Verification of a sliding window protocol in µCRL

Citation for published version (APA):

Document status and date:
Published: 01/01/2004

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 15. Nov. 2023
Verification of a Sliding Window Protocol in $\mu$CRL

Bahareh Badban $^1$, Wan Fokkink $^{1,2}$, Jan Friso Groote $^{1,3}$, Jun Pang $^1$, and Jaco van de Pol $^1$
$^1$ CWI, Department of Software Engineering
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
email: {badban, wan, pangjun, vdpol}@cwi.nl
$^2$ Vrije Universiteit Amsterdam, Department of Theoretical Computer Science
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
email: wanf@cs.vu.nl
$^3$ Eindhoven University of Technology, Department of Computer Science
PO Box 513, 5600 MB Eindhoven, The Netherlands
email: jfg@win.tue.nl

Abstract—We prove the correctness of a sliding window protocol with an arbitrary finite window size $n$ and sequence numbers modulo $2^n$. The correctness consists of showing that the sliding window protocol is branching bisimilar to a queue of capacity $2^n$. The proof is given entirely on the basis of an axiomatic theory, and has been checked in the theorem prover PVS.

Keywords—$\mu$CRL, branching bisimulation, process algebra, sliding window protocols, specification, verification techniques

* This research is partly supported by the Dutch Technology Foundation STW under the project CES5008: Improving the quality of embedded systems using formal design and systematic testing.

I. Introduction

Sliding window protocols [7] (SWPs) ensure successful transmission of messages from a sender to a receiver through a medium, in which messages may get lost. Their main characteristic is that the sender does not wait for an incoming acknowledgment before sending next messages, for optimal use of bandwidth. This is the reason why many data communication systems include the SWP, in one of its many variations.

In SWPs, both the sender and the receiver maintain a buffer. In practice the buffer at the receiver is often much smaller than at the sender, but here we make the simplifying assumption that both buffers can contain up to $n$ messages. By providing the messages with sequence numbers, reliable in-order delivery without duplications is guaranteed. The sequence numbers can be taken modulo $2^n$ (and not less, see [42] for a nice argument). The messages at the sender are numbered from $i$ to $i + n$ (modulo $2^n$); this is called a window. When an acknowledgment reaches the sender, indicating that $k$ messages have arrived correctly, the window slides forward, so that the sending buffer can contain messages with sequence numbers $i + k$ to $i + k + n$ (modulo $2^n$). The window of the receiver slides forward when the first element in this window is passed on to the environment.

Within the process algebraic community, SWPs have attracted much attention, because their precise formal verification turned out to be surprisingly difficult. We provide a comparison with verifications of SWPs from the literature in Section VIII, and restrict here to the context in which this paper was written. After the advent of process algebra in the early eighties of last century, it was observed that simple protocols, such as the alternating bit protocol, could readily be verified. In an attempt to show that more difficult protocols could also be dealt with, SWPs were considered. Middeldorp [31] and Brunkreef [5] gave specifications in ACP [1] and PSF [30], respectively. Vaandrager [43], Groenveld [12], van Wamel [44] and Bezem and Groote [3] manually verified one-bit SWPs, in which the size of the sending and receiving window is one.

Starting in 1990, we attempted to prove the most complex SWP from [42] (not taking into account additional features such as duplex message passing and piggybacking) correct using $\mu$CRL [16], which is a suitable process algebraic formalism for such purposes. This turned out to be unexpectedly hard, and has led to the development of new proof methods for protocol verification. We therefore consider the current paper as a true milestone in process algebraic verification.

Our first observation was that the external behavior of the protocol, as given in [42], was unclear. We adapted the SWP such that it nicely behaves as a queue of capacity $2^n$. The second observation was that the SWP of [42] contained a deadlock [13, Stelling...
which could only occur after at least \( n \) messages were transmitted. This error was communicated to Tanenbaum, and has been repaired in more recent editions of [42]. Another bug in the \( \mu \text{CRL} \) specification of the SWP was detected by means of a model checking analysis. A first attempt to prove the resulting SWP correct led to the verification of a bakery protocol [14], and to the development of the \textit{cones and foci} proof method [19], [9]. This method plays an essential role in the proof in the current paper, and has been used to prove many other protocols and distributed algorithms correct. But the correctness proof required an additional idea, already put forward by Schoone [37], to first perform the proof with unbounded sequence numbers, and to separately eliminate modulo arithmetic.

We present a specification in \( \mu \text{CRL} \) of a SWP with buffer size \( 2n \) and window size \( n \), for arbitrary \( n \). The medium between the sender and the receiver is modeled as a lossy queue of unbounded capacity. We manually prove that the external behavior of this protocol is branching bisimilar [10] to a FIFO queue of capacity \( 2n \). This proof is entirely based on the axiomatic theory underlying \( \mu \text{CRL} \) and the axioms characterizing the data types. It implies both safety and liveness of the protocol (the latter under the assumption of fairness). First, we linearize the specification, meaning that we get rid of parallel operators. Moreover, communication actions are stripped from their data parameters. Then we eliminate modulo arithmetic, using the proof principle CL-RSP [4]. Finally, we apply the cones and foci technique, to prove that the linear specification without modulo arithmetic is branching bisimilar to a FIFO queue of capacity \( 2n \). All lemmas for the data types, all invariants and all correctness proofs have been checked using PVS. The PVS files are available via \texttt{http://www.cwi.nl/~pangjun/swp/}.

A concise overview of other verifications of SWPs is presented in Section VIII. Many of these verifications deal with either unbounded sequence numbers, in which case the intricacies of modulo arithmetic disappear, or a fixed finite window size. The papers that do treat arbitrary finite window sizes in most cases restrict to safety properties.

This paper is set up as follows. Section II introduces the process part of \( \mu \text{CRL} \). In Section III, the data types needed for specifying the SWP and its external behavior are presented. Section IV features the \( \mu \text{CRL} \) specifications of the SWP and its external behavior. In Section V, three consecutive transformations are applied to the specification of the SWP, to linearize the specification, eliminate arguments of communication actions, and get rid of modulo arithmetic. In Section VI, properties of the data types and invariants of the transformed specification are proved. In Section VII, it is proved that the three transformations preserve branching bisimulation, and that the transformed specification behaves like a FIFO queue. Finally, Section VIII gives an overview of related work on verifying SWPs. The verification contained in this paper has been extended to verify a SWP with piggybacking.

II. \( \mu \text{CRL} \)

\( \mu \text{CRL} \) [16] (see also [18]) is a language for specifying distributed systems and protocols in an algebraic style. It is based on the process algebra \( \text{ACP} \) [1] extended with equational abstract data types [28]. In a \( \mu \text{CRL} \) specification, one part specifies the data types by means of equations \( d = e \), while a second part specifies the process behavior. We assume the data sort of booleans \( \text{Bool} \) with constants \( t \) and \( f \), and the usual connectives \( \land, \lor, \neg \), \( \implies \) and \( \iff \). For a boolean \( b \), we abbreviate \( b = t \) to \( b = f \) to \( \neg b \).

The data types needed for our \( \mu \text{CRL} \) specification of a SWP are presented in Section III. In this section we focus on the process part of \( \mu \text{CRL} \). Processes are represented by process terms, which describe the order in which the actions from a set \( A \) may happen. A process term consists of action names and recursion variables combined by process algebraic operators. Actions and recursion variables may carry data parameters. There are two predefined actions outside \( A \): \( \delta \) represents deadlock, and \( \tau \) a hidden action. These two actions never carry data parameters. \( p;q \) denotes sequential composition and \( p + q \) non-deterministic choice. Summation \( \sum_{d:D}p(d) \) provides the possibly infinite choice over a data type \( D \), and the conditional construct \( p < b > q \) with \( b \) a data term of sort \( \text{Bool} \) behaves as \( p \) if \( b \) and as \( q \) if \( \neg b \). Parallel composition \( p || q \) interleaves the actions of \( p \) and \( q \); moreover, actions from \( p \) and \( q \) may also synchronize to a communication action, when this is explicitly allowed by a predefined communication function. Two actions can only synchronize if their data parameters are equal. Encapsulation \( \partial_{H}(p) \), which renames all occurrences in \( p \) of actions from the set \( H \) into \( \delta \), can be used to force actions into communication. Hiding \( \tau_{\pi}(p) \) renames all occurrences in \( p \) of actions from the set \( \mathcal{I} \) into \( \tau \). Finally, processes can be specified by
means of recursive equations

\[ X(d_1; D_1, \ldots, d_n; D_n) \approx p \]

where \( X \) is a recursion variable, \( d_i \) a data parameter of type \( D_i \) for \( i = 1, \ldots, n \), and \( p \) a process term (possibly containing recursion variables and the parameters \( d_i \)). A recursive specification is linear, called a linear process equation (LPE), if it is of the form

\[ X(d_1; D_1, \ldots, d_n; D_n) \approx \sum_{i=1}^{\ell} \sum_{e \in \mathcal{Z}_i} a_i(e_1^i, \ldots, e_m^i) \cdot X(d_1^i, \ldots, d_n^i) \prec b \triangleright \delta. \]

To each \( \mu \text{CRL} \) specification belongs a directed graph, called a labeled transition system (LTS), which is defined by the structural operational semantics of \( \mu \text{CRL} \) (see [16]). In this labeled transition system, the states are process terms, and the edges are labeled with parameterized actions. Branching bisimulation \( \rightsquigarrow_0 \) [10] and strong bisimulation \( \rightsquigarrow \) [33] are two well-established equivalence relations on the states in labeled transition systems. Conveniently, strong bisimulation equivalence implies branching bisimulation equivalence. The proof theory of \( \mu \text{CRL} \) from [15] is sound modulo branching bisimulation equivalence, meaning that if \( p \approx q \) can be derived from it then \( p \rightsquigarrow q \).

**Definition II.1 (Branching bisimulation)** Assume an LTS. A branching bisimulation relation \( B \) is a symmetric binary relation on states such that if \( s \mathrel{B} t \) and \( s \xRightarrow{\ell} s' \), then
- either \( \ell = \tau \) and \( s' \mathrel{B} t \);
- or there is a sequence of (zero or more) \( \tau \)-transitions \( t \xRightarrow{\tau} \cdots \xRightarrow{\tau} t_0 \) such that \( s \mathrel{B} t_0 \) and \( t_0 \xrightarrow{\ell} t' \) with \( s' \mathrel{B} t' \). Two states \( s \) and \( t \) are branching bisimilar, denoted by \( s \equiv_B t \), if there is a branching bisimulation relation \( B \) such that \( s \mathrel{B} t \).

The goal of this paper is to prove that the initial state of the forthcoming \( \mu \text{CRL} \) specification of a SWP is branching bisimilar to a FIFO queue. In the proof of this fact, we will use three proof principles from the literature to derive that two \( \mu \text{CRL} \) specifications are branching (or even strongly) bisimilar: sum elimination, CL-RSP, and cones and foci.

- **Sum elimination** [14] states that a summation over a data type from which only one element can be selected can be removed. To be more precise,

\[
\sum_{d \in D} p(d) \triangleq d = e \wedge b \triangleright \delta \quad \rightsquigarrow \quad p(e) \triangleq b \triangleright \delta.
\]

- **CL-RSP** [4] states that the solutions of a linear \( \mu \text{CRL} \) specification that does not contain any infinite \( \tau \) sequence are all strongly bisimilar. This proof principle basically extends RSP [2] to a setting with data. The reader is referred to [4] for more details regarding CL-RSP.

- The cones and foci method from [9], [19] rephrases the question whether two linear \( \mu \text{CRL} \) specifications \( \tau_T(S_1) \) and \( S_2 \) are branching bisimilar, where \( S_2 \) does not contain actions from some set \( I \) of internal actions, in terms of data equalities. A state mapping \( \phi \) relates each state in \( S_1 \) to a state in \( S_2 \). Furthermore, some states in \( S_1 \) are declared to be focus points, by means of a predicate FC. The cone of a focus point consists of the states in \( S_1 \) that can reach this focus point by a string of actions from \( I \). It is required that each reachable state in \( S_1 \) is in the cone of a focus point. If a number of matching criteria are satisfied, then \( \tau_T(S_1) \) and \( S_2 \) are branching bisimilar. We give the definition of matching criteria and the general theorem as follows. The reader is referred to [9] for the technical details of the cones and foci technique.

Let \( Act \) be a set of actions. Assume an LPE \( X \)

\[
\begin{align*}
X(d; D) &= \sum_{a \in Act \cup \{\tau\}} \sum_{e : E_a} a(f_a(d, e)) \cdot X(g_a(d, e)) \\
&\prec h_a(d, e) \triangleright \delta.
\end{align*}
\]

Furthermore, assume an LPE \( Y \) without hidden actions

\[
\begin{align*}
Y(d'; D') &= \sum_{a \in Act} \sum_{e : E_a} a(f'_a(d', e)) \cdot Y(g'_a(d', e)) \\
&\prec h'_a(d', e) \triangleright \delta.
\end{align*}
\]

**Definition II.2 (Matching criteria)** A state mapping \( \phi : D \rightarrow D' \) satisfies the matching criteria for \( d : D \) if for all \( a \in Act \):

1. \( \forall e : E_a(h_\tau(d, e) \Rightarrow \phi(d) = \phi(g_\tau(d, e))) \);
2. \( \forall e : E_a(h_a(d, e) \Rightarrow h'_a(\phi(d), e)) \);
3. \( FC(d) \Rightarrow \forall e : E_a(h'_a(\phi(d), e) \Rightarrow h_a(d, e)) \);
4. \( \forall e : E_a(h_a(d, e) \Rightarrow f_a(d, e) = f'_a(\phi(d), e)) \);
5. \( \forall e : E_a(h_a(d, e) \Rightarrow \phi(g_a(d, e)) = g'_a(\phi(d), e)) \).

Matching criterion I requires that the \( \tau \)-transitions at \( d \) are inert, meaning that \( d \) and \( g_\tau(d, e) \) are branching bisimilar. Criteria II, IV and V express that each external transition of \( d \) can be simulated by \( \phi(d) \). Finally, criterion III expresses that if \( d \) is a focus point, then each external transition of \( \phi(d) \) can be simulated by \( d \).
Theorem II.3 Assume LPEs $X(d;D)$ and $Y(d';D')$ written as before. Let $I : D \rightarrow \text{Bool}$ be an invariant for $X$. Suppose that for all $d:D$ with $I(d)$, 1. $\phi : D \rightarrow D'$ satisfies the matching criteria for $d$, and 2. there is a $\hat{d}:D$ such that $FC(\hat{d})$ and $d \xrightarrow{=} \ldots \xrightarrow{=} \hat{d}$ in the LTS for $X$.

Then for all $d:D$ with $I(d)$,  
\[X(d) \xrightarrow{=} Y(\phi(d)).\]

III. DATA TYPES

In this section, the data types used in the µCRL specification of the SWP are presented: booleans, natural numbers supplied with modulo arithmetic, and buffers. Furthermore, basic properties are given for the operations defined on these data types.

A. Booleans

We introduce the data type $\text{Bool}$ of booleans.

\[
t, f \rightarrow \text{Bool} \\
\land, \lor : \text{Bool} \times \text{Bool} \rightarrow \text{Bool} \\
\neg : \text{Bool} \rightarrow \text{Bool} \\
\Rightarrow, \Leftrightarrow : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}
\]

t and $f$ denote true and false, respectively. The infix operations $\land$ and $\lor$ represent conjunction and disjunction, respectively. Finally, $\neg$ denotes negation. The defining equations are:

\[
b \land t = b \\
b \land f = f \\
b \lor t = t \\
b \lor f = b \\
\neg t = f \\
\neg f = t \\
b \Rightarrow b' = b' \lor \neg b \\
b \Leftrightarrow b' = (b \Rightarrow b') \land (b' \Rightarrow b)
\]

B. If-then-else and Equality

For each data type $D$ in this paper we assume the presence of an operation

\[if : \text{Bool} \times D \times D \rightarrow D\]

with defining equations

\[
if(t, d, e) = d \\
if(f, d, e) = e
\]

Furthermore, for each data type $D$ in this paper one can easily define a mapping $eq : D \times D \rightarrow \text{Bool}$ such that $eq(d, e)$ holds if and only if $d = e$ can be derived.

For notational convenience we take the liberty to write $d = e$ instead of $eq(d, e)$.

C. Natural Numbers

We introduce the data type $\text{Nat}$ of natural numbers.

\[
0 : \Rightarrow \text{Nat} \\
S : \text{Nat} \rightarrow \text{Nat} \\
+, -, \cdot : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \\
\leq, <, \geq, > : \text{Nat} \times \text{Nat} \rightarrow \text{Bool}
\]

$0$ denotes zero and $S(n)$ the successor of $n$. The infix operations $+$, $-$ and $\cdot$ represent addition, monus (also called proper subtraction) and multiplication, respectively. Finally, the infix operations $\leq$, $<$, $\geq$ and $>$ are the less-than-(or-equal) and greater-than-(or-equal) operations. Usually, the sign for multiplication is omitted, and $-(i = j)$ is abbreviated to $i \neq j$.

\[
i + 0 = i \\
i + S(j) = S(i + j) \\
i - 0 = i \\
0 - i = 0 \\
S(i) - S(j) = i - j \\
i - 0 = 0 \\
i - S(j) = (i - j) + i \\
0 \leq i = t \\
S(i) - S(j) = i - j \\
0 < S(i) = t \\
i < 0 = f \\
S(i) - S(j) = i - j \\
i \geq j = -(j < i) \\
i > j = -(j \leq i)
\]

We take as binding convention: $\{\_, \neq\} \times \{\_\} \times \{+, -, \cdot\} \times \{\leq, <, \geq, >\} \times \{\Rightarrow, \Leftrightarrow\}$.

D. Modulo Arithmetic

Since the size of the buffers at the sender and the receiver in the sliding window are of size $2n$, calculations modulo $2n$ play an important role. We introduce the following notation for modulo calculations:

\[
| : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \\
div : \text{Nat} \times \text{Nat} \rightarrow \text{Nat}
\]

$i|n$ denotes $i$ modulo $n$, while $i \div n$ denotes $i$ integer divided by $n$. The modulo operations are defined by the following equations (for $n > 0$):

\[
i|n = if(i < n, i, (i \div n)|n) \\
i \div n = if(i < n, 0, S((i \div n) \div n))
\]
E. Buffers

The sender and the receiver in the SWP both maintain a buffer containing the sending and the receiving window, respectively (outside these windows both buffers are empty). Let \( \Delta \) be the set of data elements that can be communicated between sender and receiver. The buffers are modeled as a list of pairs \((d, i)\) with \(d: \Delta \) and \(i: \text{Nat}\), representing that position (or sequence number) \(i\) of the buffer is occupied by datum \(d\). The data type \(\text{Buf}\) is specified as follows, where \([\ ]\) denotes the empty buffer:

\[
\begin{align*}
[] & : \rightarrow \text{Buf} \\
\text{inb} : \Delta \times \text{Nat} \times \text{Buf} & \rightarrow \text{Buf}
\end{align*}
\]

\(q_n\) denotes buffer \(q\) with all sequence numbers taken modulo \(n\).

\[
\begin{align*}
[\ ]_n & = [] \\
\text{inb}(d, i, q)_n & = \text{inb}(d, i|n, q|n)
\end{align*}
\]

\(\text{test}(i, q)\) produces \(t\) if and only if position \(i\) in \(q\) is occupied, \(\text{retrieve}(i, q)\) produces the datum that resides at position \(i\) in buffer \(q\) (if this position is occupied),\(^1\) and \(\text{remove}(i, q)\) is obtained by emptying position \(i\) in buffer \(q\).

\[
\begin{align*}
\text{test}(i, []) & = f \\
\text{test}(i, \text{inb}(d, j, q)) & = i = j \lor \text{test}(i, q) \\
\text{retrieve}(i, \text{inb}(d, j, q)) & = \text{if}(i = j \land \text{retrieve}(i, q)) \\
\text{remove}(i, []) & = [] \\
\text{remove}(i, \text{inb}(d, j, q)) & = \text{if}(i = j, \text{remove}(i, q), \text{inb}(d, j, \text{remove}(i, q)))
\end{align*}
\]

\(\text{release}(i, j, q)\) is obtained by emptying positions \(i\) up to \(j\) in \(q\). \(\text{release}_n(i, j, q)\) does the same modulo \(n\).

\[
\begin{align*}
\text{release}(i, j, q) & = \text{if}(i \geq j, q, \text{release}(S(i), j, \text{remove}(i, q))) \\
\text{release}_n(i, j, q) & = \text{if}(i|n = j|n, q, \text{release}_n(S(i), j, \text{remove}(i, q)))
\end{align*}
\]

\(\text{next-empty}(i, q)\) produces the first empty position in \(q\), counting upwards from sequence number \(i\) onward.

\(^1\)Note that \(\text{retrieve}(i, [])\) is undefined. One could choose to equate it to a default value in \(\Delta\), or to a fresh error element in \(\Delta\). However, the first approach could cover up flaws in the \(\mu\text{CRL}\) specification of the SWP, and the second approach would needlessly complicate the data type \(\Delta\). We prefer to work with a partially defined version of \(\text{retrieve}\), which is allowed in \(\mu\text{CRL}\). All operations in \(\mu\text{CRL}\) models, however, are total; partially specified operations just lead to the existence of multiple models.

\(\text{next-empty}_n(i, q)\) does the same modulo \(n\).

\[
\begin{align*}
\text{next-empty}(i, q) & = \text{if}(\text{test}(i, q), \text{next-empty}(S(i), q), i) \\
\text{next-empty}_n(i, q) & = \text{if}(\text{next-empty}_n(i|n, q|n) < n, \text{next-empty}_n(i|n, q|n), \text{next-empty}_n(0, q|n))
\end{align*}
\]

Intuitively, \(\text{in-window}(i, j, k)\) produces \(t\) if and only if \(j\) lies in the range from \(i\) to \(k - 1\), modulo \(n\), where \(n\) is greater than \(i, j\) and \(k\).

\[
\text{in-window}(i, j, k) = i \leq j < k \lor k < i \leq j \lor j < k < i
\]

Finally, we define an operation on buffers that is only needed in the derivation of some data equalities in Section VI-A: \(\text{max}(q)\) produces the greatest sequence number that is occupied in \(q\).

\[
\begin{align*}
\text{max}([\ ]) & = 0 \\
\text{max}([\text{inb}(d, i, q)]) & = \text{if}(i \geq \text{max}(q), i, \text{max}(q))
\end{align*}
\]

F. Mediums

The medium in the SWP between the sender and the receiver is modeled as a lossy channel of unbounded capacity with FIFO behavior. We model the medium containing frames from the sender to the receiver by a data type \(\text{MedK}\). It represents a list of pairs \((d, i)\) with a datum \(d: \Delta\) and its sequence number \(i: \text{Nat}\). \([\ ][\ ]^n\) denote an empty medium. \(g_n\) denotes medium \(g\) with all sequence numbers taken modulo \(n\). \(\text{member}(d, i, g)\) produces \(t\) if and only if the pair \((d, i)\) is in \(g\). \(\text{length}(g)\) denotes the length of \(g\). \(\text{return-dat}(i, g)\) and \(\text{return-seq}(i, g)\) produce the datum and the sequence number, respectively, that resides at position \(i\) in \(g\) (positions are counted from \(0\)). For convenience, we use \(\text{last-dat}(g)\) and \(\text{last-seq}(g)\) to produce the datum and the sequence number, respectively, that resides at the end of \(g\). \(\text{delete}(i, g)\) is obtained by emptying position \(i\) in \(g\). Similarly, \(\text{delete-last}(g)\) is obtained by emptying the last position in \(g\).

The medium containing the sequence numbers from the receiver to the sender by a data type \(\text{MedL}\). Similarly, we have the following defining equations.

G. Lists

We introduce the data type of \(\text{List}\) of lists, which are used in the specification of the desired external behavior of the SWP: a FIFO queue of size \(2n\). Let \([\ ]\) denote the empty list.

\[
[\ ] : \rightarrow \text{List} \\
\text{inl} : \Delta \times \text{List} \rightarrow \text{List}
\]
Furthermore, \( q[i..j] \) is the list containing the elements in buffer \( q \) at positions \( i \) up to but not including \( j \).

\[
q[i..j] = \text{if} \,(i \geq j, \emptyset, \text{inl} \,(\text{retrieve} \,(i, q), q \,[S(i)\ldots j]))
\]

IV. Sliding Window Protocol

In this section, a \( \mu \text{CRL} \) specification of a SWP is presented, together with its desired external behavior.

A. Specification of a Sliding Window Protocol

Figure 1 depicts the SWP. A sender \( S \) stores data elements that it receives via channel A in a buffer of size \( 2n \), in the order in which they are received. \( S \) can send a datum, together with its sequence number in the buffer, to a receiver \( R \) via a medium that behaves as lossy queue of unbounded capacity, represented by the medium \( K \) and the channels B and C. Upon reception, \( R \) may store the datum in its buffer, where its position in the buffer is dictated by the attached sequence number. In order to avoid a possible overlap between the sequence numbers of different data elements in the buffers of \( S \) and \( R \), no more than one half of the buffers of \( S \) and \( R \) may be occupied.
at any time; these halves are called the sending and the receiving window, respectively. \( R \) can pass on a datum that resides at the first position in its window via channel D; in that case the receiving window slides forward by one position. Furthermore, \( R \) can send the sequence number of the first empty position in (or just outside) its window as an acknowledgment to \( S \) via a medium that behaves as lossy queue of unbounded capacity, represented by the medium \( L \) and the channels \( E \) and \( F \). If \( S \) receives this acknowledgment, its window slides forward accordingly.

The sender \( S \) is modeled by the process \( S(\ell, m, q) \), where \( q \) is a buffer of size \( 2n \), \( \ell \) the first position in the sending window, and \( m \) the first empty position in (or just outside) the sending window. Data elements can be selected at random for transmission from (the filled part of) the sending window. The receiver \( R \) is modeled by the process \( R(\ell', q') \), where \( q' \) is a buffer of size \( 2n \) and \( \ell' \) the first position in the receiving window.

Finally, we specify the mediums \( K \) and \( L \), which have unbounded capacity and may lose frames between \( S \) and \( R \), and vice versa. We cannot allow reordering of messages in the medium, as this would violate the correctness of the protocol. The medium \( K \) (see Fig. 2) is modeled by the process \( K(g, p) \), where \( g:MedK \) is a buffer with unbounded capacity, and \( p:Nat \) a pointer indicating that the frames in between position 0 and \( p \) (excluding \( p \)) can still be lost, and the frames beyond \( p \) cannot be lost any more.

\( K \) receives a frame from \( S \), stores it at the front (position 0) of \( g \), and accordingly increases \( p \) by one. It sends the last frame \((last-dat(g), last-seq(g))\) in \( g \) to \( R \). A frame at position \( k \) can be lost (if \( k < p \)), and \( p \) is then decreased by one. \( K \) can also make a choice that the frame at position \( p \) cannot be lost \((p := p - 1) \). The action \( j \) expresses the nondeterministic choice whether or not a frame is lost. In a similar way, we model the medium \( L \) by the process \( L(g', p') \).

For each channel \( i \in \{B, C, E, F\} \), actions \( s_i \) and \( r_i \) can communicate, resulting in the action \( c_i \). The initial state of the SWP is expressed by

\[ \tau_T(\vartheta_H(S(0, 0, [])) \parallel R(0, [])) \parallel K([K, 0]) \parallel L([L, 0])) \]

where the set \( H \) consists of the read and send actions over the internal channels \( B, C, E, \) and \( F \), namely \( H = \{sb, rb, sc, rc, se, rE, sf, rF \} \), while the set \( I \) consists of the communication actions over these internal channels together with \( j \), namely \( I = \{cb, cc, cE, cF, j \} \).

**B. External Behavior**

Data elements that are read from channel A should be sent into channel D in the same order, and no data elements should be lost. In other words, the SWP is intended to be a solution for the linear specification. Note that \( r_A(d) \) can be performed until the list \( \lambda \) contains \( 2n \) elements, because in that situation the sending and receiving windows will be filled. Furthermore, \( s_D(top(\lambda)) \) can only be performed if \( \lambda \) is not empty.

The remainder of this paper is devoted to proving the following theorem, expressing that the external behavior of our \( \mu \)CRL specification of a SWP corresponds to a FIFO queue of size \( 2n \).

**Theorem IV.1** \[ \tau_T(\vartheta_H(S(0, 0, [])) \parallel R(0, [])) \parallel K([K, 0]) \parallel L([L, 0])) \]

**V. Transformations of the Specification**

This section witnesses three transformations, one to eliminate parallel operators, one to eliminate arguments of communication actions, and one to eliminate modulo arithmetic.

**A. Linearization**

The starting point of our correctness proof is a linear specification \( M_{mod} \), in which no parallel operators occur. \( M_{mod} \) can be obtained from the \( \mu \)CRL specification of the SWP without the hiding operator, i.e.,

\[ \partial_H(S(0, 0, [])) \parallel R(0, []) \parallel K([K, 0]) \parallel L([L, 0])) \]

by means of a linearization algorithm presented in [17].

The linear specification \( M_{mod} \) of the SWP, with encapsulation but without hiding, takes the following form. For the sake of presentation, we only present parameters whose values are changed.

**Theorem V.1** \[ \partial_H(S(0, 0, [])) \parallel R(0, []) \parallel K([K, 0]) \parallel L([L, 0])) \]

**Proof:** It is not hard to see that replacing \( M_{mod}(\ell, m, q, \ell', q', g, p, g', p') \) by \( \partial_H(S(\ell, m, q)) \parallel R(\ell', q') \parallel K(g, p) \parallel L(g', p')) \) is a solution for the recursive equation above, using the axioms of \( \mu \)CRL [15]. (The details are left to the reader.) Hence, the theorem follows by CL-RSP [4].
Fig. 1. Sliding window protocol

\[ g \text{ with } \text{length}(g) = n + 1 \]

\[ (e_0, i_0) \quad (e_1, i_1) \quad (e_m, i_m) \quad (e_n, i_n) \]

position: 0 1 \quad m \quad n

Fig. 2. The medium \( K \)

\[ S(\ell: \text{Nat}, m: \text{Nat}, q: \text{Buf}) \approx \sum_{d: \Delta} r_A(d) \cdot S(\ell, S(m) |_{2n}, \text{inb}(d, m, q)) \]
\[ \triangleleft \text{in-window}(\ell, m, (\ell + n) |_{2n}) \triangleright \delta \]
\[ + \sum_{k: \text{Nat}} s_B(\text{retrieve}(k, q), k) \cdot S(\ell, m, q) \]
\[ \triangleleft \text{test}(k, q) \triangleright \delta \]
\[ + \sum_{k: \text{Nat}} r_F(k) \cdot S(k, m, \text{release} |_{2n}(\ell, k, q)) \]

\[ R(\ell': \text{Nat}, q': \text{Buf}) \approx \sum_{d: \Delta} \sum_{k: \text{Nat}} r_C(d, k) \cdot (R(\ell', \text{inb}(d, k, q'))) \]
\[ \triangleleft \text{in-window}(\ell', k, (\ell' + n) |_{2n}) \triangleright R(\ell', q') \]
\[ + s_D(\text{retrieve}(\ell', q')) \cdot R(S(\ell') |_{2n}, \text{remove}(\ell', q')) \]
\[ \triangleleft \text{test}(\ell', q') \triangleright \delta \]
\[ + s_E(\text{next-empty} |_{2n}(\ell', q')) \cdot R(\ell', q') \]

\[ K(g: \text{MedK}, p: \text{Nat}) \approx \sum_{d: \Delta} \sum_{k: \text{Nat}} r_B(d, k) \cdot K(\text{imm}(d, k, g), p + 1) \]
\[ + \sum_{k: \text{Nat}} j \cdot K(\text{delete}(k, g), p - 1) \triangleleft k < p \triangleright \delta \]
\[ + s_C(\text{last-dat}(g), \text{last-seq}(g)) \cdot K(\text{delete-last}(g), p) \]
\[ \triangleleft p < \text{length}(g) \triangleright \delta \]
\[ + j \cdot K(g, p - 1) \triangleleft p > 0 \triangleright \delta \]

\[ L(g': \text{MedL}, p': \text{Nat}) \approx \sum_{k: \text{Nat}} r_E(k) \cdot L(\text{imm}(k, g'), p' + 1) \]
\[ + \sum_{k: \text{Nat}} j \cdot L(\text{delete}(k, g'), p' - 1) \triangleleft k < p' \triangleright \delta \]
\[ + s_F(\text{last-seq}(g')) \cdot L(\text{delete-last}(g'), p') \]
\[ \triangleleft p' < \text{length}(g') \triangleright \delta \]
\[ + j \cdot L(g', p' - 1) \triangleleft p' > 0 \triangleright \delta \]
\[ Z(\lambda: \text{List}) \approx \sum_{d, \Delta} r_A(d) \cdot Z(\text{append}(d, \lambda)) \triangleleft \text{length}(\lambda) < 2n \triangleright \delta \\
+ s_D(\text{top}(\lambda)) \cdot Z(\text{tail}(\lambda)) \triangleleft \text{length}(\lambda) > 0 \triangleright \delta \]

\[ M_{\text{mod}}(\ell: \text{Nat}, m: \text{Nat}, q: \text{Buf}, \ell': \text{Nat}, q': \text{Buf}, g: \text{MedK}, p: \text{Nat}, g': \text{MedL}, p': \text{Nat}) \approx \sum_{d, \Delta} r_A(d) \cdot M_{\text{mod}}(m := S(m)|_{2n}, q := \text{inb}(d, m, q)) \]
\[ \triangleleft \text{in-window}(\ell, m, (\ell + n)|_{2n}) \triangleright \delta \]
\[ + \sum_{k: \text{Nat}} c_B(\text{retrieve}(k, q), k) \cdot M_{\text{mod}}(g := \text{inm}(\text{retrieve}(k, q), k, g), p := p + 1) \]
\[ \triangleleft \text{test}(k, q) \triangleright \delta \]
\[ + \sum_{k: \text{Nat}} j \cdot M_{\text{mod}}(g := \text{delete}(k, g), p := p - 1) \triangleleft k < p \triangleright \delta \]
\[ + j \cdot M_{\text{mod}}(p := p - 1) \triangleleft p > 0 \triangleright \delta \]
\[ + c_C(\text{last-dat}(g), \text{last-seq}(g)) \cdot M_{\text{mod}}(q' := \text{inn}(\text{last-dat}(g), \text{last-seq}(g), q'), g := \text{delete-last}(g)) \]
\[ \triangleleft p < \text{length}(g) \land \text{in-window}(\ell', \text{last-seq}(g), (\ell' + n)|_{2n}) \triangleright \delta \]
\[ + c_C(\text{last-dat}(g), \text{last-seq}(g)) \cdot M_{\text{mod}}(g := \text{delete-last}(g)) \]
\[ \triangleleft p < \text{length}(g) \land \neg \text{in-window}(\ell', \text{last-seq}(g), (\ell' + n)|_{2n}) \triangleright \delta \]
\[ + s_D(\text{retrieve}(\ell', q')) \cdot M_{\text{mod}}(\ell' := S(\ell')|_{2n}, q' := \text{remove}(\ell', q')) \triangleleft \text{test}(\ell', q') \triangleright \delta \]
\[ + c_F(\text{next-empty}|_{2n}(\ell', q')) \cdot M_{\text{mod}}(g := \text{inm}(\text{next-empty}|_{2n}(\ell', q'), g'), p := p' + 1) \]
\[ + \sum_{k: \text{Nat}} j \cdot M_{\text{mod}}(g' := \text{delete}(k, g'), p := p' - 1) \triangleleft k < p' \triangleright \delta \]
\[ + j \cdot M_{\text{mod}}(p' := p' - 1) \triangleleft p' > 0 \triangleright \delta \]
\[ + c_F(\text{last-seq}(g')) \cdot M_{\text{mod}}(\ell := \text{last-seq}(g'), q := \text{release}|_{2n}(\ell, \text{last-seq}(g'), q), g' := \text{delete-last}(g')) \]
\[ \triangleleft p' < \text{length}(g') \triangleright \delta \]

**B. Eliminating Arguments of Communication Actions**

The linear specification \( \text{N}_{\text{mod}} \) is obtained from \( \text{M}_{\text{mod}} \) by stripping all arguments from communication actions, and renaming these actions to a fresh action \( c \).

**Theorem V.2** \( \tau_{\text{T}}(\text{M}_{\text{mod}}(0, 0, [], 0, [], \text{K}, 0, L, 0)) \Rightarrow \tau_{\{c, j\}}(\text{N}_{\text{mod}}(0, 0, [], 0, [], \text{K}, 0, L, 0)). \)

**Proof:** By a simple renaming.

**C. Getting Rid of Modulo Arithmetic**

The specification of \( \text{N}_{\text{nonmod}} \) is obtained by eliminating all occurrences of \( L \) from \( \text{N}_{\text{mod}} \), and replacing \( \text{in-window}(\ell, m, (\ell + n)|_{2n}) \) by \( m < \ell + n \) and \( \text{in-window}(\ell', \text{last-seq}(g), (\ell' + n)|_{2n}) \) by \( \ell' \leq \text{last-seq}(g) < \ell' + n \).

**Theorem V.3** \( \text{N}_{\text{mod}}(0, 0, [], 0, [], \text{K}, 0, L, 0) \Rightarrow \text{N}_{\text{nonmod}}(0, 0, [], 0, [], \text{K}, 0, L, 0). \)

The proof of Theorem V.3 is presented in Section VII-A. Next, in Section VII-B, we prove the correctness of \( \text{N}_{\text{nonmod}} \). In these proofs we will need a wide range of data equalities, which we proceed to prove in Section VI.

**VI. Properties of Data**

**A. Basic Properties**

In the correctness proof we will make use of basic properties of the operations on \textit{Nat} and \textit{Bool}, which are derivable from their axioms (using induction). Some typical examples of such properties are:

\[ \neg\neg b = b \]
\[ i + k < j + k = i < j \]
\[ i \geq j \Rightarrow (i - j) + k = (i + k) - j \]

Lemmas VI.1 and VI.2 collect basic facts on modulo arithmetic and on buffers, respectively. Lemma VI.3 contains some results on modulo arithmetic related to buffers. Lemma VI.4 presents some facts on the \textit{next-empty} operation, together with one result on \textit{max}, which is needed to derive those facts. Lemmas VI.5 and VI.6 collect some results on unbounded buffers. Finally, Lemma VI.7 contains basic facts on lists. Unless stated otherwise (this only happens in
\[ N_{\text{mod}}(\ell:\text{Nat}, m:\text{Nat}, q:\text{Buf}, \ell':\text{Nat}, q':\text{Buf}, g:\text{MedK}, p:\text{Nat}, g'\text{'MedL}, p'\text{'Nat}) \]
\approx \sum_{d:\Delta} r_A(d) \cdot N_{\text{mod}}(m:=S(m)_{2n}, q:=\text{inb}(d,m,q))
\begin{align*}
&\quad \text{in-window}(\ell, m, (\ell + n)_{2n}) > \delta \\
+ \sum_{k:\text{Nat}} c \cdot N_{\text{mod}}(g:=\text{inm}(\text{retrieve}(k,q), k,g), p:=p+1) < \text{test}(k,q) > \delta \\
+ \sum_{k:\text{Nat}} j \cdot N_{\text{mod}}(g:=\text{delete}(k,g), p:=p-1) < k < p > \delta \\
+ j \cdot N_{\text{mod}}(p:=p-1) < p > 0 > \delta \\
+ c \cdot N_{\text{mod}}(q':=\text{inb}(\text{last-dat}(g), \text{last-seq}(g), q'), g:=\text{delete-last}(g))
\begin{align*}
&\quad < p < \text{length}(g) \land \text{in-window}(\ell', \text{last-seq}(g), (\ell' + n)_{2n}) > \delta \\
+ s_D(\text{retrieve}(\ell', q')) \cdot N_{\text{mod}}(\ell':=S(\ell')_{2n}, q':=\text{remove}(\ell', q')) < \text{test}(\ell', q') > \delta \\
+ c \cdot N_{\text{mod}}(g'\text{'=}\text{inm}(\text{next-empty}_{2n}(\ell', q'), g'), p'\text{'=}p'+1) \\
+ \sum_{k:\text{Nat}} j \cdot N_{\text{mod}}(g'\text{'=}\text{delete}(k,g'), p'\text{'=}p'-1) < k < p' > \delta \\
+ j \cdot N_{\text{mod}}(p'\text{'=}p'-1) < p' > 0 > \delta \\
+ c \cdot N_{\text{mod}}(\ell':=\text{last-seq}(g'), q:=\text{release}_{2n}(\ell, \text{last-seq}(g'), q), g'\text{'=}\text{delete-last}(g'))
\begin{align*}
&\quad < p' < \text{length}(g') > \delta \\
+ N_{\text{nonmod}}(\ell:\text{Nat}, m:\text{Nat}, q:\text{Buf}, \ell':\text{Nat}, q':\text{Buf}, g:\text{MedK}, p:\text{Nat}, g'\text{'MedL}, p'\text{'Nat}) \]
\approx \sum_{d:\Delta} r_A(d) \cdot N_{\text{nonmod}}(m:=S(m)_{2n}, q:=\text{inb}(d,m,q)) < m < \ell + n > \delta \\
+ \sum_{k:\text{Nat}} c \cdot N_{\text{nonmod}}(g:=\text{inm}(\text{retrieve}(k,q), k,g), p:=p+1) < \text{test}(k,q) > \delta \\
+ \sum_{k:\text{Nat}} j \cdot N_{\text{nonmod}}(g:=\text{delete}(k,g), p:=p-1) < k < p > \delta \\
+ j \cdot N_{\text{nonmod}}(p:=p-1) < p > 0 > \delta \\
+ c \cdot N_{\text{nonmod}}(q':=\text{inb}(\text{last-dat}(g), \text{last-seq}(g), q'), g:=\text{delete-last}(g))
\begin{align*}
&\quad < p < \text{length}(g) \land \text{in-window}(\ell, \text{last-seq}(g), (\ell + n)_{2n}) > \delta \\
+ s_D(\text{retrieve}(\ell', q')) \cdot N_{\text{nonmod}}(\ell':=S(\ell')_{2n}, q':=\text{remove}(\ell', q')) < \text{test}(\ell', q') > \delta \\
+ c \cdot N_{\text{nonmod}}(g'\text{'=}\text{inm}(\text{next-empty}_{2n}(\ell', q'), g'), p'\text{'=}p'+1) \\
+ \sum_{k:\text{Nat}} j \cdot N_{\text{nonmod}}(g'\text{'=}\text{delete}(k,g'), p'\text{'=}p'-1) < k < p' > \delta \\
+ j \cdot N_{\text{nonmod}}(p'\text{'=}p'-1) < p' > 0 > \delta \\
+ c \cdot N_{\text{nonmod}}(\ell':=\text{last-seq}(g'), q:=\text{release}_{2n}(\ell, \text{last-seq}(g'), q), g'\text{'=}\text{delete-last}(g'))
\begin{align*}
&\quad < p' < \text{length}(g') > \delta
Lemmas VI.3.2-VI.3.6, VI.3.9 and VI.5.12) all variables that occur in a data lemma are implicitly universally quantified at the outside of the equality. To the purpose of presentation, all proofs of the lemmas are omitted.

**Lemma VI.1.** Let $n > 0$.
1. $(i|n + j)|n = (i + j)|n$
2. $i|n < n$
3. $(i-n)|n = 0$
4. $i = (i \text{ div } n) \cdot n + i|n$
5. $j < i \leq j + n$
⇒ $(i \text{ div } 2n = j \text{ div } 2n \& j|n \leq) i|2n \leq j|2n + n) \lor$
6. $i < j \Rightarrow i \text{ div } n < j \text{ div } n$

**Lemma VI.2.** 1. $test(i, remove(j, q)) = (test(i, q) \land i \neq j)$
2. $i \neq j \Rightarrow retrieve(i, remove(j, q)) = retrieve(i, q)$
3. $test(i, release(j, k, q)) = (test(i, q) \land \neg(j < i < k))$
4. $j < i < k \Rightarrow retrieve(i, release(j, k, q)) = retrieve(i, q)$
5. $q \neq [] \Rightarrow test(max(q), q)$

**Lemma VI.3.** 1. $test(k, q)|2n = k = k|2n$
2. $(\forall q : Nat)(test(j, q) \Rightarrow i < j < i + n \& i \leq k < i + n)$
⇒ $test(k, q) = test(k|2n, q)|2n$
3. $(\forall q : Nat)(test(j, q) \Rightarrow i < j < i + n \& test(k, q))$
⇒ $retrieve(k, q) = retrieve(k|2n, q)|2n$
4. $(\forall q : Nat)(test(j, q) \Rightarrow i < j < i + n \& i \leq k < i + n)$
⇒ $remove(k, q)|2n = remove(k|2n, q)|2n$
5. $(\forall q : Nat)(test(j, q) \Rightarrow i < j < i + n \& i < k < i + n)$
⇒ $release(i, k, q)|2n = release(i, k, q)|2n$
6. $(\forall q : Nat)(test(j, q) \Rightarrow i < j < i + n \& i < k < i + n)$
⇒ $next-empty(k, q)|2n = next-empty(k|2n, q)|2n$
7. $i < k < i + n \Rightarrow in-window(k|2n, k|2n, (i + n)|2n)$
8. $in-window(i, k, (i + n)|2n)$
⇒ $k + n < i \lor i < k < i + n \& k \geq i + 2n$
9. $(\forall q : Nat)(test(j, q) \Rightarrow i < j < i + n \& test(k, q)|2n)$
⇒ $in-window(i, k, (i + n)|2n)$

**Lemma VI.4.** 1. $test(i, q) \Rightarrow i \leq max(q)$
2. $i < j < next-empty(i, q) \Rightarrow test(i, q)$
3. $next-empty(i, q) \geq i$
4. $next-empty(i, inb(d, j, q)) \geq next-empty(i, q)$
5. $i < j \Rightarrow next-empty(i, q)$
⇒ $next-empty(i, inb(d, j, q)) = next-empty(i, q)$
6. $next-empty(i, j, k, d, q) = next-empty(S(next-empty(i, q), q))$
7. $-i < j < next-empty(i, q)$
⇒ $next-empty(i, remove(j, q)) = next-empty(i, q)$

**Lemma VI.5.** 1. $length(g) = length(g)|2n$
2. $i < length(g)$
⇒ $return-seq(i, g)|2n = return-seq(i, g)|2n$
3. $i < length(g)$
⇒ $return-dat(i, g) = return-dat(i, g)|2n$
4. $i < length(g)$
⇒ $delete(i, g)|2n = delete(i, g)|2n$
5. $length(g) > 0$
⇒ $last-dat(g) = return-dat(length(g) - 1, g)$
6. $length(g) > 0$
⇒ $last-seq(g) = return-seq(length(g) - 1, g)$
7. $length(g) > 0$
⇒ $delete-last(g) = delete(length(g) - 1, g)$
8. $(i < length(g) \& member(d, j, delete(i, g)))$
⇒ $member(d, j, g)$
9. $i < length(g)$
⇒ $length(delete(i, g)) = length(g) - 1$
10. $i < length(g)$
⇒ $member(return-dat(i, g), return-seq(i, g), g)$
11. $(i < length(g) - 1 \& j < length(g))$
⇒ $return-seq(i, delete(j, g))$
⇒ $(i < j, return-seq(i, g), return-seq(i + 1, g))$
12. $member(d, i, g)$
⇒ $\exists j : Nat (j < length(g) \& return-seq(j, g) = i \& return-dat(j, g) = d)$

**Lemma VI.6.** 1. $length(g') = length(g'|2n)$
2. $i < length(g')$
⇒ $return-seq(i, g'|2n) = return-seq(i, g'|2n)$
3. $i < length(g')$
⇒ $delete(i, g'|2n) = delete(i, g'|2n)$
4. $length(g') > 0$
⇒ $last-seq(g') = return-seq(length(g') - 1, g')$
5. $length(g') > 0$
⇒ $delete-last(g') = delete(length(g') - 1, g')$
6. $(i < length(g') \& member(j, delete(i, g')))$
⇒ $member(j, g')$
7. $i < length(g')$
⇒ $length(delete(i, g')) = length(g') - 1$
8. $i < length(g')$
⇒ $member(return-seq(i, g'), g')$
9. $(i < length(g') - 1 \& j < length(g'))$
⇒ $return-seq(i, delete(j, g'))$
⇒ $(i < j, return-seq(i, g'), return-seq(i + 1, g'))$

**Lemma VI.7.** 1. $(\lambda + \lambda') + \lambda'' = \lambda + (\lambda' + \lambda'')$
2. $length(\lambda + \lambda') = length(\lambda) + length(\lambda')$
3. $append(d, \lambda + \lambda') = \lambda + append(d, \lambda')$
4. $length(q[i_{...}]) = j - i$
5. $i \leq k \leq j \Rightarrow q[i_{...}] = q[i_{...}] + q[k_{...}]$
6. $i \leq j \Rightarrow append(d, q[i_{...}]) = inb(d, j, q)[i_{...}]S(j)$
7. test(k, q) ⇒ inb(retrieve(k, q), k, q)[i..j] = q[i..j]
8. ¬(i ≤ k < j) ⇒ remove(k, q)[i..j] = q[i..j]
9. ℓ ≤ i ⇒ release(k, ℓ, q)[i..j] = q[i..j]

B. Invariants

Invariants of a system are properties of data that are satisfied throughout the reachable state space of the system. Lemma VI.8 collects 27 invariants of the system. Lemma VI.8 are satisfied throughout the reachable state space of the invariant.

Lemma VI.8 The following invariants 1-27 hold for $N_{nonmod}(ℓ, m, q, ℓ', q', p, g, g')$.

1. $p ≤ length(g)$
2. $p' ≤ length(g')$
3. member(i, g') ⇒ i ≤ next-empty(ℓ', q')
4. ℓ ≤ next-empty(ℓ', q')
5. $i < j < length(g')$ ⇒ return-seq(i, g') ≥ return-seq(j, g')
6. member(i, g') ⇒ ℓ ≤ i
7. test(i, q) ⇒ i < m
8. member(d, i, g) ⇒ i < m
9. test(i, q') ⇒ ℓ ≤ i
10. test(i, q') ⇒ ℓ' ≤ i < ℓ' + n
11. ℓ' ≤ m
12. next-empty(ℓ', q') ≤ m
13. next-empty(ℓ', q') ≤ ℓ' + n
14. ℓ ≤ m
15. test(i, q) ⇒ ℓ ≤ i
16. ℓ ≤ i < m ⇒ test(i, q)
17. ℓ ≤ ℓ' + n
18. m ≤ ℓ + n
19. $i < j < length(g)$ ⇒ return-seq(i, g) + n > return-seq(j, g)
20. (member(d, i, g) ∧ test(j, q')) ⇒ i + n > j
21. member(d, i, g) ⇒ i + n ≥ ℓ'
22. member(d, i, g) ⇒ i + n ≥ next-empty(ℓ', q')
23. (member(d, i, g) ∧ test(i, q)) ⇒ retrieve(i, q) = d
24. (test(i, q) ∧ test(i, q')) ⇒ retrieve(i, q) = retrieve(i, q')
25. (member(d, i, g) ∧ member(e, i, g)) ⇒ d = e
26. (member(d, i, g) ∧ test(i, q')) ⇒ retrieve(i, q') = d
27. $(ℓ ≤ i ≤ m ∧ j ≤ next-empty(i, q')) ⇒ q[i..j] = q'[i..j]$

VII. Correctness of $N_{mod}$

In Section VII-A, we prove Theorem V.3, which states that $N_{mod}$ and $N_{nonmod}$ are strongly bisimilar. Next, in Section VII-B we prove that $N_{nonmod}$ behaves like a FIFO queue of size 2n. Theorem IV.1 is proved in Section VII-C.

A. Equality of $N_{mod}$ and $N_{nonmod}$

In this section we present a proof of Theorem V.3. It suffices to prove that for all $ℓ, m, ℓ', Nat, q, q': Buf, g:MedK$ and $g':MedL$,

$N_{mod}(ℓ|2n, m|2n, q|2n, ℓ'|2n, q'|2n, g|2n, p, g'|2n, p') \equiv N_{nonmod}(ℓ, m, q, ℓ', q', g, p, g', p')$

Proof: We show that $N_{mod}(ℓ|2n, m|2n, q|2n, ℓ'|2n, q'|2n, g|2n, p, g'|2n, p')$ is a solution for the defining equation of $N_{nonmod}(ℓ, m, q, ℓ', q', g, p, g', p')$.

Hence, we must derive the following equation.²

In order to prove this, we instantiate the parameters in the defining equation of $N_{mod}$ with $ℓ|2n, m|2n, q|2n, ℓ', q'|2n, g|2n, p, g'|2n, p'$.

In order to equate the eleven summands in both specifications, we obtain the following proof obligations. Cases for summands that are syntactically the same are omitted.

\[ A \quad m < ℓ + n \Leftrightarrow \text{in-window}(ℓ|2n, m|2n, ℓ, ℓ + n)|2n) \]

\[ m < ℓ + n \]

\[ ℓ ≤ m < ℓ + n \]

(Inv. VI.8.14)

\[ \text{in-window}(ℓ|2n, m|2n, ℓ, ℓ + n)|2n) \]

(Inv. VI.8.18)

Reversely,

\[ \text{in-window}(ℓ|2n, m|2n, ℓ, ℓ + n)|2n) \]

\[ \Rightarrow m + n < ℓ \vee ℓ ≤ m < ℓ + n \vee m ≥ ℓ + 2n \]

(Inv. VI.8.14 and VI.8.18)

Moreover, by Lemma VI.1.1, $(ℓ + n)|2n = (ℓ|2n, ℓ + n)|2n)$.

- $S(m)|2n = S(m)|2n$.

This follows from Lemma VI.1.1.

- inb(d, m, q)|2n = inb(d, m, q)|2n).

This follows from the definition of buffers modulo 2n.

²By abuse of notation, we use the parameters $ℓ, m, q, ℓ', q', g, p, g', p'$ in an ambiguous way. For example, $m$ refers both to the second parameter of $N_{mod}$ and to the value of this parameter.
\[
\begin{align*}
N_{\text{mod}}(\ell_{2n}, m_{2n}, g_{2n}, \ell'_{2n}, q_{2n} | 2n, p, g'_{2n}, p') \\
& \approx \sum_{d: \Delta} r_{\text{A}}(d) \cdot N_{\text{mod}}(m:=S(m_{2n})_{2n}, q:=\text{inb}(d, m_{2n}, q_{2n})) \\
& \quad \triangleq m < \ell + n \triangleright \delta \\
& + \sum_{k: \text{Nat}} c \cdot N_{\text{mod}}(g:=\text{imm}(\text{retrieve}(k, q_{2n}), k, g_{2n}), p:=p + 1) \\
& \quad \triangleq \text{test}(k, q) \triangleright \delta \\
& + \sum_{k: \text{Nat}} j \cdot N_{\text{mod}}(g:=\text{delete}(k, g_{2n}), p:=p - 1) \\
& \quad \triangleq k < p \triangleright \delta \\
& + j \cdot N_{\text{mod}}(p:=p - 1) \\
& \quad \triangleq p > 0 \triangleright \delta \\
& + c \cdot N_{\text{mod}}(q':=\text{inb}(\text{last-dat}(g), \text{last-seq}(g), q') | 2n, g:=\text{delete-last}(g_{2n})) \\
& \quad \triangleq p < \text{length}(g) \land (\ell' \leq \text{last-seq}(g) < \ell' + n) \triangleright \delta \\
& + c \cdot N_{\text{mod}}(g:=\text{delete-last}(g_{2n})) \\
& \quad \triangleq p < \text{length}(g) \land (\ell' \leq \text{last-seq}(g) < \ell' + n) \triangleright \delta \\
& + s_{\text{D}}(\text{retrieve}(\ell', q')) \cdot N_{\text{mod}}(\ell':=S(\ell'_{2n})_{2n}, q':=\text{remove}(\ell', q') | 2n) \\
& \quad \triangleq \text{test}(\ell', q') \triangleright \delta \\
& + c \cdot N_{\text{mod}}(g':=\text{inm}(\text{next-empty}(\ell', q'), g')_{2n}, p':=p' + 1) \\
& \quad \triangleq \text{test}(k, q_{2n}) \triangleright \delta \\
& + \sum_{k: \text{Nat}} j \cdot N_{\text{mod}}(g':=\text{delete}(k, g'_{2n}), p':=p' - 1) \\
& \quad \triangleq k < p \triangleright \delta \\
& + j \cdot N_{\text{mod}}(p':=p' - 1) \\
& \quad \triangleq p' > 0 \triangleright \delta \\
& + c \cdot N_{\text{mod}}(\ell:=\text{last-seq}(g')_{2n}, q:=\text{release}(\ell, \text{last-seq}(g'), q) | 2n, g':=\text{delete-last}(g'_{2n})) \\
& \quad \triangleq p' < \text{length}(g') \triangleright \delta
\end{align*}
\]
The last equality follows from the following derivation:

because it is irrelevant for this derivation.

This follows from Lemma VI.5.1.

By Invariant VI.8.1, 

(p < length(g) \Rightarrow (p < last-seq(g) < p + n = in-window(p'|2n, last-seq(g)|2n, (p' + n)|2n)).

Since 0 < length(g), Lemmas VI.5.5, VI.5.6, and VI.5.10 yield member(last-dat(g), last-seq(g), g). So by Invariant VI.8.22, next-empty(p', q') \leq last-seq(g) + n. Hence, by Lemma VI.4.3, p' \leq last-seq(g) + n. Furthermore, by Invariant VI.8.8, last-seq(g) < m, by Invariant VI.8.18, m = \ell + n, and by Invariant VI.8.17, \ell \leq \ell' + n. Hence, last-seq(g) < \ell' + 2n. So by Lemmas VI.3.7 and VI.3.8, \ell' \leq last-seq(g) < \ell' + n = in-window(\ell'|2n, last-seq(g)|2n, (\ell' + n)|2n). And by Lemma VI.1.1, (\ell' + n)|2n = (\ell'|2n, last-seq(g)|2n).

p < length(g) \Rightarrow delete-last(2n)|2n = delete-last(2n)|2n.

This follows from the definitions of buffers modulo 2n, and Lemmas VI.5.5, VI.5.6, VI.5.2 and VI.5.3.

F \quad \neg(\ell' \leq last-seq(g) < p + n) 

\iff \neg in-window(\ell'|2n, last-inserted(2n)|2n, (\ell'|2n + n)|2n).

This follows immediately from the second item of the previous case.

p < length(g) \Rightarrow delete-last(2n)|2n = delete-last(2n)|2n.

This follows immediately from the fourth item of the previous case.

G \quad test(\ell', q') = test(\ell'|2n, q'|2n)

This follows from Lemma VI.3.2 together with Invariant VI.8.10.
• \( \text{test}(\ell', q') \Rightarrow (\text{retrieve}(\ell', q') = \text{retrieve}(\ell'|_{2n}, q'|_{2n})) \).
  This follows from Lemma VI.3.3 together with Invariant VI.8.10.
• \( S(\ell'|_{2n}) = S(\ell'|_{2n})|_{2n} \).
  This follows from Lemma VI.1.1.
• \( \text{remove}(\ell', q')|_{2n} = \text{remove}(\ell'|_{2n}, q'|_{2n}) \).
  This follows from Lemma VI.3.4 together with Invariant VI.8.10.

\[
\text{FC}(\ell, m, q, \ell', q', g, p, g', p') := \ell = m \lor \ell = \ell' + n
\]

Lemma VII.1 For each \( \xi; \Xi \) where the invariants in Lemma VI.8 hold, there is a \( \xi; \Xi \) with FC(\( \xi \)) such that \( N_{\text{nonmod}}(\xi) \preceq \ldots \preceq N_{\text{nonmod}}(\xi') \), where \( c_1, \ldots, c_n \in I \).

Proof: By Invariants VI.8.12 and VI.8.13, \( \text{next-empty}(\ell', q') \leq \min\{m, \ell' + n\} \). We prove by induction on \( \min\{m, \ell' + n\} - \text{next-empty}(\ell', q') \) that for each state \( \xi \) where the invariants in Lemma VI.8 hold, a focus point can be reached by a sequence of communication actions.

Base Case: \( \text{next-empty}(\ell', q') = \min\{m, \ell' + n\} \).
Let \( y = \text{length}(g') \) and \( x = \text{next-empty}(\ell', q') \) at state \( \xi \). By summand \( H \), we reach a state \( \xi' \) with \( g' := \text{imm}(x, g') \). Hence, at state \( \xi' \) there exists a \( 0 \leq k < y \) such that \( \text{return-seq}(k, g') = x = \text{return-seq}(i, g') \) for any \( k < i < y \). In view of Invariant VI.8.5, \( k < i < y \Rightarrow x = \text{return-seq}(i, g') \). Then, by repeating summand \( J \) (\( p' \) times), we reach a state \( \xi'' \) with \( p' = 0 \). Then, by repeating summand \( K \) (\( y - (k + 1) \) times), we reach a state \( \xi''' \) such that \( \text{last-seq}(g') = x \). During these executions of \( H, J \) and \( K \) the values of \( m, \ell', q' \) remain the same. By again performing summand \( K \), we reach a state \( \xi \) where \( \ell = \text{last-seq}(g') = x = \min\{m, \ell' + n\} \). Then \( \ell = m \) or \( \ell = \ell' + n \), so FC(\( \xi \)).

Induction Case: \( \text{next-empty}(\ell', q') < \min\{m, \ell' + n\} \).
Let \( y = \text{length}(g) \) and \( x = \text{next-empty}(\ell', q') \) at state \( \xi \). By Invariants VI.8.4 and VI.8.12, \( \ell \leq x < m \). So by Invariant VI.8.16, test(x, q). Furthermore, in view of Lemma VI.4.3, \( \ell' \leq x < \ell' + n \). By summand \( B \), we perform a communication action to a state \( \xi' \) with \( g := \text{imm}(d, x, g) \) (where \( d \) denotes retrieve(x, q)). Hence, at state \( \xi' \) there exists a \( 0 \leq k < y \) such that \( \text{return-seq}(k, g') = x = \text{return-seq}(i, g') \) for any \( k < i < y \). Then, by repeating summand \( D \) (\( p \) times), we reach a state \( \xi'' \) with \( p = 0 \). Then, by repeating...
summands $E$ and $F$ ($y-(k+1)$ times), we reach a state $\xi''$ with last-dat($g$) = $d$ and last-seq($g$) = $x$. During these executions of $B, D, E$ and $F$, the values of $m, l'$ remain the same; and since during the executions of $E$ and $F$ last-seq($g$) $\neq x$, in view of Lemma VI.4.5, the value of next-empty($l', q'$) remains the same. By again performing summand $E$, we reach a state $\xi'''$ where $q' := \text{inb}(d, x, q')$. Recall that $x = \text{next-empty}(l', q')$.

\[
\text{next-empty}(l', \text{inb}(d, \text{next-empty}(l', q'), q')) = \text{next-empty}(S(\text{next-empty}(l', q'), q'))
\]

(Lem. VI.4.6)

> next-empty($l', q'$)

(Lem. VI.4.3)

So we can apply the induction hypothesis to conclude that from $\xi'''$ a focus point $\xi$ can be reached by a sequence of communication actions.

\[\tau_{(c,j)}(N_{\text{nonmod}}(0, 0, [], 0, 0)) \rightarrow Z(0).\]

\[\text{Proof:} \quad \text{By the cones and foci method we obtain the following matching criteria (see Definition II.2). Trivial matching criteria are left out.}

Class I:

\[
(p < \text{length}(g) \land l' \leq \text{last-seq}(g) < l' + n) \Rightarrow \phi(m, q, l', q') = \phi(m, q, l', \text{inb}(\text{last-dat}(g), \text{last-seq}(g), q'))
\]

\[
p' < \text{length}(q') \Rightarrow \phi(m, q, l', q') = \phi(m, \text{release}(l, \text{last-seq}(g'), q), l', q')
\]

Class II:

\[
m < l + n \Rightarrow \text{length} \phi(m, q, l', q')) < 2n
\]

\[
\text{test}(l', q') \Rightarrow \text{length} \phi(m, q, l', q')) > 0
\]

Class III:

\[
((l = m \lor l = l' + n) \land \text{length} \phi(m, q, l', q')) < 2n
\]

\[
(l = m \lor l = l' + n) \land \text{length} \phi(m, q, l', q')) > 0
\]

\[
\Rightarrow \text{test}(l', q')
\]

Class IV:

\[
\text{test}(l', q') \Rightarrow \text{retrieve}(l', q') = \text{top} \phi(m, q, l', q'))
\]

Class V:

\[
m < l + n
\Rightarrow \phi(S(m), \text{inb}(d, m, q), l', q') = \text{append}(d, \phi(m, q, l', q'))
\]

\[
\text{test}(l', q')
\Rightarrow \phi(m, q, S(l'), \text{remove}(l', q')) = \text{tail}(\phi(m, q, l', q'))
\]

I. Let $p < \text{length}(g)$. By Lemmas VI.5.5, VI.5.6 and VI.5.10, member(\text{last-dat}(g), \text{last-seq}(g), g).

Case 1: \text{last-seq}(g) $\neq$ \text{next-empty}(l', q').

By Lemma VI.4.5,\text{next-empty}(l', \text{inb}(\text{last-dat}(g), \text{last-seq}(g), q')) = \text{next-empty}(l', q')$. Hence,

\[
\phi(m, q, l', \text{inb}(\text{last-dat}(g), \text{last-seq}(g), q'))
\]

\[
\text{inb}(\text{last-dat}(g), \text{last-seq}(g), q')[l'..\text{next-empty}(l', q')]
\]

\[
\text{+q[\text{next-empty}(l', q')..m]
\]

Case 1.1: \text{last-seq}(g) < \text{next-empty}(l', q').

By Lemma VI.4.2, \text{test}(\text{last-seq}(g), q'), so by Invariant VI.8.26 together with member(\text{last-dat}(g), \text{last-seq}(g), g), retrieve(\text{last-seq}(g), q') = \text{last-dat}(g). So by Lemma VI.7.7, \text{inb}(\text{last-dat}(g), \text{last-seq}(g), q')[l'..\text{next-empty}(l', q')] = q'[l'..\text{next-empty}(l', q')].

Case 1.2: \text{last-seq}(g) < \text{next-empty}(l', q').

The derivation splits into two parts.

(1) Using Lemma VI.7.8, it follows that

\[
\text{inb}(\text{last-dat}(g), \text{last-seq}(g), q')[l'..\text{last-seq}(g)] = \text{remove}(\text{last-seq}(g), \text{inb}(\text{last-dat}(g), \text{last-seq}(g), q'))
\]

\[
[l'..\text{next-empty}(l', q')]
\]

\[
\text{remove}(\text{last-seq}(g), q')[l'..\text{last-seq}(g)] = q'[l'..\text{last-seq}(g)]
\]

(2) By Invariant VI.8.4, \text{last-seq}(g) \leq \text{last-seq}(g). By Invariant VI.8.8 and member(\text{last-dat}(g), \text{last-seq}(g), g), \text{last-seq}(g) < m. Thus, by Invariant VI.8.16, \text{test}(\text{last-seq}(g), q). So by Invariant VI.8.23 with member(\text{last-dat}(g), \text{last-seq}(g), g), retrieve(\text{last-seq}(g), q) = \text{last-dat}(g). Since $l \leq S(\text{last-seq}(g)) \leq m$, by Invariant VI.8.27,

\[
q[S(\text{last-seq}(g))..\text{next-empty}(S(\text{last-seq}(g)), q')] = q[S(\text{last-seq}(g))..\text{next-empty}(S(\text{last-seq}(g)), q')]
\]
Hence,

\[
\begin{align*}
\text{inl}(\text{last-dat}(g), \text{last-seq}(g), q') \\
\text{[last-seq}(g) \ldots \text{next-empty}(\text{S(last-seq}(g), q')) \\
= \text{inl}(\text{last-dat}(g), \text{inl}(\text{last-dat}(g), \text{last-seq}(g), q') \\
\text{[S(last-seq}(g) \ldots \text{next-empty}(\text{S(last-seq}(g), q')))
\end{align*}
\]

inl(\text{last-dat}(g), \text{remove}(\text{last-seq}(g), q'))

\[
= \text{inl}(\text{last-dat}(g), \text{remove}(\text{last-seq}(g), q') \\
\text{[S(last-seq}(g) \ldots \text{next-empty}(\text{S(last-seq}(g), q')))
\]

\[
(\text{Lem. VI.7.1, VI.4.2, VI.7.5})
\]

Finally, we combine (1) and (2). We recall that last-seq(g) = next-empty(\ell', q').

II.1 Let m < \ell + n.

\[
\begin{align*}
\text{length}(q'[\ell' \ldots \text{next-empty}(\ell', q')) &+ \\
q[\text{next-empty}(\ell', q') \ldots m]
\end{align*}
\]

\[
= \text{length}(q'[\ell' \ldots \text{next-empty}(\ell', q')) &+ \\
+ \text{length}(q[\text{next-empty}(\ell', q') \ldots m])
\]

(\text{Lem. VI.7.2})

\[
= (\text{next-empty}(\ell', q') - \ell') + (m - \text{next-empty}(\ell', q'))
\]

(\text{Lem. VI.7.4})

\[
\leq n + (m - \ell)
\]

(Inv. VI.8.13, VI.8.4)

< 2n

II.2 test(\ell', q') together with Lemma VI.4.3 yields next-empty(\ell', q') = \text{next-empty}(S(\ell'), q') \geq S(\ell'). Hence, by Lemmas VI.7.2 and VI.7.4,

\[
\text{length}(\phi(m, q, \ell', q'))
\]

\[
= (\text{next-empty}(\ell', q') - \ell') + (m - \text{next-empty}(\ell', q'))
\]

> 0

III.1 Case 1: \ell = m.

Then m < \ell + n holds trivially.

Case 2: \ell = \ell + n.

\[
\text{length}(\phi(m, q, \ell', q'))
\]

\[
= (\text{next-empty}(\ell', q') - \ell') + (m - \text{next-empty}(\ell', q'))
\]

(\text{Lem. VI.7.2, VI.7.4})

\[
\leq ((\ell' + n) - \ell') + (m - \ell)
\]

(Inv. VI.8.13, VI.8.4)

= n + (m - \ell)

So \text{length}(\phi(m, q, \ell', q')) < 2n implies m < \ell + n.

III.2 Case 1: \ell = m.

Then m - \text{next-empty}(\ell', q') \leq m - \ell (Inv. VI.8.4) = 0, so

\[
\text{length}(\phi(m, q, \ell', q'))
\]

\[
= (\text{next-empty}(\ell', q') - \ell') + (m - \text{next-empty}(\ell', q'))
\]

(\text{Lem. VI.7.2, VI.7.4})

= \text{next-empty}(\ell', q') - \ell'

Hence, \text{length}(\phi(m, q, \ell', q')) > 0 yields next-empty(\ell', q') > \ell', which implies test(\ell', q').

Case 2: \ell = \ell + n.

Then by Invariant VI.8.4, next-empty(\ell', q') \geq \ell + n, which implies test(\ell', q').

IV test(\ell', q') implies next-empty(\ell', q')

= \text{next-empty}(S(\ell'), q') \geq S(\ell') (\text{Lem. VI.4.3}). Hence,

\[
q'[\ell' \ldots \text{next-empty}(\ell', q')]
\]

= \text{inl}(\text{retrieve}(\ell', q'), q'[\text{S(\ell') \ldots \text{next-empty}(\ell', q')])

I.2 Let \text{g'} < \text{length}(g'). By Lemmas VI.6.4 and VI.6.8, member(last-seq(g'), q').

By Invariant VI.8.3, last-seq(g') \leq \text{next-empty}(\ell', q'). So by Lemma VI.7.9, release(\ell, last-seq(g'), q)

\[
[\text{next-empty}(\ell', q') \ldots m] = q[\text{next-empty}(\ell', q') \ldots m]
\]
So
\[\topof(\phi(m, q, \ell', q'))\]
= \[\topof(\text{inl}(\text{retrieve}(\ell', q'), q'[S(\ell').\text{next-empty}(\ell', q')])
++ q[\text{next-empty}(\ell', q').m))\]
= \text{retrieve}(\ell', q')

V.1
\[q'[\ell'.\text{next-empty}(\ell', q')++]\]
= \[inb(d, m, q)[\text{next-empty}(\ell', q').S(m)]\]
= \[q'[\ell'.\text{next-empty}(\ell', q')++]\]
= \[\text{append}(d, q[\text{next-empty}(\ell', q').m])]\]
= \text{append}(d, q[\ell'.\text{next-empty}(\ell', q')++]
= \[q[\text{next-empty}(\ell', q').m)]\]

(V.1.7.6, Inv. VI.8.12)

V.2 \text{test}(\ell', q') and Lemma VI.4.3 imply
\[\text{next-empty}(\ell', q') = \text{next-empty}(S(\ell'), q') \geq S(\ell').\]
Hence,
\[\text{remove}(\ell', q')[S(\ell').\]
\[\text{next-empty}(S(\ell'), \text{remove}(\ell', q'))]++
\[q[\text{next-empty}(S(\ell'), \text{remove}(\ell', q')).m]\]
= \[\text{remove}(\ell', q')[S(\ell').\text{next-empty}(S(\ell'), q')]++
\[q[\text{next-empty}(S(\ell'), q').m]\]
= \text{remove}(\ell', q')[S(\ell').\text{next-empty}(\ell', q')]++
\[q[\text{next-empty}(\ell', q').m]\]
= \[q[S(\ell').\text{next-empty}(\ell', q')]++
\[q[\text{next-empty}(\ell', q').m]\]
= \text{tail}(q'[\ell'.\text{next-empty}(\ell', q')]++
\[q[\text{next-empty}(\ell', q').m])\]

C. Correctness of the Sliding Window Protocol

Finally, we can prove Theorem IV.1.

Proof:
\[\tau_{\ell}(\partial_{H}(\text{S}(0, 0, [i] \mid \text{R}(0, [i]) \mid \text{K}([i]^{K}, 0) \mid \text{L}([i]^{L}, 0))))\]

(Thm. V.1)
\[\tau_{\ell}(\text{M}^{mod}(0, 0, [i], 0, [i], [i]^{K}, 0, [i]^{L}, 0))\]

(Thm. V.2)
\[\tau_{\ell}(\text{N}^{mod}(0, 0, [i], 0, [i], [i]^{K}, 0, [i]^{L}, 0))\]

(Thm. V.3)
\[\tau_{\ell}(\text{N}^{nonmod}(0, 0, [i], 0, [i], [i]^{K}, 0, [i]^{L}, 0))\]

(Thm. V.4)
\[\text{Z}(\emptyset)\]

(Thm. VII.2)

VIII. Related Work

Sliding window protocols have attracted considerable interest from the formal verification community. In this section we present an overview. Many of these verifications deal with unbounded sequence numbers, in which case modulo arithmetic is avoided, or with a fixed finite window size. The papers that do treat arbitrary finite window sizes mostly restrict to safety properties.

Infinitive window size. Stenning [41] studied a SWP with unbounded sequence numbers and an infinite window size, in which messages can be lost, duplicated or reordered. A timeout mechanism is used to trigger retransmission. Stenning gave informal manual proofs of some safety properties. Knuth [26] examined more general principles behind Stenning’s protocol, and manually verified some safety properties. Halperr [20] used temporal logic to formulate safety and liveness properties for Stenning’s protocol, and established their validity by informal reasoning. Jonsson [23] also verified both safety and liveness properties of the protocol, using temporal logic and a manual compositional verification technique.

Fixed finite window size. Richards et al. [34] specified a SWP in a process algebra based language Estelle/R, and verified safety properties for window size up to eight using the model checker Xesar. Madelaine and Vergamini [29] specified a SWP in Lotos, with the help of the simulation environment Lite, and proved some safety properties for window size six. Holzmann [21], [22] used the Spin model checker to verify both safety and liveness properties of a SWP with sequence numbers up to five. Käivola [25] verified safety and liveness properties using model checking for a SWP with window size up to seven. Godefroid and Long [11] specified a full duplex SWP in a guarded command language, and verified the protocol for window size two using a model checker based on Queue BDDs. Stahl et al. [40] used a combination of abstraction, data independence, compositional reasoning and model checking to verify safety and liveness properties for a SWP with window size up to sixteen. The protocol was specified in Promela, the input language for the Spin model checker. Smith and Klarlund [38] specified a SWP in the high-level language IOA, and used the theorem prover MONA to verify a safety property for unbounded sequence numbers with window size up to 256. Latvala [27] modeled a SWP using Colored Petri nets. A liveness property was model checked with fairness constraints for window size up
to eleven.

**Arbitrary finite window size.** Cardell-Oliver [6] specified a SWP using higher order logic, and manually proved and mechanically checked safety properties using HOL. (Van de Snepscheut [39] noted that what Cardell-Oliver claims to be a liveness property is in fact a safety property.) Schoone [37] manually proved safety properties for several SWPs using assertional verification. Van de Snepscheut [39] gave a correctness proof of a SWP as a sequence of correctness preserving transformations of a sequential program. Paliwoda and Sanders [32] specified a reduced version of what they call a SWP (but which is in fact very similar to the bakery protocol from [14]) in the process algebra CSP, and verified a safety property modulo trace semantics. Röckl and Esparza [35] verified the correctness of this bakery protocol modulo weak bisimulation using Isabelle/HOL, by explicitly checking a bisimulation relation. Jonsson and Nilsson [24] used an automated reachability analysis to verify safety properties for a SWP with arbitrary sending window size and receiving window size one. Rusu [36] used the theorem prover PVS to verify both safety and liveness properties for a SWP with unbounded sequence numbers. Chkliaev et al. [8] used a timed state machine in PVS to specify a SWP with a timeout mechanism and proved some safety properties with the mechanical support of PVS. Correctness is based on the timeout mechanism, which allows messages in the mediums to be reordered.

**REFERENCES**


