Formal Specification and Verification of TCP Extended with the Window Scale Option

L. Lockefeer, D.M. Williams and W.J. Fokkink

VU University Amsterdam
info@larslockefeer.nl
{d.m.williams,w.j.fokkink}@vu.nl

Abstract. We formally verify that TCP satisfies its requirements when extended with the Window Scale Option. With the aid of our μCRL specification and the `ltsmin` toolset, we verified that our specification of unidirectional TCP extended with the Window Scale Option does not deadlock, and that its external behaviour is branching bisimilar to a FIFO queue for a significantly large instance. Finally, we recommend a rewording of the specification regarding how a zero window is probed, ensuring deadlocks do not arise as a result of misinterpretation.

1 Introduction

The Transmission Control Protocol (TCP) plays an important role in the internet, providing reliable transport of messages through a possibly faulty medium to many of its applications. The original protocol (RFC 793 [17]), specified in natural language, required improvement to clarify various ambiguities and identify and address several issues resulting in the publication of a supplemental specification (RFC 1122 [8]), which also refers to numerous other documents.

Our primary contribution is the formal verification of TCP extended with the Window Scale Option, addressing the lack of consideration paid to this option in earlier verification efforts. We take care to extract our formal specification directly from the original specifications of TCP and the Window Scale Option, i.e., RFCs 793, 1122 and 1323. This work was initially triggered by a concern of Dr. Barry M. Cook, CTO at 4Links Limited, regarding the Window Scale Option proposed in RFC 1323 [14]. Specifically, he was worried that the window size being reportable only in units of 2^n bytes may conflict with a requirement that the receive buffer space available should not change downward.

We adopt the process algebra μCRL as our formal specification language. Based on ACP, μCRL is enriched with the algebraic specification of abstract data types. We found μCRL's treatment of data as a first class citizen essential for specifying TCP, and were encouraged by its previous success in verifying the Sliding Window Protocol [1, 2]. We utilise the μCRL toolset and `ltsmin` [7] to explicitly generate the statespace and perform the automated verification.

Section 2 relates our verification effort to those that precede it. Section 3 aims to bridge the gap between the RFCs and our μCRL specification. The structure

Authors	RFC 793	RFC 1122	RFC 1323	Other extensions	Conn management	Data transfer	Message Loss	Duplication	Reordering	Message direction	Conn incarnations	Window Scale
Murphy & Shankar [16]	✓				✓		✓	✓	✓	⇔	n	
Smith [21, 22], Smith & Ramakrishnan [20]	✓			✓	✓	✓	✓	✓	✓	⇒	n	
Schieferdecker [19]	✓	✓		✓	✓				✓	⇔	2	
Billington & Han [4, 11–13]	✓	✓			✓	✓	✓		✓	⇔	1	
Bishop et al. [5], Ridge et al. [18]	✓	✓	✓	✓	✓	✓	✓	✓	✓	⇔	n	
Our verification	✓	✓	✓			✓	✓	✓	✓	⇒	1	✓

Table 1: Comparison of earlier verifications of TCP

of Section 3 mirrors the structure of the μCRL process, which is illustrated more prominently in Figures 1 and 2. We describe our verification approach in Section 4 and conclude that the Window Scale Option does not adversely impact TCP. However, using μCRL we identified deadlocks that may arise due to the ambiguous formulation of how to probe zero windows in RFC 793. Finally, we give a recommendation how this RFC could be reformulated to avoid such misinterpretations.

2 Related work

Our formal verification of TCP shall address the lack of consideration paid to the Window Scale option in earlier verification efforts. Several publications aim to formally specify and verify the correctness of TCP; Table 1 shows an overview. Murphy & Shankar [16] specified a protocol with a similar service specification to TCP as defined in RFC 793. By a method of step-wise refinement, a protocol specification is defined while maintaining several correctness properties. The need for a three-way handshake and strictly increasing incarnation numbers becomes apparent with the introduction of each fault in the medium, explicitly showing why these facilities are present in TCP. Similarly, by means of a refinement mapping, Smith [21, 22] has shown that the protocol specification satisfies the specification of the user-visible behaviour. Selective acknowledgements were added in [20]. Schieferdecker [19] shows that there is an error in TCP's handling of the `ABORT` call. After stating a possible solution, a `LOTOS` specification of TCP is given and several μ-calculus properties verified using `CADP`. Bishop et al. [5] considered whether execution traces generated from real-world implementations of TCP, were accepted by a protocol specification of TCP in Higher Order Logic (HOL), which includes PAWS, the window scale option and congestion control algorithms. Of the test traces generated, the specification accepted 91.7%.

Billington & Han have studied TCP extensively considering both RFC 793 and RFC 1122 using Coloured Petri Nets. A concise overview of their TCP service specification is given in [4] and includes connection establishment, normal

data transfer, urgent data transfer, graceful connection release and abortion of connections. In [11], they give a model of the connection management service, which is further refined in [12]. This revised specification is used as a basis for a verification of connection management [13] considering a model without retransmissions and a model with retransmissions. As a result of their verification efforts, Billington & Han find several issues within connection management. For example, a deadlock may occur when one entity opens the connection passively and, after receiving and acknowledging a connection request, immediately closes the connection again. However, the work by Billington & Han on data transfer has not yet led to a verification.

As the Sliding Window Protocol (SWP) underlies TCP (see Section 3) we also compare our verification to those of SWP. We, like Bezem & Groote [3] and Badban *et al.* [1], use μCRL for our specification. Bezem & Groote and Badban *et al.* consider bidirectional communication across a medium that can lose but not duplicate nor reorder messages. Our verification considers a medium that can lose, duplicate and reorder messages, but does so only in a unidirectional setting. Whereas we, like Bezem & Groote, consider a finite window size (namely 2^2), Badban *et al.* performed the verification on an arbitrarily large window. Madelaine & Vergamini [15] modelled and verified SWP using `LOTOS` and `AUTO`. They, like us, consider the unidirectional case across a medium that can lose, duplicate and reorder messages. Finally, Chkliaev et al. [9] specify an amended version of SWP, in which the sender and receiver need not synchronise on the sequence number initially.

3 Specification

In this section we present the overall structure of our μCRL specification of TCP, which includes the core functionality specified in RFCs 793, 1122 and 1323, extended to include the window scale option. It is the aim of this section to relate the RFCs to the μCRL specification[1], assisting the reader in bridging the gap between the two. Although our μCRL specification incorporates Connection Teardown we omit discussion of this procedure for brevity. For an overview of the most prominent formal verification techniques for communication protocols using μCRL, the reader is referred to [10].

TCP receives data from some application and packages this into segments to be handed to the network layer. The TCP instance of the receiver receives segments from the network layer and should ensure the data is delivered to the receiver's application in the same order as it was sent. The purpose of a segment is twofold: (i) a segment may contain zero or more octets of data that the sender's application wishes to relay to an application at the receiver; and, (ii) a segment communicates control information between the two entities.

[1] The complete μCRL specification can be found in the Master's thesis entitled *Formal Specification and Verification of TCP Extended with the Window Scale Option* by L. Lockefeer, VU University Amsterdam, 2013, available at `http://www.cs.vu.nl/~wanf/theses/lockefeer.pdf`

TCP begins by establishing a connection with both entities reaching agreement on the configuration to use for the connection that is stored in their Transmission Control Block (TCB). This data includes the initial sequence number that the entity will use for outgoing data (`ISS`), the size of the send window for the entity (`SND.WND`), indicating the maximum number of octets that it may send at once, and the size of the receive window for the entity (`RCV.WND`), indicating the maximum number of octets that it is prepared to accept at once. We shall take as the initial state of our model, the `ESTABLISHED` state. We encourage readers unfamiliar with the TCP specifications to refer to Page 23 of RFC 793 and the amendments thereof on Page 86 of RFC 1122, which provides an illustration of all possible states of TCP. Such states include the `CLOSE_WAIT`, `FIN_WAIT_1` and `FIN_WAIT_2` states discussed in Section 3.1.

TCP uses the Sliding Window Protocol (SWP) for its data transfer. Both sender and receiver maintain a window of n sequence numbers, ranging from 0 to $n-1$, that they are allowed to send or receive, respectively. The sender may send as many octets as the size of its window before it has to wait for an acknowledgement from the receiver. Once the receiver sends an acknowledgement for m octets, its window *slides* forward m sequence numbers. Likewise, the sender's window *slides* m sequence numbers if this acknowledgement arrives. To function correctly over mediums that may lose data, the maximum size of the window is $\frac{n}{2}$ [23]. In the implementation of SWP underlying TCP, octets may be acknowledged before they are forwarded to the application layer (AL) and therefore still occupy a position in the receive buffer. In this case, the receiving entity reduces the window size by returning an acknowledgement segment, ensuring the sending entity does not send new data that will overflow its buffer. Once the octets are forwarded to the application layer, it may reopen the window.

The receiver may adjust the size of the sender's window at any time, through the value of `SEG.WND` set in acknowledgement segments. As the size of this field is limited to 16 bits, TCP can send at most 2^{16} octets into the medium before having to wait for an acknowledgement and, if the medium can hold octets, unnecessary delay will be incurred. To resolve this, RFC 1323 [14] proposes the Window Scale Option. If implemented, the send and receive windows are maintained as 32-bit numbers in the TCB of the sender and receiver, which is also extended to include variables `SND.WND.SCALE` and `RCV.WND.SCALE`. Whenever an entity receives an acknowledgement, it left-shifts the value of `SEG.WND` by the value of `SND.WND.SCALE` before it updates its send window. Likewise, whenever an entity sends an acknowledgement it sets the window field of the outgoing segment to the size of its receive window, right-shifted by the scale factor `RCV.WND.SCALE`.

We do not validate Connection Establishment, firstly because it has been well studied in the literature as discussed in Section 2 and, secondly, RFC 1323 adds no functionality to connection establishment that we expect to refute earlier verification efforts concerning this phase. We instead specify data transfer, taking care to include the core TCP functionality as well as any peripheral functionality that is potentially influenced by the Window Scale Option, in which two TCP instances (*TCP1* and *TCP2*) communicate data over a possibly faulty medium.

In this section we shall focus most of our attention on the process modelling the TCP instance as this was the primary exercise in modelling TCP. Although our μCRL specification also incorporates Connection Teardown we have omitted discussion of this procedure for brevity. In any case, in order to keep the verification tractable, the actions included to model Connection Teardown are encapsulated before instantiating the statespace in Section 4. Likewise, although we specify a generic TCP process that executes the responsibilities of the sending and receiving instances, when we come to compose our model of unidirectional TCP, including processes modelling the Application and Network layers, some actions must be encapsulated in TCP1 and TCP2 to instantiate them as the sending and receiving entities, respectively.

3.1 The TCP instance

To avoid discussing abstract notions such as connections (at the TCP level) and sessions (at the application level) that are rather detached from their contexts of sending and receiving entities in a network, we will take the point of view that we have modelled a TCP instance which only has one connection with one remote entity. This TCP instance maintains the state of the connection and the TCB. To manage its window, the sender maintains the variables `SND.UNA`, `SND.NXT` and `SND.WND` in the TCB. `SND.UNA` holds the sequence number of the first segment in the sequence number space that was sent, for which an acknowledgement has not yet been received. `SND.NXT` holds the sequence number of the next segment that the sender will send. Finally, `SND.WND` holds the number of octets that TCP may send at most before it should wait for an acknowledgement.

Some ambiguity surrounds the specification of the sequence number, as both octets and segments are assigned one. In principle, TCP numbers each octet with a unique sequence number, modulo the size of the sequence number space. A segment inherits its sequence number from the first octet it contains. However, a segment containing no octets still requires a sequence number. Here, it is still numbered with the sequence number maintained in `SND.NXT`, but `SND.NXT` is not updated. Henceforth, we adopt the convention of denoting `SND.NXT` as `SND_NXT` in μCRL, and likewise for other variables/states of the RFCs.

AL calls SEND The first call discussed in [17], `SEND`, may only be issued if the connection is in the `ESTABLISHED` or `CLOSE_WAIT` state. As long as there remains capacity in the send buffer, the TCP instance accepts `SEND` calls from the application layer via the `tcp_rcv_SND` event, adding its octets to the buffer.

Octets in send buffer? If the connection is in the `ESTABLISHED` or `CLOSE_WAIT` state and TCP is allowed to send one or more octets, a segment containing the eligible to be sent octets is passed to the network layer by issuing the call `tcp_call_SND`. This segment is labelled with the next sequence number to be used as maintained in `SND_NXT`. After the sequence number, the acknowledgement number and advertised window are included, followed by the number of

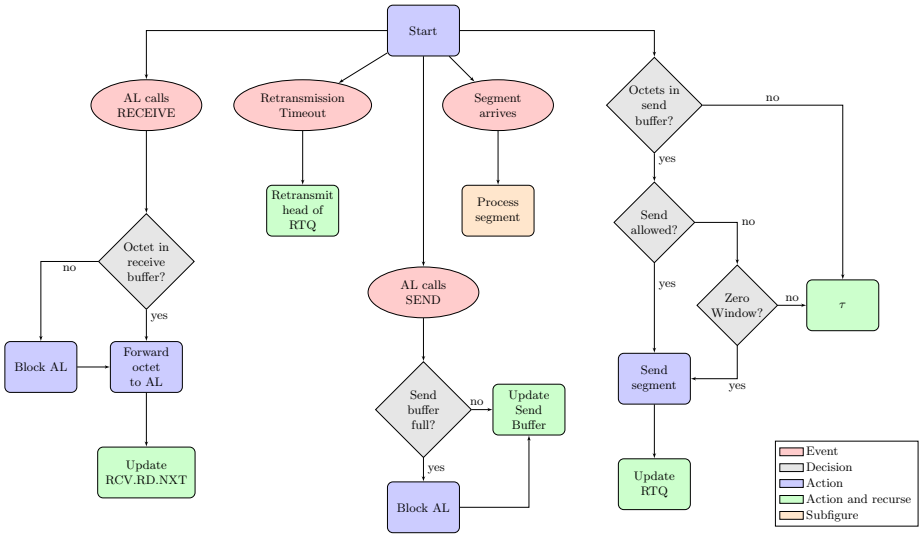

Fig. 1: Abstract overview of our specification of TCP Data Transfer

octets included in the segment and the values of the `ACK` and `FIN` flag. The actual number of octets that TCP can send at a certain time is calculated by taking the difference m between `SND.UNA` and `SND.NXT`. If $m <$ `SND.WND`, TCP may package any number of octets $n \leq m$ into a segment and send it into the medium. Note that the receive window size relayed in the segment is calculated by applying the scale factor `RCV.WND.SCALE` to the actual receive window size. Subsequently, the octets included in the segment are removed from the send buffer and the segment is added to the retransmission queue (rtq). Finally, `SND_NXT` is updated to reflect the next sequence number to be used.

In our model, the `ACK` flag will always be set to false in segments carrying data octets and therefore, the value of the acknowledgement field in the segment will not be processed by the receiver of the segment. The specification dictates that the `ACK` flag is always set to true and that the latest acknowledgement information is always included in each data segment. However, this would greatly complicate the processing of incoming segments in our model and would only be of use in case of a bidirectional connection, that we do not consider to limit the size of our state space. In a unidirectional setting, the sender's value of `RCV_NXT` will always be the same since it never receives data. Likewise, the receiver's values of `SND_NXT` and `SND_UNA` will remain constant since it never sends data. Hence, at all times during the execution of the protocol, if we have a sender A and a receiver B that have agreed on initial sequence number x, it will hold that $A_{\texttt{RCV\_NXT}} = B_{\texttt{SND\_NXT}} \wedge B_{\texttt{SND\_NXT}} = B_{\texttt{SND\_UNA}} = x$. Acknowledgement processing will not take place since $\neg(B_{\texttt{SND\_UNA}} < A_{\texttt{RCV\_NXT}})$.

More can be said about the octets that are eligible to be sent. If it holds that $\texttt{SND\_UNA} \leq \texttt{SND\_NXT} < (\texttt{SND\_UNA} + \texttt{SND\_WND})$, then the sender is allowed to send

$x = (\texttt{SND\_UNA} + \texttt{SND\_WND}) - \texttt{SND\_NXT}$ octets. To this end, we specified a function *can_send* that returns x if the length of the buffer is greater than x, or the length of the buffer otherwise. No octets may be sent if $\neg(\texttt{SND\_UNA} \leq \texttt{SND\_NXT} < (\texttt{SND\_UNA} + \texttt{SND\_WND}))$. From a modelling perspective, this solves an ambiguity in [17], namely that TCP may send octets *at its own will*.

A TCP instance must regularly transmit something to the remote entity if the variable SND_WND is set to 0 to prevent a potential deadlock. If the send window is 0 and the retransmission queue is empty, but octets are available in the send buffer, the sender will send a segment containing one octet. This segment is taken from the send buffer, put on the retransmission queue and the variable SND_NXT is updated. Note that this is the only major difference between our model and the behaviour specified in RFC 793; we delay further explanation and justification of this important revision until Section 4.

AL calls RECEIVE The second call from the application layer that the TCP instance must process is RECEIVE [17]. By issuing a tcp_rcv_RECEIVE call, parameterised with the octet pointed at by RCV_RD_NXT maintained in the TCB, that octet is offered to the application layer. The octet is removed from the receive buffer and RCV_RD_NXT is incremented. The call may only be issued if the connection is in an ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2 or CLOSE_WAIT state, if $(\texttt{RCV\_NXT} - \texttt{RCV\_RD\_NXT}) \bmod n > 0$ and if the octet with sequence number RCV_RD_NXT is available in the receive buffer. The variable RCV_RD_NXT is not mentioned in [17]. Instead, the size of the receive window, stored as RCV_WND in the TCB, is updated every time the receive buffer is manipulated. However, page 74 strictly requires the total of RCV_WND and RCV_NXT not to be reduced. It is unclear whether the total may not be reduced when an incoming segment is processed, or not at all. Either way, we believe that it relates to the requirement that the right edge of the window should never be moved to the left. To simplify the implementation but ensure this requirement we maintain RCV_WND at its initial value, and introduce the variable RCV_RD_NXT that is always the sequence number of the next octet to be forwarded to the application layer. At all times it holds that $\texttt{RCV\_NXT} \leq \texttt{RCV\_RD\_NXT} \leq (\texttt{RCV\_NXT} + \texttt{RCV\_WND})$.

Segment arrives A segment received from the network layer is deemed acceptable in the following situations [17]:

1. If the segment does not contain data octets:
   (a) If RCV.WND=0, it is required that SEG.SEQ=RCV.NXT
   (b) If RCV.WND>0, it is required that RCV.NXT≤SEG.SEQ<RCV.NXT+RCV.WND
2. If the segment does contain data octets
   (a) If RCV.WND=0, the segment is not acceptable.
   (b) If RCV.WND>0, it is required that
      – Either RCV.NXT≤SEG.SEQ<RCV.NXT+RCV.WND
      – Or: RCV.NXT≤SEG.SEQ+SEG.LEN−1<RCV.NXT+RCV.WND

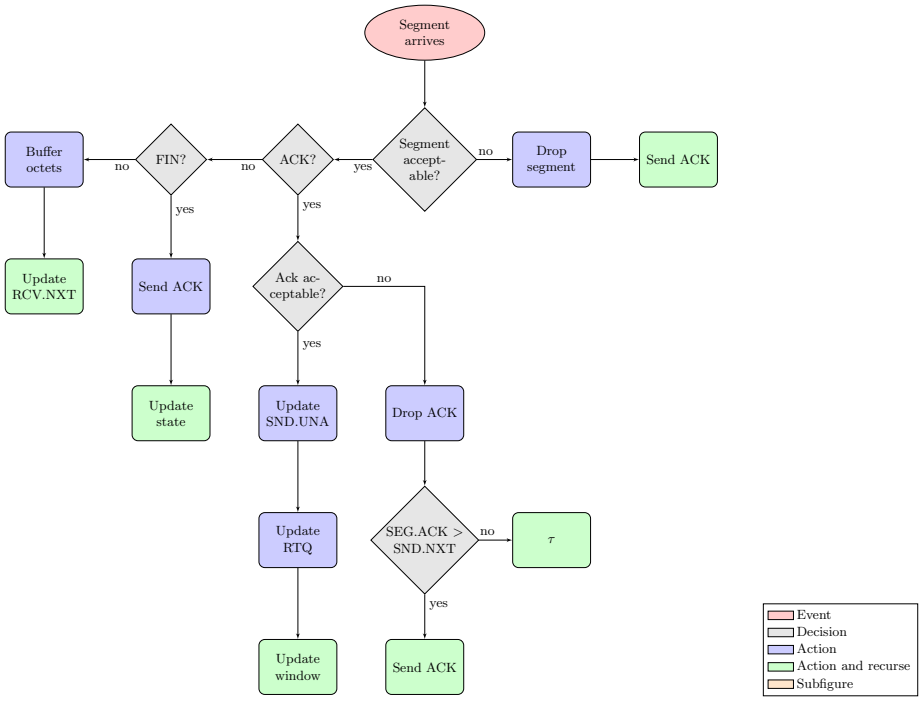

Fig. 2: Abstract overview of our specification of TCP Segment Processing

RFC 793 states that segments arriving out of order may be dropped by the receiver or suggests, to improve performance, that these segments are held in a special buffer to be processed regularly as soon as their turn arrives. In our model an unacceptable segment is dropped and an acknowledgement is sent to the sender containing the current value of `RCV.NXT`; from a modelling point of view, it makes no significant difference whether the segment is later taken from a special buffer or it is received again. Otherwise, segments continue to be processed in order of their sequence numbers. An acknowledgement is constructed including the sequence number of the octet that the TCP instance expects to receive next, the acknowledgement number and the advertised window. The `ACK`-flag of the acknowledgement segment is set to `T` while the `FIN`-flag is set to `F`.

We already stated that, in our model, a segment that carries data always has the `ACK` flag set to false. Additionally, we specify `FIN` segments to not contain acknowledgement information or data. We distinguish between segments carrying data, carrying acknowledgement information and carrying `FIN` information; we determine the type of a segment using functions *is_ack* and *fin_flag_set*.

If the incoming segment m is an acknowledgement, the TCP instance first checks that it is acceptable. This is done by verifying whether the acknowledgement number extracted from the segment is strictly between `SND_UNA` and `SND_NXT`, or indeed equal to `SND_NXT`. If so, `SND_WND` is updated to be the

size of the window contained in the segment, multiplied by the scale factor, SND_WND_SCALE. Moreover, SND_UNA is updated and segments containing octets with a sequence number of at most i are removed from the retransmission queue, where i is strictly between SND_UNA and the acknowledgement number extracted from the segment, or equal to SND_UNA. Note that, in [14] the scale factor is defined as n, resulting in integer division/multiplication by 2^n via bit shifting, whereas we maintain the scale factor as 2^n and apply scaling using division and multiplication. Where a scale factor of 1 is stated in [14], we write $2 = 2^1$.

Finally, if the acknowledgement is not acceptable it is dropped and the TCP instance remains in the same state. The official specification states that in response to an unacceptable acknowledgement an acknowledgement must be sent back if SEG.ACK > SND.NXT. In a unidirectional setting, such a situation will never occur so we exclude such behaviour from our model.

If the incoming segment is acceptable, for which (SEG_SEQ=RCV_NXT), and for which both *is_ack* and *fin_flag_set* return false, it is processed as a data segment. Its octets are added to the receive buffer and RCV_NXT is updated to reflect the next sequence number that the receiver expects to receive. Furthermore, the RCV_ACK_QUEUED flag in the TCB, which indicates that an acknowledgement should be sent, is set to true. This sets the SWP implementation of TCP apart, as an acknowledgement is sent while the octets may not yet be forwarded to the application layer and therefore occupy a position in the receive buffer. Therefore, the size of the window that is reported back represents the available capacity in the receive buffer if less than the difference between RCV.NXT and the size of the receive window that was originally agreed upon.

In [8], it states: *"a host [...] can increase efficiency [...] by sending fewer than one ACK (acknowledgement) segment per data segment received"*. We include this behaviour by setting a flag that an acknowledgement should be sent. We then enable the TCP instance to send an acknowledgement whenever the RCV_ACK_QUEUED flag in the TCB is set to true. To prevent sending an acknowledgement multiple times, this flag is then set to false again.

Retransmission Timeout For each segment the TCP instance puts on the retransmission queue, it starts a timer. When a timer expires, its segment must be retransmitted. To avoid modelling time explicitly, we allow a TCP instance to retransmit the first element on the retransmission queue at its own convenience.

3.2 The complete system

We obtain the complete system by putting two TCP instances in parallel with additional processes modeling the Application and Network Layers. The application layer continuously offers octets to the TCP instance by issuing the call al_call_SEND. Receiving data is modelled by having the application layer call al_call_RECEIVE for an arbitrary octet. Finally, we specify a network layer that may duplicate, reorder and lose data. General action names are renamed into action names specific for each component. We assume the variables to be set

as a result of the connection establishment procedure including the scale factor that each of the TCP instances will apply to their outgoing segments.

When instantiating processes for unidirectional TCP, we encapsulated (i.e., blocked) both `AL2_call_SEND` and `TCP2_rcv_SEND` to prevent AL2 from issuing the `SEND` call. Similarly, `AL1_call_RECEIVE` and `TCP1_rcv_RECEIVE` were encapsulated to prevent AL1 from issuing the `RECEIVE` call. The approach to exclude connection termination from our model is similar: by encapsulating actions `AL1_call_CLOSE`, `AL2_call_CLOSE`, `TCP1_rcv_CLOSE` and `TCP2_rcv_CLOSE`, we ensured that they will not be called in our model. Taken together, these modifications ensured that we were able to obtain a non-terminating model $TCP_{\rightarrow}$ of the data-transfer phase for a unidirectional TCP connection from our specification.

We also had to prevent deadlocks and undesired behaviour as a result of the reuse of sequence numbers. Whilst the introduction of PAWS [14] helps alleviate this problem when networks get faster (by increasing the size of the sequence number space) no protection is proposed other than to assume a limit on the time a segment can reside in the medium, namely the Maximum Segment Lifetime (MSL). Strictly speaking, this means that TCP cycles through all of its sequence numbers, waits until all segments have been acknowledged and all duplicates have drained from the network and starts a new run. The only reason that in practice there is no actual waiting period, is that data transfer speeds are guaranteed to be so 'slow' that it takes more than the MSL to send out octets with sequence numbers $i + 1 \ldots (i + (n - 1)) \, mod \, n$ after an octet with sequence number i is sent. Likewise, we ensure our specification does not get overly complex due to the addition of timing restrictions, by limiting our verification to one run of sequence numbers. We may still start anywhere in the sequence number space, since all calculations are defined modulo the size of this space.

However, the following problem remains. Assume a sequence number space ranging from $m \ldots n$. The receiving TCP instance will still accept an octet with sequence number m after receiving the octet with sequence number n. Hence, if such an octet is still in the medium as the result of duplication or retransmission, it will be accepted by the receiving TCP instance upon receipt and subsequently delivered to the application layer. Given that the assumption on the MSL holds, such behaviour cannot occur in a real-world situation. To model this we ensure that the global variable maintaining the total number of sequence numbers is greater than the number of octets, as the sequence number space is guaranteed to be larger than the number of octets sent and the problem will not occur.

4 Verification

Our verification focused on two aspects of our model: (i) we verified that its state space is deadlock free, and (ii) we compared the external behaviour of our model, defined in terms of the `SEND` and `RECEIVE` calls issued by the application layers, to the external behaviour of a FIFO queue. One can consider the `SEND` call of TCP as putting something into a queue and `RECEIVE` as taking something from it: the sender puts data elements into the transport medium and the re-

Octets Sent	Window Size	Window Scale	Medium Capacity	Levels	States	Transitions	Exploration Time
4	2	1	2	36	881.043	3.910.863	21 sec
			3	40	11.490.716	53.137.488	104 sec
			4	44	91.821.900	434.372.541	7.5 min
8	2	1	2	54	16.126.380	76.356.475	3 min
			3	58	823.501.590	4.031.264.559	49 min
	4	2	2	49	98.697.902	473.332.511	15 min
			3	56	3.505.654.685	Buffer Overflow	3 hrs, 40 min
16	4	2	2	77	3.255.174.492	3.444.088.224	4 hrs, 40 min

Table 2: Statistics of the state space generation for our model

ceiver takes them all out of this medium in precisely the same order. We first specified a behavioural specification B, then we generated an LTS from both B and $TCP_{\rightarrow}$. All actions in $TCP_{\rightarrow}$, other than `SEND` and `RECEIVE`, were defined as internal behaviour. We minimised the LTS of $TCP_{\rightarrow}$ and verified it was branching bisimilar to the LTS of B: $TCP_{\rightarrow} \leftrightarrows_B B$. Note that a fairness assumption is enforced by the minimisation algorithm; τ-loops from which an 'exit' is possible were eliminated from our minimised state space. Such τ-loops arise from segments that are continuously dropped by the network layer or a sequence of repeated retransmissions, behaviour that we can safely abstract from.

For both $TCP_{\rightarrow}$ and B, we generated a state space using the distributed state space generation tool `lps2lts-dist` of the ltsmin toolset [7]. By using the `--deadlock` option, absence of deadlocks could be checked during state space generation. In addition, we used the `--cache` option to speed up state space generation. State space generation was run on the DAS-4 cluster, more specifically on 8 nodes equipped with an Intel Sandy Bridge E5-2620 processor clocked at 2.0 GHz, 64 gigabytes of memory and a K20m Kepler GPU with 6 gigabytes of on-board memory. At each node, we utilised only 1 core to prevent the process from running out of memory. Table 2 shows some benchmarks of the state space generation for $TCP_{\rightarrow}$ for several different parameterisations. Subsequently, the state space of $TCP_{\rightarrow}$ was minimised with the `lts-reduce-dist` tool of the `ltsmin` toolset, which uses the distributed minimisation algorithm as described in [6]. Finally, the `ltsmin-compare` was used to verify that $TCP_{\rightarrow} \leftrightarrows_B B$.

As illustrated in Table 2, the capacity of the medium significantly impacts the size of the state space. We settled for a medium capacity of two segments. Since one segment may contain at most as many segments as the size of the window, a medium capacity of 2 means that a TCP instance can send at most two windows of data segments into the medium before it must 'wait' for the medium to either lose or deliver a segment. For the window scaling to be non-trivial, the size of the window should be at least 2^2 with a scale factor of 2^1 allowing three possible window sizes: zero, two and four that allow for interesting scenarios where the reported size of the window is shrunken to half of its original size.

We performed our verification assuming a sequence number space of size $2^3 + 1$, a window size of 2^2, a scale factor of 2^1 and a medium capacity of 2^1 segments, in which the sending TCP sends 2^3 octets. With these parameters, we

obtained a model that was small enough to verify within reasonable time, with characteristics that are representative for a real-world implementation of TCP. If the size of the model increases, all relevant buffers and calculations will simply scale with this increase; it is unlikely that errors are introduced as a result.

Correctness of the Window Scale Option Our initial hypothesis was that as the size of the window could only be reported in units of 2^n, problems could arise when a single octet is sent and the window that the receiving TCP entity reports must be adapted. Conceivably, situations could arise where a sending entity has a view of the size of the window at the receiving end that exceeds the maximum buffer space available. With the aid of our formal specification, we find that this is not the case. Both entities maintain the send and receive window as 32-bit numbers and maintain a scale factor by which they right/left-shift the value reported in/taken from an acknowledgement segment. Note that this shift by a factor k, has the same effect as a floored division or multiplication by a factor 2^k. Assume a receive buffer of capacity 2^{n+1} and, therefore, a window size of at most 2^n and a scale factor k, where $0 < k \leq (n-1)$, resulting in a division or multiplication by 2^k. Now, if the receiver receives a segment carrying $0 < m \leq 2^n$ bytes, two scenarios may occur: (i) the reduced buffer space (receive window) is reported in the acknowledgement segment; or, (ii) the old buffer space is reported. If (i), then the reported window size is $\lfloor (2^n - m)/2^k \rfloor < 2^{n-k} < 2^n$ else if (ii) then nothing changes and therefore the reported window size is $\lfloor 2^n/2^k \rfloor \leq 2^{n-k} < 2^n$. The reported buffer size is always $\leq 2^{n-k}$ and can never become greater than $2^{(n-k)+k} = 2^n$ when it is left-shifted at the remote end. Hence, a sending entity never views the receive buffer space available at a receiving entity to exceed the maximum buffer space available.

A second conceivable problem relates to the explicit statement in [17] that a TCP instance should not 'shrink' its receive window, meaning that the buffer capacity is reduced, i.e., the right edge of the window is moved to the left. Assume a sender and receiver have agreed upon a window size of 4. The sender sends two octets and immediately then sends another two octets. By the arrival of the first segment at the receiver, the octets are put in its receive buffer and, unfortunately, at the same time the capacity of the buffer is also reduced by one octet, causing the receiving entity to report a window of size 1 to the sender rather than 2. As the second segment, carrying two octets, is already in transit it will be discarded upon arrival at the receiver because it contains more octets than the receiver may accept. The sender will keep retransmitting this segment and it will be discarded as long as no octets are taken from the receive buffer. If window sizes get bigger, the delay incurred may significantly impact the performance of the protocol. Eventually, however, the octet will be accepted when buffer space becomes available as octets that arrived earlier are taken from the receive buffer.

When window scaling is in effect, one might expect such a scenario to occur every time an odd number of bytes is sent, due to the size of the window being reported only in multiples of 2^n. However, in this case the actual capacity of the receive buffer is not reduced and the receiver maintains the window size as a

32-bit not a 16-bit number. Therefore, the second segment that may have been in transit already, will still be accepted and an acknowledgement containing the latest size of the window will be sent back within reasonable time. In a situation where the segment was not yet sent, the difference in the number of octets that may be sent is only 1, causing a performance rather than a correctness issue.

Recommended revision of RFC 793 During our verification using μCRL, we identified a possibility of deadlock when strictly following the RFC 793 specification as we will show below. Therefore, we recommend revising the specification in its dealing with zero windows; requiring that *whenever the sender (i) has data on its send buffer, (ii) has a zero window and (iii) has an empty retransmission queue, a segment is sent to probe the zero window containing at least one octet of data from the send buffer.* This behaviour was included in our model as stated in Section 3.1 but we withheld explanation and justification until now.

Instead, the current specification states on page 42 that: *"The sending TCP must be prepared to accept from the user and send at least one octet of new data even if the send window is zero. The sending TCP must regularly retransmit to the receiving TCP even when the window is zero. [...] This retransmission is essential to guarantee that when either TCP has a zero window the re-opening of the window will be reliably reported to the other."* The latter part of this statement is confusing. It is not the *retransmission* that is essential, but rather the *transmission* of a segment (whether taken from the send buffer or the retransmission queue) when the send window is zero.

To see why, suppose that the sender has two octets on its send buffer and sends only the first of these. The receiver then acknowledges this octet, but does not yet take it from its buffer. As a result of this, both the send and receive window are now 0. In this scenario, there is still data to be sent, but the retransmission queue is empty. If the requirements above are strictly followed, the zero window will never be probed as long as the user does not provide *new data* for the sender to *accept and send at least one octet of* and therefore leads to deadlock. Implicitly, the reader may expect data on the send buffer to be sent in this case, regardless. However, this is certainly contradicted by the suggestion to *"avoid sending small segments by waiting until the window is large enough before sending data"*. Note that care should be taken when implementing this feature, since as a result of waiting to send something, no new acknowledgements will arrive to update the window information, again leading to deadlock.

Requiring the sender to be prepared to send at least one octet of new data even when the retransmission queue is non-empty also ensures that the window will be reopened, but not how one would expect. The new data is sent to the receiver, which will reject the segment since it is out of sequence. However, as a result of this rejection, the receiver sends an acknowledgement containing up-to-date window information, potentially reopening the window. Our proposed revision intentionally does not attend to the case of the retransmission queue being non-empty; it is already covered by the requirement that *"if the retransmission timeout expires on a segment in the retransmission queue, send the segment at*

the front of the retransmission queue again, reinitialize the retransmission time and return" on page 77. As an advantage, whenever the retransmission queue is non-empty an in-sequence segment will be sent and therefore accepted while its acknowledgement may also reopen the window. Only if the retransmission queue is empty, a segment containing new data probes the zero window. This segment is then guaranteed to be accepted if the receiver has reopened its window.

It may be that our revision matches the interpretation intended of the original specification, but we have shown that the wording of the specification can lead to implementations where deadlocks occur. A formal specification, given here in μCRL, leaves less scope for erroneous implementations due to misinterpretation.

5 Conclusion

TCP plays an important role in the internet, providing reliable transport of data over possibly faulty networks. The protocol is complex and its specification consists of many documents that mainly describe the proposed functioning of the protocol in natural language. We set out to formally specify TCP extended with the Window Scale Option and verify its correctness, redressing the lack of consideration paid to this option in earlier verification efforts.

Whilst formally specifying TCP, we traversed several ambiguities in RFC 793; the modelling decisions we made in this regard were stated in Section 3. Due to its formal nature, our specification may serve as a useful reference for implementors of the protocol. Moreover, our recommendation for revising RFC 793 brings attention to the potential misinterpretations of how and when to probe the zero window, which we have shown may lead to deadlock.

The process algebra that we used for our specification, μCRL, turned out to be powerful enough to mimic the required features. The size of the state space, however, was a limiting factor that forced us to split our verification efforts into a verification of TCP data transfer and a separate verification of connection teardown; the latter was omitted from this paper.

Using the `ltsmin` toolset, we were able to formally verify that our μCRL specification of unidirectional TCP extended with the Window Scale Option does not contain deadlocks, and that its external behaviour is branching bisimilar to a FIFO queue for a significantly large instance. In addition, we believe that the specification is general enough to make the introduction of errors as parameters are increased highly unlikely. Whilst our specification also supports bidirectional data transfer, only unidirectional instances were verified to avoid intractable state space explosion; a point we aim to redress in subsequent work.

Acknowledgments

The authors are indebted to Dr. Barry M. Cook, for posing the initial research question, and Dr. Kees Verstoep, for essential support in using the DAS-4 cluster. We also thank the anonymous reviewers for their insightful suggestions.

References

1. B. Badban, W.J. Fokkink, J.F. Groote, J. Pang, and J. van de Pol. Verification of a sliding window protocol in μCRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.
2. B. Badban, W.J. Fokkink, and J. van de Pol. Mechanical verification of a two-way sliding window protocol. In *CPA*, volume 66 of *CSE*, pages 179–202. IOS Press, 2008.
3. M. Bezem and J.F. Groote. A correctness proof of a one-bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.
4. J. Billington and B. Han. On defining the service provided by TCP. In *ACSC*, volume 16 of *CRPIT*, pages 129–138. ACS, 2003.
5. S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *SIGCOMM*, pages 265–276. ACM, 2005.
6. S. Blom and S. Orzan. Distributed state space minimization. *Software Tools for Technology Transfer*, 7(3):280–291, 2005. Springer.
7. S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In *CAV*, pages 345–359, 2010.
8. R. Braden. Requirements for Internet hosts-communication layers. *RFC1122*, 1989.
9. D. Chkliaev, J. Hooman, and E.P. de Vink. Verification and improvement of the sliding window protocol. In *TACAS*, volume 2619 of *LNCS*, pages 113–127. Springer, 2003.
10. W.J. Fokkink. *Modelling Distributed Systems*. Texts in theoretical computer science, An EATCS Series. Springer, 2007.
11. B. Han and J. Billington. Validating TCP connection management. In *SEFW*, pages 47–55. ACS, 2002.
12. B. Han and J. Billington. Experience using coloured Petri nets to model TCP's connection management procedures. In *CPN*, pages 57–76, 2004.
13. B. Han and J. Billington. Termination properties of TCP's connection management procedures. In *ICATPN*, volume 3536 of *LNCS*, pages 228–249. Springer, 2005.
14. V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. *RFC 1323*, 1992.
15. E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol in LOTOS. In *FORTE*, volume C-2 of *IFIP Trans.*, pages 495–510, 1991.
16. S.L. Murphy and A.U. Shankar. Service specification and protocol construction for the transport layer. In *SIGCOMM*, pages 88–97. ACM, 1988.
17. J. Postel. Transmission control protocol. *RFC 793*, 1981.
18. T. Ridge, M. Norrish, and P. Sewell. A rigorous approach to networking: TCP, from implementation to protocol to service. In *FM*, volume 5014 of *LNCS*, pages 294–309. Springer, 2008.
19. I. Schieferdecker. Abruptly terminated connections in TCP - a verification example. In *Applied Formal Methods in System Design*, pages 136–145, 1996.
20. M.A. Smith and K.K. Ramakrishnan. Formal specification and verification of safety and performance of TCP selective acknowledgement. *Trans. on Networking*, 10(2):193–207, 2002. IEEE/ACM.
21. M.A.S. Smith. Formal verification of communication protocols. In *FORTE*, volume 69 of *IFIP Conf. Proc.*, pages 129–144. Chapman & Hall, 1996.
22. M.A.S. Smith. *Formal verification of TCP and T/TCP*. PhD thesis, Massachusetts Institute of Technology, 1997.
23. A.S. Tanenbaum. *Computer Networks (4. ed.)*. Prentice Hall, 2002.

A Data types

A.1 Booleans

First of all, `T` (true) and `F` (false) are introduced as boolean variables of data type *Bool*. In addition, we define several operations on booleans.

$$\begin{array}{ll} \texttt{T} : \rightarrow \mathit{Bool} & \neg : \mathit{Bool} \rightarrow \mathit{Bool} \\ \texttt{F} : \rightarrow \mathit{Bool} & = : \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} \\ \wedge : \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} & \mathit{if} : \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} \\ \vee : \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} & \end{array}$$

Of these operations, $\wedge, \vee$ and $\neg$ are implemented as expected. $=$ defines equality of booleans while *if* defines an if-then-else construct on booleans:

$$(x = y) = \begin{cases} \texttt{T} & \text{if } x = \texttt{T} \wedge y = \texttt{T} \\ \texttt{T} & \text{if } x = \texttt{F} \wedge y = \texttt{F} \\ \texttt{F} & \text{otherwise} \end{cases} \qquad \mathit{if}(x, y, z) = \begin{cases} y & \text{if } x = \texttt{T} \\ z & \text{if } x = \texttt{F} \end{cases}$$

An operation $\mathit{if} : D \times D \rightarrow \mathit{Bool}$ is added in a similar fashion for any other data type D that we define.

A.2 Natural numbers

0 and S are introduced as constructors of data type *Nat* – the natural numbers – and several operations on natural numbers are defined:

$$\begin{array}{lll} 0 : \rightarrow \mathit{Nat} & > & : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Bool} \\ S : \mathit{Nat} \rightarrow \mathit{Nat} & \geq & : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Bool} \\ = : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Bool} & \mathrm{mod} & : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Nat} \\ + : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Nat} & \dot{-} & : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Nat} \\ * : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Nat} & \mathit{seq\_diff} & : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Nat} \\ \dot{:} : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Nat} & \mathit{seq\_between} & : \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Bool} \\ < : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Bool} & \mathit{seq\_between\_excl} & : \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Bool} \\ \leq : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Bool} & & \end{array}$$

$=, +, *, <, \leq, >, \geq$ and mod are defined as one would expect them to be. $\dot{-}$ defines the *monus* operation ($-$ with the exception that it never goes below 0) while $\dot{:}$ is defined as integer division:

$$\begin{array}{ll} 0 \dot{-} y & = 0 \\ x \dot{-} 0 & = x \\ S(x) \dot{-} S(y) & = x \dot{-} y \end{array} \qquad x \mathbin{\dot{:}} y = \begin{cases} 1 + ((x - y) \mathbin{\dot{:}} y) & \text{if } y > 0 \wedge x \geq y \\ 0 & \text{if } y > 0 \wedge x < y \end{cases}$$

In addition to these standard operations, we had to define some operations on sequence numbers since sequence numbers are used modulo n, with n the size of the sequence number space. *seq_diff* defines a difference operation on sequence numbers, while *seq_between_excl* defines whether a sequence number x is between sequence numbers y and z. *seq_between* defines whether a sequence number x is between sequence numbers y and z or $x = z$.

$$
\begin{array}{ll}
\mathit{seq\_diff}(0, y) & = y \\
\mathit{seq\_diff}(S(x), 0) & = \mathit{seq\_diff}(x, n \dot{-} 1) \\
\mathit{seq\_diff}(S(x), S(y)) & = \mathit{seq\_diff}(x, y) \\
\mathit{seq\_between}(x, y, z) & = \mathit{seq\_between\_excl}(x, y, z) \vee x = z \\
\mathit{seq\_between\_excl}(x, 0, y) & = x < y \\
\mathit{seq\_between\_excl}(S(x), S(y), 0) & = \mathit{seq\_between\_excl}(x, y, n \dot{-} 1) \\
\mathit{seq\_between\_excl}(0, S(y), S(z)) & = \mathit{seq\_between\_excl}(n - 1, y, z) \\
\mathit{seq\_between\_excl}(0, S(y), 0) & = \mathit{seq\_between\_excl}(n - 1, y, n - 1) \\
\mathit{seq\_between\_excl}(S(x), S(y), S(z)) & = \mathit{seq\_between\_excl}(x, y, z)
\end{array}
$$

A.3 Segments

Of the TCP header, only the sequence number, the acknowledgement number, the window size, the `ACK` flag and the `FIN` flag are important to our specification. In addition, we need to know the number of octets that are included in the segment as data. This can normally be calculated by subtracting the data offset from the length field that is included in the IP header. We abstract away from this implementation detail and simply include this number in the segment. Note that our segments do not contain a buffer with data, as such a buffer can easily be reconstructed from the information in the header. Hence, a data type *Sgmt* representing segments is defined with the following constructor function:

$$\mathit{sgmt} : \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \times \mathit{Bool} \times \mathit{Bool} \to \mathit{Sgmt}$$

In addition, we define the following operations on segments:

$$
\begin{array}{lllll}
= & : \mathit{Sgmt} \times \mathit{Sgmt} \to \mathit{Bool} & \quad & \mathit{get\_num\_octs} & : \mathit{Sgmt} \to \mathit{Nat} \\
\mathit{get\_seq\_nr} & : \mathit{Sgmt} \to \mathit{Nat} & & \mathit{is\_ack} & : \mathit{Sgmt} \to \mathit{Bool} \\
\mathit{get\_ack\_nr} & : \mathit{Sgmt} \to \mathit{Nat} & & \mathit{fin\_flag\_set} & : \mathit{Sgmt} \to \mathit{Bool} \\
\mathit{get\_window} & : \mathit{Sgmt} \to \mathit{Nat} & & &
\end{array}
$$

Two segments are defined to be equal, expressed by the operation =, if all of their arguments are equal. *get_seq_nr*, *get_ack_nr*, *get_window*, *get_num_octs*, *is_ack* and *fin_flag_set* are destructor methods that return the first, second, third, fourth, fifth and sixth argument of a segment, respectively.

A.4 Octet buffers

A TCP instance maintains a send and receive buffer for octets that have either just been accepted from the application layer and are waiting to be sent or that have just arrived and are ready to be forwarded to the application layer. To reduce the size of the state space, these buffers are implemented as sorted lists of natural numbers. For such lists, we defined the data type *Buffer* with constructor functions $\emptyset$ and *li*, for the empty list and an item in the list respectively. Additionally, we specify several operations on buffers:

$\emptyset$	$: \rightarrow Buffer$	$\in$	$: Nat \times Buffer \rightarrow Bool$
li	$: Nat \times Buffer \rightarrow Buffer$	*merge*	$: Buffer \times Buffer \rightarrow Buffer$
$=$	$: Buffer \times Buffer \rightarrow Bool$	*take_set*	$: Buffer \times Buffer \rightarrow Buffer$
first	$: Buffer \rightarrow Nat$	*infl*	$: Nat \times Nat \rightarrow Buffer$
rest	$: Buffer \rightarrow Buffer$	*add_ordered*	$: Nat \times Buffer \rightarrow Buffer$
add	$: Nat \times Buffer \rightarrow Buffer$	*take_n*	$: Buffer \times Nat \rightarrow Buffer$
length	$: Buffer \rightarrow Nat$		

$=$ defines equality of two buffers as the equality of their contents. *first* returns the first element in a buffer, while *rest* returns the buffer without its first element. *add* prepends an element to the front of the buffer. Likewise *add_ordered* adds an element to the buffer, but ensures that the buffer stays ordered. *length* returns the number of elements in the buffer. *merge* takes two buffers as arguments and returns an ordered buffer that is the result of merging the two. If an element x occurred in both buffers, it will occur twice in the resulting buffer. *take_n* takes one occurrence of an element x from a buffer. *take_set* takes two buffers b_1 and b_2 as arguments, and returns a buffer b_3 that consists of the buffer b_1 to which *take_n* is applied for every element in b_2. Finally, *infl* takes two sequence numbers x and y as arguments, and yields an ordered buffer that contains y sequence numbers starting at x (taking the fact that sequence numbers are taken modulo n into account).

$\emptyset = \emptyset$	$= \mathtt{T}$	$merge(\emptyset, b_1)$	$= b_1$
$li(x, b_1) = li(y, b_2)$	$= x = y \wedge b_1 = b_2$	$merge(b_1, \emptyset)$	$= b_1$
$first(li(x, b_1))$	$= x$	$merge(li(x, b_1), b_2)$	$= add\_ordered(x, merge(b_1, b_2))$
$rest(\emptyset)$	$= \emptyset$	$take\_set(b_1, \emptyset)$	$= b_1$
$rest(li(x, b_1))$	$= b_1$	$take\_set(\emptyset, b_1)$	$= \emptyset$
$add(x, b_1)$	$= li(x, b_1)$	$take\_set(b_1, li(x, b_2))$	$= take\_set(take\_n(b_1, x), b_2)$
$length(\emptyset)$	$= 0$	$infl(x, 0)$	$= \emptyset$
$length(li(x, b_1))$	$= 1 + length(b_1)$	$infl(x, S(y))$	$= add\_ordered(x, infl((x + 1) \bmod n, y)$
$x \in \emptyset$	$= \mathtt{F}$	$take\_n(\emptyset, x)$	$= \emptyset$
$x \in li(y, b_1)$	$= x = y \vee x \in b_1$		

$$take\_n(li(x, b_1), y) = \begin{cases} li(x, take\_n(b_1, y)) & \text{if } x < y \\ b_1 & \text{if } x = y \\ li(x, b_1) & \text{otherwise} \end{cases}$$

$$add\_ordered(x, li(y, b_1)) = \begin{cases} li(x, li(y, b_1)) & \text{if } x \leq y \\ li(y, add\_ordered(x, b_1)) & \text{otherwise} \end{cases}$$

A.5 Segment buffers

In addition to a buffer for octets that have just been accepted from the application layer and are waiting to be sent, the sender also maintains a retransmission queue. Furthermore, mediums may hold any number of segments at any point in time. To facilitate these requirements, we have defined *SgmtQ*, a buffer of segments, with constructor functions $\emptyset$ and *qu*, for the empty queue and a segment on the queue respectively. On these queues, we again define several operations:

$empq$	$: \rightarrow SgmtQ$	$add\_ordered$	$: Sgmt \times SgmtQ \rightarrow SgmtQ$
qu	$: Sgmt \times SgmtQ \rightarrow SgmtQ$	$remove$	$: Sgmt \times SgmtQ \rightarrow SgmtQ$
$first$	$: SgmtQ \rightarrow Sgmt$	$length$	$: SgmtQ \rightarrow Nat$
$rest$	$: SgmtQ \rightarrow SgmtQ$	$=$	$: SgmtQ \times SgmtQ \rightarrow Bool$
add	$: Sgmt \times SgmtQ \rightarrow SgmtQ$	$\in$	$: SgmtQ \rightarrow Bool$

The operations *first*, *rest*, *add*, *add_ordered*, *length*, $=$ and $\in$ are defined in similarly to the operations on *Buffer* as discussed in the previous section. *remove* takes a segment s and a buffer b_1 as a parameter and returns b_1' without s.

A.6 Connection states

Since a connection may progress through several states, a data type *ConnState* is defined, with the following constructors:

$CLOSED$	$: \rightarrow ConnState$	$FINWAIT2$	$: \rightarrow ConnState$
$LISTEN$	$: \rightarrow ConnState$	$CLOSEWAIT$	$: \rightarrow ConnState$
$SYNSENT$	$: \rightarrow ConnState$	$CLOSING$	$: \rightarrow ConnState$
$SYNRECEIVED$	$: \rightarrow ConnState$	$LASTACK$	$: \rightarrow ConnState$
$ESTABLISHED$	$: \rightarrow ConnState$	$TIMEWAIT$	$: \rightarrow ConnState$
$FINWAIT1$	$: \rightarrow ConnState$		

Apart from the equality function $=$ and conditional function *if*, no other functions have been defined for this data type. Additionally, *ConnectionStates* represents a list of connection states to achieve notational convenience in the guards of our specification.

A.7 The Transmission Control Block

The Transmission Control Block (TCB) maintains all variables for a connection. It is specified as data type *TransCtrlBlk*, with the following constructor:

$$tcb : Buffer \times SgmtQ \times Nat \times Nat \times Nat \times Nat \times Nat \times Nat \times Nat \times Nat \times Buffer \times Nat \times Nat \times Nat \times Nat \times Nat \times Nat \times Bool \rightarrow TransCtrlBlk$$

The TCB contains the following variables (listed in order of the constructor) :

For sending segments

- The send buffer
- The retransmission queue
- `SND_UNA`
- `SND_NXT`
- `SND_WND`
- `SND_UP`
- `SND_WL1`
- `SND_WL2`
- `SND_ISS`†
- `SND_WND_SCALE`†

For receiving segments

- The receive buffer
- `RCV_NXT`
- `RCV_WND`†
- `RCV_UP`
- `RCV_IRS`†
- `RCV_WND_SCALE`†
- `RCV_RD_NXT`
- `RCV_ACK_QUEUED`

`RCV_RD_NXT` keeps track of the next octet that the TCP instance should forward to the application layer, while `RCV_ACK_QUEUED` is used to determine whether an acknowledgement should be sent to the remote entity to acknowledge the reception of a segment. For each of the variables in the TCB, a getter and setter function are defined. Furthermore, equality of TCBs tcb_1 and tcb_2 is defined as equality of all variables of tcb_1 and tcb_2.

For the variables marked with a †, the value is determined during connection establishment and remains constant during the execution of the protocol. We chose to include them in the TCP for future extensibility; if we wish to add connection establishment to our specification at a later stage, the variables are already included. The same holds for the variables `SND_UP` and `RCV_UP`, which are not used currently since we have not included the urgent function in our specification. Also note that, in case of a unidirectional connection, variables related to sending segments will remain constant during the execution of the protocol at the sending TCP instance while variables related to receiving segments will remain constant at the receiving TCP instance.

A.8 Utility functions

Apart from the data types and functions as discussed above, we have also implemented several utility functions. First of all, *can_send* is used by a TCP instance to determine the number of octets that it may send. *rcv_wnd* is used to calculate the size of the receive window that should be advertised to the remote

entity. *adv_wnd* also calls this method and subsequently applies the scale factor to its result by means of the operator $\dot{\div}$. *may_accept_ack* determines whether an acknowledgement segment is acceptable while *must_update_window* determines whether the window information in an acknowledgement is 'fresh' enough to be used to update the size of the send window. *receiver_may_accept* is used to determine whether an incoming segment is eligible to be accepted. Finally, *update_rtq* is used to remove segments from the retransmission queue that are acknowledged by an incoming acknowledgement and *construct_ack* takes the current state of the TCB as a parameter and yields an acknowledgement segment.

$$
\begin{array}{ll}
\mathit{can\_send} & : \mathit{TransCtrlBlk} \to \mathit{Nat} \\
\mathit{rcv\_wnd} & : \mathit{TransCtrlBlk} \to \mathit{Nat} \\
\mathit{adv\_wnd} & : \mathit{TransCtrlBlk} \to \mathit{Nat} \\
\mathit{may\_accept\_ack} & : \mathit{Sgmt} \times \mathit{TransCtrlBlk} \to \mathit{Bool} \\
\mathit{must\_update\_window} & : \mathit{Sgmt} \times \mathit{TransCtrlBlk} \to \mathit{Bool} \\
\mathit{receiver\_may\_accept} & : \mathit{Sgmt} \times \mathit{TransCtrlBlk} \to \mathit{Bool} \\
\mathit{update\_rtq} & : \mathit{SgmtQ} \times \mathit{Nat} \times \mathit{Nat} \to \mathit{SgmtQ} \\
\mathit{construct\_ack} & : \mathit{TransCtrlBlk} \to \mathit{Sgmt}
\end{array}
$$

B Specification of TCP

All updates to variables in the TCB are of the form $a \mapsto b$. If a is also used at the right side of the substitution, this means that the old value is first retrieved from the TCB for manipulation.

B.1 The application layer

$$
\begin{aligned}
\mathit{AL}(x{:}\mathit{Nat}, y{:}\mathit{Nat}) = {} & \mathit{al\_call\_SEND}(x) \cdot \mathit{AL}((x+1) \bmod n, y-1) \ \triangleleft\ y > 0 \ \triangleright\ \delta \\
& + \sum_{z:\mathit{Nat}} (\mathit{al\_call\_RECEIVE}(z) \cdot \mathit{AL}(x,y) \ \triangleleft\ z < n \ \triangleright\ \delta) \\
& + \mathit{al\_call\_CLOSE} \cdot \mathit{AL}(x,y) \ \triangleleft\ y = 0 \ \triangleright\ \delta
\end{aligned}
$$

The application layer does not change state after accepting `RECEIVE` call, as we abstract from the actual processing of the data at the application layer.

B.2 The TCP instance

AL calls SEND

$$
\begin{aligned}
& \mathit{TCP}(s{:}\mathit{ConnectionState}, t{:}\mathit{TransmissionControlBlock}) = \\
& \sum_{x:\mathit{Nat}} (\mathit{tcp\_rcv\_SND}(x) \cdot \mathit{TCP}(s, t[\mathit{send\_buffer} \mapsto \mathit{add\_ordered}(x, \mathit{send\_buffer})]) \\
& \triangleleft\ s \in \{\mathit{ESTABLISHED},\ \mathit{CLOSEWAIT}\} \\
& \quad \wedge x < n \wedge \mathit{length}(\mathit{get\_send\_buffer}(t)) < \mathit{buffer\_capacity} \ \triangleright\ \delta)
\end{aligned}
$$

Octets in send buffer?

$$+\ tcp\_call\_SND(sgmt(get\_SND\_NXT(t), get\_RCV\_NXT(t), adv\_wnd(t), can\_send(t), \mathtt{F}, \mathtt{F}))\cdot$$

$$TCP(s, t[rtq \mapsto add(sgmt(SND\_NXT, RCV\_NXT, adv\_wnd(t), can\_send(t), \mathtt{F}, \mathtt{F}), rtq),$$

$$send\_buffer \mapsto take\_set(send\_buffer, infl(SND\_NXT, can\_send(t))),$$

$$SND\_NXT \mapsto (SND\_NXT + can\_send(t)) \bmod n])$$

$$\lhd\ s \in \{\textit{ESTABLISHED, CLOSEWAIT}\} \wedge can\_send(t) > 0\ \rhd\ \delta$$

$$+\ tcp\_call\_SND(sgmt(get\_SND\_NXT(t), get\_RCV\_NXT(t), adv\_wnd(t), 1, \mathtt{F}, \mathtt{F}))\cdot$$

$$TCP(s, t[rtq \mapsto add(sgmt(SND\_NXT, RCV\_NXT, adv\_wnd(t), 1, \mathtt{F}, \mathtt{F}), rtq),$$

$$send\_buffer \mapsto take\_set(send\_buffer, infl(SND\_NXT, 1)),$$

$$SND\_NXT \mapsto (SND\_NXT + 1) \bmod n])$$

$$\lhd\ can\_send(t) = 0 \wedge get\_SND\_WND(t) = 0$$

$$\wedge length(get\_rtq(t)) = 0 \wedge length(get\_send\_buffer(t)) > 0\ \rhd\ \delta$$

AL calls RECEIVE

$$+\ tcp\_rcv\_RECEIVE(get\_RCV\_RD\_NXT(t))\cdot$$

$$TCP(s, t[receive\_buf \mapsto take\_n(receive\_buf,\ RCV\_RD\_NXT)$$

$$RCV\_RD\_NXT \mapsto (RCV\_RD\_NXT + 1) \bmod n])$$

$$\lhd\ s \in \{\textit{ESTABLISHED, FINWAIT1, FINWAIT2, CLOSEWAIT}\}$$

$$\wedge seq\_diff(get\_RCV\_RD\_NXT(t), get\_RCV\_NXT(t)) > 0$$

$$\wedge length(get\_receive\_buf(t)) > 0 \wedge get\_RCV\_RD\_NXT(t) \in get\_receive\_buf(t)\ \rhd\ \delta$$

AL calls CLOSE

$$+\ tcp\_rcv\_CLOSE\cdot$$

$$tcp\_call\_SND(sgmt(get\_SND\_NXT(t), get\_RCV\_NXT(t), adv\_wnd(t), 0, \mathtt{F}, \mathtt{T}))\cdot$$

$$TCP(\textsc{finwait1}, t[rtq \mapsto add(sgmt(SND\_NXT, RCV\_NXT, adv\_wnd(t), 0, \mathtt{F}, \mathtt{T}), rtq),$$

$$SND\_NXT \mapsto (SND\_NXT + 1) \bmod n])$$

$$\lhd\ s = \textsc{established} \wedge length(get\_send\_buffer(t)) = 0\ \rhd\ \delta$$

$$+\ tcp\_rcv\_CLOSE\cdot$$

$$tcp\_call\_SND(sgmt(get\_SND\_NXT(t), get\_RCV\_NXT(t), adv\_wnd(t), 0, \mathtt{F}, \mathtt{T}))\cdot$$

$$TCP(\textsc{lastack}, t[rtq \mapsto add(sgmt(SND\_NXT, RCV\_NXT, adv\_wnd(t), 0, \mathtt{F}, \mathtt{T}), rtq),$$

$$SND\_NXT \mapsto (SND\_NXT + 1) \bmod n])$$

$$\lhd\ s = \textsc{closewait} \wedge length(get\_send\_buffer(t)) = 0\ \rhd\ \delta$$

Segment Arrives

$$+ \sum_{m:Sgmt} (tcp\_call\_RECV(m) \cdot tcp\_call\_SND(construct\_ack(t)) \cdot TCP(s,t)$$

$$\lhd\ s \in \{\textsc{established, closewait, finwait1, finwait2, closing, lastack, timewait}\}$$

$$\wedge(\neg receiver\_may\_accept(m,t) \vee get\_seq\_nr(m) \neq get\_RCV\_NXT(t))\ \rhd\ \delta)$$

When an unacceptable segment is received, an acknowledgement is sent immediately. Acknowledgements are constructed using *construct_ack*, which takes the current state of the TCB as a parameter. It then constructs an acknowledgement including the sequence number of the octet that the TCP instance is expected to receive next, the acknowledgement number and the advertised window. Obviously, the `ACK`-flag of the acknowledgement segment is set to `T` while the `FIN`-flag is set to `F`:

$$construct\_ack(t) = sgmt(get\_SND\_NXT(t), get\_RCV\_NXT(t), adv\_wnd(t), \mathtt{T}, \mathtt{F})$$

If the incoming segment m is a `FIN` segment, it is processed as described on pages 75-76 of [17].

$$+ \sum_{m:Sgmt} (tcp\_call\_RECV(m) \cdot$$

$$tcp\_call\_SND(construct\_ack(t[RCV\_NXT \mapsto (RCV\_NXT + 1) \bmod n])) \cdot$$

$$TCP(s', t[RCV\_NXT \mapsto (RCV\_NXT + 1) \bmod n, RCV\_ACK\_QUEUED \mapsto \mathtt{F}])$$

$$\lhd\ s \in \{\textit{ESTABLISHED, FINWAIT1, FINWAIT2, CLOSEWAIT, CLOSING, LASTACK, TIMEWAIT}\}$$

$$\wedge receiver\_may\_accept(m, t) \wedge get\_seq\_nr(m) = get\_RCV\_NXT(t)$$

$$\wedge fin\_flag\_set(m)\ \rhd\ \delta)$$

If the incoming segment is an acknowledgement, it is verified to be acceptable.

$$+ \sum_{m:Sgmt} (tcp\_call\_RECV(m) \cdot$$

$$TCP(s', t[rtq \mapsto update\_rtq(rtq, get\_ack\_nr(m), SND\_UNA),$$

$$SND\_WL2 \mapsto get\_ack\_nr(m), SND\_WL1 \mapsto get\_seq\_nr(m),$$

$$SND\_WND \mapsto get\_window(m) * SND\_WND\_SCALE,$$

$$SND\_UNA \mapsto get\_ack\_nr(m)])$$

$$\lhd\ s \in \{\textit{ESTABLISHED, CLOSEWAIT, FINWAIT1, FINWAIT2, CLOSING, LASTACK}\}$$

$$\wedge receiver\_may\_accept(m, t) \wedge get\_seq\_nr(m) = get\_RCV\_NXT(t)$$

$$\wedge is\_ack(m) \wedge may\_accept\_ack(m, t) \wedge must\_update\_window(m, t)\ \rhd\ \delta)$$

$$+ \sum_{m:Sgmt} (tcp\_call\_RECV(m) \cdot$$

$$TCP(s', t[rtq \mapsto update\_rtq(rtq, get\_ack\_nr(m), SND\_UNA), SND\_UNA \mapsto get\_ack\_nr(m)])$$

$$\lhd\ s \in \{\textit{ESTABLISHED, CLOSEWAIT, FINWAIT1, FINWAIT2, CLOSING, LASTACK}\}$$

$$\wedge receiver\_may\_accept(m, t) \wedge get\_seq\_nr(m) = get\_RCV\_NXT(t)$$

$$\wedge is\_ack(m) \wedge may\_accept\_ack(m, t) \wedge \neg must\_update\_window(m, t)\ \rhd\ \delta)$$

$$+ \sum_{m:Sgmt} (tcp\_call\_RECV(m) \cdot TCP(s,t)$$

$$\lhd\ s \in \{\textit{ESTABLISHED, CLOSEWAIT, FINWAIT1, FINWAIT2, CLOSING, LASTACK}\}$$

$$\wedge receiver\_may\_accept(m,t) \wedge get\_seq\_nr(m) = get\_RCV\_NXT(t)$$

$$\wedge is\_ack(m) \wedge \neg may\_accept\_ack(m,t)\ \rhd\ \delta)$$

If the segment is acceptable, for which (`SEG_SEQ`=`RCV_NXT`), and for which both *is_ack* and *fin_flag_set* return false, it is processed as a data segment.

$$+ \sum_{m:Sgmt} (tcp\_call\_RECV(m) \cdot$$

$$TCP(s, t[RCV\_NXT \mapsto (RCV\_NXT + get\_num\_octs(m)) \bmod n,$$

$$receive\_buf \mapsto merge(receive\_buf, infl(get\_seq\_nr(m), get\_num\_octs(m)))$$

$$RCV\_ACK\_QUEUED \mapsto \mathtt{T}])$$

$$\lhd\ s \in \{\textit{ESTABLISHED, FINWAIT1, FINWAIT2}\}$$

$$\wedge receiver\_may\_accept(m, tcb) \wedge get\_seq\_nr(m) = get\_RCV\_NXT(t)$$

$$\wedge \neg is\_ack(m) \wedge \neg fin\_flag\_set(m)\ \rhd\ \delta)$$

$$+\ tcp\_call\_SND(construct\_ack(t)) \cdot TCP(s, t[RCV\_ACK\_QUEUED \mapsto \mathtt{F}])$$

$$\lhd\ get\_RCV\_ACK\_QUEUED(t) = \mathtt{T}\ \rhd\ \delta$$

Retransmission Timeout and Time-Wait Timeout

$$+\ tcp\_call\_SND(first(get\_rtq(t))) \cdot TCP(s,t)\ \lhd\ length(get\_rtq(t)) > 0\ \rhd\ \delta$$

$$+\ tcp\_TW\_TIMEOUT \cdot TCP(CLOSED, t)\ \lhd\ s = \textit{TIMEWAIT}\ \rhd\ \delta$$

The following summand is added as μCRL cannot cope with terminating processes:

$$+\ tcp\_idle \cdot TCP(s,t)\ \lhd\ s = \textit{CLOSED}\ \rhd\ \delta$$

B.3 The network layer

We specify a faulty network layer that may duplicate, reorder and lose data:

$$
\begin{aligned}
&NL(q{:}SgmtQ) = \\
&\quad \sum_{m:Sgmt} (nl\_rcv\_SEND(m) \cdot NL(add\_ordered(m,q)) \\
&\qquad \lhd\ length(q) < medium\_capacity\ \rhd\ \delta) \\
&+ \sum_{m:Sgmt} (nl\_rcv\_RECV(m) \cdot NL(remove(m,q)) \\
&\qquad \lhd\ get\_seq\_nr(m) < n \wedge m \in q\ \rhd\ \delta) \\
&+ \sum_{m:Sgmt} (nl\_rcv\_SEND(m) \cdot NL(q) \\
&\qquad \lhd\ length(q) < medium\_capacity\ \rhd\ \delta) \\
&+ \sum_{m:Sgmt} (nl\_rcv\_RECV(m) \cdot NL(q) \\
&\qquad \lhd\ get\_seq\_nr(m) < n \wedge m \in q\ \rhd\ \delta) \\
&+ medium\_drain \cdot NL(rest(q))\ \lhd\ length(q) = medium\_capacity\ \rhd\ \delta
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{SystemSpecification} = (\\
&\quad \rho_{\{al\_call\_SEND \to AL1\_call\_SEND, al\_call\_RECEIVE \to AL1\_call\_RECEIVE,}\\
&\qquad \rho_{al\_call\_CLOSE \to AL1\_call\_CLOSE\}}(\\
&\qquad AL(initial\_seq\_num\_TCP1, total\_octets\_to\_send\_TCP1)) \parallel\\
&\quad \rho_{\{tcp\_rcv\_SND \to TCP1\_rcv\_SEND, tcp\_rcv\_RECEIVE \to TCP1\_rcv\_RECEIVE,}\\
&\qquad \rho_{tcp\_call\_SND \to TCP1\_call\_SEND, tcp\_call\_RECV \to TCP1\_call\_RECV,}\\
&\qquad \rho_{tcp\_TW\_TIMEOUT \to TCP1\_TW\_TIMEOUT, tcp\_idle \to TCP1\_idle,}\\
&\qquad \rho_{tcp\_rcv\_CLOSE \to TCP1\_rcv\_CLOSE\}}(\\
&\qquad TCP(ESTABLISHED,\ tcb(\emptyset,\ \emptyset,\ initial\_seq\_num\_TCP1,\ initial\_seq\_num\_TCP1,\\
&\qquad initial\_window\_size\_TCP2,\ 0,\ initial\_seq\_num\_TCP2,\ initial\_seq\_num\_TCP1,\ initial\_seq\_num\_TCP1,\\
&\qquad window\_scale\_TCP2,\ \emptyset,\ initial\_seq\_num\_TCP2,\ initial\_window\_size\_TCP1,\ 0,\ initial\_seq\_num\_TCP2,\\
&\qquad window\_scale\_TCP1,\ initial\_seq\_num\_TCP2, \mathtt{F}))) \parallel\\
&\quad \rho_{\{nl\_rcv\_SEND \to NL1\_rcv\_SEND, nl\_rcv\_RECV \to NL1\_rcv\_RECV\}}(NL(\emptyset)) \parallel\\
&\quad \rho_{\{nl\_rcv\_SEND \to NL2\_rcv\_SEND, nl\_rcv\_RECV \to NL2\_rcv\_RECV\}}(NL(\emptyset)) \parallel\\
&\quad \rho_{\{tcp\_rcv\_SND \to TCP2\_rcv\_SEND, tcp\_rcv\_RECEIVE \to TCP2\_rcv\_RECEIVE,}\\
&\qquad \rho_{tcp\_call\_SND \to TCP2\_call\_SEND, tcp\_call\_RECV \to TCP2\_call\_RECV,}\\
&\qquad \rho_{tcp\_TW\_TIMEOUT \to TCP2\_TW\_TIMEOUT, tcp\_idle \to TCP2\_idle,}\\
&\qquad \rho_{tcp\_rcv\_CLOSE \to TCP2\_rcv\_CLOSE\}}(\\
&\qquad TCP(ESTABLISHED,\ tcb(\emptyset,\ \emptyset,\ initial\_seq\_num\_TCP2,\ initial\_seq\_num\_TCP2,\\
&\qquad initial\_window\_size\_TCP1,\ 0,\ initial\_seq\_num\_TCP1,\ initial\_seq\_num\_TCP2,\ initial\_seq\_num\_TCP2,\\
&\qquad window\_scale\_TCP1,\ \emptyset,\ initial\_seq\_num\_TCP1,\ initial\_window\_size\_TCP2,\ 0,\ initial\_seq\_num\_TCP1,\\
&\qquad window\_scale\_TCP2,\ initial\_seq\_num\_TCP1, \mathtt{F}))) \parallel\\
&\quad \rho_{\{al\_call\_SEND \to AL2\_call\_SEND, al\_call\_RECEIVE \to AL2\_call\_RECEIVE,}\\
&\qquad \rho_{al\_call\_CLOSE \to AL2\_call\_CLOSE\}}(\\
&\qquad AL(initial\_seq\_num\_TCP2, total\_octets\_to\_send\_TCP2))\\
&)
\end{aligned}
$$

Fig. 3: Our *SystemSpecification*, obtained by putting our components together