Bull. Nov. Comp. Center, Comp. Science, 32 (2011), 37–56

Specification and verification of the classical sliding window protocol

D. A. Chkliaev, V. A. Nepomniaschy

Abstract. We consider the well-known Sliding Window Protocol (SWP) which provides reliable and efficient transmission of data over unreliable channels. It seems quite important to give a formal proof of correctness for the SWP, especially because the high degree of parallelism in this protocol creates a significant potential for errors. However, the efforts to provide a deductive verification for the SWP had only a limited success so far. To fill this gap, we offer a new approach, in which the protocol is specified by a state machine in the language of the verification system PVS. We also formalize its safety property and prove it using the interactive proof checker of PVS.

Keywords: *computer networks, communication protocols, Sliding Window Protocol, formal specification, automated verification, interactive theorem proving, verification system PVS*

1. Introduction

One of the best known protocols for reliable transmission of data over unreliable channels is the Sliding Window Protocol (SWP) [2, 13, 14]. Many popular communication protocols, such as TCP and HDLC, are based on the SWP.

Correctness of the SWP is far from obvious, because the protocol involves a subtle interaction of several distributed components and has a high degree of parallelism. This is why many researchers attempted to verify the protocol formally, by specifying it in some specification language and giving a formal proof of correctness using mathematical techniques. The verification was often supported by tools, such as a model checker or an interactive theorem prover. It turned out that such a rigorous study of the SWP is a rather difficult task, and there is currently no consensus among researchers on which approaches and formal techniques are best suited for the formal verification of the protocol.

In the dissertation [3], we previously presented a promising general method for specification and verification of distributed protocols and used it to verify several non-trivial examples from the field of databases. Here we demonstrate that our method can also be successfully applied to the Sliding Window protocol, leading to a compact specification with rather natural data structures and an intuitively understandable proof of correctness. Just as in [3], all our proofs are automated with the verification system PVS.

In our previous work [4], we have already verified a version of the Sliding Window protocol using our method from [3]. In that version (invented by us), an unusual timing mechanism was used to periodically remove old messages from the channels; this allowed messages in the channels to be reordered.

Here we consider the "classical" version of the SWP protocol, i.e. the version aimed at data link channels that do not allow reordering of messages. This is the version that is usually studied in the literature in the context of formal verification. It corresponds to the "go-back-n" protocol from [14], i.e. the protocol with the sending window of an arbitrary size and the receiving window equal to 1. However, unlike [14], we assume that there is no limit on message size in our protocol. Therefore, in this version the sender uses the original index of each frame in the input sequence when it transmits the frame to the receiver, instead of its remainder with respect to some fixed modulus. Such simplification of the protocol is rather common in the works dedicated to its formal verification; for example, in the context of deductive verification it is used in [9].

The rest of the paper is organized as follows. In Section 2, we give a brief introduction to the PVS system. In Section 3, an informal description of the SWP protocol is given. In Section 4, we formalize the protocol by a state machine. Section 5 presents specification and verification of the safety property for our protocol. Finally, Section 6 gives a review of related works and some remarks on the possible future work.

2. The PVS verification system

The PVS system [7], created at the Stanford Research Institute about two decades ago, is widely used for formal specification and verification of complex computer protocols and systems, especially in the area of fault-tolerant computing. It consists of a specification language, a large number of predefined theories and an interactive prover, as well as documentation, tutorials and examples, illustrating the use of PVS in several domains. PVS is able to combine an expressive specification language with powerful automated deduction, which allows it to handle many examples that present considerable difficulties for other verification systems.

The specification language of PVS is based on the classical higher-order logic. The main types include uninterpreted types, which may be introduced by a user, as well as built-in types such as Booleans, natural and real numbers. The constructors of types include functions, sets, records, tuples and enumerations, as well as recursively defined abstract datatypes (for example, lists and binary trees). It is also possible to define predicate subtypes such as the type of prime numbers. The specifications are organized into a hierarchy of parameterized theories, which contain assumptions, definitions, axioms and theorems. Expressions of the PVS language provide

usual arithmetical and logical operations, as well as application of functions, lambda abstraction and quantifiers, all with natural syntax. The predefined theories contain hundreds of useful definitions and lemmas.

In the prover of PVS, every goal or subgoal is displayed in the following form:

{-1} A1
{-2} A2
[-3] A3
...
|———
[1] B1
{2} B2
{3} B3
...

This display is a *sequent*: formulas above the dashed line (A1, A1, A3...) are called *antecedents* and those below (B1, B1, B3...) are called *consequents*. The sequent is interpreted as follows: conjunction of the antecedents implies disjunction of the consequents. The lists of antecedents and consequents may both be empty (an empty antecedent is equivalent to *true*, and an empty consequent is equivalent to *false*).

The proof of every theorem in PVS begins with a single consequent (representing the theorem). The objective of the proof is to create a *proof tree* of sequents in which all leaves are trivially true. The prover is always attempting to prove some unproved leaf in the tree. It can accomplish this task by invoking one of its commands, which either proves the current sequent (usually by applying some of the decision procedures) or splits it into several easier subgoals. When there are no more unproven branches in the tree, the prover notifies the user that the proof is complete. The resulting proof is automatically stored in a file and can be run again later. The PVS system has extensive facilities for managing the proofs and displaying information about them.

In our PVS specification of the SWP protocol, we use uninterpreted types to represent the frames transmitted by our protocol and abstract datatypes to represent the actions (transitions) of the protocol. Naturally, we also use Booleans and natural numbers to represent some of the variables in the protocol. Additional datatypes are constructed from these basic types by applying records, finite and infinite sequences and predicate subtypes. Many predicates and lambda functions are also used to generate the whole specification. Verification of the protocol relies on the PVS decision procedures for Boolean logic and arithmetical operations on natural numbers; mathematical induction is also used to prove several lemmas.

3. Informal description of the SWP protocol

Sender and receiver. In the SWP protocol there are two main components: the sender and the receiver. The sender obtains an infinite sequence of data from the *sending host*. We call indivisible blocks of data in this sequence "frames", and the sequence itself the "input sequence". The input sequence must be transmitted to the receiver via an unreliable network. After receiving a frame via the channel, the receiver may decide to *accept* the frame and eventually *deliver* it to the *receiving host*. The safety condition for the SWP protocol says that the receiver should deliver the frames to the receiving host in the same order in which they appear in the input sequence. The liveness condition expresses that each frame in the input sequence should eventually be delivered by the receiver. In this paper, we are only concerned with the safety property.

Messages and channels. In order to transmit a frame, the sender puts it into a *frame message* together with some additional information and sends it to the *frame channel*. After the receiver eventually accepts the frame message from this channel, it sends an *acknowledgment message* for the corresponding frame back to the sender. This acknowledgment message is transmitted via the *acknowledgment channel*. After receiving an acknowledgment message, the sender knows that the corresponding frame has been received by the receiver.

Sequence numbers. The sender sends the frames in the same order in which they appear in its input sequence. However, the frame channel is unreliable, so the receiver may receive these frames in a very different order (if receive at all). Therefore it is clear that each frame message must contain some information about the order of the corresponding frame in the input sequence. Such additional information is called "a sequence number". If there is no natural limit on the size of a frame message, then we can simply send the initial position of the frame in the input sequence (such as 0, 1, 2, 3 etc.) together with the frame. This is the situation that we consider in this paper. Therefore, an infinite range of sequence numbers is used by this version of the protocol.

To acknowledge a frame, the receiver sends an acknowledgment message with the sequence number with which the frame was received. Acknowledgments are "accumulative"; for example, when the sender acknowledges a frame with the sequence number 3, it means that frames with sequence numbers 0, 1 and 2 have also been accepted.

Sending window. At any time, the sender maintains a sequence of sequence numbers corresponding to frames permitted to be sent. These frames are said to be a part of the *sending window*. Similarly, the receiver maintains a *receiving window* of sequence numbers permitted to be accepted. In our protocol, the size of the sending window is represented by an arbitrary

integer N, whereas the receiving window is equal to 1.

At some point during the execution it is possible that some frames in the beginning of the sending window have already been sent but not yet acknowledged, and the remaining frames have not been sent yet. When an acknowledgment arrives for a frame in the sending window that has already been sent, this frame and all preceding frames are removed from the window as acknowledgments are accumulative. Simultaneously, the window is shifted forward, so that it again contains N frames. As a result, more frames can be sent. Acknowledgments that fall outside the window are discarded. If a sent frame is not acknowledged for a long time, it usually means that either this frame or an acknowledgment for it has been lost. To ensure the progress of the protocol, such a frame is eventually *resent*. Many different policies exist for sending and resending of frames [14], which take into account, e.g., the efficient allocation of resources and the need to avoid network congestion. Here we abstract from such details of the transmission policy and specify only those restrictions on protocol's behavior that are needed to ensure its safety property.

Receiving window. When the receiving window is equal to 1, the receiver is always waiting for a frame message with one particular sequence number. When a message arrives with a sequence number matching the expected one, it is accepted and the frame in it is delivered to the receiving host; otherwise the message is discarded. Thus in this version the actions for acceptance and delivery of frames by the receiver are combined to simplify the specification. After the delivery of a frame, the expected sequence number is updated by incrementing it by 1.

The sequence number of the last delivered frame can be sent back to the sender to acknowledge the frame. Not every frame must be acknowledged; it is possible to deliver a few frames in a row and then acknowledge only the last of them. If the receiver does not deliver any new frames for a long time, it may resend the last acknowledgment to ensure the progress of the protocol. Just as with resending of frames, no discipline is enforced in this version with regard to resending of acknowledgments: basically the acknowledgment messages may be resent at any time.

4. Specification of the protocol in PVS

4.1. Our model of distributed computation

Here we use the same methods of specification as in [3]. In that dissertation we presented a comprehensive approach to the specification and verification of fault-tolerant distributed protocols with nondeterministic behavior. In our method, a protocol itself is represented by an abstract state machine with a possibly infinite number of states and transitions between them. The desired properties of the protocol are formalized as logical formulas on all traces of

states and actions that can be generated by the protocol. We refer to [3] for a more detailed presentation of our method, as well as its comparison with related frameworks, such as process algebra and I/O automata.

Formally, in our approach a protocol is defined by the notion of a *state*, representing a snapshot of the state-of-affairs during protocol execution, and a set of *actions*. The state consists of data and control variables (which are not formally distinguished), each belonging to one of the distributed components. For example, in the SWP protocol we have four components: the sender, the receiver, the frames channel, and the acknowledgements channel. Each action is executed by one of these distributed components and changes its own variables, and, possibly, also the variables of an adjacent component with which it is interacting. Actions, which may have an arbitrary number of parameters, are specified by a precondition and an effect predicate which relates the states before and after the action execution. Execution of our protocol, or a *run*, is represented by an infinite sequence of the form $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ...s_i \xrightarrow{a_i} s_{i+1} \xrightarrow{a_{i+1}} ...$, where s_i are states, a_i are executed actions, s_0 is the initial state, each s_i satisfies the precondition of a_i, and every pair (s_i, s_{i+1}) corresponds to the effect of a_i.

In the PVS specification, the definition of runs is implemented by giving the initial state Ini, the precondition Pre, i.e. the Boolean predicate on pairs (s_i, a_i), and the effect predicate Eff, i.e. the Boolean predicate on triplets (s_i, a_i, s_{i+1}). After that, a run r is defined as a pair consisting of an infinite sequence of states $st(r)$ and an infinite sequence of actions $act(r)$ and satisfying the following three properties:
1. $st(r)(0) = Ini$;
2. for each natural index i, we have $Pre(st(r)(i), act(r)(i)) = true$;
3. for each natural index i, we have $Eff(st(r)(i), act(r)(i), st(r)(i+1)) = true$.

The rest of this section is organized as follows. In subsection 4.2, we define the data structures of the protocol in PVS. In subsection 4.3, the atomic actions of the protocol are defined and their effect on the state of the protocol is shown and briefly explained.

4.2. Data structures of the protocol

There are different ways to model both the sender and the receiver in the SWP protocol. In our model of the sender, the window "slides" over the infinite input sequence *input*. We do not specify the nature of the frames in the input sequence. Thus frames are represented by an uninterpreted (unempty) type $Frames$. The window size is represented by a positive natural number N.

```
Frames : TYPE+
N : { n : nat |  n > = 1 }
```

The variable $first$ denotes the first frame in the sending window, $ftsend$ is the first frame that has not been sent yet, and we always have $first \leq ftsend \leq first + N$. Thus, at any moment of time, frames with indices from $first$ to $ftsend - 1$ (if any) have been sent but not yet acknowledged, and frames with indices from $ftsend$ to $first + N - 1$ (if any) are in the sending window but not sent yet. The complete data structure for the sender is encoded as follows.

```
Sender : TYPE = [# input  : sequence[Frames],
                  first  : nat,
                  ftsend : nat #]
```

For the receiver, *output* is the finite output sequence, *seqnum* is the currently expected sequence number, *ackseqnum* is the last received sequence number, *mayack* is a boolean variable which tells whether we are allowed to send the acknowledgment for *ackseqnum* to the sender.

```
Receiver : TYPE = [# output    : finite_sequence[Frames],
                     seqnum    : nat,
                     ackseqnum : nat,
                     mayack    : bool #]
```

Both channels are modeled as lossy queues of unbounded capacity. Formally, the frame channel is represented by a finite sequence of frame messages, and the acknowledgment channel is represented by a finite sequence of acknowledgment messages. The messages contain a sequence number and possibly a frame. The complete state of the protocol, represented by the PVS type *SWStates*, consists of the sender, the receiver and the two channels $frameChannel$ and *ackChannel*.

```
FrameMessage : TYPE = [# seqnum : nat,
                         frame  : Frames #]

AckMessage : TYPE = [# ackseqnum : nat #]

SWStates : TYPE = [# sender       : Sender,
                     receiver     : Receiver,
                     frameChannel : finite_sequence[FrameMessage],
                     ackChannel   : finite_sequence[AckMessage] #]
```

The initial state of the protocol *SWinit* is defined in a rather obvious way, i.e. 0 or an empty sequence is assigned to most fields. Here the constant

F represents an arbitrary sequence of frames that is obtained by the sender during the execution of the protocol. At the start of the execution, no frames have been received yet. Therefore, the variable *mayack* is set to *FALSE* to disallow sending of any acknowledgements (and the variable *ackseqnum* may be set to an arbitrary value, in this case 1).

```
F : sequence[Frames]

SWinit : SWStates =
   (# sender :=
        (# input  := F,
           first  := 0,
           ftsend := 0 #),
      receiver :=
        (# output    := empty_seq,
           seqnum    := 0,
           ackseqnum := 1,
           mayack    := FALSE #),
      frameChannel := empty_seq,
      ackChannel   := empty_seq #)
```

4.3. Atomic actions of the protocol

There are six atomic actions in our protocol: two for the sender (*send* and *receiveAck*), two for the receiver (*receive* and *sendAck*), one for the frame channel (*loseFrame*) and one for the acknowledgment channel (*loseAck*). In PVS these actions are represented by the abstract datatype *Actions*, which is shown below.

```
Actions [FrameMessage, AckMessage : TYPE] : DATATYPE
BEGIN

send(index : nat) : send?
receiveAck(ackmes : AckMessage, accept : bool, index : nat) :
     receiveAck?
receive(framemes : FrameMessage, accept : bool) : receive?
sendAck : sendAck?
loseFrame(index : nat) : loseFrame?
loseAck(index : nat) : loseAck?

END Actions
```

The informal meaning of the actions and their parameters is as follows:

- *send* is an action for both the initial sending and re-sending of frames. It has a parameter indicating the index of the frame being sent in the initial sequence. For our data structure, there is no need to include the frame itself as a parameter, because it can be easily computed from its index in the initial sequence;
- *receiveAck* is an action for the receiving of acknowledgements by the sender. It has three parameters: the acknowledgement message *ackmes*, the Boolean variable *accept* indicating whether this message is accepted or rejected, and finally the index of the frame in the initial sequence to which this acknowledgement corresponds. The parameter *accept* is necessary, because only the accepted messages change the state of the sender, whereas the rejected messages are simply removed from the channel. The removal of messages can also be achieved by applying the action *loseAck*, but we believe that immediate removal of unwanted messages corresponds more closely to the informal definition of the protocol;
- *receive* is an action for the receiving of frames by the receiver. It has two parameters: the frame message *framemes* and the Boolean variable *accept* indicating whether this message is accepted or rejected. Just as for the action *receiveAck*, the parameter *accept* is necessary, because only the accepted messages change the state of the receiver;
- *sendAck* is an action for sending of an acknowledgement for the last frame received by the receiver;
- *loseFrame* is an action that erases a message with a particular index from the frame channel;
- *loseAck* is an action that erases a message with a particular index from the acknowledgement channel.

The precondition for the actions of our protocol is defined by the predicate *SWPre* which says whether the action A is allowed in the current state s:

```
SWPre(s, A) : bool =
   CASES A OF
send(i) : (i = ftsend(sender(s)) &
              ftsend(sender(s)) < first(sender(s)) + N) OR
           (i >= first(sender(s)) & i < ftsend(sender(s))),
receiveAck(ackmes, accept, i) :
   length(ackChannel(s)) > 0 & ackmes = ackChannel(s)(0) &
   (accept <=> (i = ackseqnum(ackmes) &
                i >= first(sender(s)) & i < ftsend(sender(s)))),
```

```
receive(framemes, accept) :
   length(frameChannel(s)) > 0 & framemes = frameChannel(s)(0) &
   (accept <=> seqnum(framemes) = seqnum(receiver(s))),
sendAck : mayack(receiver(s)),
loseFrame(i) : i < length(frameChannel(s)),
loseAck(i) : i < length(ackChannel(s))
   ENDCASES
```

The informal meaning of the predicate *SWPre* for particular actions can usually be easily deduced from the description of these actions given above:

- for the action *send*, the predicate checks that its index lies within an acceptable range;
- for the action *receiveAck*, the predicate checks that *ackmes* is the first message in the acknowledgement channel and its sequence number corresponds to some sequence number that is currently in the sending window;
- for the action *receive*, the predicate demands that *framemes* is the first message in the frame channel and its sequence number is equal to the sequence number that is currently expected by the receiver;
- for the action *sendAck*, the predicate demands that this action is allowed by the boolean variable *mayack*;
- for the action *loseFrame*, its parameter must be meaningful, i.e. it should not exceed the length of the frame channel;
- for the action *loseAck*, its parameter must be meaningful, i.e. it should not exceed the length of the acknowledgement channel.

The effect of the actions of our protocol is defined by the predicate *SWEffect* which says whether a new state $s1$ can be obtained from the current state $s0$ by applying the action A:

```
SWEffect(s0, A, s1) : bool =
   CASES A OF

send(i) : SendEffect(s0, i, s1),
receiveAck(ackmes, accept, i) : ReceiveAckEffect(s0, accept, i, s1),
receive(framemes, accept) : ReceiveEffect(s0, framemes, accept, s1),
sendAck : SendAckEffect(s0, s1),
loseFrame(i) : LoseFrameEffect(s0, i, s1),
loseAck(i) : LoseAckEffect(s0, i, s1)

   ENDCASES
```

The effect predicates for particular actions are somewhat large and cumbersome, but for our model this probably cannot be avoided, because each action may change several variables at once. Due to their large size, these effect predicates are defined separately in the preceding text of the specification. Below we show and briefly explain the effect for each of the six actions of our protocol. Note that in PVS, if a finite sequence FS is of length L, then its elements are numbered from 0 to $L-1$. We call the initial element $FS(0)$ *the beginning* of FS, and the final element $FS(L-1)$ *the end* of *FS*.

send(i). This action sends the first frame that has not been sent yet, i.e. a frame with the index *ftsend*, and this index is included into the message as its sequence number. It may also resend a frame that has already been sent but not yet acknowledged, i.e. a frame with the index i such that $i \geq first$ and $i < ftsend$. The definition of *SendEffect* uses an auxiliary function *addLastFM*: if *fmes* is a frame message and *fmesFS* a finite sequence of frame messages, then *addLastFM(fmes, fmesFS)* is a finite sequence of frame messages in which *fmes* is appended to the end of *fmesFS*.

```
SendEffect(s0, i, s1) : bool =
   IF (i = ftsend(sender(s0))) THEN
      s1 = s0 WITH
         [ sender := sender(s0) WITH
              [ ftsend := ftsend(sender(s0)) + 1 ],
           frameChannel :=
              addLastFM((# seqnum := i,
                           frame := input(sender(s0))(i) #),
                        frameChannel(s0)) ]
   ELSE s1 = s0 WITH
         [ frameChannel :=
              addLastFM((# seqnum := i,
                           frame := input(sender(s0))(i) #),
                        frameChannel(s0)) ]
   ENDIF
```

receiveAck(ackmes, accept, i). This action receives the acknowledgment message *ackmes* and checks whether its sequence number lies within the sending window. If so, the frames with sequence numbers up to *acksegnum(ackmes)* are removed from the window and the window is shifted accordingly. The definition of *ReceiveAckEffect* uses an auxiliary function *removeFirstAM*: if *amesFS* is a finite sequence of acknowledgement messages with at least one element, then *removeFirstAM(amesFS)* removes from *amesFS* its initial element.

```
ReceiveAckEffect(s0, accept, i, s1) : bool =
   IF (accept = TRUE) THEN
         s1 = s0 WITH
             [ sender := sender(s0) WITH [ first := i + 1 ],
               ackChannel := removeFirstAM(ackChannel(s0)) ]
   ELSE
      s1 = s0 WITH [ ackChannel := removeFirstAM(ackChannel(s0))]
   ENDIF
```

receive(frameмes, accept). This action receives the frame message *frameмes* and checks whether its sequence number corresponds to the sequence number expected by the receiver. If so, it accepts the message and appends its frame to the output sequence; otherwise the message is discarded. The definition of *ReceiveEffect* uses an auxiliary function *removeFirstFM*: if *fmesFS* is a finite sequence of frame messages with at least one element, then *removeFirstFM(fmesFS)* removes from *fmesFS* its initial element. Also, an additional function *addLastFrame* is used: if *fr* is a frame and *frFS* is a finite sequence of frames, then *addLastFrame(fr, frFS)* is a finite sequence of frames in which *fr* is appended to the end of *frFS*.

```
ReceiveEffect(s0, frameмes, accept, s1) : bool =
   IF (accept = TRUE) THEN
       s1 = s0 WITH
          [ receiver := receiver(s0) WITH
             [ output :=
                  addLastFrame(frame(frameмes),
                         output(receiver(s0))),
               seqnum := seqnum(receiver(s0)) + 1,
               ackseqnum := seqnum(receiver(s0)),
               mayack := TRUE ],
            frameChannel := removeFirstFM(frameChannel(s0)) ]
   ELSE
      s1 = s0 WITH [ frameChannel :=
                removeFirstFM(frameChannel(s0)) ]
   ENDIF
```

sendAck. If *mayack* is true, this action sends an acknowledgment for the last received frame, i.e. the frame with the sequence number *ackseqnum*. The definition of *SendAckEffect* uses an auxiliary function *addLastAM*: if *amеs* is an acknowledgment message and *amesFS* is a finite sequence of acknowledgment messages, then *addLastAM(amеs, amesFS)* is a finite sequence of acknowledgment messages in which *amеs* is appended to the end of *amesFS*.

```
SendAckEffect(s0, s1) : bool =
   s1 = s0 WITH [ ackChannel :=
       addLastAM((# ackseqnum := ackseqnum(receiver(s0)) #),
          ackChannel(s0)) ]
```

loseFrame(i). This action formalizes a loss of a message with the index i from the frame channel. The definition of its effect uses an auxiliary function *removeFM*: if *fmesFS* is a finite sequence of frame messages with at least one element and i is an index smaller than the length of *fmesFS*, then *removeFM(fmesFS, i)* removes from *fmesFS* its element with the index i.

```
LoseFrameEffect(s0, i, s1) : bool =
   s1 = s0 WITH [ frameChannel := removeFM(frameChannel(s0), i)]
```

loseAck(i). This action models a loss of a message with the index i from the acknowledgment channel. The definition of its effect uses an auxiliary function *removeAM*: if *amesFS* is a finite sequence of acknowledgment messages with at least one element and i is an index smaller than the length of *amesFS*, then *removeAM(amesFS, i)* removes from *amesFS* its element with the index i.

```
LoseAckEffect(s0, i, s1) : bool =
   s1 = s0 WITH [ ackChannel := removeAM(ackChannel(s0), i) ]
```

5. Specification and verification of the safety property

The SWP protocol is correct with respect to safety, if the receiver always delivers the frames to the receiving host in the same order in which they appear in the input sequence. In our model, we prefer to define correctness in terms of states rather than actions. Note that in each state frames that have already been delivered to the receiving host are represented by the output sequence. Therefore, the safety property for a particular state s can be expressed by a predicate which says that the output sequence is the prefix of the input sequence (note that in PVS the elements of a sequence are enumerated starting with 0, not with 1):
$Safe(s) = \forall\, i : i < length(output(s)) \Rightarrow output(s)(i) = input(s)(i)$

As defined in Section 4.1, *st(r)* and *act(r)* denote the sequence of states and sequence of actions of a run r, respectively. We define the run r to be safe, if it is safe in every state:
$Safety(r) = \forall\, j : Safe(st(r)(j))$

In order to establish the safety property for our protocol, we need to prove in PVS the following theorem called Main:
$\forall\, r : Safety(r)$ Main

The proof of the theorem Main consists of about 20 PVS theorems and lemmas, and it took the first author from 2 to 3 weeks to develop the proof. Checking the proof takes less than 25 seconds on a regular PC. Below we present the proof itself.

Proof of the theorem Main. Like all PVS proofs, our proof is structured as a tree. The root of our tree is the theorem Main, and most of its leaves are lemmas IniLem, PreLem and EffLem, which will be given below. These lemmas, which we call *elementary lemmas*, follow directly from the definition of runs as it was given in Section 4.1. Note that in that general definition we must replace the initial state Ini and the predicates Pre and Eff by their instances SWinit, SWPre and SWEffect given in Sections 4.2 and 4.3.

The lemma IniLem expresses that the first state in any run must be equal to the initial state. It follows directly from clause 1 in the definition of runs.

$$\forall r : st(r)(0) = SWinit \qquad \text{IniLem}$$

The lemma PreLem means that each action with an index i must be allowed in the state with the same index by the precondition predicate. It follows directly from clause 2 in the definition of runs.

$$\forall r, i : SWPre(st(r)(i), act(r)(i)) \qquad \text{PreLem}$$

Finally, the elementary lemma EffLem expresses that each action with an index i must transform the state with the same index according to the effect predicate. It follows directly from clause 3 in the definition of runs.

$$\forall r, i : SWEffect(st(r)(i), act(r)(i), st(r)(i+1)) \qquad \text{Efflem}$$

Now we continue with the proof. Let r be an arbitrary run. We denote as $output_r(i)$ and $input_r(i)$ the output and input sequences in a state with the index i, and the length of the output sequence as $LO_r(i)$. The proof is by induction on the length of the output sequence. We prove the following theorem MainInduct, which expresses the relation between the length of the output and the safety property, by induction on k:

$$\forall r, k, i : LO_r(i) = k \Rightarrow Safe(st(r)(i)) \qquad \text{MainInduct}$$

It is obvious that MainInduct implies our main theorem (if we take as k the current length of the output sequence). In the proof of MainInduct, the basis of the induction is trivial, because a sequence of length 0 has no elements. It remains to prove the induction step. Suppose that the theorem has been proved for any output length not greater than k, and that we are currently in a state with the index i such that $LO_r(i) = k + 1$. In order to prove $Safe(st(r)(i))$, we need to introduce 5 additional lemmas L1, L2, L3, L4, and L5, the proofs of which will be given later. The lemma L1 expresses that the output can only be changed by the action *receive*, which accepts the message being received:

$\forall\, r, i : output_r(i+1) \neq output_r(i) \Rightarrow$
$receive?(act(r)(i))\ \&\ accept(act(r)(i))$ L1

We will also use the lemma L2, which expresses that if the current length of the output is equal to some arbitrary natural number *n + 1*, then there was a preceding state in the run, when the length of the output increased from *n* to *n + 1*, and the output remained the same from the resulting state to the current state:

$\forall\, r, k, n : LO_r(k) = n+1 \Rightarrow$
$\exists\, l : l < k\ \&\ LO_r(l) = n\ \&\ LO_r(l+1) = n+1\ \&$
$output_r(l+1) = output_r(k)$ L2

Applying the lemmas L1 and L2, we obtain that there exists an index l such that $l < i$, $LO_r(l) = k$, $LO_r(l+1) = k+1$, $act(r)(l) = receive$, $accept(act(r)(l)) = true$ and $output_r(l+1) = output_r(i)$. Thus in a state with the index l we increased the length of the output from k to the current value of $k+1$, and after this increase the output sequence remained the same. We can now apply the induction hypothesis to the state with the index l, which gives us $Safe(st(r)(l))$. Now we split the proof of $Safe(st(r)(i))$ into two parts (*) and (**):

(*) Safe(st(r)(l)) ⇒ Safe(st(r)(l + 1))
(**) Safe(st(r)(l + 1)) ⇒ Safe(st(r)(i))

First we prove the easy part (**). Its proof is based on the following lemma L3, which says that the input sequence remains the same in all states of any run:

$\forall\, r, i, j : input_r(i) = input_r(j)$ L3

Applying L3, we obtain $input_r(l+1) = input_r(i)$. But we have already proved that $output_r(l+1) = output_r(i)$. So both the input and output are the same in the state with the index i as in the state with the index $l+1$. Therefore if the output is the prefix of the input in the state with the index $l+1$, this is still the case in the state with the index i. This completes the proof of (**).

The lemma L3 allows us to introduce a new abbreviation: in the rest of the proof we denote the input sequence in *any* state of a run r as $Input_r$. We also denote the frame channel in a state with the index i as $FChan_r(i)$, and the sequence number expected by the receiver in a state with the index i as $SNum_r(i)$.

Next we prove the more difficult part (*). The proof uses two additional lemmas L4 and L5. The definition of L4 includes an auxiliary predicate *Fbelongs* expressing that a frame message *fmes* belongs to a finite sequence of frame messages *fmesFS*. The lemma L4 says that if a frame message *fmes* belongs to the frame channel of any state in a run r and its sequence number is some natural number n, then the frame of *fmes* originated from the same index n in the input sequence.

$Fbelongs(fmes, fmesFS) = \exists\, m : m < length(fmesFS)\,\&\, fmes = fmesFS(m)$
$\forall\, r, fmes, i : Fbelongs(fmes, FChan_r(i))\,\&\, seqnum(fmes) = n \Rightarrow$
$frame(fmes) = Input_r(n)$ L4

The lemma L5 expresses that if the length of the output in any state of a run r is some natural number k, then the receiver is currently expecting the frame message with the same sequence number k:

$\forall r, k, i : LO_r(i) = k \Rightarrow SNum_r(i) = k$ L5

Now we denote as FM the frame message that was received and accepted in a state with the index l: FM = *frameme(act(r)(l))*. Applying the elementary lemma PreLem, we obtain $FM = FChanr(l)(0)$ and $seqnum(FM) = SNum_r(l)$. Applying the lemma L5, we obtain $SNum_r(l) = k$, so $seqnum(FM) = k$. It is clear that FM belongs to the frame channel in a state with the index l, so the lemma L4 gives us $frame(FM) = Input_r(k)$. If we now apply the elementary lemma EffLem, we obtain $output_r(l+1) = addLastFrame(Input_r(k), output_r(l))$. The definition of *addLastFrame* now gives us that in a state with the index *l + 1* the output is still the prefix of the input. This completes the proof of (*) and of the theorem Main.

We present brief proofs of lemmas L1 - L5 in the appendix of this paper.

6. Conclusion

A significant number of publications have already been dedicated to verification of the Sliding Window protocol, so here we can mention only some of them. In the area of model checking, some versions of the protocol have been checked in [8, 5, 12]. These works cannot be considered particularly successful, because the state explosion (due to the massive parallelism present in the protocol) impedes the checking of the SWP for large parameter values. For example, in [12] the SWP was verified for a window of size 16, which is at least two orders less than the window size in some industrial implementations of the protocol (such as TCP).

The deductive verification of the SWP also faced considerable challenges. In one of the first papers on the protocol [13], only an informal manual proof is given. A semi-formal manual proof is also presented in [6]. An excellent verification of the protocol was given in [1] in the framework of the process algebra, and the proof was automated with PVS. Unfortunately, their proof is somewhat incomplete: it relies on some complicated results from the field of the process algebra which have not been proved in PVS. Some researchers also considered verification of the SWP for transport channels that allow reordering of messages. For such channels, an interesting verification of the protocol for an untimed system is given in [11] (with some automation of the proof), and for a timed system in [10] (with only a manual proof).

In the introduction, we have already mentioned verification of the SWP with an unlimited message size presented in [9]. Unlike our paper, they consider the receiving window of an arbitrary size instead of just 1. Their approach to specification of the protocol and its safety property (based on extended automata) is somewhat similar to the one that was presented in this work, and their proof is also automated with PVS. However, in their specification the frames are represented by natural numbers, whereas we represent them by an arbitrary datatype. This makes our modeling considerably more general. We also believe that the data structures used by our specification are more natural. For example, most sequences in our specification are finite, whereas in [9] only infinite sequences are used. We believe that the use of finite sequences (which are easily programmed using arrays) makes our specification closer to possible implementations of the protocol (such as that presented in [14]), because infinite sequences cannot be directly represented in a programming language.

In our future work, we would like to verify a version of the SWP with the receiving window of an arbitrary size, i.e. corresponding to the "selective repeat" protocol from [14]. This would make our investigation of the protocol more complete and also allow us to more accurately compare our modeling of it with the approach from [9].

References

[1] Badban B., Fokkink W., Groote J.F., Pang J., van de Pol J. Verification of a sliding window protocol in μCRL and PVS // Formal Aspects of Computing. – 2005. – Vol. 17(3). – P. 342–388.

[2] Cerf V.G., Kahn R.E. A protocol for packet network intercommunication // IEEE Trans. on Communs. – 1974. – Vol. COM-22. – P. 637–648.

[3] Chkliaev D. Mechanical Verification of Concurrency Control and Recovery Protocols: PhD thesis. / Eindhoven University of Technology. – 2001. – Available at http://alexandria.tue.nl/extra2/200112908.pdf

[4] Chkliaev D., Hooman J., de Vink E.P. Verification and improvement of the sliding window protocol // Lect. Notes Comput. Sci. – 2003. – Vol. 2619. – P. 113–127.

[5] Kaivola R. Using compositional preorders in the verification of sliding window protocol // Lect. Notes Comput. Sci. – 1997. – Vol. 1254. – P. 48–59.

[6] Knuth, D.E. Verification of link-level protocols // BIT. – 1981. – Vol. 21. – P. 31–36.

[7] Owre S., Rushby J.M., Shankar N. PVS: A prototype verification system // Lect. Notes Comput. Sci. – 1992. – Vol. 607. – P. 748–752.

[8] Richier J.L., Rodriguez C., Sifakis J., Voiron J. Verification in Xesar of the sliding Window protocol // Protocol specification, testing and verification. – 1987. – Vol. 7. – P. 235–248.

[9] Rusu V. Verifying a sliding-Window Protocol using PVS // Formal Description Techniques (FORTE'01). – 2001. – P. 251–266.

[10] Shankar A.U. Verified data transfer protocols with variable flow control // ACM Trans. on Comput. Systems. – 1989. – Vol. 7. – P. 281–316.

[11] Smith M., Klarlund N. Verification of a sliding window protocol using IOA and MON // Formal methods for distributed system development. – 2000. – P. 19–34.

[12] Stahl K., Baukus K., Lakhnech Y., Steffen M. Divide, abstract, and model-check // Lect. Notes Comput. Sci. – 1999. – Vol. 1680. – P. 57–76.

[13] Stenning N.V. A data transfer protocol // Computer Networks. – 1976. – Vol. 1. – P. 99–110.

[14] Tanenbaum A.S. Computer Networks (Third Edition). – Prentice-Hall International, 1996.

Appendix: the proofs of lemmas L1–L5

Proof of lemma L1. The proof trivially follows from the elementary lemma EffLem and the definition of the SWEffect predicate. Indeed, suppose that the output in the state *st(r)(i + 1)* is different from the output in the previous state *st(r)(i)*. In the state *st(r)(i)*, all 6 actions of our protocol are possible. By examining the parts of the SWEffect predicate corresponding to the actions *send*, *receiveAck*, *sendAck*, *loseFrame* and *loseAck*, we can easily determine that these actions do not change the receiver at all. It is also clear from this predicate that if the receiver receives a frame message but rejects it, then its variables do not change as well. Thus the variable of the receiver *output* can only be changed by the action *receive* that accepts the message being received, and this completes the proof of the lemma L1.

Proof of lemma L2. The proof uses an additional lemma L2A, which expresses that receiving and accepting a frame message increases the length of the output exactly by 1:

$$\forall r, i : receive?(act(r)(i)) \,\&\, accept(act(r)(i)) \Rightarrow$$
$$LO_r(i+1) = LO_r(i) + 1 \qquad \text{L2A}$$

The proof of the lemma L2A follows directly from the elementary lemma EffLem and the predicate SWEffect. If in a state with the index i the receiver receives and accepts some frame message *FM*, they imply that the output sequence is transformed in the following way: $output_r(i + 1) =$

$addLastFrame(frame(FM), output_r(i))$. The definition of *addLastFrame* now implies that, indeed, the length of the output is increased exactly by 1.

Using the lemmas L1 and L2A, we can now prove the lemma L2 by induction on a natural index k. If $k = 0$, we have $LO_r(0) = n + 1$, i.e. the length of the output in a state with the index 0 is greater than 0. But this contradicts the elementary lemma IniLem and the definition of the initial state for our protocol, because according to this definition the initial output sequence is empty. This proves the basis of induction. Now we must make the induction step. Suppose $LO_r(k + 1) = n + 1$. According to the lemmas L1 and L2A, the output either did not change from the previous state, or its length was increased exactly by 1. Let us consider each of these cases:

- $LO_r(k) = n + 1$ and $output_r(k + 1) = output_r(k)$. Applying the induction hypothesis, we obtain that there exists l such that $l < k$, $LO_r(l) = n, LO_r(l + 1) = n + 1$ and $output_r(l + 1) = output_r(k)$. It is clear that $l < k + 1$ and $output_r(l + 1) = output_r(k + 1)$, so we can instantiate the same index l as the index required by the lemma. This completes the proof of the first subcase.
- $LO_r(k) = n$. If we instantiate $l = k$, we can see that all conditions on the index l required by the lemma are trivially satisfied. This completes the proof of the second subcase and of the lemma L2.

Proof of lemma L3. The proof is based on the following lemma L3Induct, which is proved by induction on a natural index k:

$\forall r, i, k : input_r(i + k) = input_r(i)$ L3Induct

In the proof of L3Induct, the basis of induction is trivial. Let us consider the induction step. Suppose $input_r(i + k) = input_r(i)$. We must prove $input_r(i + k + 1) = input_r(i)$. Similar to the proof of L1, we apply the elementary lemma EffLem and consider all 6 actions that are possible in a state with the index $i+k$. It follows from the parts of the SWEffect predicate corresponding to these actions that none of them changes the input sequence. Thus $input_r(i + k + 1) = input_r(i + k)$, and this obviously completes the induction step and the proof of L3Induct.

The lemma L3Induct trivially implies the lemma L3: if $i > j$, we take $k = i - j$, otherwise we take $k = j - i$. This completes the proof of the lemma L3.

Proof of lemma L4. The proof is based on the following lemma L4A:

$\forall r, fmes, i : \neg Fbelongs(fmes, FChan_r(i))$
$\& Fbelongs(fmes, FChan_r(i + 1)) \Rightarrow send?(act(r)(i))$ L4A

The lemma L4A intuitively means that a frame message may enter the frame channel only as a result of a send action. Like several previous lemmas, it is easily proved using the elementary lemma EffLem and the definition of

the SWEffect predicate. The action *send* is not the only one that changes the frame channel: the channel is also changed by the actions *receive* and *loseFrame*. However, by considering the parts of the SWEffect predicate corresponding to these actions, we prove that both of them can only remove messages from the channel. Therefore the action *send* is the only one that can add messages to the frame channel, and this completes the proof of the lemma L4A.

Using L4A, the lemma L4 can easily be proved. If a frame message *fmes* is currently in the frame channel, it is easy to prove that there was a moment in the past when it was added to the channel. Thus there exists an index j ($j < i$) such that *fmes* does not belong to the frame channel in a state with the index j but belongs to the channel in a state with the index $j + 1$. Now we apply the lemma L4A and obtain that *fmes* was added to the channel as a result of the action *send*. It remains to consider the part of the SWEffect predicate corresponding to the action *send*. This immediately gives us that the frame in *fmes* was taken from the position in the input sequence that is exactly equal to the sequence number in *fmes*, completing the proof of the lemma L4.

Proof of lemma L5. The lemma is proved by induction on a natural index k. First we prove the basis of induction. If $k = 0$, then in the current state we have $LO_r(i) = 0$. It is easy to prove, using the SWEffect predicate, that as long as the length of the output does not change, the sequence number expected by the receiver stays the same as well. Here the length of the output did not change from the initial state, and this implies that $SNum_r$ remained the same as well. Thus it must be currently equal to 0, and this completes the basis of induction.

Now we must prove the induction step. Suppose $LO_r(i) = k + 1$. Applying the lemmas L1 and L2 in the same way as we did in the proof of the theorem MainInduct, we obtain that there exists an index l such that $l < i$, $LO_r(l) = k, LO_r(l+1) = k+1, act(r)(l) = receive, accept(act(r)(l)) = true$ and $output_r(l + 1) = output_r(i)$. We now apply the induction hypothesis to a state with the index l, which gives us $SNum_r(l) = k$. The effect of the action *receive* in a state with the index l now gives us $SNum_r(l+1) = k+1$. If the output did not change from the state with the index $l + 1$ to the state with the index i, then its length also did not change. But we already know from the proof of the basis of induction, that in this case $SNum_r$ also stayed the same during that interval. This implies $SNum_r(i) = k + 1$, and this completes the induction step and the proof of L5.