Formal Verification of an Improved Sliding Window Protocol

Dmitri Chkliaev(1) Jozef Hooman(2) Erik de Vink(1)

(1) Eindhoven University of Technology
Dept. of Computing Science
P.O. Box 513, 5600 MB Eindhoven
the Netherlands
Email: { dmitri,evink }@win.tue.nl

(2) University of Nijmegen
Dept. of Computing Science
P.O. Box 9010, 6500GL Nijmegen
the Netherlands
Email: hooman@cs.kun.nl

Abstract— **The well-known Sliding Window protocol caters for the reliable and efficient transmission of data over unreliable channels that can lose, reorder and duplicate messages. Despite the practical importance of the protocol and its high potential for errors, it has never been formally verified for the general setting. We try to fill this gap by giving a fully formal specification and verification of an improved version of the protocol. The protocol is specified by a timed state machine in the language of the verification system PVS. This allows a mechanical check of the proof by the interactive proof checker of PVS. Our modelling is very general and includes such important features of the protocol as sending and receiving windows of arbitrary size, bounded sequence numbers and the three types of channel faults mentioned above.**

Keywords— **distributed networks, communication protocols, formal specification, mechanized verification, PVS**

I. INTRODUCTION

Reliable transmission of data over unreliable channels is an old and well-studied problem in computer science. Without a satisfactory solution, computer networks would be useless, because they transmit data over channels that often lose, duplicate or reorder messages. One of the most efficient protocols for reliable transmission is the Sliding Window (SW) protocol [20]. Many popular communication protocols such as TCP and HDLC are based on the SW protocol.

Communication protocols usually involve a subtle interaction of a number of distributed components and have a high degree of parallelism. This is why their correctness is difficult to ensure, and many protocols turned out to be erroneous. One of the most promising solutions to this problem is the use of formal verification, which requires the precise specification of the protocol in some specification language and a formal proof of its correctness by mathematical techniques. Formal verification is especially useful when it uses some form of mechanical support, such as a model checker or an interactive theorem prover.

This research is supported by the PROGRESS project EES5202, "Modelling and performance analysis of telecommunication systems".

However, formal verification of communication protocols is notoriously difficult. Although a number of specification and verification techniques exist, including Hoare logic [11], temporal logic [15], automata [14] and process algebra [3], many of them have only been applied to toy examples. Even verification of a version of the Alternating Bit protocol [4] (which is one of the simplest communication protocols), namely the bounded retransmission protocol (BRP) turned out to be nontrivial. The use of model checking for verification of the BRP is problematic due to the infinite state space of the protocol (caused by unboundedness of the message data, the retransmission bound, and the file length), and mechanization of the correctness proof by interactive theorem provers revealed many technical difficulties [8], [10], [9].

Despite the practical significance of the Sliding Window protocol, relatively little has been done on its formal verification. Stenning [20] only gave an informal manual proof for his protocol. A semi-formal manual proof is also presented in [13], and some versions of the protocol have been model-checked for small parameter values in [16], [12], [7]. The combination of abstraction techniques and model-checking in [19] allowed to verify the SW protocol for a relatively large window size of 16 (which is still a few orders less than a possible window size in TCP). The verifications [16], [12], [7], [19] assume *data link channels*, which can only lose messages. The protocols for such channels, called *data link protocols*, are important (they include, e.g., HDLC, SLIP and PPP protocols), but they are only used for transmission of data over relatively short distances.

In this paper, we study the verification of sliding window protocols for more general *transport channels*, which can also reorder and duplicate messages. Such channels are already considered in the original paper on the SW protocol by Stenning [20]. The protocols for such channels (called *transport protocols*), such as TCP, can transmit data over very large networks such as Internet.

Note that not for all types of transport channels a SW protocol exists. As [2] shows, for a fully asynchronous system and channels that can both lose and reorder messages, it is impossible to design an efficient transmission protocol that uses bounded sequence numbers. Similar result is proved for systems that can both reorder and duplicate messages [23]. In [18], unbounded sequence numbers are assumed for verification of the SW protocol for transport channels. This makes the verification rather simple, because it is known that the repetition of sequence numbers is the main source of errors for SW protocols [22].

Unfortunately, transmission protocols that use unbounded sequence numbers are usually not practical. Because of the impossibility results mentioned above, a SW protocol for transport channels with bounded sequence numbers can only be designed for systems, in which each message in a channel has a maximum lifetime[1]. Such a SW protocol is a part of the TCP protocol, which operates over transport channels with a given maximum packet lifetime. The theoretical basis of that protocol is presented in [21]. TCP uses 2^{32} sequence numbers, which is enough to represent 4 gigabytes of data. The transmission mechanism of TCP uses a complicated timing mechanism to implement sequence numbers in such a way that their periodical repetition does not cause ambiguity. It often requires the sender and the receiver to synchronize on the sequence numbers they use. Such synchronization is provided by the three-way handshake protocol, which is not a part of the SW protocol and correctness of which is not easy to ensure. In general, the transmission mechanism of TCP seems too complicated and too specific for TCP to serve as a good starting point for verification of SW protocols for transport channels.

Another approach is chosen in [17]. Shankar presents a version of the SW protocol for transport channels with the maximum packet lifetime, which does not require any synchronization between the sender and the receiver, and also does not impose any restrictions on the transmission policy. However, the range of sequence numbers, required to ensure the correctness of his protocol, depends on the maximum transmission rate of the sender. In the case of TCP, his protocol would only work correctly if the sender did not send into the channel more than some 30 megabytes of data per second (if we take 120 seconds for the maximum packet lifetime in TCP, as in [22]). Such restriction may not be practical for modern networks, which are getting faster every year. Indeed, the range of sequence numbers in a large industrial protocol like TCP is fixed. Therefore, if the available transmission rate at some point exceeds our expectations, we would need to re-design the whole protocol to allow for faster transmission, which may be costly. From the formal point of view, the need to reason about the maximum transmission rate made the verification of Shankar's protocol in [17] very large and complicated.

In this paper, we present a new version of the SW protocol for transport channels. In our opinion, it combines some of the best features of the transmission mechanism of TCP and Shankar's protocol. We do not require any synchronization between the sender and the receiver. Maximum packet lifetime and appropriate transmission and acknowledgment policies are used to ensure the correct recognition of sequence numbers. These policies are rather simple; roughly speaking, they require the sender (receiver) to stop and wait for the maximum packet lifetime after sending (receiving) the maximum sequence number. Unlike some previous works [20], [17], the range of sequence numbers used by our protocol does not depend on the transmission rate of the sender. Therefore, between the required periods of waiting, the sender may transmit data arbitrarily fast, even if the range of sequence numbers is fixed[2], e.g. as in TCP. If implemented for TCP, our protocol would allow to transmit files up to 4 gigabytes arbitrarily fast.

Even for relatively simple communication protocols such as a One-bit Sliding Window protocol [5], manual formal verification is so lengthy and complicated that it can easily be erroneous. This is why we need some form of mechanical support. Despite its relative simplicity, our protocol highly depends on complex data structures. It also uses several parameters of arbitrary size, such as the window size and the range of sequence numbers. Hence completely automatic verification is not feasible for us. This is why we use an interactive theorem prover. We have chosen PVS [1], because we have an extensive experience with it and successfully applied it to verification of several complicated protocols [6]. PVS, which is based on a higher-order logic, has a convenient specification language and is relatively easy to learn and to use.

The rest of the paper is organized as follows. In section II, we give an informal description of our protocol. In section III, we formalize the protocol by a timed state machine. Section IV outlines the proof of correctness property for our protocol. Some concluding remarks are given

[1]Such protocols can also be designed for untimed systems which limit the reordering of messages in a channel [13], but such systems seem to be only of theoretical interest.

[2]Of course, the *average* transmission rate of our protocol over the long run does depend on the range of sequence numbers, because the fewer sequence numbers the protocol has, the more often it has to stop and wait after the maximum number.

in section V.

II. Protocol overview

Sender and receiver. In a SW 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 the frames, the receiver eventually *delivers* them to the *receiving host.* The correctness condition for a SW 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.

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 receives 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. Thus the communication between the sender and the receiver is bi-directional; the sender transmits frames to the receiver via the frame channel, and the receiver transmits acknowledgments for these frames to the sender via the acknowledgment channel.

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 "sequence number". If we include as a sequence number the exact position of the frame in the input sequence, it would make sequence numbers used by our protocol unbounded (because conceptually the input sequence is infinite). As we already explained in the introduction, unbounded sequence numbers are not practical. This is why in a SW protocol, instead of the exact position of the frame in the input sequence, the sender sends the remainder of this position with respect to some fixed modulus K. The value of K varies greatly among protocols: it is only 16 for the Mascara protocol for wireless ATM networks, but 2^{32} for TCP. To acknowledge a frame, the receiver sends in the acknowledgment message the sequence number for which the frame was received. It should be noted that acknowledgments are "accumulative"; for example, when the sender acknowledges a frame with sequence number 3, it means that frames with sequence numbers 0, 1 and 2 have also been received.

Sending and receiving windows. At any time, the sender maintains a sequence of sequence numbers corresponding to frames it is permitted to send. These frames are said to be a part of the *sending window.* Similarly, the receiver maintains a *receiving window* of sequence numbers it is permitted to receive. In our protocol, the sizes of sending and receiving windows are equal and represented by an arbitrary integer N.

At some point during the execution it is possible that some frames in the beginning of the sending window are already sent, but not yet acknowledged, and the remaining frames are not sent yet. When an acknowledgment arrives for a frame in the sending window that is already sent, this frame and all preceding frames are removed from the window as acknowledgments are accumulative. Simultaneously, the window is shifted forward, such that it again contains N frames. As a result, more frames can be sent either immediately or later. 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 frame is eventually *resent.* Many different policies for sending and resending of frames exist [22], which take into account, e.g., the efficient allocation of resources and the need to avoid network congestion. Here we are only concerned with the correctness of the protocol, so we abstract from the details of the transmission policy and specify only those restrictions on protocol's behaviour that are needed to ensure safety.

During the execution, the receiving window is usually a mix of sequence numbers corresponding to frames that have been received out of order and sequence numbers corresponding to "empty spaces", i.e. frames that are still expected. When a frame arrives with a sequence number corresponding to some empty space, it is inserted in the window, otherwise it is discarded. At any time, if the first element of the receiving window is a frame, it can be delivered to the receiving host, and the window is shifted by one. The sequence number of the last delivered frame can be sent back to the sender to acknowledge the frame (for convenience reasons, in this version we acknowledge delivered frames instead of received frames). It should be noted that 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.

Potential ambiguity. It is shown in [22], that for data link channels we need $K \geq 2 * N$ to ensure the unambiguous recognition of sequence numbers. However, for transport channels this condition is not sufficient. Indeed, suppose that window size $N = 1$ and we use $K = 2$ sequence numbers, so we only have sequence numbers 0 and 1. Suppose the sender sends the first two frames $f0$ and $f1$ to the receiver, which are successfully received, delivered and acknowledged. Suppose, however, that the first of these frames has been duplicated in the frame buffer, so the buffer still contains frame $f0$ with sequence number 0. The receiver now has 0 in its window, so it can receive frame $f0$ for the second time, violating the safety property.

This simple example clearly shows that we need additional restrictions on the protocol to recognize sequence numbers correctly. Traditional approaches [20], [17] introduce a stronger restriction on K, which essentially has the form $K \geq 2 * N + f(Rmax, Lmax)$, where $Rmax$ is the maximum transmission rate of the sender, $Lmax$ is the maximum message lifetime in the frame and acknowledgment channels, and f is some function. As we already explained in the introduction, such dependence between the range of sequence numbers and the maximum transmission rate is undesirable. This is why in our protocol we only require $K \geq 2 * N$, but introduce timing restrictions on the transmission and acknowledgment policies to ensure that frames and acknowledgments are not received more than once. These timing restrictions of our protocol are explained below.

A. *Timing restrictions*

In our protocol, the sender is allowed to resend sequence number 0 and all subsequent sequence numbers only after more than $Lmax$ time units have passed since the receipt of the acknowledgment for the maximum sequence number $K - 1$. This is necessary to ensure that when sequence number 0 is resent, all "old" acknowledgments, i.e. those for frames preceding the current frame, are already removed from the acknowledgment channel (because their timeouts expired), and cannot be mistaken for "new" acknowledgments, i.e. those for the current frame and its successors.

Similarly, the receiver is allowed to receive sequence number 0 and all subsequent sequence numbers for any time but the first only after more than $Lmax$ time units have passed since the delivery of a frame with the maximum sequence number $K - 1$. This is necessary to ensure that all "old" frames are already removed from the frame channel and cannot be mistaken for "new" frames. To implement these restrictions, our protocol keeps two variables $tackmax$ and $tdelmax$, expressing the time when we received an acknowledgment for sequence number $K - 1$, and, delivered a frame with sequence number $K - 1$, respectively.

We were surprised to discover during the verification, that these restrictions are not quite sufficient to guarantee the unambiguous recognition of sequence numbers. It is the acknowledgment for the maximum sequence number $K - 1$ that causes the problem. In the initial version of the protocol, all acknowledgments could be resent at any time (by resending we mean here sending an acknowledgment for *the same frame* more than once, not just repetition of a sequence number). Suppose that between the receipt of an acknowledgment for sequence number $K - 1$ and the sending of sequence number 0 by the sender, the acknowledgment for sequence number $K - 1$ is resent by the receiver. Then this acknowledgment may still be in the channel at the time when sequence number 0 is sent by the sender (remember that all other acknowledgments are eliminated from the channel during the timeout period). As a result, this "old" acknowledgment may eventually be mistaken for a newly sent acknowledgment. So, the sender will think that a frame with sequence number $K - 1$ is acknowledged, whereas in fact it could have been lost.

We constructed a (lengthy) scenario in which such incorrect receipt of acknowledgments eventually leads to incorrect receipt of frames and violation of the safety property. To fix this error, in the revised version of the protocol we do not allow to acknowledge a particular frame with sequence number $K - 1$ more than once. Considering that acknowledgments can be lost, this results in a possibility of deadlock. We are not very concerned about this, since in any reasonable implementation of the SW protocol, it is only allowed to resent a message if there is a strong suspicion that the original message has been lost. In our protocol, we prefer to abstract away from such implementation details.

III. Formal specification

A. *Data structures*

First we define the data structure of the protocol. For the sender, the window "slides" over the infinite input sequence $input$. We do not specify the nature of the frames in the input sequence. Variable $first$ denotes the first frame in the sending window, $ftsend$ is the first frame that is not sent yet, and we always have $first(s0) \leq ftsend(s0) \leq first(s0) + N$. Thus, at any moment of time, frames with indices from $first$ to $ftsend - 1$ (if any) are already sent but not yet acknowledged, and frames with indices from $ftsend$ to $first + N - 1$ (if any) are in the sending window but not yet sent. Variable $tackmax$ expresses the time when we received the acknowledgment with the maximum sequence number $K-1$ for the last time. As a time domain *Time*, we take the set of non-negative real numbers.

Sender:

1) $input\ :\ sequence[Frames]$,
2) $first\ :\ nat$,
3) $ftsend\ :\ nat$,
4) $tackmax\ :\ Time$

For the receiver, $output$ is the output sequence, *buffer* is a record with two fields *snumber* and frn, that represents the receiving window with N elements (which are either frames or empty spaces, denoted by ε), $lastdel$ is the last delivered sequence number, $acklastdel$ is a boolean variable which tells whether we are allowed to send the acknowledgment for $lastdel$ to the sender, $delmax$ is a boolean variable which tells whether we already delivered the maximum sequence number $K-1$ at least once, and variable $tdelmax$ expresses the time when we delivered the frame with the maximum sequence number $K-1$ for the last time (the importance of variables *tackmax* and $tdelmax$ is explained in subsection II-A).

Receiver:

1) $output\ :\ finite\_sequence[Frames]$,
2) $buffer\ :\ \{0,\ 1,\ \ldots\ N-1\} \longrightarrow$
   $(snumber\ :\ \{0,\ 1,\ \ldots\ K-1\}$,
   $frn\ :\ Frames \cup \{\varepsilon\})$,
3) $lastdel\ :\ \{0,\ 1,\ \ldots\ K-1\}$,
4) $acklastdel\ :\ bool$,
5) $delmax\ :\ bool$,
6) $tdelmax\ :\ Time$

The frame channel and the acknowledgment channel are represented by its contents, namely a set of frame messages and a set of acknowledgment messages, respectively. Besides a sequence number and possibly a frame, each message includes its *timeout*, i.e. the latest time when it must be removed from the channel. When a message is sent, we assign as its timeout the current time plus *Lmax*, where *Lmax* is the maximum message lifetime. Although timeout is formally a part of a message, it is never used by the recipient of this message.

FrameMessage:

1) $snumber\ :\ \{0,\ 1,\ \ldots\ K-1\}$,
2) $frame\ :\ Frames$,
3) $timeout\ :\ Time$

AckMessage:

1) $snumber\ :\ \{0,\ 1,\ \ldots\ K-1\}$,
2) $timeout\ :\ Time$

The complete state of the protocol consists of the sender, the receiver and the two channels *fchannel* and *achannel*, together with the variable $time$, indicating the current time.

State:

1) $sender\ :\ Sender$,
2) $receiver\ :\ Receiver$,
3) $fchannel \subseteq FrameMessage$,
4) $achannel \subseteq AckMessage$,
5) $time\ :\ Time$

The initial state of the protocol is defined below in a rather obvious way. The only subtlety is the values of $tackmax$, $lastdel$ and $tdelmax$; they are initialized as 0, but we can easily determine from other variables that these values are initial and should not be used.

InitialState:

1) $sender$
   1) $input\ =\ arbitrary\ sequence\ of\ frames$,
   2) $first\ =\ 0$,
   3) $ftsend\ =\ 0$,
   4) $tackmax\ =\ 0$
2) $receiver$
   1) $output\ =\ empty\ sequence$,
   2) $buffer\ =\ \forall\ (i\ :\ \{0,\ 1,\ \ldots\ N-1\})\ :$
      $(snumber\ =\ i,\ frn\ =\ \varepsilon)$,
   3) $lastdel\ =\ 0$,
   4) $acklastdel\ =\ FALSE$,
   5) $delmax\ =\ FALSE$,
   6) $tdelmax\ =\ 0$
3) $fchannel\ =\ \emptyset$,
4) $achannel\ =\ \emptyset$,
5) $time\ =\ 0$

The protocol is specified by a state machine with 7 atomic actions: 1 general, 3 for the sender and 3 for the receiver, where some actions have a parameter. Below we show the precondition and the effect of each of them, using some abbreviations and PVS-like notation. The precondition is defined for an arbitrary state $s0$, which is transformed according to the effect predicate to a new state $s1$. In our specifications, the effect is given in an imperative style close to specifications of PVS, and operator *with* is used to overwrite values of records and functions. For instance, record of the form $s1 = s0\ with\ [time := time(s0) + t]$ indicates that in the new state $s1$, the variable $time$ is changed from its current value $time(s0)$ to the new value $time(s0) + t$, and all other variables are left unchanged. Operator $rem(K)$ gives a remainder to a modulus K, operator *choose* gives an arbitrary element from a nonempty set, and $if - then - else$ operator and operators on sets have a usual meaning.

Note that actions for sending and receiving messages are nondeterministic, which can be noticed from the use of $\vee$ in their effect predicates. Actions for sending messages (*send*, *resend* and *sendack*) either add a message to the channel (which models its successful sending) or let the channel unchanged (which models loss of a message). Actions for receiving messages (*receiveack* and *receive*) either remove a message from the channel (which models its "normal" reception) or let the channel unchanged (which models duplication of a message). Note that we model reordering of messages by representing both channels by unordered sets.

B. The delay action

Action $Delay(t)$ expresses the passing of t units of time. The precondition of this action, using a "time-lock" construction, ensures that any message in a channel is removed from the channel before its timeout expires.

```
Delay(t)
Precondition:
    $\forall fm:\ fm \in fchannel(s0) \Longrightarrow$
        $time(s0) + t \leq timeout(fm),$
    $\forall am:\ am \in achannel(s0) \Longrightarrow$
        $time(s0) + t \leq timeout(am)$

Effect:
    $s1 = s0\ with\ [time := time(s0) + t]$
```

C. Actions of the sender

The precondition *ReuseZeroPre* of action *Send* expresses that sequence number 0 can only be reused after more than *Lmax* time units has passed since the last acknowledgment of sequence number $K-1$ (and only if all preceding frames have already been acknowledged). The precondition of action *Resend* allows to resend any frame in the sending window that has been already sent. In the effect of action *Receiveack*, *aset* is a set of indices of frames in the sending window that are acknowledged by the acknowledgment message *am*. It is easy to see that this set consists of at most one index.

```
Send
Precondition:
LET
$ReuseZeroPre(s0) =$
    $(rem(K)(ftsend(s0)) = 0\ \&\ ftsend(s0) > 0) \Longrightarrow$
        $(ftsend(s0) = first(s0)\ \&$
        $time(s0) > tackmax(s0) + Lmax)$
IN
    $first(s0) \leq ftsend(s0) < first(s0) + N,$
    $ReuseZeroPre(s0)$

Effect:
LET
$SendNS(s0) = s0\ with$
    $[ftsend := ftsend(s0) + 1]$
IN
    $s1 = SendNS(s0)\ with$
        $[fchannel := fchannel(s0) \cup$
            $\{(rem(K)(ftsend(s0)),$
            $input(s0)(ftsend(s0)),$
            $time(s0) + Lmax)\}]$
    $\vee\ s1 = SendNS(s0)$
```

```
Resend(i)
Precondition:
    $i \geq first(s0),$
    $i < ftsend(s0)$

Effect:
    $s1 = s0\ with$
     $[fchannel := fchannel(s0) \cup$
     $\{(rem(K)(i),\ input(s0)(i),\ time(s0) + Lmax)\}]$
    $\vee\ s1 = s0$
```

```
Receiveack(am)
Precondition:
    $am \in achannel(s0)$

Effect:
LET
$aset(s0,\ am)\ =\ \{\ j\ \mid\ j \geq first(s0)\ \&$
    $j < ftsend(s0)\ \&$
    $rem(K)(j) = snumber(am)\ \}$
AND
$RANS(s0,\ i,\ bk) = s0\ with$
    $[first := i,$
    $tackmax :=$ if $bk = K - 1$ then $time(s0)$
                    else $tackmax(s0)]$
IN
   if $aset(s0, am) \neq \emptyset$ then
     $s1 = RANS(s0, choose(aset(s0, am)) + 1,$
                    $snumber(am))\ with$
                $[achannel := achannel(s0) \setminus \{am\}]$
     $\vee\ s1 = RANS(s0,\ choose(aset(s0, am)) + 1,$
            $snumber(am))$
    else $s1 = s0$ *with*
            $[achannel := achannel(s0) \setminus \{am\}]$
        $\vee\ s1 = s0$
```

D. Actions of the receiver

The precondition of action *Receive* ensures that we can only receive messages after more than *Lmax* time units has passed since the last delivery of a frame with sequence number $K-1$. In the effect of the *Receive* action, $fset$ is a set of indices in the receiving window into which the frame from message fm can be inserted. It is easy to see that this set consists of at most one index.

Receive(fm)

```
Precondition:
    $fm \in fchannel(s0)$,
    $delmax(s0) \Longrightarrow time(s0) > tdelmax(s0) + Lmax$

Effect:
LET
$fset(s0,\ fm) = \{\ j \mid j < N$ &
    $snumber(buffer(s0)(j)) = snumber(fm)$ &
    $frn(buffer(s0)(j)) = \varepsilon$ &
    $snumber(buffer(s0)(j)) \geq j\ \}$
AND
$RNS(s0,\ bn,\ fr) = s0$ *with*
    $[buffer := buffer(s0)$ *with*
        $[(bn) := (snumber(buffer(s0)(bn)),\ fr)]]$
IN
    if $fset(s0, fm) \neq \emptyset$ then
        $s1 = RNS(s0, choose(fset(s0, fm)),$
                $frame(fm))$ *with*
            $[fchannel := fchannel(s0) \setminus \{fm\}]$
        $\vee\ s1 = RNS(s0,\ choose(fset(s0, fm)),$
                $frame(fm))$
    else $s1 = s0$ *with*
            $[fchannel := fchannel(s0) \setminus \{fm\}]$
        $\vee\ s1 = s0$
```

Sendack

```
Precondition:
    $acklastdel = TRUE$

Effect:
LET
$SendackNS(s0) = s0$ *with*
    $[acklastdel :=$ if $lastdel(s0) = K-1$
        then $FALSE$ else $acklastdel(s0)]$
IN
    $s1 = SendackNS(s0)$ *with*
        $[achannel := achannel(s0) \cup$
            $\{(lastdel(s0),\ time(s0) + Lmax)\}]$
    $\vee\ s1 = SendackNS(s0)$
```

Deliver

```
Precondition:
    $frn(buffer(s0)(0)) \neq \varepsilon$

Effect:
    $s1 = s0$ *with*
        $[output := output(s0)\ o\ one(frn(buffer(s0)(0))),$
        $buffer := shift(buffer(s0),$
            $(rem(K)(snumber(buffer(s0)(N-1)) + 1),\ \varepsilon)),$
        $lastdel := snumber(buffer(s0)(0)),$
        $acklastdel := TRUE,$
        $delmax :=$ if $snumber(buffer(s0)(0)) = K-1$
                then $TRUE$ else $delmax(s0),$
        $tdelmax :=$ if $snumber(buffer(s0)(0)) = K-1$
                then $time(s0)$ else $tdelmax(s0)]$
```

Here for a frame fr, $one(fr)$ denotes the sequence of frames of length one with the only element fr; o is the operator that performs concatenation of two finite sequences of frames; and $shift$ is the operator that removes the first element of a buffer and adds another element to its end, i.e. for a buffer *buff* with N elements and a buffer element *be*, the expression $shift(buff, be)$ is defined as follows:

$shift(buff, be) = \forall\ (i\ :\ \{0,\ 1,\ \ldots\ N-1\})$:
    if $i < N-1$ then $buff(i+1)$ else be

IV. Formal verification

A. Correctness condition

An execution of our protocol, or a *run*, is represented by an infinite sequence of the form $s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \ldots \overset{a_{i-1}}{\to} s_i \overset{a_i}{\to} s_{i+1} \overset{a_{i+1}}{\to} \ldots$, 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 (where initial state, precondition and effect are defined in the previous section). As we already explained, a SW 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 the correctness property 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:

$$Safe(s) = \forall\, i:\ i < length(output(s)) \Longrightarrow \\ output(s)(i) = input(s)(i)$$

Let $st(r)$ and $act(r)$ denote the sequence of states and sequence of actions of a run r, respectively. A run r can now be defined as correct, if each state in this run satisfies the safety property.

$$Safety(r) = \forall\, i : Safe(st(r)(i))$$

In the next subsection, we outline how to prove $Safety(r)$ for each run of our protocol. All our proofs are done with the interactive theorem prover PVS, so below we show some mathematical proofs "extracted" from the PVS proofs.

B. Proof of correctness condition

We need to prove the following theorem:

$$\forall\, r,\ i:\ Safe(st(r)(i)) \qquad Main$$

The proof is based on the following important invariant *OriginOK*, in which *bn* is a variable for an integer not greater than $N-1$, *buffer* is the buffer of the receiver, and function *LO* gives the length of the output sequence in a particular state:

$$\forall\, r,\ i,\ bn:\ frn(buffer(st(r)(i))(bn)) \neq \varepsilon \Longrightarrow \\ frn(buffer(st(r)(i))(bn)) = \\ input(st(r)(i))(LO(st(r)(i)) + bn) \qquad OriginOK$$

Invariant *OriginOK* determines the "origin" of each frame in the buffer of the receiver: a frame in the position *bk* was sent by the sender from the position in the input sequence, equal to the sum of *bk* and the current length of the output sequence. Assuming *OriginOK*, it is easy to prove theorem *Main*.

Proof of *Main*. First, we prove that all actions of our protocol don't change the input sequence. Now let r be an arbitrary run. The proof is by induction on the length of the output. If it is equal to 0, the statement is trivially true. Now suppose that the theorem is proved for any output length not greater than k, and that we are in the state with index i such that $LO(st(r)(i)) = k+1$. It is easy to see that action *Deliver* increases the length of the output exactly by one, and all other actions of our protocol don't change the output. Therefore, there exists index l such that $l < i$, $LO(st(r)(l)) = k$, $act(r)(l) = Deliver$ and $output(st(r)(l+1)) = output(st(r)(i))$. By the induction hypothesis, it follows that in state $st(r)(l)$, the input is the prefix of the output. We can now apply invariant *OriginOK* for r, l and 0, and obtain that the frame delivered by action $act(r)(l)$ originates from position $LO(st(r)(l))$ in the input. Thus in state $st(r)(l)$, output includes frames $input(0)$, $input(1)$, ... $input(LO(st(r)(l)) - 1)$, and frame $input(LO(st(r)(l)))$ is added to it by action $act(r)(l)$. Therefore, in states $st(r)(l+1)$ and $st(r)(i)$ the output is still the prefix of the input, and this completes the proof.

To prove invariant *OriginOK*, the following important invariants *AckOK* and *FrOK* are needed:

$$\forall\, r,\ i:\ first(st(r)(i)) \leq LO(st(r)(i)) \qquad AckOK$$

Intuitively, invariant *AckOK* means "we cannot receive acknowledgments for frames that are not acknowledged yet". Indeed, the value of *first* is equal to the number of frames for which acknowledgments from the receiver have been received. But the receiver acknowledges only those frames which it had already delivered, and all such frames are included in the output. Therefore, *first* can become greater than the length of the output only if the sender receives some acknowledgments more than once.

$$\forall\, r,\ i,\ bn:\ frn(buffer(st(r)(i))(bn)) \neq \varepsilon \Longrightarrow \\ LO(st(r)(i)) + bn < ftsend(st(r)(i)) \qquad FrOK$$

Intuitively, invariant *FrOK* means "we cannot receive frames that are not sent yet". Indeed, if the buffer of the sender has a frame in position *bn*, it implies that at least $LO(st(r)(i)) + bn + 1$ frames have been sent by the sender, but the exact number of such frames is represented by variable *ftsend*. Therefore, the invariant can only be violated if the receiver receives some frames more than once.

Together, invariants *AckOK* and *FrOK* mean that the length of the output is always very close to the borders of the sending window. Despite the clear intuitive meaning of these invariants, it turned out very difficult to prove them in PVS, and we are still working on their proofs. In this paper, we assume these invariants to be true, and show how to use them to prove invariant *OriginOK*. Below we give a brief sketch of the proof, which is based on dozens of PVS lemmas.

Proof of *OriginOK*. Let's consider arbitrary r, i and *bn*, and suppose there is a frame in the buffer of the sender in position *bn*. It is easy to prove that as long as a frame stays in the buffer, the sum of its position and the length of the output remains the same. Therefore, we can assume without loss of generality that this frame has just been put into the buffer, i.e. action $act(r)(i-1)$ is a receive action that receives message with frame $frn(buffer(st(r)(i))(bn))$ from the channel. It is also easy to prove that a frame in position bn has a sequence number $rem(K)(LO(st(r)(i))+bn)$. Thus in state $st(r)(i-1)$, the frame channel includes a message with frame $frn(buffer(st(r)(i))(bn))$ and sequence number $rem(K)(LO(st(r)(i))+bn)$. We can prove that each message in the frame channel was sent by the sender at some moment in the past. This implies that the message originates from some frame with position j in the input sequence, and from the way in which messages are constructed we obtain $input(st(r)(i))(j) = frn(buffer(st(r)(i))(bn))$ and $rem(K)(j) = rem(K)(LO(st(r)(i))+bn)$. To finish the proof, it is now sufficient to show $j = LO(st(r)(i))+bn$.

It is easy to see that $j < ftsend(st(r)(i-1))$. We can also prove $ftsend(st(r)(i-1)) - j \leq K$. Indeed, it is obvious that in state $st(r)(i-1)$, all frames in positions from $j+1$ to $ftsend(st(r)(i-1))-1$ have already been sent. If $ftsend(st(r)(i-1)) - j > K$, then there are at least K such positions, so at least one of them has a remainder 0 with respect to K. Thus after sending the frame in position j, we sent a frame with sequence number 0 at least once. But our protocol waits for $Lmax$ time units before resending sequence number 0, and this ensures that by the time of this resending all preceding messages disappear from the channel. Contradiction, because in state $st(r)(i-1)$ we received a message originating from position j in the input.

Now we use invariants $AckOK$ and $FrOK$. Invariant $FrOK$ implies $LO(st(r)(i)) + bn < ftsend(st(r)(i))$. Invariant $AckOK$ gives $first(st(r)(i)) \leq LO(st(r)(i))$. We know that $ftsend(st(r)(i)) - first(st(r)(i)) \leq N$. This implies $ftsend(st(r)(i)) - (LO(st(r)(i)) + bn) \leq N$. Action $act(r)(i-1)$ is not a send action, so we have $LO(st(r)(i)) + bn < ftsend(st(r)(i-1))$ and $ftsend(st(r)(i-1)) - (LO(st(r)(i)) + bn) \leq N$. Comparing this with our results about j, we obtain that both j and $LO(st(r)(i)) + bn$ are less than $ftsend(st(r)(i-1))$, and the difference between each of them and $ftsend(st(r)(i-1))$ is not greater than K. Therefore the difference between j and $LO(st(r)(i)) + bn$ is less than K. But we already know that these two numbers have the same remainder with respect to K. Thus they are equal, and this completes the proof.

V. Conclusions

We presented the formal specification and verification of the Sliding Window protocol for transport channels. As explained in the introduction, our version of the protocol offers a significant improvement over some previously published versions, and can potentially be used as a part of the TCP protocol. Unlike some previous papers, our modelling of the protocol is very general, and the verification is supported by the interactive theorem prover PVS. Our work on the verification helped us to obtain an increased insight into the protocol, and to discover and to eliminate a surprising erroneous scenario in an earlier version of the protocol (see subsection II-A), which otherwise could have been left unnoticed.

In the immediate future, we plan to finish the PVS proof outlined in section IV. As a part of our future work, we also would like to specify and verify a simplified version of our protocol for data link channels, and to compare both specifications and proofs with the more general protocol for transport channels. Finally, we are also interested in the study of liveness properties for Sliding Window protocols and their formal verification.

References

[1] *PVS Specification and Verification System, http://pvs.csl.sri.com/.*

[2] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D. Wang, and L. Zuck. Reliable communication over unreliable channels. *Journal of the ACM*, 41(6):1267–1297, 1994.

[3] J.C.M. Baeten and W.P. Weijland. *Process algebra*. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[4] K.A. Barlett, R.A. Scantlebury, and P.C. Wilkinson. A note on reliable transmission over half duplex links. *Communications of the ACM*, 12:260–261, 1969.

[5] M.A. Bezem and J.F. Groote. A correctness proof of a one-bit sliding window protocol in *μCRL*. *The Computer Journal*, 37(4):1–19, 1994.

[6] D. Chkliaev. *Mechanical Verification of Concurrency Control and Recovery Protocols*. PhD thesis, Technische Universiteit Eindhoven, 2001.

[7] P. Godefroid and D. E. Long. Symbolic protocol verification with queue BDDs. *Formal Methods in System Design: An International Journal*, 14(3):257–271, May 1999.

[8] J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. A case study in computer-checked verification. In *AMAST'96*, pages 536–550. LNCS 1101, 1996.

[9] K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 662–681. Springer-Verlag, 1996.

[10] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In *International Workshop TYPES'93*, pages 127–165. LNCS 806, 1994.

[11] C.A.R. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM*, volume 12(10), pages 576–583, 1969.

[12] R. Kaivola. Using compositional preorders in the verification of sliding window protocal. In *Computer Aided Verification*, pages 48–59, 1997.

[13] D.E. Knuth. Verification of link-level protocols. *BIT*, 21:31–36, 1981.

[14] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[15] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: specification.* Springer Berlin, 1991.

[16] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in Xesar of the sliding window protocol. In *Protocol specification, testing and verification 7*, pages 235–248, 1987.

[17] A. Udaya Shankar. Verified data transfer protocols with variable flow control. *ACM Transactions on Computer Systems*, 7(3):281–316, 1989.

[18] M. Smith and N. Klarlund. Verification of a sliding window protocol using IOA and MONA. Formal methods for distributed system development, pages 19–34. Kluwer Academic Publishers, 2000.

[19] K. Stahl, K. Baukus, Y. Lakhnech, and M. Steffen. Divide, abstract, and model-check. In *The 5th International SPIN Workshop on Theoretical Aspects of Model Checking*, pages 57–76. LNCS 1680, 1999.

[20] N.V. Stenning. A data transfer protocol. *Computer Networks*, 1(2):99–110, 1976.

[21] C. Sunshine and Y. Dalal. Connection management in transport protocols. *Computer Networks*, 2:454–473, 1978.

[22] A.S. Tanenbaum. *Computer Networks*. Prentice-Hall International, Inc., 1996.

[23] D. Wang and L. Zuck. Tight bounds for the sequence transmission problem. The 8th ACM Symposium on Principles of Distributed Computing, pages 73–83. ACM, 1989.