Verification and Improvement of the Sliding Window Protocol*

Dmitri Chkliaev[1], Jozef Hooman[2], and Erik de Vink[1]

[1] Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{dmitri, evink}@win.tue.nl

[2] Computing Science Institute, University of Nijmegen, The Netherlands
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 channels that may lose, reorder and duplicate messages.

1 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 [Ste76]. 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 Dutch PROGRESS project EES5202, "Modelling and performance analysis of telecommunication systems".

However, formal verification of communication protocols is notoriously difficult. Even verification of a version of the Alternating Bit protocol [BSW69] (which is one of the simplest communication protocols), namely the Bounded Retransmission Protocol (BRP) of Philips Electronics, turned out to be non-trivial. 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). In [DKRT97], only restricted versions of the protocol could be model-checked; the manually derived constraints have been checked by parametric model-checking in [HRSV01], revealing a small error. More general correctness proofs by theorem provers also encounter many technical difficulties [GP96,HSV94,HS96].

Despite the practical significance of the Sliding Window protocol, the work on its formal verification had only a limited success so far. Stenning [Ste76] only gave an informal manual proof for his protocol. A semi-formal manual proof is also presented in [Knu81]. A more formal, but not fully automated proof for the window size of one is given in [BG94], and for the arbitrary window size in [FGP03]. Some versions of the protocol have been model-checked for small parameter values in [RRSV87,Kai97]. The combination of abstraction techniques and model-checking in [SBLS99] 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). Almost all of these verifications 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 [Ste76]. The protocols for such channels (called *transport protocols*), such as TCP, can transmit data over very large networks such as the Internet.

Note that an SW protocol does not exists for all types of transport channels. As [AAF$^+$94] 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. A similar result is proved for systems that can both reorder and duplicate messages [WZ89]. In [SK00], 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 [Tan96].

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[3]. Such a SW protocol is a part of the TCP protocol, which operates over

[3] Such protocols can also be designed for untimed systems which limit the reordering of messages [Knu81], but such systems seem to be only of theoretical interest.

transport channels with a given maximum packet lifetime. The theoretical basis of that protocol is presented in [SD78]. 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 [Sha89]. 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 [Tan96]). 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.

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 receiving acknowledgment for (delivering) the maximum sequence number, respectively. Unlike some previous works [Ste76,Sha89], 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[4], 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, manual formal verification is so lengthy and complicated that it can easily be erroneous. This is why we need some form of mechanical support. Our protocol highly depends on com-

[4] 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.

plex data structures and 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 [PVS], because we have an extensive experience with it and successfully applied the tool to verification of several complicated protocols [Chk01]. 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 2, we give an informal description of our protocol. In section 3, we formalize the protocol by a timed state machine. Section 4 outlines the proof of correctness property for our protocol. Some concluding remarks are given in section 5.

2 Protocol overview

In section 2.1 we present the basics of a sliding window protocol. The required relation between sequence numbers and window size is described in section 2.2. Timing restrictions are discussed in section 2.3.

2.1 Basic notions

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 a frame via the channel, the receiver may decide to *accept* the frame and eventually *deliver* it 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 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 "sequence number". In the 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, in the acknowledgment message the receiver sends the sequence number with which the frame was received. 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 accepted.

Sending window. 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 accept. 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 have been already 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 been 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. 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 for sending and resending of frames exist [Tan96], 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 behaviour that are needed to ensure its safety property.

Receiving window. During the execution, the receiving window is usually a mix of sequence numbers corresponding to frames that have been accepted 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 accepted, i.e. 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 accepted frames). 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.

2.2 Relating sequence numbers and window size

It is explained in [Tan96], 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 accepted, delivered and acknowledged. Suppose, however, that $f0$ has been duplicated in the frame channel and subsequently reordered with $f1$, so the channel still contains frame $f0$ with sequence number 0. The receiver now has a window with an empty space and sequence number 0, so it can receive frame $f0$ for the second time, violating the safety property. In this case, the error is caused by the combination of message reordering and duplication; a similar erroneous scenario can be constructed using reordering and loss.

This simple example clearly shows that we need additional restrictions on the protocol to recognize sequence numbers correctly. Traditional approaches [Ste76,Sha89] 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, 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 some timing restrictions on the transmission and acknowledgment policies (explained below) to ensure that frames and acknowledgments are not received more than once.

2.3 Timing restrictions

In our protocol, the sender is allowed to reuse sequence number 0 and all subsequent sequence numbers only after more than $Lmax$ time units have passed since the receipt of the acknowledgment with 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 accept sequence number 0 (or any subsequent sequence numbers) 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 have been 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. It is the acknowledgment for the maximum sequence number $K-1$ that causes the problem. In the initial version of the protocol, acknowledgments for a particular frame could be resent at any time. 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 $K-1$ is sent by the sender again (assuming the fast and correct transmission of all $K-1$ frames within $Lmax$ time units). As a result, this "old" acknowledgment may 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 the acknowledgment with sequence number $K-1$ must be sent immediately after the corresponding frame is delivered, and it cannot be resent. Considering that acknowledgments can be lost, this results in a possibility of deadlock. We are not very concerned about this, since any reasonable implementation of the SW protocol also does not allow to resend acknowledgments at any time (only 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. However, we also constructed a modification of our protocol (presented below) that does not suffer from this deadlock problem.

Possible improvement. The simplest way to prevent the acknowledgment with sequence number $K-1$ from being accepted again is to introduce the additional waiting period for the sender. Before the sender resends sequence number $K-1$, it should wait for more than $Lmax$ time units after accepting the acknowledgment with sequence number $K-2$. This ensures the elimination of all "old" acknowledgments with sequence number $K-1$. As a result, it becomes possible to acknowledge a particular frame with sequence number $K-1$ more than once, and this resolves the deadlock problem. However, it is obvious that this additional waiting period greatly reduces the performance of the protocol.

We specified this modification of our protocol in PVS, but we did not verify it for two reasons. Firstly, it is not clear whether the degradation in performance is justified. In a complex distributed system, it is not reasonable to avoid deadlock at any cost. It may be more efficient to allow a deadlock in some rare situations, and to use an additional protocol to resolve it. Secondly, this additional waiting period would make the verification of the protocol even more complex without adding much theoretical value to it.

3 Formal specification

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. For the SW protocol, we have actions of sender and receiver, and a delay action. Our timing model can be considered a simplified version of timed automata of Alur and Dill [AD94], in which there is only one clock (called *time*) that is never reset. Actions are specified by a precondition and an effect predicate which relates the states before and after action execution. An execution of our protocol, or a *run*, is represented by an infinite sequence of the form $s_0 \stackrel{a_0}{\to} s_1 \stackrel{a_1}{\to} \ldots \stackrel{a_{i-1}}{\to} s_i \stackrel{a_i}{\to} s_{i+1} \stackrel{a_{i+1}}{\to} \ldots$, 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. In this section, we describe the structure of states and actions in our specification.

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 has not been sent yet, and we always have $\mathit{first} \leq \mathit{ftsend} \leq \mathit{first} + N$. 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. 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.

<u>Sender</u>:
1) *input* : $sequence[Frames]$,
2) *first* : *nat*,
3) *ftsend* : *nat*,
4) *tackmax* : $Time$

For the receiver, *output* is the finite output sequence, *rwindow* is 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, 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 *acklastdel* and *tdelmax* is explained in subsection 2.3).

<u>Receiver</u>:
1) *output* : $finite\_sequence[Frames]$,
2) *rwindow* : $\{0,\ 1,\ \ldots\ N-1\} \longrightarrow$
   $(\mathit{snumber}\ :\ \{0,\ 1,\ \ldots\ K-1\},\ \mathit{frn}\ :\ Frames \cup \{\varepsilon\})$,
3) *lastdel* : $\{0,\ 1,\ \ldots\ K-1\}$,
4) *acklastdel* : *bool*,
5) *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, in our model 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. Note that timeout is only used to model maximum message lifetime, it cannot be used by the recipient of a 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. The initial state of the protocol is defined in a rather obvious way, i.e. 0 is assigned to most fields.

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

There are seven atomic actions in our protocol: one general ($Delay$), three for the sender ($Send$, $Resend$ and $Receiveack$) and three for the receiver ($Receive$, $Sendack$ and $Deliver$). Due to space limitations, we only give an informal description of these actions. The full specification of the protocol can be found in the PVS files, available via [URL]. In the rest of the paper, operator *mod* K gives a remainder to a modulus K.

Note that actions for sending and receiving messages are nondeterministic. 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). The reordering of messages is modelled by representing both channels by unordered sets.

Also note that the sender and the receiver are "input enabled" (e.g., as in I/O automata of Lynch [Lyn96]); they are always willing to receive a message from the channel. This is necessary to ensure that all messages can be received before their timeouts expire. If certain conditions are met, the received message is accepted, changing the state of the sender (receiver), otherwise it is discarded.

3.1 The delay action

Action $Delay(t)$ expresses the passing of t units of time. The precondition of this action expresses that time cannot be advanced above the minimal time-out value in channels. Hence, any message in a channel must be removed from the channel before its timeout expires. The effect predicate expresses that the new state equals the old state except for the value of time which is incremented by t.

3.2 Actions of the sender

- **Send** This action sends the first frame that has not been sent yet, i.e. the frame with index $ftsend$, and $ftsend \underline{mod} K$ is included into the message as its sequence number. The precondition of this action expresses that sequence number 0 can be reused only after more than $Lmax$ time units have passed since the last acknowledgment of sequence number $K - 1$, and only if 0 is the first sequence number in the window, i.e. if $ftsend \underline{mod} K = 0$, then we require 1) $time > tackmax + Lmax$ and 2) $first = ftsend$.

- **Resend(i)** This action resends a frame that has already been sent but not yet acknowledged, i.e. a frame with index i such as $i \geq first$ and $i < ftsend$.

- **Receiveack(am)** This action receives acknowledgment message am and checks whether $snumber(am)$ lies within the sending window. If so, the frames with sequence numbers up to $snumber(am)$ are removed from the window and the window is shifted accordingly, and if $snumber(am) = K-1$, then the current time is assigned to the variable $tackmax$. Otherwise, the message is discarded.

3.3 Actions of the receiver

- **Receive(fm)** This action receives frame message fm and checks whether $snumber(fm)$ corresponds to some empty space in the receiving window. If so, it accepts the message if more than $Lmax$ time units have passed since the last delivery of a frame with sequence number $K - 1$. The exact formalization of this requirement is subtle because frames can be accepted and delivered in different order; it is defined by giving a restriction on the position of $snumber(fm)$ in the window in addition to the timing restriction $time > tdelmax + Lmax$ (see [URL] for details). If these conditions for acceptance are satisfied, then the frame from the message is inserted into the corresponding place in the window; otherwise the message is discarded.

- **Sendack** This action sends an acknowledgment for the last delivered frame, i.e. the frame with sequence number $lastdel$. If $lastdel = K - 1$, it changes the value of variable $acklastdel$ to false. The precondition of Sendack requires $acklastdel$ to be true, and this prevents the acknowledgment for $K - 1$ from being resent.

- **Deliver** The precondition of this action requires that the first element of the receiving window is a frame. If it is the case, the frame is appended to the output sequence and removed from the window, i.e. the window is shifted by one. Also, the sequence number of the frame is assigned to $lastdel$, if this sequence number is $K-1$, then the current time is assigned to $tdelmax$, and $acklastdel$ becomes equal to true.

In addition to the preconditions of individial actions, we also specify an additional assertion expressing that each delivery of the sequence number $K-1$ is immediately followed by action $Sendack$ acknowledging this sequence number.

4 Formal verification

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 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:

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

Let $st(r)$ and $act(r)$ denote the sequence of states and sequence of actions of a run r, respectively. Run r is safe if it is safe in each state, i.e. we define

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

To verify our SW protocol, we proved $Safety(r)$ for each run r, using the interactive theorem prover of PVS. The proof was performed by the first author in about 4 months (from scratch, without hand-written proofs). It consists of about 150 PVS lemmas and theorems and some 10 thousand PVS commands, see [URL]. Our experience with PVS has been very positive, e.g., there are more than sufficient lemmas about modulo arithmetic in the PVS prelude file.

Our verification efforts are comparable with the efforts to verify the BRP protocol in [GP96,HSV94]. However, our proof seems to be much more involved, and includes some lemmas that are far from trivial. In general, this is not surprising, considering that our protocol tolerates more types of channel faults, and it has several parameters, a relatively complex data structure and timing aspects. It is reported in [HSV94] that they could prove most invariants by simple induction on the length of the execution. For our proof, this is certainly not the case. The proof of many lemmas about the current state required a rather subtle analysis of all states and actions preceeding this state in a run. Another problem we faced is that correctness of one difficult invariant for the receiver depended on a similar invariant for the sender, and vice versa. Significant efforts and some advanced techniques were needed to break this cycle in the proof. In the next subsection, we briefly outline the proof of the main correctness condition.

4.1 Proof of correctness condition

We need to prove the following theorem:

$$\forall\, r:\ Safety(r) \qquad\qquad Main$$

The following abbreviations are used for a run r: bn is a variable for an integer not greater than $N-1$, $rwindow_r(i, bn) = rwindow(st(r)(i))(bn)$ (i.e. $rwindow_r(i, bn)$ is the bn-th element of the receiving window in state i), $first_r(i) = first(st(r)(i))$, $ftsend_r(i) = ftsend(st(r)(i))$, and $LO_r(i)$ is the length of the output sequence in state i. It is easy to prove that all actions of our protocol don't change the input sequence, so we denote by $input_r$ the input sequence in each state of r.

The proof of $Main$ is based on the following important invariant $OriginOK$:

$$\forall\, r,\ i,\ bn:\ frn(rwindow_r(i, bn)) \neq \varepsilon \Longrightarrow$$
$$frn(rwindow_r(i, bn)) = input_r(LO_r(i) + bn) \qquad\qquad OriginOK$$

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

Proof of $Main$. Let r be an arbitrary run. The proof is by induction on the length of the output. If it is 0, the statement is trivially true. Now suppose that the theorem has been proved for any output length not greater than k, and that we are in the state with index i such that $LO_r(i) = k + 1$. 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 $l < i$, $LO_r(l) = k$, $act(r)(l) = Deliver$ 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 $LO_r(l)$ in the input. Thus in state $st(r)(l)$, output includes frames $input_r(0)$, $input_r(1)$, ... $input_r(LO_r(l) - 1)$, and frame $input_r(LO_r(l))$ 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, which completes the proof.

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

$$\forall\, r,\ i:\ first_r(i) \leq LO_r(i) \qquad\qquad AckOK$$

Intuitively, invariant $AckOK$ implies that acknowledgments are accepted only once. Indeed, the value of $first$ is equal to the number of frames for which acknowledgments from the receiver have been accepted. But the receiver acknowledges only delivered frames which are included in the output. Therefore, $first$ can become greater than the length of the output only if the sender accepts some acknowledgments more than once.

$$\forall\ r,\ i,\ bn:\ frn(rwindow_r(i,bn)) \neq \varepsilon \Longrightarrow LO_r(i) + bn < ftsend_r(i) \qquad FrOK$$

Invariant $FrOK$ informally means that frames are accepted only once. Indeed, if the receiving window has a frame in position bn, it implies that at least $LO_r(i) + bn + 1$ 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 accepts some frames more than once.

Together, invariants $AckOK$ and $FrOK$ mean that the length of the output is always within the borders of the sending window. Despite the clear intuitive meaning of these invariants, their proofs are fairly large and complicated, and they use some advanced techniques, such as counting of the number of certain actions preceeding the current state. Due to space limitations, we cannot present these proofs here. In this paper, we only show how to use invariants $AckOK$ and $FrOK$ 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 receiving window in position bn. It is easy to prove that as long as a frame stays in the window, 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 window, i.e. action $act(r)(i-1)$ is a receive action that accepts message with frame $frn(rwindow_r(i,bn))$ from the channel. It is also easy to prove that a frame in position bn has a sequence number $LO_r(i) + bn\ \underline{mod}\ K$. Thus in state $st(r)(i-1)$, the frame channel includes a message with frame $frn(rwindow_r(i,bn))$ and sequence number $LO_r(i) + bn\ \underline{mod}\ K$. 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, i.e. $input_r(j) = frn(rwindow_r(i,bn))$, and from the way in which messages are constructed we obtain $j\ \underline{mod}\ K = LO_r(i) + bn\ \underline{mod}\ K$. To finish the proof, it is now sufficient to show $j = LO_r(i) + bn$.

It is easy to see that $j < ftsend_r(i-1)$. We can also prove $ftsend_r(i-1) - j \leq K$. Indeed, it is obvious that in state $st(r)(i-1)$, all frames in positions from $j+1$ to $ftsend_r(i-1) - 1$ have already been sent. If $ftsend_r(i-1) - j > K$, 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 $AckOK$ gives $first_r(i) \leq LO_r(i)$, hence $first_r(i) \leq LO_r(i)+bn$. We know that $ftsend_r(i)-first_r(i) \leq N$, so $ftsend_r(i) - (LO_r(i) + bn) \leq N$. Invariant $FrOK$ implies $LO_r(i) + bn < ftsend_r(i)$. Action $act(r)(i-1)$ is not a send action, so we have $LO_r(i) + bn < ftsend_r(i-1)$ and $ftsend_r(i-1)-(LO_r(i)+bn) \leq N$. Comparing this with our results about j, we obtain that both j and $LO_r(i)+bn$ are less than $ftsend_r(i-1)$, and the difference between each of them and $ftsend_r(i-1)$ is not greater than K. Therefore the difference between j and $LO_r(i) + bn$ 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.

5 Conclusions

We presented the formal specification and verification of the Sliding Window protocol for transport channels. Our version of the protocol offers an interesting improvement over some previously published versions (as it tolerates arbitrary transmission rates), and can potentially be used as a part of the TCP protocol. Unlike most previous papers, our modelling of the protocol is very general, and the verification is supported by the interactive theorem prover PVS.

An interesting lesson we learned from this project is that the traditional (untimed) Sliding Window protocol cannot correctly handle the combination of message reordering and duplication, or reordering and loss, as explained in section 2.2. The timed version of the protocol presented in this paper eliminates the potential errors arising from such a combination, but at a cost of a significant performance loss. It should be noted, however, that some performance loss seems to be unavoidable for any Sliding Window protocol operating over transport channels. E.g., the complicated timing mechanism of TCP, as mentioned in the introduction, also requires waiting if there is a danger of overlap of sequence numbers. For a good comparison of performances, in our future work we would like to analyse the performance of our protocol for different retransmission policies, possibly using simulations, and to study its compatibility with TCP. It would be also interesting to apply our verification techniques (which have already been used for several concurrency control protocols in [Chk01]) to other types of communication protocols.

References

[AAF+94] 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*, 41:1267–1297, 1994.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[BG94] M.A. Bezem and J.F. Groote. A correctness proof of a one-bit sliding window protocol in μCRL. *The Computer Journal*, 37:1–19, 1994.

[BSW69] 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.

[Chk01] D. Chkliaev. *Mechanical Verification of Concurrency Control and Recovery Protocols.* PhD thesis, Technische Universiteit Eindhoven, 2001.

[DKRT97] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *TACAS'97*, pages 416 – 431. LNCS 1217, 1997.

[FGP03] W. Fokkink, J.F. Groote, and J. Pang. Verification of a sliding window protocol in μCRL. *Unfinished article*, 2003.

[GP96] 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.

[HRSV01] T. Hune, J.M.T. Romijn, M.I.A. Stoelinga, and F.W. Vaandrager. Linear parametric model checking of timed automata. In *TACAS'01*, pages 189–203. LNCS 2031, 2001.

[HS96] 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. LNCS 1051, 1996.

[HSV94] 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.

[Kai97] R. Kaivola. Using compositional preorders in the verification of sliding window protocol. In *Computer Aided Verification*, pages 48–59. LNCS 1254, 1997.

[Knu81] D.E. Knuth. Verification of link-level protocols. *BIT*, 21:31–36, 1981.

[Lyn96] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

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

[RRSV87] 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.

[SBLS99] 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.

[SD78] C. Sunshine and Y. Dalal. Connection management in transport protocols. *Computer Networks*, 2:454–473, 1978.

[Sha89] A. Udaya Shankar. Verified data transfer protocols with variable flow control. *ACM Transactions on Computer Systems*, 7:281–316, 1989.

[SK00] M. Smith and N. Klarlund. Verification of a sliding window protocol using IOA and MONA. In *Formal methods for distributed system development*, pages 19–34. Kluwer Academic Publishers, 2000.

[Ste76] N.V. Stenning. A data transfer protocol. *Computer Networks*, 1:99–110, 1976.

[Tan96] A.S. Tanenbaum. *Computer Networks.* Third Edition, Prentice-Hall International, 1996.

[URL] *PVS specifications and proofs, http://www.cs.kun.nl/~hooman/SWP.html.*

[WZ89] D. Wang and L. Zuck. Tight bounds for the sequence transmission problem. In *The 8th ACM Symposium on Principles of Distributed Computing*, pages 73–83. ACM, 1989.