Verifying a Sliding Window Protocol in \( \mu \text{CRL} \)

Wan Fokkink\(^1,2\), Jan Friso Groote\(^1,3\), Jun Pang\(^1\),
Bahareh Badban\(^1\), and Jaco van de Pol\(^1\)

\(^1\) CWI, Embedded Systems Group
\{wan, pang\jun, badban, vd\pol\}@cwi.nl
\(^2\) Vrije Universiteit Amsterdam, Theoretical Computer Science Group
\(^3\) Eindhoven University of Technology, Systems Engineering Group
J.F.Groote@tue.nl

Abstract. We prove the correctness of a sliding window protocol with
an arbitrary finite window size \( n \) and sequence numbers modulo \( 2n \). We
show that the sliding window protocol is branching bisimilar to a queue
of capacity \( 2n \). The proof is given entirely on the basis of an axiomatic
theory, and was checked with the help of PVS.

1 Introduction

Sliding window protocols [6] (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 ac-
knowledgement 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 \( 2n \) (and
not less, see [42] for a nice argument). The messages at the sender are numbered
from \( i \) to \( i+n \) (modulo \( 2n \)); this is called a \textit{window}. When an acknowledgement
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 \( 2n \)). 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 atten-
tion. We provide a comparison with verifications of SWPs in Section 8, and
restrict here to the context in which the current paper was written. After the
advent of process algebra in the early 80's 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 Brunekreef [4] 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 windows
have size 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 the process algebraic language μCRL [16]. This turned out to be unexpectedly hard, which is shown by the 13 year it took to finish the current paper, and led to significant developments in the realm of process algebraic proof techniques for protocol verification. We therefore consider the current paper as a true milestone in process algebraic verification.

Our first observation was that the external behaviour of the protocol, as given in [42], was unclear. We adapted the SWP such that it nicely behaves as a queue of capacity 2n. The second observation was that the SWP of [42] contained a deadlock [13, Stelling 7], 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 μCRL 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 cones and foci proof method [19,9]. This method rephrases the question whether two system specifications are branching bisimilar in terms of proof obligations on relations between data objects. It 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 μCRL of a SWP with buffer size 2n and window size n, for arbitrary n. The medium between the sender and the receiver is modelled as a lossy queue of capacity one. We manually prove that the external behaviour of this protocol is branching bisimilar [10] to a FIFO queue of capacity 2n. This proof is entirely based on the axiomatic theory underlying μCRL and the axioms characterising the data types. It implies both safety and liveness of the protocol (the latter under the assumption of fairness). First, we linearise 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 [2], which states that each linear specification has a unique solution (modulo branching bisimulation). 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 http://www.cwi.nl/~badban/swp.html. Ongoing research is to extend the current verification to a setting where the medium is modelled as a lossy queue of unbounded capacity, and to include duplex message passing and piggybacking.

In this extended abstract we omitted most equational definitions of the data types, most lemmas regarding these data types, part of the invariants and part of the correctness proofs. The reader is referred to the full version of the paper [8], for these definitions and proofs.
2 \( \mu \text{CRL} \)

\( \mu \text{CRL} \) [16] (see also [18]) is a language for specifying distributed systems and protocols in an algebraic style. It is based on the process algebra ACP [1] extended with equational abstract data types [28]. In a \( \mu \text{CRL} \) specification, one part specifies the data types, while a second part specifies the process behaviour.

The data types needed for our \( \mu \text{CRL} \) specification of a SWP are presented in Section 3. 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 \( \mathcal{A} \) 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 \( \mathcal{A} \): \( \delta \) represents deadlock, and \( \tau \) a hidden action. These two actions never carry data parameters. \( p \cdot q \) denotes sequential composition and \( p + q \) non-deterministic choice. Summation \( \sum_{d \in D} p(d) \) provides the possibly infinite choice over a data type \( D \), and the conditional construct \( p < b \triangleright q \) with \( b \) a data term of sort \( \text{Bool} \) behaves as \( p \) if \( b \) and as \( q \) if \( \neg b \). Parallel composition \( p \parallel q \) interleaves the actions of \( p \) and \( q \); moreover, actions from \( p \) and \( q \) may also synchronise to a communication action, when this is explicitly allowed by a predefined communication function. Two actions can only synchronise if their data parameters are equal. Encapsulation \( \theta_{\mathcal{H}}(p) \), which renames all occurrences in \( p \) of actions from the set \( \mathcal{H} \) into \( \delta \), can be used to force actions into communication. Hiding \( \tau_{\mathcal{T}}(p) \) renames all occurrences in \( p \) of actions from the set \( \mathcal{T} \) 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 if it is of the form

\[
X(d_1; D_1, \ldots, d_n; D_n) \approx \sum_{i=1}^{t} \sum_{x_i=1}^{x_i} a_i(e_1^i, \ldots, e_{m_i}^i) \cdot X(d_1^i, \ldots, d_n^i) \triangleleft b_i \triangleright \delta.
\]

To each \( \mu \text{CRL} \) specification belongs a directed graph, called a labelled transition system, which is defined by the structural operational semantics of \( \mu \text{CRL} \) (see [16]). In this labelled transition system, the states are process terms, and the edges are labelled with parameterised actions. Branching bisimulation \( \approx_b \) [10] and strong bisimulation \( \approx \) [33] are two well-established equivalence relations on the states in labelled 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 \( p \approx_b q \).

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. We use three proof principles from the literature:

- **Sum elimination** [14] states that a summation over a data type from which only one element can be selected can be removed.
- **CL-RSP** [2] states that the solutions of a linear \( \mu \text{CRL} \) specification that does not contain any infinite \( \tau \) sequence are all strongly bisimilar.
The cones and foci method from [9, 19] rephrases the question whether two linear μCRL specifications \( \tau_\mathcal{I}(S_1) \) and \( S_2 \) are branching bisimilar, where \( S_2 \) does not contain actions from some set \( \mathcal{I} \) 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 \( \mathcal{I} \). 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 \( \tau_\mathcal{I}(S_1) \) and \( S_2 \) are branching bisimilar.

3 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.

**Booleans and Natural Numbers.** \textit{Bool} is the data type of booleans. \texttt{t} and \texttt{f} denote true and false, \( \land \) and \( \lor \) conjunction and disjunction, \( \rightarrow \) and \( \leftrightarrow \) implication and logic equivalence, and \( \neg \) negation. For a boolean \( b \), we abbreviate \( b = \texttt{t} \) to \( b \) and \( b = \texttt{f} \) to \( \neg b \). Unless otherwise stated, data parameters in boolean formulas are universally quantified.

For each data type \( D \) in this paper there is an operation \( if : \textit{Bool} \times D \times D \rightarrow D \) with as defining equations \( if(\texttt{t}, d, e) = d \) and \( if(\texttt{f}, d, e) = e \). Furthermore, for each data type \( D \) in this paper one can easily define a mapping \( eq : D \times D \rightarrow \textit{Bool} \) 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 \( eq(d, e) \).

\textit{Nat} is the data type of natural numbers. \texttt{0} denotes zero, \( S(n) \) the successor of \( n \), \( + \), \( - \) and \( \cdot \) addition, monus (also called proper subtraction) and multiplication, and \( \leq, <, >, \geq, \) and \( \textless, > \) less-than-(or-equal) and greater-than-(or-equal). Usually, the sign for multiplication is omitted, and \( \neg(i = j) \) is abbreviated to \( i \neq j \). As binding convention, \( \{\texttt{=}, \texttt{\neq}\} > \{\cdot\} > \{+, -, \cdot\} > \{\leq, <, \geq, >\} > \{\neg\} > \{\land, \lor\} > \{\rightarrow, \leftrightarrow\} \).

Since the buffers at the sender and the receiver in the sliding window are of size \( 2n \), calculations modulo \( 2n \) play an important role. \( i \bmod n \) denotes \( i \) modulo \( n \), while \( i \texttt{div} n \) denotes \( i \) integer divided by \( n \).

**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 modelled as a list of pairs \((d, i)\) with \( d \in \Delta \) and \( i \in \textit{Nat} \), representing that position (or sequence number) \( i \) of the buffer is occupied by datum \( d \). The data type \textit{Buf} is specified as follows, where \( [] \) denotes the empty buffer: \([\cdot] : \rightarrow \textit{Buf} \) and \( in : \Delta \times \textit{Nat} \times \textit{Buf} \rightarrow \textit{Buf} \). \( q|_n \) denotes buffer \( q \) with all sequence numbers taken modulo \( n \). \([\cdot]|_n = [] \) and \( in(d, i, q)|_n = in(d, i, q) \). \textit{test}(i, q) produces \( t \) if and only if position \( i \) in \( q \) is occupied, \textit{retrieve}(i, q) produces the datum that resides at position \( i \) in buffer \( q \) (if this position is occupied), and \textit{remove}(i, q) is obtained by emptying position
\(i\) in buffer \(q\). \(\text{release}(i, j, q)\) is obtained by emptying positions \(i\) up to \(j\) excluded 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. \(\text{next-empty}_{n}(i, q)\) does the same modulo \(n\).

\[
\begin{align*}
\text{next-empty}(i, q) &= \text{if} (\text{test}(i, q), \text{next-empty}(S(i), q), i) \\
\text{next-empty}_{n}(i, q) &= \begin{cases} 
\text{next-empty}(i_{n}, q_{n}) & \text{if } \text{next-empty}(i_{n}, q_{n}) < n \\
\text{next-empty}(0, q_{n}) & \text{otherwise}
\end{cases}
\end{align*}
\]

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

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

**Lists.** The data type \(\text{List}\) of lists is used in the specification of the desired external behaviour of the SWP: a FIFO queue of capacity \(2n\). It is specified by the empty list \(\langle \rangle \rightarrow \text{List}\) and in : \(\Delta \times \text{List} \rightarrow \text{List}\). \(\text{length}(\lambda)\) denotes the length of \(\lambda\), \(\text{top}(\lambda)\) produces the datum at the top of \(\lambda\), \(\text{tail}(\lambda)\) is obtained by removing the top position in \(\lambda\), \(\text{append}(d, \lambda)\) adds datum \(d\) at the end of \(\lambda\), and \(\lambda + \lambda'\) represents list concatenation. Furthermore, \(q[i..j]\) is the list containing the elements in buffer \(q\) at positions \(i\) up to but not including \(j\). An empty position in \(q\), in between \(i\) and \(j\), gives rise to an occurrence of the default datum \(d_0\) in \(q[i..j]\).

\[
q[i..j] = \begin{cases} 
\langle \rangle & \text{if } i \geq j \\
\text{in}((\text{retrieve}(i, q), q[S(i)..j])) & \text{if } i < j \land \text{test}(i, q) \\
\text{in}(d_0, q[S(i)..j]) & \text{if } i < j \land \neg \text{test}(i, q)
\end{cases}
\]

### 4 Sliding Window Protocol

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

**Specification of a Sliding Window Protocol.** 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 capacity one, 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 acknowledgement to \( S \) via a medium that behaves as lossy queue of capacity one, represented by the medium \( L \) and the channels \( E \) and \( F \). If \( S \) receives this acknowledgement, its window slides accordingly.

The sender \( S \) is modelled 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.

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

The receiver \( R \) is modelled by the process \( R(\ell', q') \), where \( q' \) is a buffer of size \( 2n \) and \( \ell' \) the first position in the receiving window.

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

For \( i \in \{B, C, E, F\} \), \( s_i \) and \( r_i \) can communicate, resulting in \( c_i \).

Finally, the mediums \( K \) and \( L \), which have capacity one, may lose frames between \( S \) and \( R \). The action \( j \) indicates an internal choice.

\[
K \approx \sum_{d : \Delta} \sum_{k : \text{Nat}} r_B(d, k) \cdot (j \cdot sc(d, k) + j) \cdot K \\
L \approx \sum_{k : \text{Nat}} r_E(k) \cdot (j \cdot sf(k) + j) \cdot L
\]

The initial state of the SWP is expressed by \( \tau_I(\partial_H(S(0, 0, []) \parallel R(0, [])) \parallel K \parallel L) \), where the set \( H \) consists of the read and send actions over the internal
channels B, C, E, and F, while the set I consists of the communication actions over these internal channels together with j.

**External Behaviour.** Data elements that are read from channel A by S should be sent into channel D by R 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

\[ Z(\lambda; \text{List}) \approx \sum_d \lambda r_A(d) Z(\text{append}(d, \lambda)) < \text{length}(\lambda) < 2n \triangleright \delta \\
+ s_D(\text{top}(\lambda)) Z(\text{tail}(\lambda)) < \text{length}(\lambda) > 0 \triangleright \delta \]

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, \( s_D(\text{top}(\lambda)) \) can only be performed if \( \lambda \) is not empty.

The remainder of this paper is devoted to proving the following theorem.

**Theorem 1.** \( \tau_\pi(\partial_h(S(0, 0, [], []) \parallel R(0, [], []) \parallel K \parallel L)) \rightarrow_b Z(\langle \rangle) \).

### 5 Transformations of the Specification

The starting point of our correctness proof is a linear specification \( N_{\text{mod}} \), in which no parallel operators occur. \( N_{\text{mod}} \) can be obtained from the \( \mu \text{CRL} \) specification of the SWP without the hiding operator, by means of a linearisation algorithm presented in [17]. \( N_{\text{mod}} \) contains five extra parameters: \( e:D \) and \( g,g',h,h':\text{Nat} \). Intuitively, \( g \) (resp. \( g' \)) equals zero when medium \( K \) (resp. \( L \)) is inactive, equals one when \( K \) (resp. \( L \)) just received a datum, and equals two if \( K \) (resp. \( L \)) decides to pass on this datum. Furthermore, \( e \) (resp. \( h \)) equals the datum that is being sent from \( S \) to \( R \) (resp. the position of this datum in the sending window) while \( g = 0 \), and equals the dummy value \( d_0 \) (resp. 0) while \( g = 0 \). Finally \( h' \) equals the first empty position in the receiving window while \( g' = 0 \) and equals 0 while \( g' = 0 \). Furthermore, data arguments are stripped from communication actions, and these actions are renamed to a fresh action c. For the sake of presentation, we only present parameters whose values are changed.

\[
N_{\text{mod}}(\ell; \text{Nat}, m; \text{Nat}, q; \text{Buf}, \ell'; \text{Nat}, q'; \text{Buf}, g; \text{Nat}, e; D, h; \text{Nat}, g'; \text{Nat}, h'; \text{Nat}) \\
\approx \sum_d \lambda r_A(d) N_{\text{mod}}(m := S(m) \parallel \langle \rangle, q := \text{in}(d, m, q)) < \text{in-window}(\ell, m, (\ell + n) \parallel \langle \rangle) > \delta \\
+ \sum_k \text{Nat} c N_{\text{mod}}(g := 1, e := \text{retrieve}(k, q), h := k) < \text{test}(k, q) \wedge g = 0 > \delta \\
+ j N_{\text{mod}}(g := 0, e := d_0, h := 0) \wedge g = 1 > \delta \\
+ c N_{\text{mod}}(g := 2) < g = 1 > \delta \\
+ c N_{\text{mod}}(q' := \text{in}(e, h, q'), g := 0, e := d_0, h := 0) < \text{in-window}(\ell', h, (\ell' + n) \parallel \langle \rangle) \\
\wedge g = 2 > \delta \\
+ c N_{\text{mod}}(q := 0, e := d_0, h := 0) < \neg \text{in-window}(\ell', h, (\ell' + n) \parallel \langle \rangle) \wedge g = 2 > \delta \\
+ s_D(\text{retrieve}(\ell', q')) N_{\text{mod}}(\ell := S(\ell), q' := \text{remove}(\ell', q')) < \text{test}(\ell', q') > \delta \\
+ c N_{\text{mod}}(g' := 1, h' := \text{next-empty}(\ell', q')) < g' = 0 > \delta \\
+ j N_{\text{mod}}(g' := 0, h' := 0) < g' = 1 > \delta \\
+ j N_{\text{mod}}(g' := 2) < g' = 1 > \delta \\
+ c N_{\text{mod}}(\ell := h', q := \text{release}(\ell', q'), g' := 0, h' := 0) < g' = 2 > \delta 
\]
Theorem 2.

\[ \tau(\partial_H(S(0,0,[] \parallel R(0,[]) \parallel K \parallel L)) \Leftrightarrow \tau_{\langle e,j \rangle}(N_{\text{mod}}(0,0,[],0,[],0,d_0,0,0,0)). \]

The specification of \( N_{\text{nonmod}} \) is obtained by eliminating all occurrences of \( \langle 2n \rangle \) from \( N_{\text{mod}} \), and replacing \( \text{in-window}(\ell,m,(\ell+n)\langle 2n \rangle) \) by \( m < \ell + n \) and \( \text{in-window}(\ell',h,(\ell' + n)\langle 2n \rangle) \) by \( \ell' \leq h < \ell' + n \).

Theorem 3. \( N_{\text{mod}}(0,0,0,0,0,0,0,0,0) \Leftrightarrow N_{\text{nonmod}}(0,0,0,0,0,0,0,0,0) \).

The proof of Theorem 2, using a linearisation algorithm [17] and a simple renaming, is omitted. The proof of Theorem 3 is shown in Section 7.1.

6 Properties of Data and Invariants of \( N_{\text{nonmod}} \)

Lemma 1 collects results on modulo arithmetic related to buffers. Simpler lemmas on modulo arithmetic, buffers, the \( \text{next-empty} \) operation, and lists can be found in the full version of this paper [8]. We use those lemmas without mention.

Lemma 1. The lemmas below hold for modulo arithmetic related to buffers.

1. \( \forall j: \text{Nat}(\text{test}(j,q) \rightarrow i < j + n \land i \leq k \leq i + n \rightarrow \text{test}(k,q) = \text{test}(k\langle 2n \rangle,q\langle 2n \rangle) \)
2. \( \forall j: \text{Nat}(\text{test}(j,q) \rightarrow i < j + n \land \text{test}(k,q) \rightarrow \text{retrieve}(k,q) = \text{retrieve}(k\langle 2n \rangle,q\langle 2n \rangle) \)
3. \( i < k \leq i + n \rightarrow \text{in-window}(i\langle 2n \rangle,k\langle 2n \rangle,(i + n)\langle 2n \rangle) \)
4. \( \text{in-window}(i\langle 2n \rangle,k\langle 2n \rangle,(i + n)\langle 2n \rangle) \rightarrow k + n < i \lor i \leq k < i + n \lor k \geq i + 2n \)
5. \( \forall j: \text{Nat}(\text{test}(j,q) \rightarrow i < j + n \land \text{test}(k,q\langle 2n \rangle) \rightarrow \text{in-window}(i\langle 2n \rangle,k,(i + n)\langle 2n \rangle) \)

Invariants of a system are properties of data that are satisfied throughout the reachable state space of the system. Lemma 2 collects 9 invariants of \( N_{\text{nonmod}} \) that are needed in the correctness proofs in the current paper.

Lemma 2. The invariants below hold for \( N_{\text{nonmod}}(\ell,m,q,\ell',q',g,e,h,g',h') \).

1. \( \max(h',\ell) \leq \text{next-empty}(\ell',q') \)
2. \( g \neq 0 \rightarrow h < m \)
3. \( \text{next-empty}(\ell',q') \leq \min(m,\ell' + n) \)
4. \( \text{test}(i,q) \rightarrow \ell \leq i < m \)
5. \( \ell \leq m \leq \ell + n \leq \ell' + 2n \)
6. \( g \neq 0 \rightarrow \text{next-empty}(\ell',q') \leq h + n \)
7. \( g \neq 0 \land \text{test}(h,q) \rightarrow \text{retrieve}(h,q) = e \)
8. \( g \neq 0 \land \text{test}(h,q') \rightarrow \text{retrieve}(h,q') = e \)
9. \( \ell \leq i \land j \leq \text{next-empty}(i,q') \rightarrow q[i..j] = q'[i..j] \)

7 Correctness of \( N_{\text{mod}} \)

7.1 Equality of \( N_{\text{mod}} \) and \( N_{\text{nonmod}} \)

In this section we present a proof of Theorem 3. It suffices to prove that for all \( \ell,m,\ell',h,h': \text{Nat},q,q': \text{Buf},e: \Delta \) and \( g,g' \leq 2, \)

\[ N_{\text{mod}}(\ell|2n,m|2n,q|2n,\ell'|2n,q'|2n,g,e,h|2n,g',h'|2n) \]

\[ \Leftrightarrow N_{\text{nonmod}}(\ell|2n,m|2n,q|2n,\ell'|2n,q'|2n,g,e,h|2n,g',h'|2n) \]
Proof. We show that $N_{\text{mod}}(\ell|2n, m|2n, q|2n, \ell'|2n, q'|2n, g, e, h|2n, g', h'|2n)$ is a solution for the defining equation of $N_{\text{nonmod}}(\ell, m, q, \ell', q', g, e, h, g', h')$. Hence, we must derive the following equation.

$$\begin{align*}
N_{\text{mod}}(\ell|2n, m|2n, q|2n, \ell'|2n, q'|2n, g, e, h|2n, g', h'|2n)
\approx & \sum_{d, \Delta} r_{A}(d) \cdot N_{\text{mod}}(m:=S(m)|2n, q:=\text{in}(d, m, q)|2n) < m < \ell + n \triangleright \delta \\
+ & \sum_{k, \text{Nat}} c \cdot N_{\text{mod}}(g:=1, e:=\text{retrieve}(k, q), h:=k|2n) < \text{test}(k, q) \land g = 0 \triangleright \delta \\
+ & j \cdot N_{\text{mod}}(g:=0, e:=d_0, h:=0) < g = 1 \triangleright \delta \\
+ & j \cdot N_{\text{mod}}(g:=2) < g = 1 \triangleright \delta \\
+ & c \cdot N_{\text{mod}}(q':=\text{in}(e, h, q')|2n, g:=0, e:=d_0, h:=0) < \ell' \leq h < \ell' + n \land g = 2 \triangleright \delta \\
+ & c \cdot N_{\text{mod}}(g:=0, e:=d_0, h:=0) < \neg(\ell' \leq h < \ell' + n) \land g = 2 \triangleright \delta \\
+ & \text{sp}(\text{retrieve}(\ell', q')) \cdot N_{\text{mod}}(\ell':=S(\ell')|2n, q':=\text{remove}(\ell', q')|2n) < \text{test}(\ell', q') \triangleright \delta \\
+ & c \cdot N_{\text{mod}}(g':=1, h':=\text{next-empty}(\ell', q')|2n) < g' = 0 \triangleright \delta \\
+ & j \cdot N_{\text{mod}}(g':=0, h':=0) < g' = 1 \triangleright \delta \\
+ & j \cdot N_{\text{mod}}(g':=2) < g' = 1 \triangleright \delta \\
+ & c \cdot N_{\text{mod}}(\ell':=h'|2n, q:=\text{release}(\ell, h', q)|2n, g:=0, h':=0) < g' = 2 \triangleright \delta
\end{align*}$$

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

$$\begin{align*}
N_{\text{mod}}(\ell|2n, m|2n, q|2n, \ell'|2n, q'|2n, g, e, h|2n, g', h'|2n)
\approx & \sum_{d, \Delta} r_{A}(d) \cdot N_{\text{mod}}(m:=S(m)|2n, q:=\text{in}(d, m, q)|2n) < \text{in-window}(\ell|2n, m|2n, (\ell'|2n + n)|2n) \triangleright \delta \\
+ & \sum_{k, \text{Nat}} c \cdot N_{\text{mod}}(g:=1, e:=\text{retrieve}(k, q)|2n, h:=k) < \text{test}(k, q)|2n \land g = 0 \triangleright \delta \\
+ & j \cdot N_{\text{mod}}(g:=0, e:=d_0, h:=0) < g = 1 \triangleright \delta \\
+ & j \cdot N_{\text{mod}}(g:=2) < g = 1 \triangleright \delta \\
+ & c \cdot N_{\text{mod}}(q':=\text{in}(e, h|2n, q'|2n), g:=0, e:=d_0, h:=0) < \text{in-window}(\ell'|2n, h|2n, (\ell'|2n + n)|2n) \land g = 2 \triangleright \delta \\
+ & c \cdot N_{\text{mod}}(g:=0, e:=d_0, h:=0) < \neg\text{in-window}(\ell'|2n, h|2n, (\ell'|2n + n)|2n) \land g = 2 \triangleright \delta \\
+ & \text{sp}(\text{retrieve}(\ell'|2n, q'|2n)) \cdot N_{\text{mod}}(\ell':=S(\ell'|2n), q':=\text{remove}(\ell'|2n, q'|2n)) < \text{test}(\ell'|2n, q'|2n) \triangleright \delta \\
+ & c \cdot N_{\text{mod}}(g':=1, h':=\text{next-empty}(\ell'|2n, q'|2n)) < g' = 0 \triangleright \delta \\
+ & j \cdot N_{\text{mod}}(g':=0, h':=0) < g' = 1 \triangleright \delta \\
+ & j \cdot N_{\text{mod}}(g':=2) < g' = 1 \triangleright \delta \\
+ & c \cdot N_{\text{mod}}(\ell:=h'|2n, q:=\text{release}(\ell|2n, h'|2n, q|2n), g:=0, h':=0) < g' = 2 \triangleright \delta
\end{align*}$$

To equate the eleven summands in both specifications, we obtain a number of proof digations. Here, we focus on summands A, B, and E.

$m < \ell + n = \text{in-window}(\ell|2n, m|2n, (\ell|2n + n)|2n)$.

$m < \ell + n \rightarrow \ell \leq m < \ell + n \quad \text{(Inv. 2.5)} \rightarrow \quad \text{in-window}(\ell|2n, m|2n, (\ell + n)|2n)$ (Lem. 1.3). Reversely, $\text{in-window}(\ell|2n, m|2n, (\ell + n)|2n) \rightarrow m + n < \ell \forall \ell \leq m < \ell + n \vee m \geq \ell + 2n$ (Lem. 1.4) $\rightarrow m < \ell + n \quad \text{(Inv. 2.5)}$. Since $(\ell + n)|2n = (\ell|2n + n)|2n$, we have $m < \ell + n = \text{in-window}(\ell|2n, m|2n, (\ell|2n + n)|2n)$. 

Below we equate the entire summand \(B\) of the two specifications. The conjunction \(g = 0\) and the argument \(g:=1\) of summand \(B\) are omitted, because they are irrelevant for this derivation.

By Inv. 2.4 and 2.5, \(\text{test}(j, q) \rightarrow \ell \leq j < \ell + n\). So by Lem. 1.5, \(\text{test}(k', q'_{2n})\) implies \(\text{in-window}(\ell_{2n}, k', (\ell + n)_{2n})\). \(\text{test}(k', q'_{2n})\) implies \(k' = k'_{2n}\), and by Lem. 1.4, \(k' + n < \ell_{2n} \vee \ell_{2n} \leq k' < \ell_{2n} + n \vee k' \geq \ell + 2n\). \(k' = k'_{2n} < 2n\) implies \(k' + n < \ell_{2n} \vee \ell_{2n} \leq k' < \ell_{2n} + n\).

\[
\sum_{k':\text{Nat}} \text{N}_{\text{mod}}(e:=\text{retrieve}(k, q), h:=k_{2n}) < \text{test}(k, q) \triangleright \delta
\]

\(\approx \sum_{k':\text{Nat}} \text{N}_{\text{mod}}(e:=\text{retrieve}(k, q), h:=k_{2n}) < \text{test}(k, q) \land \ell \leq k < \ell + n \triangleright \delta \quad \text{(Inv. 2.4, 2.5)}
\]

\(\approx \sum_{k:\text{Nat}} \sum_{k':\text{Nat}} \text{N}_{\text{mod}}(e:=\text{retrieve}(k|2n, q|2n), h:=k_{2n}) < \text{test}(k|2n, q|2n) \land \ell \leq k < \ell + n \triangleright \delta \quad \text{(Lem. 1.1, 1.2)}
\]

\(\approx \sum_{k':\text{Nat}} \sum_{k:\text{Nat}} \text{N}_{\text{mod}}(e:=\text{retrieve}(k', q|2n), h:=k') < \text{test}(k', q|2n) \land \ell \leq k < \ell + n \land k' = k|2n \triangleright \delta \quad \text{(sum elimination)}
\]

\(\approx \sum_{k':\text{Nat}} \sum_{k:\text{Nat}} \text{N}_{\text{mod}}(e:=\text{retrieve}(k', q|2n), h:=k') < \text{test}(k', q|2n) \land \ell | 2n = (\ell | \text{div} 2n) 2n + k' \land k' < \ell|2n + \ell \land k' = k|2n \triangleright \delta \quad \text{(see above)}
\]

Let \(g = 2\). We have \(\ell' \leq h < \ell' + n = \text{in-window}(\ell'|2n, h|2n, (\ell' + n)|2n)\).

Equality of other summands can be derived without much difficulty. Hence, we prove that \(\text{N}_{\text{mod}}(\ell|2n, m|2n, q'|2n, \ell'|2n, q'|2n, g, e, h|2n, g', h'|2n)\) is a solution for the specification of \(\text{N}_{\text{nonmod}}(\ell, m, q, \ell', q', g, e, h, g', h')\). By CL-RSP, they are strongly bisimilar.

### 7.2 Correctness of \(\text{N}_{\text{nonmod}}\)

We prove that \(\text{N}_{\text{nonmod}}\) is branching bisimilar to the FIFO queue \(Z\) of capacity \(2n\) (see Section 4), using the cones and foci method [9].

Let \(\Xi\) abbreviate \(\text{Nat} \times \text{Nat} \times \text{Buf} \times \text{Nat} \times \text{Buf} \times \Delta \times \text{Nat} \times \text{Nat} \times \text{Nat}\). Furthermore, let \(\xi:\Xi\) denote \((\ell, m, q, \ell', q', g, e, h, g', h')\). The state mapping \(\phi: \Xi \rightarrow \text{List}\), which maps states of \(\text{N}_{\text{nonmod}}\) to states of \(Z\), is defined by:

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

Intuitively, \(\phi\) collects the data elements in the sending and receiving windows, starting at the first position of the receiving window (i.e., \(\ell'\)) until the first empty position in this window, and then continuing in the sending window until the
first empty position in that window (i.e., \(m\)). Note that \(\phi\) is independent of \(e, g, \ell, h, g', h'\); we therefore write \(\phi(m, q, \ell', q')\).

The focus points are those states where either the sending window is empty (meaning that \(\ell = m\)), or the receiving window is full and all data elements in the receiving window have been acknowledged, meaning that \(\ell = \ell' + n\). That is, the focus condition for \(N_{\text{nonmod}}(\ell, m, q, \ell', q', g, e, h, g', h')\) is

\[
FC(\ell, m, q, \ell', q', g, e, h, g', h') := \ell = m \lor \ell = \ell' + n
\]

**Lemma 3.** For each \(\xi : \Xi\) where the invariants in Lemma 2 hold, there is a \(\xi : \Xi\) with \(FC(\xi)\) such that \(N_{\text{nonmod}}(\xi) \subseteq \ldots \subseteq N_{\text{nonmod}}(\xi),\) where \(c_1, \ldots, c_n \in \mathcal{T}\).

**Proof.** In case \(g \neq 0\) in \(\xi\), by summands \(C, E\) and \(F\), we can perform one or two communication actions to a state where \(g = 0\). By Inv. 2.3, next-empty(\(\ell', q'\)) \(\leq\) \(\min\{m, \ell' + n\}\).

We prove by induction on \(\min\{m, \ell' + n\}\) -- next-empty(\(\ell', q'\)) that for each state \(\xi'\) where \(g = 0\) and the invariants in Lemma 2 hold, a focus point can be reached.

**Base Case:** next-empty(\(\ell', q'\)) = \(\min\{m, \ell' + n\}\).

In case \(g' \neq 0\) in \(\xi\), by summands \(I\) and \(K\), we can perform communication actions to a state where \(g' = 0\) and next-empty(\(\ell', q'\)) = \(\min\{m, \ell' + n\}\). By summands \(H\), \(J\) and \(K\) we can perform three communication actions to a state \(\xi\) where \(\ell = h' =\) next-empty(\(\ell', q'\)) = \(\min\{m, \ell' + n\}\). Then \(\ell = m\) or \(\ell = \ell' + n\), so \(FC(\xi)\).

**Induction Case:** next-empty(\(\ell', q'\)) < \(\min\{m, \ell' + n\}\).

By Inv. 2.1, \(\ell \leq\) next-empty(\(\ell', q'\)) < \(m\). By Inv. 2.4, test(\(\text{next-empty}(\ell', q'), q\)). Furthermore, \(\ell' \leq\) next-empty(\(\ell', q'\)) < \(\ell' + n\). Hence, by summands \(B, D\) and \(E\) from \(\xi\) we can perform three communication actions to a state \(\xi''\). In \(\xi''\), \(g = 0\), and in comparison to \(\xi', m\) and \(\ell'\) remain the same, while \(q' =\) \(\text{in}(d, \text{next-empty}(\ell', q'), q')\) where \(d\) denotes retrieve(\(\text{next-empty}(\ell', q'), q\)). Since next-empty(\(\ell', \text{in}(d, \text{next-empty}(\ell', q'), q')\)) = next-empty(S\(\text{next-empty}(\ell', q'), q')\), \(\xi''\) > next-empty(\(\ell', q'\)), we can apply the induction hypothesis to conclude that from \(\xi''\) a focus point can be reached.

**Theorem 4.** For all \(e : \Delta\), \(\tau(e, f)\) (\(N_{\text{nonmod}}(0, 0, [0], [0], 0, e, 0, 0, 0) \Leftrightarrow Z(\langle\rangle)\)).

**Proof.** By the cones and foci method we obtain the following matching criteria (cf. [9]). Trivial matching criteria are left out.

\[
\begin{align*}
\text{I.1: } & \quad \ell' \leq h < \ell' + n \land g = 2 \rightarrow \phi(m, q, \ell', q') = \phi(m, q, \ell', \text{in}(e, h, q')) \\
\text{I.2: } & \quad g' = 2 \rightarrow \phi(m, q, \ell', q') = \phi(m, \text{release}(\ell', h', q), \ell', q') \\
\text{I.1.1: } & \quad (\ell = m \lor \ell = \ell' + n) \land \text{length}(\phi(m, q, \ell', q')) < 2n \\
\text{I.2.2: } & \quad \text{test}(\ell', q') \rightarrow \text{length}(\phi(m, q, \ell', q')) > 0 \\
\text{III.1: } & \quad (\ell = m \lor \ell = \ell' + n) \land \text{length}(\phi(m, q, \ell', q')) < 2n \rightarrow m < \ell + n \\
\text{III.2: } & \quad (\ell = m \lor \ell = \ell' + n) \land \text{length}(\phi(m, q, \ell', q')) > 0 \rightarrow \text{test}(\ell', q') \\
\text{IV.1: } & \quad m < \ell + n \rightarrow \phi(S(m), \text{in}(d, m, q), \ell', q') = \text{append}(d, \phi(m, q, \ell', q')) \\
\text{IV.2: } & \quad \text{test}(\ell', q') \rightarrow \phi(m, q, S(\ell'), \text{remove}(\ell', q', q')) = \text{tail}(\phi(m, q, \ell', q'))
\end{align*}
\]

I.1 \(\ell' \leq h < \ell' + n \land g = 2 \rightarrow \phi(m, q, \ell', q') = \phi(m, q, \ell', \text{in}(e, h, q'))\).

**Case 1:** \(h \neq \text{next-empty}(\ell', q')\).

Let \(g = 2\). Since next-empty(\(\ell', \text{in}(e, h, q')\)) = next-empty(\(\ell', q'\)), it follows that \(\phi(m, q, \ell', \text{in}(e, h, q')) = \text{in}(e, h, q')\).[\(\ell'\).next-empty(\(\ell', q'\))] = \(q'\).[\(\ell'\).next-empty(\(\ell', q'\))].
CASE 1.2: \(-(\ell' \leq h \leq \text{next-empty}(\ell', q')).\)
\[
in(e, h, q')[\ell'..\text{next-empty}(\ell', q')] = q'[\ell'..\text{next-empty}(\ell', q')].
\]
CASE 2: \(h = \text{next-empty}(\ell', q').\)
Let \(g = 2\). The derivation splits into two parts.
(1) \(\text{in}(e, h, q')[\ell'..h] = q'[\ell'..h].\)
(2) By Inv 2.1, \(\ell \leq h\), and by Inv 2.2 together with \(g = 2\), \(h < m\). Thus, by Inv 2.4, \(\text{test}(h, q).\) So by Inv 2.7 together with \(g = 2\), \(\text{retrieve}(h, q) = e.\) Hence,

\[
in(e, h, q'[\ell'..\text{next-empty}(S(h), q')])
= \text{in}(e, \text{in}(e, h, q'[\ell'..\text{next-empty}(S(h), q')]))
= \text{in}(e, q'[\ell'..\text{next-empty}(S(h), q')])
= \text{in}(e, q[S(h)..'\text{next-empty}(S(h), q')])
= q[h..\text{next-empty}(S(h), q')])
\]

Finally, we combine (1) and (2). We recall that \(h = \text{next-empty}(\ell', q').\)

\[
in(e, h, q'[\ell'..\text{next-empty}(\ell', \text{in}(e, h, q'))])
= \text{in}(e, \text{in}(e, h, q'[\ell'..\text{next-empty}(S(h), q')]))
= \text{in}(e, q'[\ell'..\text{next-empty}(S(h), q')])
= \text{in}(e, q[S(h)..'\text{next-empty}(S(h), q')])
= q'[\ell'..h]++q[h..m]
\]

I.2 \(g' = 2 \rightarrow \phi(m, q, \ell', q') = \phi(m, \text{release}(\ell, h', q), \ell', q').\)
By Inv 2.1, \(h' \leq \text{next-empty}(\ell', q').\)
So \(\text{release}(\ell, h', q)[\text{next-empty}(\ell', q')..m] = q'[\text{next-empty}(\ell', q')..m].\)

II.1 \(m < \ell + n \rightarrow \text{length}(\phi(m, q, \ell', q')) < 2n.\)
Let \(m < \ell + n.\) By Inv 2.3, \(\text{next-empty}(\ell', q') \leq \ell' + n.\) Hence,

\[
\text{length}(q'[\ell'..\text{next-empty}(\ell', q'))++)q[\text{next-empty}(\ell', q')..m])
= \text{length}(q'[\ell'..\text{next-empty}(\ell', q'))++)q[\text{next-empty}(\ell', q')..m])
= (\text{next-empty}(\ell', q') \sim \ell') + (m - \text{next-empty}(\ell', q'))
\leq n + (m - \ell)
< 2n
\]

II.2 \(\text{test}(\ell', q') \rightarrow \text{length}(\phi(m, q, \ell', q')) > 0.\)
\(\text{test}(\ell', q')\) yields \(\text{next-empty}(\ell', q') = \text{next-empty}(S(\ell'), q') \geq S(\ell').\) Hence,

\[
\text{length}(\phi(m, q, \ell', q')) = (\text{next-empty}(\ell', q') \sim \ell') + (m - \text{next-empty}(\ell', q')) > 0.
\]

III.1 \((\ell = m \lor \ell = \ell' + n) \land \text{length}(\phi(m, q, \ell', q')) < 2n \rightarrow m < \ell + n.\)
CASE 1: \(\ell = m.\)
Then \(m < \ell + n\) holds trivially.
CASE 2: \(\ell = \ell' + n.\)
By Inv 2.3, \(\text{next-empty}(\ell', q') \leq \ell' + n.\) Hence,

\[
\text{length}(\phi(m, q, \ell', q'))
= (\text{next-empty}(\ell', q') \sim \ell') + (m - \text{next-empty}(\ell', q'))
\leq ((\ell' + n) \sim \ell') + (m - \ell)
= n + (m - \ell)
\]
So \(\text{length}(\phi(m, q, \ell', q')) < 2n\) implies \(m < \ell + n.\)
III.2 \( (\ell = m \lor \ell = \ell + n) \land \text{length}(\phi(m, q, \ell', q')) > 0 \rightarrow \text{test}(\ell', q') \).

**CASE 1:** \( \ell = m \).
Since \( m - \text{next-empty}(\ell', q') \leq (m - \ell) \) (Inv. 2.1) = 0, we have
\[
\text{length}(\phi(m, q, \ell', q')) = \text{next-empty}(\ell', q') + \ell'.
\]
Hence, \( \text{length}(\phi(m, q, \ell', q')) > 0 \) yields \( \text{next-empty}(\ell', q') > \ell' \), which implies
\( \text{test}(\ell', q') \).

**CASE 2:** \( \ell = \ell + n \).
Then by Inv. 2.1, \( \text{next-empty}(\ell', q') \geq \ell' + n \), which implies \( \text{test}(\ell', q') \).

IV \( \text{test}(\ell', q') \rightarrow \text{retrievel}(\ell', q') = \text{top}(\phi(m, q, \ell', q')) \).
\( \text{test}(\ell', q') \) implies \( \text{next-empty}(\ell', q') = \text{next-empty}(S(\ell'), q') \geq S(\ell') \).
So \( q'[\ell'..\text{next-empty}(\ell', q')] = \text{in}(\text{retrievel}(\ell', q'), q'[S(\ell')..\text{next-empty}(\ell', q')]) \).
Hence, \( \text{top}(\phi(m, q, \ell', q')) = \text{retrievel}(\ell', q') \).

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

V.2 \( \text{test}(\ell', q') \rightarrow \phi(m, q, S(\ell'), \text{remove}(\ell', q')) = \text{tail}(\phi(m, q, \ell', q')) \).
\( \text{test}(\ell', q') \) implies \( \text{next-empty}(\ell', q') = \text{next-empty}(S(\ell'), q') \). Hence,
\[
\text{remove}(\ell', q')[S(\ell')..\text{next-empty}(S(\ell'), \text{remove}(\ell', q'))] + q[\text{next-empty}(S(\ell'), q')..m]
\]
\[
= \text{remove}(\ell', q')[S(\ell')..\text{next-empty}(S(\ell'), q')] + q[\text{next-empty}(S(\ell'), q')..m]
\]
\[
= \text{remove}(\ell', q')[S(\ell')..\text{next-empty}(\ell', q')] + q[\text{next-empty}(\ell', q')..m]
\]
\[
= \text{tail}(q'[\ell'..\text{next-empty}(\ell', q')] + q[\text{next-empty}(\ell', q')..m])
\]

### 7.3 Correctness of the Sliding Window Protocol

Finally, we can prove Theorem 1.

**Proof.**

\[
\tau_2(\partial_2(S(0, 0, []) \parallel R(0, []) \parallel K \parallel L))
\]
\[
\equiv \tau_{\{c,j\}}(N_{\text{mod}}(0, 0, 0, 0, 0, 0, 0, 0)) \quad \text{(Thm. 2)}
\]
\[
\equiv \tau_{\{c,j\}}(N_{\text{nonmod}}(0, 0, 0, 0, 0, 0, 0, 0)) \quad \text{(Thm. 3)}
\]
\[
\equiv Z() \quad \text{(Thm. 4)}
\]

### 8 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.

**Infinite 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. Hailpern [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.* Richier 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. Kaivola [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 [5] specified a SWP using higher or-
der logic, and manually proved and mechanically checked safety properties using
HOL. (Van de Snepscheut [39] noted that what Cardell-Oliver claims to be a liven-
ness 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 cor-
rectness 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 ar-
bitrary 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. [7] used a timed state machine in
PVS to specify a SWP in which messages can be lost, duplicated or reordered,
and proved some safety properties with the mechanical support of PVS.
References


