Safely Sliding Windows: The Specs

Thorsten Gerdsmeier, Dirk Henkel, Peter B. Ladkin
RVS Group Research Report RVS-RR-97-05b
Universität Bielefeld

1 July, 1997
last changed 19 November, 1997

Introduction

This paper introduces five TLA$^+$ specifications that have been written for the safety part of a verification proof for a sliding-window-style message retransmission mechanism[1] with linear arithmetic (SWLA). This safety proof is provided in [Hen97]. SWLA is the linear arithmetic version of the sliding window message retransmission mechanism with modular arithmetic (SWMA), which is part of the Radio Link Protocol (RLP1), the datalink layer protocol for TDMA mobile cellular radio being analysed by the FCAR project of the Telecommunications Software Group at INRS-Télécommunications in Montréal [Fer96]. These specifications and the accompanying proofs contribute to the verification of SWLA using TLA and TLA$^+$.

Documents on the Temporal Logic of Actions (TLA) calculus and the corresponding specification language TLA$^+$ can be found at the TLA website of Leslie Lamport [Lam]. A tutorial on TLA verification has been written by Peter Ladkin [Lad97].

TLA provides a clear separation between safety and liveness properties. The following specifications concern only safety properties. To obtain complete specifications we must add liveness properties. Liveness properties in TLA specifications have the form of Boolean combinations of fairness properties. The specifications have been designed, so to speak, with these liveness properties in mind, so we don't believe that adding them will change the general idea of how the five specified systems work. The only sure way to know, however, would be to add them and provide the relevant proofs.

System verification must be performed relative to stated requirements. For SWLA (as well as for RLP1) we require that the protocol implements a reliable, asynchronous first-in first-out message-transfer stream. We specify this by means of a variable *buffer* which contains the messages that have been *sent* but not *received*, in order. Reliability is ensured by the implicit criterion that sent messages do not disappear; asynchrony by the independence of *send* and *receive* actions (corresponding to the *buffer*'s *push* and *pop* actions); FIFO by

[1] It has been pointed out by Michael Ferguson that the meaning of the word "protocol" in telecommunications engeneering is far richer than its informal use by specification engineers and network textbook writers. We find the mathematical operation far better described as a "retransmission mechanism", as Ferguson suggests.

the standard FIFO property of a buffer. The *AbstractBuffer* (AB) is thus a state machine which embodies exactly the requirements of the communication system. The AB specification in TLA was first formulated and used in [Lad96].

For both SWLA and AB we give formal specifications of their safety properties: the TLA formulas $I.Safety$ and $V.Safety$. Proving that SWLA-safety implies AB-safety is equivalent to proving that $I.Safety$ logically implies $V.Safety$. One may try this directly, but it's easier to do it in several small steps via a hierarchy of specifications. We use three intermediate specifications $II.Safety$, $III.Safety$ and $IV.Safety$, which show a continuous development from the simplicity of $I.Safety$ to the details of $V.Safety$. The whole safety proof [Hen97] is accomplished in four steps of the form "$(N+1).Safety$ implies $N.Safety$".

The approach to the system verification differs from that of Mokkedem, Ferguson and Johnston [MFJ97]. They describe invariants, and prove using the TLA logic that these invariants logically follow from an implementation specified in TLA. We follow the "traditional" TLA format, in which all specifications, *including the requirement*, have the form of a state machine. In this example, the requirement state machine embodies, as we have argued, exactly the properties of a reliable, asynchronous FIFO channel. It is notable that the required invariant in [MFJ97] does not appear to be logically equivalent to a TLA specification as defined by Lamport [Lam94], thus it is not guaranteed that the TLA logic suffices to prove it (but Mokkedem, Ferguson and Johnston have indeed succeeded in proving it).

The five safety specifications are given in six TLA$^+$ modules. The first one is a collection of some common definitions. It is EXTEND'd by each of the five specification modules I to V. Module I, the "requirement", specifies the safety properties of AB, which is just a sequence with fixed maximum length and the two actions *Push* and *Pop*. As we said above, the purpose of any reliable, asynchronous FIFO message-transmission mechanism must be to simulate this precisely. Module II is a translation of AB into set notation. Module III introduces a distinction between sender and receiver sides. In Module IV the message transfer from sender to receiver is split into two seperate actions. Module V specifies the safety properties of SWLA. Besides *Push* and *Pop*, it defines four actions that make up the message transfer. Each module is preceeded by some commentary that should give an idea of how the specified system works.

Acknowledgements

The specifications were developed jointly with Prof. Dr. Jean-Charles Grégoire of INRS-Télécommunications à l'Université de Québec in Montréal. Petra Korff also contributed to the analysis and verification of some of the former high-level specifications with which we worked. Some of the explanatory commentary to the specifications is based on notes by Prof. Dr. Grégoire.

References

[Fer] Michael J. Ferguson: Selected publications. Available through `http://www.inrs-telecom.uquebec.ca/users/telesoft/`

[Fer96] Michael J. Ferguson: Formalization and validation of the Radio Link Protocol (RLP1). To be published in *Computer Networks and ISDN Systems*, 1996. Available through [Fer].

[Hen97] Dirk Henkel: Safely sliding windows. Technical Report RVS-RR-97-05a, RVS Group, Universität Bielefeld, Technische Fakultät, May 1997. Available through [Lad].

[Lad] Peter B. Ladkin: Abstracts of RVS research publications. Available through `http://www.rvs.uni-bielefeld.de`

[Lad96] Peter B. Ladkin: Formal but lively buffers in TLA$^+$. Technical Report RVS-RR-96-07, RVS Group, Universität Bielefeld, Technische Fakultät, January 1996. Available through [Lad].

[Lad97] Peter B. Ladkin: Using the Temporal Logic of Actions: A tutorial on TLA verification. Technical Report RVS-RR-97-08, RVS Group, Universität Bielefeld, Technische Fakultät, June 1997. Available through [Lad].

[Lam] Leslie Lamport: TLA – The Temporal Logic of Actions. `http://www.research.digital.com/SRC/tla/`

[Lam94] Leslie Lamport: The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872-923, May 1994. Also available through [Lam].

[MFJ97] Abdelillah Mokkedem, Michael J. Ferguson, and Robert deB. Johnston: A TLA solution to the specification and verification of the RLP1 retransmission protocol. *Formal Methods Europe* symposium, September 1997. Available through [Fer].

[Gre97] Jean-Charles Gregoire: TLA + PROMELA: conjecture, check, proof – engineering new protocols using methods and formal notations. *Formal Methods Europe* symposium, September 1997.

1 Common Definitions

Some declarations, assumptions and definitions will be needed in several of the following specifications. They are collected in the module *CommonDefinitions*.

Messages is the Set of all Objects that can be passed through the buffer we specify. It specifically does not contain the constant $\perp_{Messages}$. *BufferLength* is the maximum number of such objects that can be held in the buffer at any time. We require $Bufferlength \geq 1$.

We introduce a simple notation for intervals of natural numbers and some operators on sequences: $Len(s)$ denotes the length of a sequence s; $Tail(s)$ is "s without its first element"; $s \circ t$ the concatenation of the sequences s and t.

The last four definitions refer to sets of pairs of the form $(index, message)$ where *index* is a natural number. $MinPair(S)$ is that pair (or one of those pairs) with the least *index* in S, $MinIndex(S)$ is this least index. Similarly for *Max*-functions with the greatest index. These functions are defined only on nonempty sets.

module *CommonDefinitions*

EXTENDS *Naturals*

CONSTANTS $BufferLength, Messages, \perp_{Messages}$
ASSUMPTION $BufferLength > 0$
ASSUMPTION $\perp_{Messages} \notin Messages$

$$l \,..\, u \triangleq \{i \in Nat : l \leq i \land i \leq u\}$$

$$Len(s) \triangleq \text{CHOOSE } n : \land\ n \in Nat \\ \land\ \text{DOMAIN } s = 1 \,..\, n$$

$$Tail(s) \triangleq [i \in 1 \,..\, (Len(s) - 1) \mapsto s[i + 1]]$$

$$s \circ t \triangleq [i \in 1 \,..\, (Len(s) + Len(t)) \mapsto \textbf{if } i \leq Len(s) \textbf{ then } s[i] \textbf{ else } t[i - Len(s)]]$$

$$MinPair(S) \triangleq \text{CHOOSE } z : \land\ z \in S \\ \land\ \forall\, y \in S : z[1] \leq y[1]$$

$$MaxPair(S) \triangleq \text{CHOOSE } z : \land\ z \in S \\ \land\ \forall\, y \in S : z[1] \geq y[1]$$

$$MinIndex(S) \triangleq MinPair(S)[1]$$

$$MaxIndex(S) \triangleq MaxPair(S)[1]$$

2 An Abstract Buffer

Module I contains a straight-forward specification of an abstract buffer. Its only variable is the sequence *buffer*. At the beginning, *buffer* is empty. If the buffer is at any time not full (i.e. shorter than *BufferLength*), a message can be *Push*'d onto it at the rear end. If the buffer is not empty, its first message can be *Pop*'d off. Thus the messages are *Pop*'d in FIFO order.

module I

EXTENDS *CommonDefinitions*

VARIABLES *buffer*

$$Init \triangleq buffer = \langle\rangle$$

$$\begin{aligned} Push(a) \triangleq\ & \land\ a \in Messages \\ & \land\ Len(buffer) < BufferLength \\ & \land\ buffer' = buffer \circ \langle a \rangle \\ Pop \triangleq\ & \land\ buffer \neq \langle\rangle \\ & \land\ buffer' = Tail(buffer) \end{aligned}$$

$$\begin{aligned} SomeAction \triangleq\ & \lor \exists\, a : Push(a) \\ & \lor Pop \\ Safety \triangleq\ & \land\ Init \\ & \land\ \Box[SomeAction]_{buffer} \end{aligned}$$

3 Set Notation

Specification *II* is a translation of the buffer into set notation. Instead of the sequence *buffer* we now use the set *tSet*. Like *buffer* in specification *I*, *tSet* starts empty.

To preserve the FIFO property of the buffer, we must know in which order the messages in the buffer have been *Push*'d. Thus $Push(a)$ does not simply add the message a to *tSet* but a pair $(index, a)$ where *index* is the current number of the *Push* action. Accordingly, *Pop* takes the elements out of *tSet* in ascending order of *index*.

The current indices for the *Push* and *Pop* actions are stored in the counter variables $index_s$ and $index_r$. Both start at 0; every *Push* increments $index_s$; every *Pop* increments $index_r$. Thus $index_s$ is the index of the next element to be *Push*'d and $index_r$ is the index of the next element to be *Pop*'d.

The difference between $index_s$ and $index_r$ equals the size of *tSet*, thus it can be used to specify that the buffer must not be full at the beginning of a *Push*.

module *II*

EXTENDS *CommonDefinitions*

VARIABLES $tSet, index_r, index_s$

$$
\begin{array}{lcl}
Vars & \triangleq & \langle tSet, index_s, index_r \rangle \\
Init & \triangleq & \land\ tSet = \emptyset \\
 & & \land\ index_s = 0 \\
 & & \land\ index_r = 0
\end{array}
$$

$$
\begin{array}{lcl}
Push(a) & \triangleq & \land\ a \in Messages \\
 & & \land\ index_s - index_r < BufferLength \\
 & & \land\ tSet' = tSet \cup \{\langle index_s, a \rangle\} \\
 & & \land\ index'_s = index_s + 1 \\
 & & \land\ \text{UNCHANGED}\ index_r \\
Pop & \triangleq & \land\ \exists\, a : \land\ \langle index_r, a \rangle \in tSet \\
 & & \quad\ \land\ tSet' = tSet \setminus \{\langle index_r, a \rangle\} \\
 & & \land\ index'_r = index_r + 1 \\
 & & \land\ \text{UNCHANGED}\ index_s
\end{array}
$$

$$
\begin{array}{lcl}
SomeAction & \triangleq & \lor\ \exists\, a : Push(a) \\
 & & \lor\ Pop \\
Safety & \triangleq & \land\ Init \\
 & & \land\ \Box[SomeAction]_{Vars}
\end{array}
$$

4 Sender and Receiver

In this module, we split *tSet* into two different variables, *rSet* and *sSet*. We are slowly evolving towards a specification with distinct sender and receiver sides. At this stage though, we keep $index_r$ and $index_s$ as global variables. In particular, the sender action *Push* uses the information in the receiver variable $index_r$.

The buffer is now the (disjoint) union of *rSet* and *sSet*. Its actual size still equals the difference between $index_s$ and $index_r$.

The only changes to *Push* and *Pop* are that we now *Push* to *sSet* but *Pop* from *rSet*. A new action *Transfer* is added to the model, to represent the transfer of information from *rSet* to *sSet*. This action picks non-deterministically a pair from *sSet* and moves it to *rSet*. Indexing and $index_r$ become critical as pairs are *Transfer*'d in non-deterministic order, but must be *Pop*'d in FIFO order.

module *III*

EXTENDS *CommonDefinitions*

VARIABLES $sSet, rSet, index_s, index_r$

$$
\begin{array}{lcl}
Vars & \triangleq & \langle sSet, rSet, index_s, index_r \rangle \\
Init & \triangleq & \land\ index_r = 0 \\
 & & \land\ index_s = 0 \\
 & & \land\ sSet = \emptyset \\
 & & \land\ rSet = \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
Push(a) & \triangleq & \land\ a \in Messages \\
 & & \land\ index_s - index_r < BufferLength \\
 & & \land\ sSet' = sSet \cup \{\langle index_s, a \rangle\} \\
 & & \land\ index'_s = index_s + 1 \\
 & & \land\ \text{UNCHANGED}\ \langle rSet, index_r \rangle \\
Pop & \triangleq & \land\ \exists\, a : \land\ \langle index_r, a \rangle \in rSet \\
 & & \qquad \land\ rSet' = rSet \setminus \{\langle index_r, a \rangle\} \\
 & & \land\ index'_r = index_r + 1 \\
 & & \land\ \text{UNCHANGED}\ \langle sSet, index_s \rangle \\
Transfer & \triangleq & \land\ \exists\, p : \land\ p \in sSet \\
 & & \qquad \land\ sSet' = sSet \setminus \{p\} \\
 & & \qquad \land\ rSet' = rSet \cup \{p\} \\
 & & \land\ \text{UNCHANGED}\ \langle index_s, index_r \rangle
\end{array}
$$

$$
\begin{array}{lcl}
SomeAction & \triangleq & \lor\ \exists\, a : Push(a) \\
 & & \lor\ Pop \\
 & & \lor\ Transfer \\
Safety & \triangleq & \land\ Init \\
 & & \land\ \Box[SomeAction]_{Vars}
\end{array}
$$

5 Copy and Purge

Specification *IV* introduces a first degree of asynchronism between both sides of communication. An index-message pair is no longer *Transfer*'d but rather *Copy*'d from *sSet* to *rSet* and later *Purge*'d from *sSet*, when it has been *Pop*'d for certain. As a consequence *rSet* is not disjoint from, but rather a subset of *sSet*. This asynchronism is a first step towards handling loss of data between sender and receiver: the sender became careful, she keeps a sent message until she knows it has been *Pop*'d.

In the Specifications *II* and *III*, the difference $index_s - index_r$ was controlled to ensure that the buffer didn't exceed the maximum length. Here, we use the term $index_s - MinIndex(sSet)$ instead, since $MinIndex(sSet)$ can linger below $index_r$. The size control is now completely local to the sender side.

This is the only change to *Push*. *Pop* has not changed at all. In contrast to *III.Transfer*, *Copy* allows multiple copying of the same pair to *rSet*. Since we are dealing with sets, such multiple copies are idempotent. *Purge* can handle several messages at once: it non-deterministically picks an already-*Pop*'d message in *sSet* and deletes this and all older messages.

The sender and the receiver side are not yet completely separated. *Copy* is a common action using both *sSet* and *rSet*. *Purge* could be regarded as a sender action, but it still uses the receiver information in $index_r$, thus $index_r$ remains a global variable.

module *IV*

EXTENDS *CommonDefinitions*

VARIABLES $sSet, rSet, index_r, index_s$

$$
\begin{array}{lcl}
Vars & \triangleq & \langle sSet, rSet, index_r, index_s \rangle \\
Init & \triangleq & \land\ index_r = 0 \\
 & & \land\ index_s = 0 \\
 & & \land\ sSet = \emptyset \\
 & & \land\ rSet = \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
Push(a) & \triangleq & \land\ a \in Messages \\
 & & \land\ \lor\ sSet = \emptyset \\
 & & \quad \lor\ index_s - MinIndex(sSet) < BufferLength \\
 & & \land\ sSet' = sSet \cup \{\langle index_s, a \rangle\} \\
 & & \land\ index'_s = index_s + 1 \\
 & & \land\ \text{UNCHANGED}\ \langle rSet, index_r \rangle \\
Pop & \triangleq & \land\ \exists\, a : \land\ \langle index_r, a \rangle \in rSet \\
 & & \quad \land\ rSet' = rSet \setminus \{\langle index_r, a \rangle\} \\
 & & \land\ index'_r = index_r + 1 \\
 & & \land\ \text{UNCHANGED}\ \langle sSet, index_s \rangle \\
Copy & \triangleq & \land\ \exists\, p : \land\ p \in sSet \\
 & & \quad \land\ p[1] \geq index_r \\
 & & \quad \land\ rSet' = rSet \cup \{p\} \\
 & & \land\ \text{UNCHANGED}\ \langle sSet, index_s, index_r \rangle
\end{array}
$$

$$
\begin{array}{lcl}
Purge & \triangleq & \land\ \exists\, p \in sSet\ :\ \land\ p[1] < index_r \\
 & & \qquad\qquad\ \land\ sSet' = sSet \setminus \{pair \in sSet\ :\ pair[1] \leq p[1]\} \\
 & & \land\ \text{UNCHANGED}\ \langle rSet, index_r, index_s \rangle
\end{array}
$$

$$
\begin{array}{lcl}
SomeAction & \triangleq & \lor\ \exists\, a\ :\ Push(a) \\
 & & \lor\ Pop \\
 & & \lor\ Copy \\
 & & \lor\ Purge \\
Safety & \triangleq & \land\ Init \\
 & & \land\ \Box[SomeAction]_{Vars}
\end{array}
$$

6 Sliding Window with Linear Arithmetic

Module V specifies the sliding-window-style message retransmission mechanism with linear arithmetic. We introduce full decoupling between sender and receiver by modeling communications as two one-way communication channels: a forward channel *fBuffer* for the messages and a backward channel *bBuffer* for acknowledgements. These are the only common variables of the sender and the receiver sides.

The communication channels are handled as if they might lose data: sender and receiver can send messages rsp. acknowledgements repeatedly; the sender keeps the sent messages until they have been acknowledged, and the receiver acknowledges only what he has already *Pop*'d.

Rather than *Copy*ing a pair from one side to the other, the pair is put into *fBuffer* with *SendMsg*, and read from there by *ReadMsg*. *fBuffer* contains either one index-message pair or the zero element $\langle 0, \perp_{Messages} \rangle$ which indicates "no message". *SendMsg* copies an arbitrary message from *sSet* into *fBuffer*, no matter if this overwrites an unread earlier message. Reading a message deletes it from *fBuffer* and puts it into *rSet*.

SendAck and *ReadAck* replace *IV.Purge*: *SendAck* copies $index_r$ into *bBuffer*. Thus *bBuffer* contains the index of the oldest message, that might not yet be *Pop*'d. It can be understood as meaning "precisely those messages older than this index are guaranteed to be *Pop*'d". *ReadAck* removes all messages from *sSet* with an index older (i.e. smaller) than that in *bBuffer*. In contrast to *fBuffer*, *bBuffer* is never reset to 0.

module V

EXTENDS *CommonDefinitions*

VARIABLES $sSet, rSet, index_r, index_s, fBuffer, bBuffer$

$$
\begin{array}{lcl}
Vars & \triangleq & \langle sSet, rSet, index_r, index_s, fBuffer, bBuffer \rangle \\
Init & \triangleq & \land\ index_r = 0 \\
 & & \land\ index_s = 0 \\
 & & \land\ sSet = \emptyset \\
 & & \land\ rSet = \emptyset \\
 & & \land\ fBuffer = \langle 0, \perp_{Messages} \rangle \\
 & & \land\ bBuffer = 0
\end{array}
$$

$$
\begin{array}{lcl}
Push(a) & \triangleq & \land\ a \in Messages \\
 & & \land\ (sSet = \emptyset) \lor (index_s - MinIndex(sSet) < BufferLength) \\
 & & \land\ sSet' = sSet \cup \{\langle index_s, a \rangle\} \\
 & & \land\ index'_s = index_s + 1 \\
 & & \land\ \text{UNCHANGED}\ \langle rSet, index_r, fBuffer, bBuffer \rangle \\
Pop & \triangleq & \land\ \exists\, a : \land\ \langle index_r, a \rangle \in rSet \\
 & & \qquad\ \land\ rSet' = rSet \setminus \{\langle index_r, a \rangle\} \\
 & & \land\ index'_r = index_r + 1 \\
 & & \land\ \text{UNCHANGED}\ \langle sSet, index_s, fBuffer, bBuffer \rangle
\end{array}
$$

$$
\begin{array}{lcl}
SendMsg & \triangleq & \land\ \exists\, p : \land\ p \in sSet \\
 & & \qquad\land\ fBuffer' = p \\
 & & \land\ \text{UNCHANGED}\ \langle index_s, index_r, sSet, rSet, bBuffer\rangle \\
ReadMsg & \triangleq & \land\ fBuffer \neq \langle 0, \bot_{Messages}\rangle \\
 & & \land\ fBuffer[1] < index_r + BufferLength \\
 & & \land\ fBuffer[1] \geq index_r \\
 & & \land\ rSet' = rSet \cup \{fBuffer\} \\
 & & \land\ fBuffer' = \langle 0, \bot_{Messages}\rangle \\
 & & \land\ \text{UNCHANGED}\ \langle index_s, index_r, sSet, bBuffer\rangle \\
SendAck & \triangleq & \land\ bBuffer' = index_r \\
 & & \land\ \text{UNCHANGED}\ \langle index_s, index_r, sSet, rSet, fBuffer\rangle \\
ReadAck & \triangleq & \land\ sSet' = sSet \setminus \{pair \in sSet : pair[1] < bBuffer\} \\
 & & \land\ \text{UNCHANGED}\ \langle index_s, index_r, rSet, fBuffer, bBuffer\rangle
\end{array}
$$

$$
\begin{array}{lcl}
SomeAction & \triangleq & \lor\ \exists\, a : Push(a) \\
 & & \lor\ Pop \\
 & & \lor\ SendMsg \\
 & & \lor\ ReadMsg \\
 & & \lor\ SendAck \\
 & & \lor\ ReadAck \\
Safety & \triangleq & \land\ Init \\
 & & \land\ \Box[SomeAction]_{Vars}
\end{array}
$$