Verification of a Sliding Window Protocol in μCRL

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 $2n$. The correctness consists of showing 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 has been checked in the theorem prover PVS.**

Keywords— **μCRL, 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 $2n$ (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 *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 $2n$). 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 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 Brunekreef [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 μCRL [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 $2n$. 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 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 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 μCRL 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 `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 μCRL. In Section III, the data types needed for specifying the SWP and its external behavior are presented. Section IV features the μCRL 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. μCRL

μCRL [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 μCRL 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 *Bool* with constants t and f, and the usual connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$ and $\Leftrightarrow$. For a boolean b, we abbreviate $b = \mathsf{t}$ to b and $b = \mathsf{f}$ to $\neg b$.

The data types needed for our μCRL specification of a SWP are presented in Section III. In this section we focus on the process part of μCRL. 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 $\mathcal{A}$: δ represents deadlock, and τ a hidden action. These two actions never carry data parameters. $p{\cdot}q$ denotes sequential composition and $p + q$ non-deterministic choice. Summation $\sum_{d:D} p(d)$ provides the possibly infinite choice over a data type D, and the conditional construct $p \lhd b \rhd q$ with b a data term of sort *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 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 $\partial_{\mathcal{H}}(p)$, which renames all occurrences in p of actions from the set $\mathcal{H}$ into δ, can be used to force actions into communication. Hiding $\tau_{\mathcal{I}}(p)$ renames all occurrences in p of actions from the set $\mathcal{I}$ into τ. 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

$$X(d_1{:}D_1,\ldots,d_n{:}D_n) \approx \\ \sum_{i=1}^{\ell} \sum_{z_i:Z_i} a_i(e_1^i,\ldots,e_{m_i}^i)\cdot X(d_1^i,\ldots,d_n^i) \lhd b_i \rhd \delta.$$

To each μCRL specification belongs a directed graph, called a labeled transition system (LTS), which is defined by the structural operational semantics of μCRL (see [16]). In this labeled transition system, the states are process terms, and the edges are labeled with parameterized actions. Branching bisimulation $\leftrightarrow_b$ [10] and strong bisimulation $\leftrightarrow$ [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 μCRL from [15] is sound modulo branching bisimulation equivalence, meaning that if $p \approx q$ can be derived from it then $p \leftrightarrow_b q$.

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

The goal of this paper is to prove that the initial state of the forthcoming μCRL 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 μCRL 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,

$$\sum_{d:D} p(d) \lhd d = e \wedge b \rhd \delta \;\leftrightarrow\; p(e) \lhd b \rhd \delta.$$

• *CL-RSP* [4] states that the solutions of a linear μCRL specification that does not contain any infinite τ 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 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* ϕ 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. We give the definition of matching criteria and the general thoerem 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

$$\begin{array}{rl} & X(d{:}D) \\ = & \sum_{a\in Act\cup\{\tau\}} \sum_{e:E_a} a(f_a(d,e))\cdot X(g_a(d,e)) \\ & \lhd h_a(d,e) \rhd \delta. \end{array}$$

Furthermore, assume an LPE Y without hidden actions

$$\begin{array}{rl} & Y(d'{:}D') \\ = & \sum_{a\in Act} \sum_{e:E_a} a(f'_a(d',e))\cdot Y(g'_a(d',e)) \\ & \lhd h'_a(d',e) \rhd \delta. \end{array}$$

Definition II.2 (Matching criteria) A state mapping $\phi : D \to D'$ satisfies the *matching criteria* for $d{:}D$ if for all $a \in Act$:
I $\quad \forall e{:}E_a\,(h_\tau(d,e) \Rightarrow \phi(d) = \phi(g_\tau(d,e)))$;
II $\quad \forall e{:}E_a\,(h_a(d,e) \Rightarrow h'_a(\phi(d),e))$;
III $\quad FC(d) \Rightarrow \forall e{:}E_a\,(h'_a(\phi(d),e) \Rightarrow h_a(d,e))$;
IV $\quad \forall e{:}E_a\,(h_a(d,e) \Rightarrow f_a(d,e) = f'_a(\phi(d),e))$;
V $\quad \forall e{:}E_a\,(h_a(d,e) \Rightarrow \phi(g_a(d,e)) = g'_a(\phi(d),e))$.

Matching criterion I requires that the τ-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 $\mathcal{I} : D \to Bool$ be an invariant for X. Suppose that for all $d{:}D$ with $\mathcal{I}(d)$,

1. $\phi : D \to D'$ satisfies the matching criteria for d, and
2. there is a $\hat{d}{:}D$ such that $FC(\hat{d})$ and $d \xrightarrow{\tau} \cdots \xrightarrow{\tau} \hat{d}$ in the LTS for X.

Then for all $d{:}D$ with $\mathcal{I}(d)$,

$$X(d) \leftrightarrow_b Y(\phi(d)).$$

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 *Bool* of booleans.

$$\begin{array}{l} \mathsf{t}, \mathsf{f} :\to Bool \\ \wedge, \vee : Bool \times Bool \to Bool \\ \neg : Bool \to Bool \\ \Rightarrow, \Leftrightarrow : Bool \times Bool \to Bool \end{array}$$

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

$$\begin{array}{lcl} b \wedge \mathsf{t} & = & b \\ b \wedge \mathsf{f} & = & \mathsf{f} \\ b \vee \mathsf{t} & = & \mathsf{t} \\ b \vee \mathsf{f} & = & b \\ \neg\mathsf{t} & = & \mathsf{f} \\ \neg\mathsf{f} & = & \mathsf{t} \\ b \Rightarrow b' & = & b' \vee \neg b \\ b \Leftrightarrow b' & = & (b \Rightarrow b') \wedge (b' \Rightarrow b) \end{array}$$

B. If-then-else and Equality

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

$$if : Bool \times D \times D \to D$$

with as defining equations

$$\begin{array}{lcl} if(\mathsf{t}, d, e) & = & d \\ if(\mathsf{f}, d, e) & = & e \end{array}$$

Furthermore, for each data type D in this paper one can easily define a mapping $eq : D \times D \to Bool$ 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 *Nat* of natural numbers.

$$\begin{array}{l} 0 :\to Nat \\ S : Nat \to Nat \\ +, \dot{-}, \cdot : Nat \times Nat \to Nat \\ \leq, <, \geq, > : Nat \times Nat \to Bool \end{array}$$

0 denotes zero and $S(n)$ the successor of n. The infix operations $+$, $\dot{-}$ 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 $\neg(i = j)$ is abbreviated to $i \neq j$.

$$\begin{array}{lcl} i + 0 & = & i \\ i + S(j) & = & S(i + j) \\ i \dot{-} 0 & = & i \\ 0 \dot{-} i & = & 0 \\ S(i) \dot{-} S(j) & = & i \dot{-} j \\ i \cdot 0 & = & 0 \\ i \cdot S(j) & = & (i \cdot j) + i \\ 0 \leq i & = & \mathsf{t} \\ S(i) \leq 0 & = & \mathsf{f} \\ S(i) \leq S(j) & = & i \leq j \\ 0 < S(i) & = & \mathsf{t} \\ i < 0 & = & \mathsf{f} \\ S(i) < S(j) & = & i < j \\ i \geq j & = & \neg(j < i) \\ i > j & = & \neg(j \leq i) \end{array}$$

We take as binding convention: $\{=, \neq\} > \{\cdot\} > \{+, \dot{-}\} > \{\leq, <, \geq, >\} > \{\neg\} > \{\wedge, \vee\} > \{\Rightarrow, \Leftrightarrow\}$.

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:

$$\begin{array}{l} | : Nat \times Nat \to Nat \\ div : Nat \times Nat \to Nat \end{array}$$

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

$$\begin{array}{lcl} i|_n & = & if(i < n, i, (i \dot{-} n)|_n) \\ i \, div \, n & = & if(i < n, 0, S((i \dot{-} n) \, div \, n)) \end{array}$$

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 Δ 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 $i{:}Nat$, representing that position (or sequence number) i of the buffer is occupied by datum d. The data type *Buf* is specified as follows, where $[]$ denotes the empty buffer:

$$[] :\rightarrow Buf$$
$$inb : \Delta \times Nat \times Buf \rightarrow Buf$$

$q|_n$ denotes buffer q with all sequence numbers taken modulo n.

$$\begin{array}{lcl} []|_n & = & [] \\ inb(d,i,q)|_n & = & inb(d, i|_n, q|_n) \end{array}$$

$test(i, q)$ produces t if and only if position i in q is occupied, $retrieve(i, q)$ produces the datum that resides at position i in buffer q (if this position is occupied),[1] and $remove(i, q)$ is obtained by emptying position i in buffer q.

$$\begin{array}{lcl} test(i,[]) & = & \mathsf{f} \\ test(i, inb(d,j,q)) & = & i{=}j \vee test(i,q) \\ retrieve(i, inb(d,j,q)) & = & if(i{=}j, d, retrieve(i,q)) \\ remove(i,[]) & = & [] \\ remove(i, inb(d,j,q)) & = & if(i{=}j, remove(i,q), \\ & & \quad inb(d,j,remove(i,q))) \end{array}$$

$release(i, j, q)$ is obtained by emptying positions i up to j in q. $release|_n(i, j, q)$ does the same modulo n.

$$\begin{array}{lcl} release(i,j,q) & = & if(i \geq j, q, \\ & & \quad release(S(i), j, remove(i,q))) \\ release|_n(i,j,q) & = & if(i|_n{=}j|_n, q, \\ & & \quad release|_n(S(i), j, remove(i,q))) \end{array}$$

$next\text{-}empty(i, q)$ produces the first empty position in q, counting upwards from sequence number i onward. $next\text{-}empty|_n(i, q)$ does the same modulo n.

$$\begin{array}{lcl} next\text{-}empty(i,q) & = & if(test(i,q), \\ & & \quad next\text{-}empty(S(i),q), i) \\ next\text{-}empty|_n(i,q) & = & if(next\text{-}empty(i|_n, q|_n) < n, \\ & & \quad next\text{-}empty(i|_n, q|_n), \\ & & \quad next\text{-}empty(0, q|_n)) \end{array}$$

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

$$in\text{-}window(i,j,k) = i \leq j < k \vee k < i \leq j \vee j < k < i$$

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

$$\begin{array}{lcl} max([]) & = & 0 \\ max(inb(d,i,q)) & = & if(i \geq max(q), i, max(q)) \end{array}$$

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 *MedK*. It represents a list of pairs (d, i) with a datum $d{:}\Delta$ and its sequence number $i{:}Nat$. Let $[]^K$ denote an empty medium. $g|_n$ denotes medium g with all sequence numbers taken modulo n. $member(d, i, g)$ produces t if and only if the pair (d, i) is in g. $length(g)$ denotes the length of g. $return\text{-}dat(i, g)$ and $return\text{-}seq(i, g)$ produce the datum and the sequence number, respectively, that resides at position i in g (positions are counted from 0). For convenience, we use $last\text{-}dat(g)$ and $last\text{-}seq(g)$ to produce the datum and the sequence number, respectively, that resides at the end of g. $delete(i, g)$ is obtained by emptying position i in g. Similarly, $delete\text{-}last(g)$ 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 *MedL*. Similarly, we have the following defining equations.

G. Lists

We introduce the data type of *List* of lists, which are used in the specification of the desired external behavior of the SWP: a FIFO queue of size $2n$. Let $\langle\rangle$ denote the empty list.

$$\langle\rangle :\rightarrow List$$
$$inl : \Delta \times List \rightarrow List$$

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

$$[]^K :\to MedK$$
$$inm : \Delta \times Nat \times MedK \to MedK$$

$$\begin{array}{lcl} []^K|_n & = & []^K \\ inm(d,i,g)|_n & = & inm(d,i|_n,g|_n) \end{array}$$

$$\begin{array}{lcl}
member(d,i,[]^K) & = & \mathsf{f} \\
member(d,i,inm(e,j,g)) & = & (d=e \wedge i=j) \vee member(d,i,g) \\
length([]^K) & = & 0 \\
length(inm(d,i,g)) & = & S(length(g)) \\
return\text{-}dat(i,inm(d,j,g)) & = & if(i=0,d,return\text{-}dat(i-1,g)) \\
return\text{-}seq(i,inm(d,j,g)) & = & if(i=0,j,return\text{-}seq(i-1,g)) \\
last\text{-}dat(inm(d,i,g)) & = & if(length(g)=0,d,last\text{-}dat(g)) \\
last\text{-}seq(inm(d,i,g)) & = & if(length(g)=0,i,last\text{-}dat(g)) \\
delete(i,inm(d,j,g)) & = & if(i=0,g,inm(d,j,delete(i-1,g))) \\
delete\text{-}last(inm(d,i,g)) & = & if(length(g)=0,g,inm(d,i,delete\text{-}last(g)))
\end{array}$$

$$[]^L :\to MedL$$
$$inm : Nat \times MedL \to MedL$$

$$\begin{array}{lcl} []^L|_n & = & []^L \\ inm(i,g')|_n & = & inm(i|_n,g'|_n) \end{array}$$

$$\begin{array}{lcl}
member(i,[]^L) & = & \mathsf{f} \\
member(i,inm(j,g)) & = & i=j \vee member(d,i,g) \\
length([]^L) & = & 0 \\
length(inm(i,g')) & = & S(length(g')) \\
return\text{-}seq(i,inm(j,g')) & = & if(i=0,j,return\text{-}seq(i-1,g')) \\
last\text{-}seq(inm(i,g')) & = & if(length(g')=0,i,last\text{-}seq(g')) \\
delete(i,inm(j,g')) & = & if(i=0,g',inm(j,delete(i-1,g'))) \\
delete\text{-}last(inm(j,g')) & = & if(length(g')=0,g',inm(j,delete\text{-}last(g')))
\end{array}$$

$length(\lambda)$ denotes the length of λ, $top(\lambda)$ produces the datum that resides at the top of λ, $tail(\lambda)$ is obtained by removing the top position in λ, $append(d,\lambda)$ adds datum d at the end of λ, and $\lambda{+\!\!+}\lambda'$ represents list concatenation.

$$\begin{array}{lcl}
length(\langle\rangle) & = & 0 \\
length(inl(d,\lambda)) & = & S(length(\lambda)) \\
top(inl(d,\lambda)) & = & d \\
tail(inl(d,\lambda)) & = & \lambda \\
append(d,\langle\rangle) & = & inl(d,\langle\rangle) \\
append(d,inl(e,\lambda)) & = & inl(e,append(d,\lambda)) \\
\langle\rangle{+\!\!+}\lambda & = & \lambda \\
inl(d,\lambda){+\!\!+}\lambda' & = & inl(d,\lambda{+\!\!+}\lambda')
\end{array}$$

Furthermore, $q[i..j\rangle$ is the list containing the elements in buffer q at positions i up to but not including j.

$$q[i..j\rangle \quad = \quad if(i \geq j, \langle\rangle, inl(retrieve(i,q), q[S(i)..j\rangle))$$

IV. Sliding Window Protocol

In this section, a μCRL 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 $\mathbf{S}(\ell, m, q)$, where q is a buffer of size $2n$, ℓ 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 $\mathbf{R}(\ell', q')$, where q' is a buffer of size $2n$ and ℓ' 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 $\mathbf{K}(g, p)$, where $g{:}\mathit{MedK}$ is a buffer with unbounded capacity, and $p{:}\mathit{Nat}$ 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 $(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g))$ 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 $\mathbf{L}(g', p')$.

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

$$\tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^K,0) \parallel \mathbf{L}([]^L,0)))$$

where the set $\mathcal{H}$ consists of the read and send actions over the internal channels B, C, E, and F, namely $\mathcal{H} = \{s_\mathrm{B}, r_\mathrm{B}, s_\mathrm{C}, r_\mathrm{C}, s_\mathrm{E}, r_\mathrm{E}, s_\mathrm{F}, r_\mathrm{F}\}$, while the set $\mathcal{I}$ consists of the communication actions over these internal channels together with j, namely $\mathcal{I} = \{c_\mathrm{B}, c_\mathrm{C}, c_\mathrm{E}, c_\mathrm{F}, j\}$.

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 $r_\mathrm{A}(d)$ can be performed until the list λ contains $2n$ elements, because in that situation the sending and receiving windows will be filled. Furthermore, $s_\mathrm{D}(\mathit{top}(\lambda))$ can only be performed if λ is not empty.

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

Theorem IV.1 $\tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^K,0) \parallel \mathbf{L}([]^L,0))) \leftrightarrow_b \mathbf{Z}(\langle\rangle)$.

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 $\mathbf{M}_{mod}$, in which no parallel operators occur. $\mathbf{M}_{mod}$ can be obtained from the μCRL specification of the SWP without the hiding operator, i.e.,

$$\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^K,0) \parallel \mathbf{L}([]^L,0))$$

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

The linear specification $\mathbf{M}_{mod}$ 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 $\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^K,0) \parallel \mathbf{L}([]^L,0)) \leftrightarrow \mathbf{M}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0)$.

Proof: It is not hard to see that replacing $\mathbf{M}_{mod}(\ell, m, q, \ell', q', g, p, g', p')$ by $\partial_{\mathcal{H}}(\mathbf{S}(\ell, m, q) \parallel \mathbf{R}(\ell', q') \parallel \mathbf{K}(g, p) \parallel \mathbf{L}(g', p'))$ is a solution for the recursive equation above, using the axioms of μCRL [15]. (The details are left to the reader.) Hence, the theorem follows by CL-RSP [4]. ∎

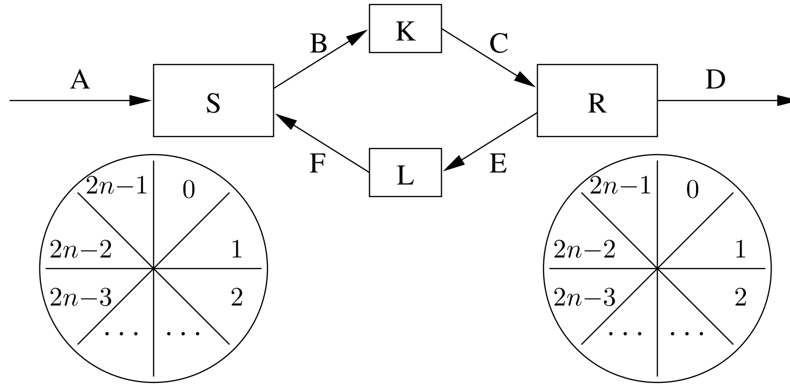

Fig. 1. Sliding window protocol

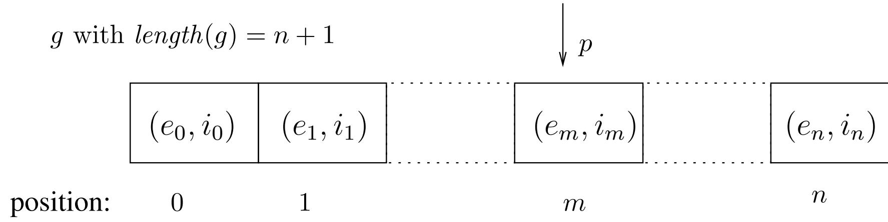

Fig. 2. The medium $\mathbf{K}$

$$\begin{aligned}
\mathbf{S}(\ell{:}Nat, m{:}Nat, q{:}Buf) \;\approx\;& \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{S}(\ell, S(m)|_{2n}, inb(d,m,q)) \\
&\qquad\qquad \lhd\ \textit{in-window}(\ell, m, (\ell+n)|_{2n})\ \rhd\ \delta \\
+\;& \textstyle\sum_{k:Nat} s_{\mathrm{B}}(retrieve(k,q),k)\cdot\mathbf{S}(\ell,m,q) \\
&\qquad\qquad \lhd\ test(k,q)\ \rhd\ \delta \\
+\;& \textstyle\sum_{k:Nat} r_{\mathrm{F}}(k)\cdot\mathbf{S}(k, m, release|_{2n}(\ell,k,q))
\end{aligned}$$

$$\begin{aligned}
\mathbf{R}(\ell'{:}Nat, q'{:}Buf) \;\approx\;& \textstyle\sum_{d:\Delta}\sum_{k:Nat} r_{\mathrm{C}}(d,k)\cdot(\mathbf{R}(\ell', inb(d,k,q')) \\
&\qquad\qquad \lhd\ \textit{in-window}(\ell', k, (\ell'+n)|_{2n})\ \rhd\ \mathbf{R}(\ell', q')) \\
+\;& s_{\mathrm{D}}(retrieve(\ell',q'))\cdot\mathbf{R}(S(\ell')|_{2n}, remove(\ell',q')) \\
&\qquad\qquad \lhd\ test(\ell',q')\ \rhd\ \delta \\
+\;& s_{\mathrm{E}}(\textit{next-empty}|_{2n}(\ell',q'))\cdot\mathbf{R}(\ell',q')
\end{aligned}$$

$$\begin{aligned}
\mathbf{K}(g{:}MedK, p{:}Nat) \;\approx\;& \textstyle\sum_{d:\Delta}\sum_{k:Nat} r_{\mathrm{B}}(d,k)\cdot\mathbf{K}(inm(d,k,g), p+1) \\
+\;& \textstyle\sum_{k:Nat} j\cdot\mathbf{K}(delete(k,g), p-1)\ \lhd\ k<p\ \rhd\ \delta \\
+\;& s_{\mathrm{C}}(\textit{last-dat}(g), \textit{last-seq}(g))\cdot\mathbf{K}(\textit{delete-last}(g), p) \\
&\qquad\qquad \lhd\ p < length(g)\ \rhd\ \delta \\
+\;& j\cdot\mathbf{K}(g, p-1)\ \lhd\ p>0\ \rhd\ \delta
\end{aligned}$$

$$\begin{aligned}
\mathbf{L}(g'{:}MedL, p'{:}Nat) \;\approx\;& \textstyle\sum_{k:Nat} r_{\mathrm{E}}(k)\cdot\mathbf{L}(inm(k,g'), p'+1) \\
+\;& \textstyle\sum_{k:Nat} j\cdot\mathbf{L}(delete(k,g'), p'-1)\ \lhd\ k<p'\ \rhd\ \delta \\
+\;& s_{\mathrm{F}}(\textit{last-seq}(g'))\cdot\mathbf{L}(\textit{delete-last}(g'), p') \\
&\qquad\qquad \lhd\ p' < length(g')\ \rhd\ \delta \\
+\;& j\cdot\mathbf{L}(g', p'-1)\ \lhd\ p'>0\ \rhd\ \delta
\end{aligned}$$

$$\begin{array}{rcl}\mathbf{Z}(\lambda{:}List) & \approx & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{Z}(append(d,\lambda)) \lhd length(\lambda) < 2n \rhd \delta \\ & + & s_{\mathrm{D}}(top(\lambda))\cdot\mathbf{Z}(tail(\lambda)) \lhd length(\lambda) > 0 \rhd \delta\end{array}$$

$$\begin{array}{ll} & \mathbf{M}_{mod}(\ell{:}Nat, m{:}Nat, q{:}Buf, \ell'{:}Nat, q'{:}Buf, g{:}MedK, p{:}Nat, g'{:}MedL, p'{:}Nat) \\ \approx & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{M}_{mod}(m{:=}S(m)|_{2n}, q{:=}inb(d,m,q)) \\ & \qquad \lhd in\text{-}window(\ell, m, (\ell+n)|_{2n}) \rhd \delta \\ + & \sum_{k:Nat} c_{\mathrm{B}}(retrieve(k,q),k)\cdot\mathbf{M}_{mod}(g{:=}inm(retrieve(k,q),k,g), p{:=}p+1) \\ & \qquad \lhd test(k,q) \rhd \delta \\ + & \sum_{k:Nat} j\cdot\mathbf{M}_{mod}(g{:=}\ delete(k,g), p{:=}p-1) \lhd\ k<p \rhd \delta \\ + & j\cdot\mathbf{M}_{mod}(p{:=}p-1) \lhd\ p>0 \rhd \delta \\ + & c_{\mathrm{C}}(last\text{-}dat(g), last\text{-}seq(g))\cdot\mathbf{M}_{mod}(q'{:=}inb(last\text{-}dat(g), last\text{-}seq(g), q'), g{:=}delete\text{-}last(g)) \\ & \qquad \lhd p < length(g) \wedge in\text{-}window(\ell', last\text{-}seq(g), (\ell'+n)|_{2n}) \rhd \delta \\ + & c_{\mathrm{C}}(last\text{-}dat(g), last\text{-}seq(g))\cdot\mathbf{M}_{mod}(g{:=}delete\text{-}last(g)) \\ & \qquad \lhd p < length(g) \wedge \neg in\text{-}window(\ell', last\text{-}seq(g), (\ell'+n)|_{2n}) \rhd \delta \\ + & s_{\mathrm{D}}(retrieve(\ell',q'))\cdot\mathbf{M}_{mod}(\ell'{:=}S(\ell')|_{2n}, q'{:=}remove(\ell',q')) \lhd\ test(\ell',q') \rhd \delta \\ + & c_{\mathrm{E}}(next\text{-}empty|_{2n}(\ell',q'))\cdot\mathbf{M}_{mod}(g'{:=}inm(next\text{-}empty|_{2n}(\ell',q'),g'), p'{:=}p'+1) \\ + & \sum_{k:Nat} j\cdot\mathbf{M}_{mod}(g'{:=}delete(k,g'), p'{:=}p'-1) \lhd\ k<p' \rhd \delta \\ + & j\cdot\mathbf{M}_{mod}(p'{:=}p'-1) \lhd\ p'>0 \rhd \delta \\ + & c_{\mathrm{F}}(last\text{-}seq(g'))\cdot\mathbf{M}_{mod}(\ell{:=}last\text{-}seq(g'), q{:=}release|_{2n}(\ell, last\text{-}seq(g'), q), g'{:=}delete\text{-}last(g')) \\ & \qquad \lhd p' < length(g') \rhd \delta\end{array}$$

B. Eliminating Arguments of Communication Actions

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

Theorem V.2 $\tau_{\mathcal{I}}(\mathbf{M}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0)) \leftrightarrow \tau_{\{c,j\}}(\mathbf{N}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0))$.

Proof: By a simple renaming. ∎

C. Getting Rid of Modulo Arithmetic

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

Theorem V.3 $\mathbf{N}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0) \leftrightarrow \mathbf{N}_{nonmod}(0,0,[],0,[],[]^K,0,[]^L,0)$.

The proof of Theorem V.3 is presented in Section VII-A. Next, in Section VII-B, we prove the correctness of $\mathbf{N}_{nonmod}$. 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 *Nat* and *Bool*, which are derivable from their axioms (using induction). Some typical examples of such properties are:

$$\begin{array}{rcl}\neg\neg b & = & b \\ i+k<j+k & = & i<j \\ i \geq j \Rightarrow (i \dot{-} j)+k & = & (i+k) \dot{-} j\end{array}$$

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 *next-empty* operation, together with one result on *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

$$
\begin{array}{ll}
& \mathbf{N}_{mod}(\ell{:}Nat, m{:}Nat, q{:}Buf, \ell'{:}Nat, q'{:}Buf, g{:}MedK, p{:}Nat, g'{:}MedL, p'{:}Nat) \\
\approx & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{N}_{mod}(m{:=}S(m)|_{2n}, q{:=}inb(d,m,q)) \\
& \qquad \lhd\ in\text{-}window(\ell, m, (\ell+n)|_{2n}) \rhd \delta \\
+ & \sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k,q),k,g), p{:=}p+1) \lhd\ test(k,q) \rhd \delta \\
+ & \sum_{k:Nat} j\cdot\mathbf{N}_{mod}(g{:=}\ delete(k,g), p{:=}p-1) \lhd\ k<p \rhd \delta \\
+ & j\cdot\mathbf{N}_{mod}(p{:=}p-1) \lhd\ p>0 \rhd \delta \\
+ & c\cdot\mathbf{N}_{mod}(q'{:=}inb(last\text{-}dat(g), last\text{-}seq(g), q'), g{:=}delete\text{-}last(g)) \\
& \qquad \lhd p<length(g) \wedge in\text{-}window(\ell', last\text{-}seq(g), (\ell'+n)|_{2n}) \rhd \delta \\
+ & c\cdot\mathbf{N}_{mod}(g{:=}delete\text{-}last(g)) \\
& \qquad \lhd p<length(g) \wedge \neg in\text{-}window(\ell', last\text{-}seq(g), (\ell'+n)|_{2n}) \rhd \delta \\
+ & s_{\mathrm{D}}(retrieve(\ell',q'))\cdot\mathbf{N}_{mod}(\ell'{:=}S(\ell')|_{2n}, q'{:=}remove(\ell',q')) \lhd\ test(\ell',q') \rhd \delta \\
+ & c\cdot\mathbf{N}_{mod}(g'{:=}inm(next\text{-}empty|_{2n}(\ell',q'), g'), p'{:=}p'+1) \\
+ & \sum_{k:Nat} j\cdot\mathbf{N}_{mod}(g'{:=}delete(k,g'), p'{:=}p'-1) \lhd\ k<p' \rhd \delta \\
+ & j\cdot\mathbf{N}_{mod}(p'{:=}p'-1) \lhd\ p'>0 \rhd \delta \\
+ & c\cdot\mathbf{N}_{mod}(\ell{:=}last\text{-}seq(g'), q{:=}release|_{2n}(\ell, last\text{-}seq(g'), q), g'{:=}delete\text{-}last(g')) \\
& \qquad \lhd p'<length(g') \rhd \delta
\end{array}
$$

$$
\begin{array}{llr}
& \mathbf{N}_{nonmod}(\ell{:}Nat, m{:}Nat, q{:}Buf, \ell'{:}Nat, q'{:}Buf, g{:}MedK, p{:}Nat, g'{:}MedL, p'{:}Nat) & \\
\approx & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{N}_{nonmod}(m{:=}S(m), q{:=}inb(d,m,q)) \lhd\ m<\ell+n \rhd \delta & (A) \\
+ & \sum_{k:Nat} c\cdot\mathbf{N}_{nonmod}(g{:=}inm(retrieve(k,q),k,g), p{:=}p+1) \lhd\ test(k,q) \rhd \delta & (B) \\
+ & \sum_{k:Nat} j\cdot\mathbf{N}_{nonmod}(g{:=}\ delete(k,g), p{:=}p-1) \lhd\ k<p \rhd \delta & (C) \\
+ & j\cdot\mathbf{N}_{nonmod}(p{:=}p-1) \lhd\ p>0 \rhd \delta & (D) \\
+ & c\cdot\mathbf{N}_{nonmod}(q'{:=}inb(last\text{-}dat(g), last\text{-}seq(g), q'), g{:=}delete\text{-}last(g)) & \\
& \qquad \lhd p<length(g) \wedge (\ell' \leq last\text{-}seq(g) < \ell'+n) \rhd \delta & (E) \\
+ & c\cdot\mathbf{N}_{nonmod}(g{:=}delete\text{-}last(g)) & \\
& \qquad \lhd p<length(g) \wedge \neg(\ell' \leq last\text{-}seq(g) < \ell'+n) \rhd \delta & (F) \\
+ & s_{\mathrm{D}}(retrieve(\ell',q'))\cdot\mathbf{N}_{nonmod}(\ell'{:=}S(\ell'), q'{:=}remove(\ell',q')) \lhd\ test(\ell',q') \rhd \delta & (G) \\
+ & c\cdot\mathbf{N}_{nonmod}(g'{:=}inm(next\text{-}empty(\ell',q'), g'), p'{:=}p'+1) & (H) \\
+ & \sum_{k:Nat} j\cdot\mathbf{N}_{nonmod}(g'{:=}delete(k,g'), p'{:=}p'-1) \lhd\ k<p' \rhd \delta & (I) \\
+ & j\cdot\mathbf{N}_{nonmod}(p'{:=}p'-1) \lhd\ p'>0 \rhd \delta & (J) \\
+ & c\cdot\mathbf{N}_{nonmod}(\ell{:=}last\text{-}seq(g'), q{:=}release(\ell, last\text{-}seq(g'), q), g'{:=}delete\text{-}last(g')) & \\
& \qquad \lhd p'<length(g') \rhd \delta & (K)
\end{array}
$$

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. $(i{\cdot}n)|_n = 0$
4. $i = (i \; div \; n){\cdot}n + i|_n$
5. $j \leq i \leq j + n$
$\Rightarrow \; (i \; div \; 2n = j \; div \; 2n \wedge j|_{2n} \leq i|_{2n} \leq j|_{2n} + n) \vee (i \; div \; 2n = S(j \; div \; 2n) \wedge i|_{2n} + n \leq j|_{2n})$
6. $i \leq j \;\Rightarrow\; i \; div \; n \leq j \; div \; n$

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

Lemma VI.3 1. $test(k, q|_{2n}) \;\Rightarrow\; k = k|_{2n}$
2. $(\forall j{:}Nat(test(j, q) \Rightarrow i \leq j < i + n) \wedge i \leq k \leq i + n)$
$\Rightarrow \; test(k, q) = test(k|_{2n}, q|_{2n})$
3. $(\forall j{:}Nat(test(j, q) \Rightarrow i \leq j < i + n) \wedge test(k, q))$
$\Rightarrow \; retrieve(k, q) = retrieve(k|_{2n}, q|_{2n})$
4. $(\forall j{:}Nat(test(j, q) \Rightarrow i \leq j < i + n) \wedge i \leq k \leq i + n)$
$\Rightarrow \; remove(k, q)|_{2n} = remove(k|_{2n}, q|_{2n})$
5. $(\forall j{:}Nat(test(j, q) \Rightarrow i \leq j < i + n) \wedge i \leq k \leq i + n)$
$\Rightarrow \; release(i, k, q)|_{2n} = release|_{2n}(i, k, q|_{2n})$
6. $(\forall j{:}Nat(test(j, q) \Rightarrow i \leq j < i + n) \wedge i \leq k \leq i + n)$
$\Rightarrow \; next\text{-}empty(k, q)|_{2n} = next\text{-}empty|_{2n}(k|_{2n}, q|_{2n})$
7. $i \leq k < i + n \;\Rightarrow\; in\text{-}window(i|_{2n}, k|_{2n}, (i + n)|_{2n})$
8. $in\text{-}window(i|_{2n}, k|_{2n}, (i + n)|_{2n})$
$\Rightarrow \; k + n < i \vee i \leq k < i + n \vee k \geq i + 2n$
9. $(\forall j{:}Nat(test(j, q) \Rightarrow i \leq j < i + n) \wedge test(k, q|_{2n}))$
$\Rightarrow \; in\text{-}window(i|_{2n}, k, (i + n)|_{2n})$

Lemma VI.4 1. $test(i, q) \;\Rightarrow\; i \leq max(q)$
2. $i \leq j < next\text{-}empty(i, q) \;\Rightarrow\; test(j, q)$
3. $next\text{-}empty(i, q) \geq i$
4. $next\text{-}empty(i, inb(d, j, q)) \geq next\text{-}empty(i, q)$
5. $j \neq next\text{-}empty(i, q)$
$\Rightarrow \; next\text{-}empty(i, inb(d, j, q)) = next\text{-}empty(i, q)$
6. $next\text{-}empty(i, inb(d, next\text{-}empty(i, q), q))$
$= next\text{-}empty(S(next\text{-}empty(i, q)), q)$
7. $\neg(i \leq j < next\text{-}empty(i, q))$
$\Rightarrow \; next\text{-}empty(i, remove(j, q)) = next\text{-}empty(i, q)$

Lemma VI.5 1. $length(g) = length(g|_{2n})$
2. $i < length(g)$
$\Rightarrow \; return\text{-}seq(i, g)|_{2n} = return\text{-}seq(i, g|_{2n})$
3. $i < length(g)$
$\Rightarrow \; return\text{-}dat(i, g) = return\text{-}dat(i, g|_{2n})$
4. $i < length(g)$
$\Rightarrow \; delete(i, g)|_{2n} = delete(i, g|_{2n})$
5. $length(g) > 0$
$\Rightarrow \; last\text{-}dat(g) = return\text{-}dat(length(g) - 1, g)$
6. $length(g) > 0$
$\Rightarrow \; last\text{-}seq(g) = return\text{-}seq(length(g) - 1, g)$
7. $length(g) > 0$
$\Rightarrow \; delete\text{-}last(g) = delete(length(g) - 1, g)$
8. $(i < length(g) \wedge member(d, j, delete(i, g)))$
$\Rightarrow \; member(d, j, g)$
9. $i < length(g)$
$\Rightarrow \; length(delete(i, g)) = length(g) - 1$
10. $i < length(g)$
$\Rightarrow \; member(return\text{-}dat(i, g), return\text{-}seq(i, g), g)$
11. $(i < length(g) - 1 \wedge j < length(g))$
$\Rightarrow \; return\text{-}seq(i, delete(j, g))$
$= if(i < j, return\text{-}seq(i, g), return\text{-}seq(i + 1, g))$
12. $member(d, i, g)$
$\Rightarrow \; \exists j{:}Nat \; (j < length(g) \wedge return\text{-}seq(j, g) = i \wedge return\text{-}dat(j, g) = d)$

Lemma VI.6 1. $length(g') = length(g'|_{2n})$
2. $i < length(g')$
$\Rightarrow \; return\text{-}seq(i, g')|_{2n} = return\text{-}seq(i, g'|_{2n})$
3. $i < length(g')$
$\Rightarrow \; delete(i, g')|_{2n} = delete(i, g'|_{2n})$
4. $length(g') > 0$
$\Rightarrow \; last\text{-}seq(g') = return\text{-}seq(length(g') - 1, g')$
5. $length(g') > 0$
$\Rightarrow \; delete\text{-}last(g') = delete(length(g') - 1, g')$
6. $(i < length(g') \wedge member(j, delete(i, g')))$
$\Rightarrow \; member(j, g')$
7. $i < length(g')$
$\Rightarrow \; length(delete(i, g')) = length(g') - 1$
8. $i < length(g')$
$\Rightarrow \; member(return\text{-}seq(i, g'), g')$
9. $(i < length(g') - 1 \wedge j < length(g'))$
$\Rightarrow \; return\text{-}seq(i, delete(j, g'))$
$= if(i < j, return\text{-}seq(i, g'), return\text{-}seq(i + 1, g'))$

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. $length(q[i..j\rangle) = j \dot{-} i$
5. $i \leq k \leq j \;\Rightarrow\; q[i..j\rangle = q[i..k\rangle{+\!\!+}q[k..j\rangle$
6. $i \leq j \;\Rightarrow\; append(d, q[i..j\rangle) = inb(d, j, q)[i..S(j)\rangle$

7. $test(k,q) \ \Rightarrow \ inb(retrieve(k,q),k,q)[i..j\rangle = q[i..j\rangle$
8. $\neg(i \leq k < j) \ \Rightarrow \ remove(k,q)[i..j\rangle = q[i..j\rangle$
9. $\ell \leq i \ \Rightarrow \ release(k,\ell,q)[i..j\rangle = q[i..j\rangle$

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 $\mathbf{N}_{nonmod}$ that are needed in the correctness proof. Occurrences of variables $i,j{:}Nat$ and $d,e{:}\Delta$ in an invariant are always implicitly universally quantified at the outside of the invariant.

Lemma VI.8 The following invariants 1-27 hold for $\mathbf{N}_{nonmod}(\ell,m,q,\ell',q',g,p,g',p')$.
1. $p \leq length(g)$
2. $p' \leq length(g')$
3. $member(i,g') \ \Rightarrow \ i \leq next\text{-}empty(\ell',q')$
4. $\ell \leq next\text{-}empty(\ell',q')$
5. $i < j < length(g')$
$\Rightarrow \ return\text{-}seq(i,g') \geq return\text{-}seq(j,g')$
6. $member(i,g') \ \Rightarrow \ \ell \leq i$
7. $test(i,q) \ \Rightarrow \ i < m$
8. $member(d,i,g) \ \Rightarrow \ i < m$
9. $test(i,q') \ \Rightarrow \ i < m$
10. $test(i,q') \ \Rightarrow \ \ell' \leq i < \ell'+n$
11. $\ell' \leq m$
12. $next\text{-}empty(\ell',q') \leq m$
13. $next\text{-}empty(\ell',q') \leq \ell'+n$
14. $\ell \leq m$
15. $test(i,q) \ \Rightarrow \ \ell \leq i$
16. $\ell \leq i < m \ \Rightarrow \ test(i,q)$
17. $\ell \leq \ell'+n$
18. $m \leq \ell+n$
19. $i \leq j < length(g)$
$\Rightarrow \ return\text{-}seq(i,g)+n > return\text{-}seq(j,g)$
20. $(member(d,i,g) \wedge test(j,q')) \ \Rightarrow \ i+n > j$
21. $member(d,i,g) \ \Rightarrow \ i+n \geq \ell'$
22. $member(d,i,g) \ \Rightarrow \ i+n \geq next\text{-}empty(\ell',q')$
23. $(member(d,i,g) \wedge test(i,q))$
$\Rightarrow \ retrieve(i,q) = d$
24. $(test(i,q) \wedge test(i,q'))$
$\Rightarrow \ retrieve(i,q) = retrieve(i,q')$
25. $(member(d,i,g) \wedge member(e,i,g))$
$\Rightarrow \ d = e$
26. $(member(d,i,g) \wedge test(i,q'))$
$\Rightarrow \ retrieve(i,q') = d$
27. $(\ell \leq i \leq m \wedge j \leq next\text{-}empty(i,q'))$
$\Rightarrow \ q[i..j\rangle = q'[i..j\rangle$

VII. Correctness of $\mathbf{N}_{mod}$

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

A. Equality of $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$

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

$$\begin{array}{rl} & \mathbf{N}_{mod}(\ell|_{2n},m|_{2n},q|_{2n},\ell'|_{2n},q'|_{2n},g|_{2n},p,g'|_{2n},p') \\ \leftrightarrow & \mathbf{N}_{nonmod}(\ell,m,q,\ell',q',g,p,g',p') \end{array}$$

Proof: We show that
$\mathbf{N}_{mod}(\ell|_{2n},m|_{2n},q|_{2n},\ell'|_{2n},q'|_{2n},g|_{2n},p,g'|_{2n},p')$
is a solution for the defining equation of
$\mathbf{N}_{nonmod}(\ell,m,q,\ell',q',g,p,g',p')$.
Hence, we must derive the following equation.[2]

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

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

$$\begin{array}{rl} & m<\ell+n \\ \Leftrightarrow & \ell\leq m<\ell+n \\ & \text{(Inv. VI.8.14)} \\ \Rightarrow & in\text{-}window(\ell|_{2n},m|_{2n},(\ell+n)|_{2n}) \\ & \text{(Lem. VI.3.7)} \end{array}$$

Reversely,

$$\begin{array}{rl} & in\text{-}window(\ell|_{2n},m|_{2n},(\ell+n)|_{2n}) \\ \Rightarrow & m+n<\ell \vee \ell\leq m<\ell+n \vee m\geq \ell+2n \\ & \text{(Lem. VI.3.8)} \\ \Leftrightarrow & m<\ell+n \\ & \text{(Inv. VI.8.14 and VI.8.18)} \end{array}$$

Moreover, by Lemma VI.1.1, $(\ell+n)|_{2n} = (\ell|_{2n}+n)|_{2n}$.
• $S(m)|_{2n} = S(m|_{2n})|_{2n}$.
This follows from Lemma VI.1.1.
• $inb(d,m,q)|_{2n} = inb(d,m|_{2n},q|_{2n})$.
This follows from the definition of buffers modulo $2n$.

[2] By abuse of notation, we use the parameters ℓ, m, q, ℓ', q', g, g' in an ambiguous way. For example, m refers both to the second parameter of $\mathbf{N}_{mod}$ and to the value of this parameter.

$$
\begin{array}{lll}
 & \mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g|_{2n}, p, g'|_{2n}, p') & \\
\approx & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{N}_{mod}(m{:=}S(m)|_{2n}, q{:=}inb(d,m,q)|_{2n}) \lhd m < \ell + n \rhd \delta & (A) \\
+ & \sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k,q),k,g)|_{2n}, p{:=}p+1) \lhd test(k,q) \rhd \delta & (B) \\
+ & \sum_{k:Nat} j\cdot\mathbf{N}_{mod}(g{:=}\ delete(k,g)|_{2n}, p{:=}p-1) \lhd k < p \rhd \delta & (C) \\
+ & j\cdot\mathbf{N}_{mod}(p{:=}p-1) \lhd p > 0 \rhd \delta & (D) \\
+ & c\cdot\mathbf{N}_{mod}(q'{:=}inb(last\text{-}dat(g), last\text{-}seq(g), q')|_{2n}, g{:=}delete\text{-}last(g)|_{2n}) & \\
 & \qquad \lhd p < length(g) \wedge (\ell' \leq last\text{-}seq(g) < \ell' + n) \rhd \delta & (E) \\
+ & c\cdot\mathbf{N}_{mod}(g{:=}delete\text{-}last(g)|_{2n}) & \\
 & \qquad \lhd p < length(g) \wedge \neg(\ell' \leq last\text{-}seq(g) < \ell' + n) \rhd \delta & (F) \\
+ & s_{\mathrm{D}}(retrieve(\ell',q'))\cdot\mathbf{N}_{mod}(\ell'{:=}S(\ell')|_{2n}, q'{:=}remove(\ell',q')|_{2n}) \lhd test(\ell',q') \rhd \delta & (G) \\
+ & c\cdot\mathbf{N}_{mod}(g'{:=}inm(next\text{-}empty(\ell',q'),g')|_{2n}, p'{:=}p'+1) & (H) \\
+ & \sum_{k:Nat} j\cdot\mathbf{N}_{mod}(g'{:=}delete(k,g')|_{2n}, p'{:=}p'-1) \lhd k < p' \rhd \delta & (I) \\
+ & j\cdot\mathbf{N}_{mod}(p'{:=}p'-1) \lhd p' > 0 \rhd \delta & (J) \\
+ & c\cdot\mathbf{N}_{mod}(\ell{:=}last\text{-}seq(g')|_{2n}, q{:=}release(\ell, last\text{-}seq(g'), q)|_{2n}, g'{:=}delete\text{-}last(g')|_{2n}) & \\
 & \qquad \lhd p' < length(g') \rhd \delta & (K)
\end{array}
$$

$$
\begin{array}{lll}
 & \mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g|_{2n}, p, g'|_{2n}, p') & \\
\approx & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{N}_{mod}(m{:=}S(m|_{2n})|_{2n}, q{:=}inb(d, m|_{2n}, q|_{2n})) & \\
 & \qquad \lhd in\text{-}window(\ell|_{2n}, m|_{2n}, (\ell|_{2n}+n)|_{2n}) \rhd \delta & (A) \\
+ & \sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k, q|_{2n}), k, g|_{2n}), p{:=}p+1) & \\
 & \qquad \lhd test(k, q|_{2n}) \rhd \delta & (B) \\
+ & \sum_{k:Nat} j\cdot\mathbf{N}_{mod}(g{:=}\ delete(k, g|_{2n}), p{:=}p-1) \lhd k < p \rhd \delta & (C) \\
+ & j\cdot\mathbf{N}_{mod}(p{:=}p-1) \lhd p > 0 \rhd \delta & (D) \\
+ & c\cdot\mathbf{N}_{mod}(q'{:=}inb(last\text{-}dat(g|_{2n}), last\text{-}seq(g|_{2n}), q'|_{2n}), g{:=}delete\text{-}last(g|_{2n})) & \\
 & \qquad \lhd p < length(g|_{2n}) \wedge in\text{-}window(\ell'|_{2n}, last\text{-}seq(g|_{2n}), (\ell'|_{2n}+n)|_{2n}) \rhd \delta & (E) \\
+ & c\cdot\mathbf{N}_{mod}(g{:=}delete\text{-}last(g|_{2n})) & \\
 & \qquad \lhd p < length(g|_{2n}) \wedge \neg in\text{-}window(\ell'|_{2n}, last\text{-}seq(g|_{2n}), (\ell'|_{2n}+n)|_{2n}) \rhd \delta & (F) \\
+ & s_{\mathrm{D}}(retrieve(\ell'|_{2n}, q'|_{2n}))\cdot\mathbf{N}_{mod}(\ell'{:=}S(\ell'|_{2n})|_{2n}, q'{:=}remove(\ell'|_{2n}, q'|_{2n})) \lhd test(\ell'|_{2n}, q'|_{2n}) \rhd \delta & (G) \\
+ & c\cdot\mathbf{N}_{mod}(g'{:=}inm(next\text{-}empty|_{2n}(\ell'|_{2n}, q'|_{2n}), g'|_{2n}), p'{:=}p'+1) & (H) \\
+ & \sum_{k:Nat} j\cdot\mathbf{N}_{mod}(g'{:=}delete(k, g'|_{2n}), p'{:=}p'-1) \lhd k < p' \rhd \delta & (I) \\
+ & j\cdot\mathbf{N}_{mod}(p'{:=}p'-1) \lhd p' > 0 \rhd \delta & (J) \\
+ & c\cdot\mathbf{N}_{mod}(\ell{:=}last\text{-}seq(g'|_{2n})|_{2n}, q{:=}release|_{2n}(\ell|_{2n}, last\text{-}seq(g'|_{2n})|_{2n}, q|_{2n}), g'{:=}delete\text{-}last(g'|_{2n})) & (K) \\
 & \qquad \lhd p' < length(g'|_{2n}) \rhd \delta &
\end{array}
$$

$$\begin{array}{lll}
& \sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k,q),k,g)|_{2n}) & \\
& \lhd\, test(k,q) \,\rhd\, \delta & \\
\approx & \sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k,q),k|_{2n},g|_{2n})) & \\
& \lhd\, test(k,q) \wedge \ell \leq k < \ell+n \,\rhd\, \delta & \text{(Inv. VI.8.7, VI.8.15, VI.8.18)} \\
\approx & \sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k|_{2n},q|_{2n}),k|_{2n},g|_{2n})) & \\
& \lhd\, test(k|_{2n},q|_{2n}) \wedge \ell \leq k < \ell+n \,\rhd\, \delta & \text{(Lem. VI.3.2, VI.3.3)} \\
\approx & \sum_{k':Nat}\sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k',q|_{2n}),k',g|_{2n})) & \\
& \lhd\, test(k',q|_{2n}) \wedge \ell \leq k < \ell+n \wedge k'=k|_{2n} \,\rhd\, \delta & \text{(sum elim.)} \\
\approx & \sum_{k':Nat}\sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k',q|_{2n}),k',g|_{2n})) & \\
& \lhd\, test(k',q|_{2n}) \wedge k=(\ell\, div\, 2n)2n+k' \wedge & \\
& \ell|_{2n} \leq k' < \ell|_{2n}+n \wedge k'=k|_{2n} \,\rhd\, \delta & \\
+ & \sum_{k':Nat}\sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k',q|_{2n}),k',g|_{2n})) & \\
& \lhd\, test(k',q|_{2n}) \wedge k=S(\ell\, div\, 2n)2n+k' \wedge & \\
& k'+n < \ell|_{2n} \wedge k'=k|_{2n} \,\rhd\, \delta & \text{(Lem. VI.1.4, VI.1.5)} \\
\approx & \sum_{k':Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k',q|_{2n}),k',g|_{2n})) & \\
& \lhd\, test(k',q|_{2n}) \wedge \ell|_{2n} \leq k' < \ell|_{2n}+n \wedge k'=k' \,\rhd\, \delta & \\
+ & \sum_{k':Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k',q|_{2n}),k',g|_{2n})) & \\
& \lhd\, test(k',q|_{2n}) \wedge k'+n < \ell|_{2n} \wedge k'=k' \,\rhd\, \delta & \text{(sum elim., Lem. VI.1.3)} \\
\approx & \sum_{k':Nat} c\cdot\mathbf{N}_{mod}(g{:=}inm(retrieve(k',q|_{2n}),k',g|_{2n})) & \\
& \lhd\, test(k',q|_{2n}) \,\rhd\, \delta & \text{(see below)}
\end{array}$$

B Below we equate the entire summand B of the two specifications. The argument $p := p+1$ is omitted, because it is irrelevant for this derivation.
The last equality follows from the following derivation:

$$\begin{array}{ll}
& test(k',q|_{2n}) \\
\Rightarrow & test(k'|_{2n},q|_{2n}) \\
& \text{(Lem. VI.3.1)} \\
\Rightarrow & \ell \leq k'|_{2n} < \ell+n \\
& \text{(Inv. VI.8.7, VI.8.15, VI.8.18)} \\
\Rightarrow & in\text{-}window(\ell|_{2n},k'|_{2n},(\ell+n)|_{2n}) \\
& \text{(Lem. VI.3.9)} \\
\Rightarrow & k'+n < \ell|_{2n} \vee \ell|_{2n} \leq k' < \ell|_{2n}+n \\
& \vee k' \geq \ell|_{2n}+2n \\
& \text{(Lem. VI.1.1, VI.3.8)} \\
\Leftrightarrow & k'+n < \ell|_{2n} \vee \ell|_{2n} \leq k' < \ell|_{2n}+n \\
& \text{(Lem. VI.1.2, VI.3.1)}
\end{array}$$

C $k < p \Rightarrow delete(k,g)|_{2n} = delete(k,g|_{2n})$.
By Invariant VI.8.1, $k < p \leq length(g)$. So this follows from Lemma VI.5.4.
E • $length(g) = length(g|_{2n})$.
This follows from Lemma VI.5.1.
• $p < length(g) \Rightarrow (\ell' \leq last\text{-}seq(g) < \ell'+n = in\text{-}window(\ell'|_{2n}, last\text{-}seq(g)|_{2n}, (\ell'|_{2n}+n)|_{2n}))$.
Since $0 < length(g)$, Lemmas VI.5.5, VI.5.6, and VI.5.10 yield $member(last\text{-}dat(g), last\text{-}seq(g), g)$. So by Invariant VI.8.22, $next\text{-}empty(\ell',q') \leq last\text{-}seq(g)+n$. Hence, by Lemma VI.4.3, $\ell' \leq last\text{-}seq(g)+n$. Furthermore, by Invariant VI.8.8, $last\text{-}seq(g) < m$, by Invariant VI.8.18, $m \leq \ell+n$, and by Invariant VI.8.17, $\ell \leq \ell'+n$. Hence, $last\text{-}seq(g) < \ell'+2n$. So by Lemmas VI.3.7 and VI.3.8, $\ell' \leq last\text{-}seq(g) < \ell'+n = in\text{-}window(\ell'|_{2n}, last\text{-}seq(g)|_{2n}, (\ell'+n)|_{2n})$. And by Lemma VI.1.1, $(\ell'+n)|_{2n} = (\ell'|_{2n}+n)|_{2n}$.
• $p < length(g) \Rightarrow inb(last\text{-}dat(g), last\text{-}seq(g), q')|_{2n} = inb(last\text{-}dat(g|_{2n}), last\text{-}seq(g|_{2n}), q'|_{2n})$.
This follows from the definitions of buffers modulo $2n$, and Lemmas VI.5.5, VI.5.6, VI.5.2 and VI.5.3.
• $p < length(g) \Rightarrow delete\text{-}last(g)|_{2n} = delete\text{-}last(g|_{2n})$.
This follows from Lemmas VI.5.7 and VI.5.4.
F • $\neg(\ell' \leq last\text{-}seq(g) < \ell'+n)$
$\Leftrightarrow \neg in\text{-}window(\ell'|_{2n}, last\text{-}seq(g)|_{2n}, (\ell'|_{2n}+n)|_{2n})$.
This follows immediately from the second item of the previous case.
• $p < length(g) \Rightarrow delete\text{-}last(g)|_{2n} = delete\text{-}last(g|_{2n})$.
This follows immediately from the fourth item of the previous case.
G • $test(\ell',q') = test(\ell'|_{2n},q'|_{2n})$.
This follows from Lemma VI.3.2 together with Invariant VI.8.10.

- $test(\ell', q') \Rightarrow (retrieve(\ell', q') = retrieve(\ell'|_{2n}, q'|_{2n}))$.

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.
- $remove(\ell', q')|_{2n} = remove(\ell'|_{2n}, q'|_{2n})$.

This follows from Lemma VI.3.4 together with Invariant VI.8.10.

H $inm(next\text{-}empty(\ell', q')|_{2n}, g')|_{2n}$
$= inm(next\text{-}empty|_{2n}(\ell'|_{2n}, q'|_{2n}), g'|_{2n})$.
By Lemma VI.3.6 and Invariant VI.8.10,
$next\text{-}empty(\ell', q')|_{2n} = next\text{-}empty|_{2n}(\ell'|_{2n}, q'|_{2n})$. So the desired equality follows the definition of mediums modulo $2n$.

I $k < p' \Rightarrow delete(k, g')|_{2n} = delete(k, g'|_{2n})$.
By Invariant VI.8.2, $k < p' \leq length(g')$. So the desired equality follows from Lemma VI.6.3.

K • $length(g') = length(g'|_{2n})$.
This follows from Lemma VI.6.1.
- $p' < length(g') \Rightarrow last\text{-}seq(g')|_{2n} = last\text{-}seq(g'|_{2n})|_{2n}$.

This follows from Lemmas VI.6.4, VI.6.2 and VI.1.1.
- $release(\ell, last\text{-}seq(g'), q)|_{2n}$ $= release|_{2n}(\ell|_{2n}, last\text{-}seq(g')|_{2n}, q|_{2n})$.

By Lemmas VI.6.4 and VI.6.8, $p' < length(g')$ implies $member(last\text{-}seq(g'), g')$. So by Invariant VI.8.6, $\ell \leq last\text{-}seq(g')$. By Invariants VI.8.3 and VI.8.12, $last\text{-}seq(g') \leq next\text{-}empty(\ell', q') \leq m$. And by Invariant VI.8.18, $m \leq \ell + n$. So $\ell \leq last\text{-}seq(g') \leq \ell + n$. Furthermore, by Invariants VI.8.7, VI.8.15 and VI.8.18, $test(i, q) \Rightarrow \ell \leq i < \ell + n$. Hence, the desired equation follows from Lemma VI.3.5.
- $p' < length(g')$ $\Rightarrow delete\text{-}last(g')|_{2n} = delete\text{-}last(g'|_{2n})$.

This follows from Lemmas VI.6.3 and VI.6.5.
Hence, $\mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g|_{2n}, p, g'|_{2n}, p')$ is a solution for the defining equation of $\mathbf{N}_{nonmod}(\ell, m, q, \ell', q', g, p, g', p')$. So by CL-RSP, they are strongly (and thus branching) bisimilar. ■

B. Correctness of $\mathbf{N}_{nonmod}$

We prove that $\mathbf{N}_{nonmod}$ is branching bisimilar to the FIFO queue $\mathbf{Z}$ of size $2n$ (see Section IV-B), using the cones and foci method [9].

Let Ξ abbreviate $Nat \times Nat \times Buf \times Nat \times Buf \times MedK \times Nat \times MedL \times Nat$. Furthermore, let $\xi{:}\Xi$ denote $(\ell, m, q, \ell', q', g, p, g', p')$. The state mapping $\phi : \Xi \Rightarrow List$, which maps states of $\mathbf{N}_{nonmod}$ to states of $\mathbf{Z}$, is defined by: $\phi(\xi) = q'[\ell'..next\text{-}empty(\ell', q')\rangle{+\!\!+}q[next\text{-}empty(\ell', q')..m\rangle$

Intuitively, ϕ collects the data elements in the sending and receiving windows, starting at the first position of the receiving window (i.e., ℓ') 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 ϕ is independent of ℓ, g, p, g', p'; 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 $\mathbf{N}_{nonmod}(\ell, m, q, \ell', q', g, p, g', p')$ is

$$FC(\ell, m, q, \ell', q', g, p, g', p') := \ell = m \vee \ell = \ell' + n$$

Lemma VII.1 For each $\xi{:}\Xi$ where the invariants in Lemma VI.8 hold, there is a $\hat{\xi}{:}\Xi$ with $FC(\hat{\xi})$ such that $\mathbf{N}_{nonmod}(\xi) \xrightarrow{c_1} \cdots \xrightarrow{c_n} \mathbf{N}_{nonmod}(\hat{\xi})$, where $c_1, \ldots, c_n \in \mathcal{I}$.

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

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

INDUCTION CASE: $next\text{-}empty(\ell', q') < \min\{m, \ell' + n\}$.
Let $y = length(g)$ and $x = next\text{-}empty(\ell', q')$ at state ξ. 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 ξ' with $g{:=}inm(d, x, g)$ (where d denotes $retrieve(x, q)$). Hence, at state ξ' there exists a $0 \leq k < y$ such that $return\text{-}seq(k, g) = x$ and $return\text{-}seq(i, g) \neq x$ for any $k < i < y$. Then, by repeating summand D (p times), we reach a state ξ'' with $p = 0$. Then, by repeating

summands E and F ($y-(k+1)$ times), we reach a state ξ''' with $\mathit{last\text{-}dat}(g) = d$ and $\mathit{last\text{-}seq}(g) = x$. 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 $\mathit{last\text{-}seq}(g) \neq x$, in view of Lemma VI.4.5, the value of $\mathit{next\text{-}empty}(\ell', q')$ remains the same. By again performing summand E, we reach a state ξ'''' where $q' := \mathit{inb}(d, x, q')$. Recall that $x = \mathit{next\text{-}empty}(\ell', q')$.

$$\begin{array}{ll} & \mathit{next\text{-}empty}(\ell', \mathit{in}(d, \mathit{next\text{-}empty}(\ell', q'), q')) \\ = & \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(\ell', q')), q') \\ & \text{(Lem. VI.4.6)} \\ > & \mathit{next\text{-}empty}(\ell', q') \\ & \text{(Lem. VI.4.3)} \end{array}$$

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

■

Theorem VII.2 For all $e{:}\Delta$,

$$\tau_{\{c,j\}}(\mathbf{N}_{nonmod}(0, 0, [], 0, [], []^K, 0, []^L, 0)) \underline{\leftrightarrow}_b \mathbf{Z}(\langle\rangle).$$

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

Class I:

$$\begin{array}{l} (p < \mathit{length}(g) \wedge \ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n) \\ \Rightarrow\ \phi(m, q, \ell', q') = \phi(m, q, \ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \\ \mathit{last\text{-}seq}(g), q')) \end{array}$$

$$\begin{array}{l} p' < \mathit{length}(g') \\ \Rightarrow\ \phi(m, q, \ell', q') = \phi(m, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q), \ell', q') \end{array}$$

Class II:

$$m < \ell + n\ \Rightarrow\ \mathit{length}(\phi(m, q, \ell', q')) < 2n$$

$$\mathit{test}(\ell', q')\ \Rightarrow\ \mathit{length}(\phi(m, q, \ell', q')) > 0$$

Class III:

$$\begin{array}{l} ((\ell = m \vee \ell = \ell' + n) \wedge \mathit{length}(\phi(m, q, \ell', q')) < 2n) \\ \Rightarrow\ m < \ell + n \end{array}$$

$$\begin{array}{l} ((\ell = m \vee \ell = \ell' + n) \wedge \mathit{length}(\phi(m, q, \ell', q')) > 0) \\ \Rightarrow\ \mathit{test}(\ell', q') \end{array}$$

Class IV:

$$\mathit{test}(\ell', q')\ \Rightarrow\ \mathit{retrieve}(\ell', q') = \mathit{top}(\phi(m, q, \ell', q'))$$

Class V:

$$\begin{array}{l} m < \ell + n \\ \Rightarrow\ \phi(S(m), \mathit{inb}(d, m, q), \ell', q') = \\ \mathit{append}(d, \phi(m, q, \ell', q')) \end{array}$$

$$\begin{array}{l} \mathit{test}(\ell', q') \\ \Rightarrow\ \phi(m, q, S(\ell'), \mathit{remove}(\ell', q')) = \mathit{tail}(\phi(m, q, \ell', q')) \end{array}$$

I.1 Let $p < \mathit{length}(g)$. By Lemmas VI.5.5, VI.5.6 and VI.5.10, $\mathit{member}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), g)$.
CASE 1: $\mathit{last\text{-}seq}(g) \neq \mathit{next\text{-}empty}(\ell', q')$.
By Lemma VI.4.5,
$\mathit{next\text{-}empty}(\ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'))$
$= \mathit{next\text{-}empty}(\ell', q')$. Hence,

$$\begin{array}{ll} & \phi(m, q, \ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) \\ = & \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle \\ & +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle \end{array}$$

CASE 1.1: $\ell' \leq \mathit{last\text{-}seq}(g) < \mathit{next\text{-}empty}(\ell', q')$.
By Lemma VI.4.2, $\mathit{test}(\mathit{last\text{-}seq}(g), q')$, so by Invariant VI.8.26 together with
$\mathit{member}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), g)$, $\mathit{retrieve}(\mathit{last\text{-}seq}(g), q')$
$= \mathit{last\text{-}dat}(g)$. So by Lemma VI.7.7,
$\mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle =$
$q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle$.
CASE 1.2: $\neg(\ell' \leq \mathit{last\text{-}seq}(g) < \mathit{next\text{-}empty}(\ell', q'))$.
Using Lemma VI.7.8, it follows that

$$\begin{array}{ll} & \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle \\ = & \mathit{remove}(\mathit{last\text{-}seq}(g), \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) \\ & [\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle \\ = & \mathit{remove}(\mathit{last\text{-}seq}(g), q')[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle \\ = & q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle \end{array}$$

CASE 2: $\mathit{last\text{-}seq}(g) = \mathit{next\text{-}empty}(\ell', q')$.
The derivation splits into two parts.
(1) Using Lemma VI.7.8, it follows that

$$\begin{array}{ll} & \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\ell'..\mathit{last\text{-}seq}(g)\rangle \\ = & \mathit{remove}(\mathit{last\text{-}dat}(g), \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) \\ & [\ell'..\mathit{last\text{-}seq}(g)\rangle \\ = & \mathit{remove}(\mathit{last\text{-}dat}(g), q')[\ell'..\mathit{last\text{-}seq}(g)\rangle \\ = & q'[\ell'..\mathit{last\text{-}seq}(g)\rangle \end{array}$$

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

$$\begin{array}{ll} & q'[S(\mathit{last\text{-}seq}(g))..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle \\ = & q[S(\mathit{last\text{-}seq}(g))..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle \end{array}$$

Hence,

$$\begin{aligned}
& \textit{inb}(\textit{last-dat}(g), \textit{last-seq}(g), q') \\
& [\textit{last-seq}(g)..\textit{next-empty}(S(\textit{last-seq}(g)), q')\rangle \\
= \; & \textit{inl}(\textit{last-dat}(g), \textit{inb}(\textit{last-dat}(g), \textit{last-seq}(g), q') \\
& [S(\textit{last-seq}(g))..\textit{next-empty}(S(\textit{last-seq}(g)), q')\rangle) \\
= \; & \textit{inl}(\textit{last-dat}(g), \textit{remove}(\textit{last-seq}(g), \\
& \textit{inb}(\textit{last-dat}(g), \textit{last-seq}(g), q')) \\
& [S(\textit{last-seq}(g))..\textit{next-empty}(S(\textit{last-seq}(g)), q')\rangle) \\
& \text{(Lem. VI.7.8)} \\
= \; & \textit{inl}(\textit{last-dat}(g), \textit{remove}(\textit{last-seq}(g), q') \\
& [S(\textit{last-seq}(g))..\textit{next-empty}(S(\textit{last-seq}(g)), q')\rangle) \\
= \; & \textit{inl}(\textit{last-dat}(g), \\
& q'[S(\textit{last-seq}(g))..\textit{next-empty}(S(\textit{last-seq}(g)), q')\rangle) \\
& \text{(Lem. VI.7.8)} \\
= \; & \textit{inl}(\textit{last-dat}(g), \\
& q[S(\textit{last-seq}(g))..\textit{next-empty}(S(\textit{last-seq}(g)), q')\rangle) \\
& \text{(see above)} \\
= \; & q[\textit{last-seq}(g)..\textit{next-empty}(S(\textit{last-seq}(g)), q')\rangle
\end{aligned}$$

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

$$\begin{aligned}
& \textit{inb}(\textit{last-dat}(g), \textit{last-seq}(g), q') \\
& [\ell'..\textit{next-empty}(\ell', \textit{inb}(\textit{last-dat}(g), \textit{last-seq}(g), q'))\rangle \\
& +\!\!+ q[\textit{next-empty}(\ell', \textit{inb}(\textit{last-dat}(g), \textit{last-seq}(g), q')) \\
& ..m\rangle \\
= \; & \textit{inb}(\textit{last-dat}(g), \textit{last-seq}(g), q') \\
& [\ell'..\textit{next-empty}(S(\textit{last-seq}(g)), q')\rangle \\
& +\!\!+ q[\textit{next-empty}(S(\textit{last-seq}(g)), q')..m\rangle \\
& \text{(Lem. VI.4.6)} \\
= \; & (\textit{inb}(\textit{last-dat}(g), \textit{last-seq}(g), q') \\
& [\ell'..\textit{last-seq}(g)\rangle \\
& +\!\!+ \textit{inb}(\textit{last-dat}(g), \textit{last-seq}(g), q') \\
& [\textit{last-seq}(g)..\textit{next-empty}(S(\textit{last-seq}(g)), q')\rangle) \\
& +\!\!+ q[\textit{next-empty}(S(\textit{last-seq}(g)), q')..m\rangle \\
& \text{(Lem. VI.4.3, VI.7.5)} \\
= \; & (q'[\ell'..\textit{last-seq}(g)\rangle \\
& +\!\!+ q[\textit{last-seq}(g)..\textit{next-empty}(S(\textit{last-seq}(g)), q')\rangle) \\
& +\!\!+ q[\textit{next-empty}(S(\textit{last-seq}(g)), q')..m\rangle \\
& \text{(1), (2)} \\
= \; & q'[\ell'..\textit{last-seq}(g)\rangle +\!\!+ q[\textit{last-seq}(g)..m\rangle \\
& \text{(Lem. VI.7.1, VI.4.2, VI.7.5)}
\end{aligned}$$

I.2 Let $p' < \textit{length}(g')$. By Lemmas VI.6.4 and VI.6.8, $\textit{member}(\textit{last-seq}(g'), g')$.
By Invariant VI.8.3, $\textit{last-seq}(g') \leq \textit{next-empty}(\ell', q')$.
So by Lemma VI.7.9, $\textit{release}(\ell, \textit{last-seq}(g'), q)$ $[\textit{next-empty}(\ell', q')..m\rangle = q[\textit{next-empty}(\ell', q')..m\rangle$

II.1 Let $m < \ell + n$.

$$\begin{aligned}
& \textit{length}(q'[\ell'..\textit{next-empty}(\ell', q')\rangle +\!\!+ \\
& q[\textit{next-empty}(\ell', q')..m\rangle) \\
= \; & \textit{length}(q'[\ell'..\textit{next-empty}(\ell', q')\rangle) \\
& + \textit{length}(q[\textit{next-empty}(\ell', q')..m\rangle)) \\
& \text{(Lem. VI.7.2)} \\
= \; & (\textit{next-empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \textit{next-empty}(\ell', q')) \\
& \text{(Lem. VI.7.4)} \\
\leq \; & n + (m \dot{-} \ell) \\
& \text{(Inv. VI.8.13, VI.8.4)} \\
< \; & 2n
\end{aligned}$$

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

$$\begin{aligned}
& \textit{length}(\phi(m, q, \ell', q')) \\
= \; & (\textit{next-empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \textit{next-empty}(\ell', q')) \\
> \; & 0
\end{aligned}$$

III.1 CASE 1: $\ell = m$.
Then $m < \ell + n$ holds trivially.
CASE 2: $\ell = \ell' + n$.

$$\begin{aligned}
& \textit{length}(\phi(m, q, \ell', q')) \\
= \; & (\textit{next-empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \textit{next-empty}(\ell', q')) \\
& \text{(Lem. VI.7.2, VI.7.4)} \\
\leq \; & ((\ell' + n) \dot{-} \ell') + (m \dot{-} \ell) \\
& \text{(Inv. VI.8.13, VI.8.4)} \\
= \; & n + (m \dot{-} \ell)
\end{aligned}$$

So $\textit{length}(\phi(m, q, \ell', q')) < 2n$ implies $m < \ell + n$.
III.2 CASE 1: $\ell = m$.
Then $m \dot{-} \textit{next-empty}(\ell', q') \leq m \dot{-} \ell$ (Inv. VI.8.4) $= 0$, so

$$\begin{aligned}
& \textit{length}(\phi(m, q, \ell', q')) \\
= \; & (\textit{next-empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \textit{next-empty}(\ell', q')) \\
& \text{(Lem. VI.7.2, VI.7.4)} \\
= \; & \textit{next-empty}(\ell', q') \dot{-} \ell'
\end{aligned}$$

Hence, $\textit{length}(\phi(m, q, \ell', q')) > 0$ yields $\textit{next-empty}(\ell', q') > \ell'$, which implies $\textit{test}(\ell', q')$.
CASE 2: $\ell = \ell' + n$.
Then by Invariant VI.8.4, $\textit{next-empty}(\ell', q') \geq \ell' + n$, which implies $\textit{test}(\ell', q')$.
IV $\textit{test}(\ell', q')$ implies $\textit{next-empty}(\ell', q')$ $= \textit{next-empty}(S(\ell'), q') \geq S(\ell')$ (Lem. VI.4.3). Hence,

$$\begin{aligned}
& q'[\ell'..\textit{next-empty}(\ell', q')\rangle \\
= \; & \textit{inl}(\textit{retrieve}(\ell', q'), q'[S(\ell')..\textit{next-empty}(\ell', q')\rangle)
\end{aligned}$$

So

$$\begin{array}{ll} & top(\phi(m,q,\ell',q')) \\ = & top(inl(retrieve(\ell',q'), q'[S(\ell')..next\text{-}empty(\ell',q')\rangle \\ & {+\!\!+}q[next\text{-}empty(\ell',q')..m\rangle)) \\ = & retrieve(\ell',q') \end{array}$$

V.1

$$\begin{array}{ll} & q'[\ell'..next\text{-}empty(\ell',q')\rangle{+\!\!+} \\ & inb(d,m,q)[next\text{-}empty(\ell',q')..S(m)\rangle \\ = & q'[\ell'..next\text{-}empty(\ell',q')\rangle{+\!\!+} \\ & append(d,q[next\text{-}empty(\ell',q')..m\rangle) \\ & \text{(Lem. VI.7.6, Inv. VI.8.12)} \\ = & append(d,q'[\ell'..next\text{-}empty(\ell',q')\rangle{+\!\!+} \\ & q[next\text{-}empty(\ell',q')..m\rangle) \\ & \text{(Lem. VI.7.3)} \end{array}$$

V.2 $test(\ell',q')$ and Lemma VI.4.3 imply $next\text{-}empty(\ell',q') = next\text{-}empty(S(\ell'),q') \geq S(\ell')$. Hence,

$$\begin{array}{ll} & remove(\ell',q')[S(\ell').. \\ & next\text{-}empty(S(\ell'),remove(\ell',q'))\rangle \\ & {+\!\!+}q[next\text{-}empty(S(\ell'),remove(\ell',q'))..m\rangle \\ = & remove(\ell',q')[S(\ell')..next\text{-}empty(S(\ell'),q')\rangle \\ & {+\!\!+}q[next\text{-}empty(S(\ell'),q')..m\rangle \\ & \text{(Lem. VI.4.7)} \\ = & remove(\ell',q')[S(\ell')..next\text{-}empty(\ell',q')\rangle \\ & {+\!\!+}q[next\text{-}empty(\ell',q')..m\rangle \\ = & q'[S(\ell')..next\text{-}empty(\ell',q')\rangle \\ & {+\!\!+}q[next\text{-}empty(\ell',q')..m\rangle \\ & \text{(Lem. VI.7.8)} \\ = & tail(q'[\ell'..next\text{-}empty(\ell',q')\rangle \\ & {+\!\!+}q[next\text{-}empty(\ell',q')..m\rangle) \end{array}$$

■

C. Correctness of the Sliding Window Protocol

Finally, we can prove Theorem IV.1.

Proof:

$$\begin{array}{ll} & \tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^K,0) \parallel \mathbf{L}([]^L,0))) \\ \underline{\leftrightarrow} & \tau_{\mathcal{I}}(\mathbf{M}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0)) \\ & \text{(Thm. V.1)} \\ \underline{\leftrightarrow} & \tau_{\{c,j\}}(\mathbf{N}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0)) \\ & \text{(Thm. V.2)} \\ \underline{\leftrightarrow} & \tau_{\{c,j\}}(\mathbf{N}_{nonmod}(0,0,[],0,[],[]^K,0,[]^L,0)) \\ & \text{(Thm. V.3)} \\ \underline{\leftrightarrow}_b & \mathbf{Z}(\langle\rangle) \\ & \text{(Thm. VII.2)} \end{array}$$

■

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.

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 [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

[1] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60:109–137, 1984.

[2] J.A. Bergstra and J.W. Klop. Verification of an alternating bit protocol by means of process algebra. In *Proc. Spring School on Mathematical Methods of Specification and Synthesis of Software Systems*, LNCS 215, pp. 9–23. Springer, 1986.

[3] M.A. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.

[4] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In *Proc. 5th Conference on Concurrency Theory*, LNCS 836, pp. 401–416. Springer, 1994.

[5] J.J. Brunekreef. Sliding window protocols. In *Algebraic Specification of Protocols.* Cambridge Tracts in Theoretical Computer Science 36, pp. 71–112. Cambridge University Press, 1993.

[6] R. Cardell-Oliver. sing higher order logic for modelling real-time protocols. In *Proc. 4th Joint Conference on Theory and Practice of Software Development*, LNCS 494, pp. 259–282. Springer, 1991.

[7] V.G. Cerf and R.E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, COM-22:637–648, 1974.

[8] D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In *Proc. 9th Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2619, pp. 113–127. Springer, 2003.

[9] W.J. Fokkink and J. Pang. Cones and foci for protocol verification revisited. In *Proc. 6th Conference on Foundations of Software Science and Computation Structures*, LNCS 2620, pp. 267–281, Springer, 2003.

[10] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:555–600, 1996.

[11] P. Godefroid and D.E. Long. Symbolic protocol verification with Queue BDDs. *Formal Methods and System Design*, 14(3):257–271, 1999.

[12] R.A. Groenveld. Verification of a sliding window protocol by means of process algebra. Report P8701, University of Amsterdam, 1987.

[13] J.F. Groote. *Process Algebra and Structured Operational Semantics.* PhD thesis, University of Amsterdam, 1991.

[14] J.F. Groote and H.P. Korver. Correctness proof of the bakery protocol in μCRL. In *Proc. 1st Workshop on the Algebra of Communicating Processes*, Workshops in Computing, pp. 63–86. Springer, 1995.

[15] J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In *Proc. Workshop on Semantics of Specification Languages*, Workshops in Computing, pp. 232–251. Springer, 1994.

[16] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Proc. 1st Workshop on the Algebra of Communicating Processes*, Workshops in Computing Series, pp. 26–62. Springer, 1995.

[17] J. F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1/2):39–72, 2001.

[18] J.F. Groote and M. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, eds, *Handbook of Process Algebra*, pp. 1151–1208. Elsevier, 2001.

[19] J.F. Groote and J. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. *Journal of Logic and Algebraic Programming*, 49(1/2):31–60, 2001.

[20] B.T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic.* LNCS 129, Springer, 1982.

[21] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[22] G.J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279-295, 1997.

[23] B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Department of Computer Science, Uppsala University, 1987.

[24] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Proc. 6th Conference on Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1785, pp. 220–234. Springer, 2000

[25] R. Kaivola. Using compositional preorders in the verification of sliding window protocol. In *Proc. 9th Conference on Computer Aided Verification*, LNCS 1254, pp. 48–59. Springer, 1997.

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

[27] T. Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In *Proc. 22nd Conference on Application and Theory of Petri Nets*, LNCS 2075, pp. 242–262. Springer, 2001.

[28] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley/Teubner, 1996.

[29] E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol in Lotos. In *Proc. 4th Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, IFIP Transactions, pp. 495-510. North-Holland, 1991.

[30] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, 13(2):85–139, 1990.

[31] A. Middeldorp. Specification of a sliding window protocol within the framework of process algebra. Report FVI 86-19, University of Amsterdam, 1986.

[32] K. Paliwoda and J.W. Sanders. An incremental specification of the sliding-window protocol. *Distributed Computing*, 5:83–94, 1991.

[33] D.M.R. Park. Concurrency and automata on infinite sequences. In *Proc. 5th GI-Conference on Theoretical Computer Science*, LNCS 104, pp. 167–183. Springer, 1981.

[34] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in Xesar of the sliding window protocol. In *Proc. 7th Conference on Protocol Specification, Testing and Verification*, pp. 235–248. North-Holland, 1987.

[35] C. Röckl and J. Esparza. Proof-checking protocols using bisimulations. In *Proc. 10th Conference on Concurrency Theory*, LNCS 1664, pp. 525–540. Springer, 1999.

[36] V. Rusu. Verifying a sliding-window protocol using PVS. In *Proc. 21st Conference on Formal Techniques for Networked and Distributed Systems*, IFIP Conference Proceedings 197, pp. 251-268. Kluwer, 2001.

[37] A.A. Schoone. *Assertional Verification in Distributed Computing*. PhD thesis, Utrecht University, 1991.

[38] M.A. Smith and N. Klarlund. Verification of a sliding window protocol using IOA and MONA. In *Proc. 2oth Conference on Formal Techniques for Distributed System Development*, IFIP Conference Proceedings 183, pp. 19–34. Kluwer, 2000.

[39] J.L.A. van de Snepscheut. The sliding window protocol revisited. *Formal Aspects of Computing*, 7(1):3–17, 1995.

[40] K. Stahl, K. Baukus, Y. Lakhnech, and M. Steffen. Divide, abstract, and model-check. In *Proc. 6th International SPIN Workshop*, LNCS 1680, pp. 57–76. Springer, 1999.

[41] N.V. Stenning. A data transfer protocol. *Computer Networks*, 1(2):99–110, 1976.

[42] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1981.

[43] F.W. Vaandrager. Verification of two communication protocols by means of process algebra. Report CS-R8608, CWI, Amsterdam, 1986.

[44] J.J. van Wamel. A study of a one bit sliding window protocol in ACP. Report P9212, University of Amsterdam, 1992.