Under consideration for publication in Formal Aspects of Computing

Verification of a Sliding Window Protocol in μCRL and PVS

Bahareh Badban[1], Wan Fokkink[1,4], Jan Friso Groote[1,3], Jun Pang[2], and Jaco van de Pol[1,3]

[1]CWI
Department of Software Engineering
PO Box 94079, 1090 GB Amsterdam, The Netherlands
[2] INRIA Futurs and LIX, École Polytechnique
Rue de Saclay, 91128 Palaiseau Cedex, France
[3] Eindhoven University of Technology
Department of Computer Science
PO Box 513, 5600 MB Eindhoven, The Netherlands
[4] Vrije Universiteit Amsterdam
Department of Computer Science
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

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

1. Introduction

Sliding window protocols [CeK74] (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 [Tan81]

Correspondence and offprint requests to: Wan Fokkink, Department of Theoretical Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. email: `wanf@cs.vu.nl` 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 2, 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 [Mid86] and Brunekreef [Bru93] gave specifications in ACP [BeK84] and PSF [MaV90], respectively. Vaandrager [Vaa86], Groenveld [Gro87], van Wamel [Wam92] and Bezem and Groote [BeG94a] 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 [Tan81] (not taking into account additional features such as duplex message passing and piggybacking) correct using μCRL [GrP95], 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 [Tan81], 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 [Tan81] contained a deadlock [Gro91, 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 [Tan81]. 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 [GrK95], and to the development of the *cones and foci* proof method [GrS01, FoP03]. 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 [Sch91], 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 [GlW96] 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 [BeG94b]. 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 [ORR$^+$96]. The PVS files are available via `http://homepages.cwi.nl/~vdpol/swp.html`.

A concise overview of other verifications of SWPs is presented in Section 2. 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 2 gives an overview of related work on verifying SWPs. Section 3 introduces the process part of μCRL. In Section 4, the data types needed for specifying the SWP and its external behavior are presented. Section 5 features the μCRL specifications of the SWP and its external behavior. In Section 6, 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 7, properties of the data types and invariants of the transformed specification are proved. In Section 8, it is proved that the three transformations preserve branching bisimilarity, and that the transformed specification behaves like a FIFO queue. In Section 9, we present the formalization and verification of the SWP in PVS [ORR$^+$96]. Finally, we conclude the paper in Section 10.

An earlier version of this paper appeared as [FGP$^+$04], where the medium between the sender and the receiver was modeled as a lossy queue of capacity one. Here, we model the medium as a lossy queue of unbounded capacity, which is more realistic and further complicates the verification effort. In this paper, we also present equational definitions of the data types, lemmas regarding these data types, all invariants, and detailed correctness proofs, which were for a large part omitted in [FGP$^+$04]. Moreover, in Section 9 we report on the formalization and verification of the SWP in PVS.

2. 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 buffer and window size at the sender and the receiver. Case studies that do treat arbitrary finite buffer and window sizes mostly restrict to safety properties.

Unbounded sequence numbers Stenning [Ste76] 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 [Knu81] examined more general principles behind Stenning's protocol, and manually verified some safety properties. Hailpern [Hai82] used temporal logic to formulate safety and liveness properties for Stenning's protocol, and established their validity by informal reasoning. Jonsson [Jon87] also verified safety and liveness properties of the protocol, using temporal logic and a manual compositional verification technique. Rusu [Rus01] used the theorem prover PVS to verify safety and liveness properties for a SWP with unbounded sequence numbers.

Fixed finite window size Richier *et al.* [RRS$^+$87] specified a SWP in a process algebra based language Estelle/R, and verified safety properties for window size up to eight using the model checker Xesar. Madelaine and Vergamini [MaV91] specified a SWP in Lotos, with the help of the simulation environment Lite, and proved some safety properties for window size six. Holzmann [Hol91, Hol97] used the Spin model checker to verify safety and liveness properties of a SWP with sequence numbers up to five. Kaivola [Kai97] verified safety and liveness properties using model checking for a SWP with window size up to seven. Godefroid and Long [GoL99] specified a full duplex SWP in a guarded command language, and verified the protocol for window size two using a model checker based on Queue BDDs. Stahl *et al.* [SBL$^+$99] 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 [SmK00] 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. Jonsson and Nilsson [JoN00] used an automated reachability analysis to verify safety properties for a SWP with a receiving window of size one. Latvala [Lat01] 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 [Car91] specified a SWP using higher order logic, and manually proved and mechanically checked safety properties using HOL. (Van de Snepscheut [Sne95] noted that what Cardell-Oliver claims to be a liveness property is in fact a safety property.) Schoone [Sch91] manually proved safety properties for several SWPs using assertional verification. Van de Snepscheut [Sne95] gave a correctness proof of a SWP as a sequence of correctness preserving transformations of a sequential program. Paliwoda and Sanders [PaS91] specified a reduced version of what they call a SWP (but which is in fact very similar to the bakery protocol from [GrK95]) in the process algebra CSP, and verified a safety property modulo trace semantics. Röckl and Esparza [RoE99] verified the correctness of this bakery protocol modulo weak bisimilarity using Isabelle/HOL, by explicitly checking a bisimulation relation. Chkliaev *et al.* [CHV03] 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.

3. μCRL

μCRL [GrP95] (see also [GrR01]) is a language for specifying distributed systems and protocols in an algebraic style. It is based on the process algebra ACP [BeK84] extended with equational abstract data types [LEW96]. We will use $\approx$ for equality between process terms and $=$ for equality between data terms.

A μCRL specification of data types consists of two parts: A signature, consisting of function symbols from which one can build data terms, and axioms that induce an equality relation on data terms of the same type. They provide a loose semantics, meaning that it is allowed to have multiple models. The data types needed for our μCRL specification of a SWP are presented in Section 4. In particular we have 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 process part of μCRL is specified using a number of pre-defined process algebraic operators, which we will present below. From these operators one can build process terms, which describe the order in which the atomic actions from a set $\mathcal{A}$ may happen. A process term consists of actions and recursion variables combined by the 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.

Two elementary operators to construct processes are *sequential composition*, written $p{\cdot}q$, and *alternative composition*, written $p + q$. The process $p{\cdot}q$ first executes p, until p terminates, and then continues with executing q. The process $p+q$ non-deterministically behaves as either p or q. *Summation* $\sum_{d:D} p(d)$ provides the possibly infinite non-deterministic choice over a data type D. For example, $\sum_{n:Nat} a(n)$ can perform the action $a(n)$ for all natural numbers n. 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* $p \parallel q$ performs the processes p and q in parallel; in other words, it consists of the arbitrary interleaving of actions of the processes p and q. For example, if there is no communication possible between actions a and b, then $a \parallel b$ behaves as $a{\cdot}b + b{\cdot}a$. 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. For example, if actions a and b communicate to c, then $\partial_{\{a,b\}}(a \parallel b) \approx c$. *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). For example, let $X(n{:}Nat) \approx a(n){\cdot}X(n+1)$; then $X(0)$ can execute the infinite sequence of actions $a(0){\cdot}a(1){\cdot}a(2) \cdot \cdots$.

A recursive specification is a *linear process equation (LPE)* if it is of the form

$$X(d{:}D) \approx \sum_{a\in\mathcal{A}} \sum_{e_a:E_a} a(f_a(d, e_a)){\cdot}X(g_a(d, e_a)) \lhd h_a(d, e_a) \rhd \delta$$

with $f_a : D \times E_a \to D_a$, $g_a : D \times E_a \to D$, and $h_a : D \times E_a \to Bool$. Note that an LPE does not contain parallel composition, encapsulation and hiding, and uses only one recursion variable. Groote, Ponse and Usenko [GPU01] presented an algorithm that transforms each μCRL specification into an LPE.

The μCRL specification of the data part of a SWP is presented in Section 4, while the process part is presented in Section 5.1. The μCRL specification of the external behaviour of this SWP, being a FIFO queue, is presented in Section 5.2. Section 6.1 contains the LPE that results from applying this linearization algorithm to the μCRL specification of a SWP in Section 5.1

To each μCRL specification belongs a directed graph, called a labeled transition system. In this labeled transition system, the states are process terms, and the edges are labeled with parameterized actions. For example, given the μCRL specification $X(n{:}Nat) \approx a(n){\cdot}X(n + 1)$, we have transitions $X(n) \overset{a(n)}{\to} X(n + 1)$. Branching bisimilarity $\underline{\leftrightarrow}_b$ [GlW96] and strong bisimilarity $\underline{\leftrightarrow}$ [Par81] are two well-established equivalence relations on states in labeled transition systems.[1] Conveniently, strong bisimilarity implies branching bisim-

[1] The definitions of these relations often take into account a special predicate on states to denote succesful termination. This predicate is missing here, as successful termination does not play a role in our SWP specification.

ilarity. The proof theory of μCRL from [GrP94] is sound modulo branching bisimilarity, meaning that if $p \approx q$ can be derived from it then $p \mathrel{\underline{\leftrightarrow}_b} q$.

Definition 3.1 (Branching bisimulation) Given a labeled transition system. A *strong bisimulation relation* $\mathcal{B}$ is a symmetric binary relation on states such that if $s \,\mathcal{B}\, t$ and $s \xrightarrow{\ell} s'$, then $t \xrightarrow{\ell} t'$ with $s' \,\mathcal{B}\, t'$. Two states s and t are *strongly bisimilar*, denoted by $s \mathrel{\underline{\leftrightarrow}} t$, if there is a strong bisimulation relation $\mathcal{B}$ such that $s \,\mathcal{B}\, t$.

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 $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} \hat{t}$ such that $s \,\mathcal{B}\, \hat{t}$ and $\hat{t} \xrightarrow{\ell} t'$ with $s' \,\mathcal{B}\, t'$.

Two states s and t are *branching bisimilar*, denoted by $s \mathrel{\underline{\leftrightarrow}_b} t$, if there is a branching bisimulation relation $\mathcal{B}$ such that $s \,\mathcal{B}\, t$.

See [Gla94] for a lucid exposition on why branching bisimilarity constitutes a sensible equivalence relation for concurrent processes.

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, in Section 8, we will use four proof techniques from the literature to derive that two μCRL specifications are branching (or even strongly) bisimilar: sum elimination, invariants, CL-RSP, and cones and foci.

Sum elimination [GrK95] states that a summation over a data type from which only one element can be selected can be removed.

Theorem 3.2 (Sum elimination) $\sum_{d:D} p(d) \lhd d = e \wedge b \rhd \delta \;\; \underline{\leftrightarrow} \;\; p(e) \lhd b \rhd \delta$.

An *invariant* $I : D \to \mathit{Bool}$ characterizes the set of reachable states of an LPE $X(d{:}D)$. That is, if $I(d) = \mathsf{t}$ and X can evolve from d to d' in zero or more transitions, then $I(d') = \mathsf{t}$.

Definition 3.3 (Invariant) $I : D \to \mathit{Bool}$ is an *invariant* for an LPE

$$X(d{:}D) \approx \sum_{a \in \mathcal{A}} \sum_{e_a : E_a} a(f_a(d, e_a)) \cdot X(g_a(d, e_a)) \lhd h_a(d, e_a) \rhd \delta$$

if for all $d{:}D$, $a{:}\mathcal{A}$ and $e_a{:}E_a$,

$$(I(d) \wedge h_a(d, e_a)) \;\Rightarrow\; I(g_a(d, e_a)).$$

If I holds in a state d and $X(d)$ can perform a transition, meaning that $h_a(d, e_a) = \mathsf{t}$ for some $e_a{:}E_a$, then it is ensured by the definition above that I holds in the resulting state $g_a(d, e_a)$.

CL-RSP [BeG94b] states that the solutions of an LPE are all strongly bisimilar. This proof principle basically extends RSP [BeK86] to a setting with data. It says that if process terms $t(d)$ are solutions for recursion variables $X(d)$ for $d{:}D$, where $X(d{:}D)$ is an LPE, then $t(d)$ and $X(d)$ are strongly bisimilar for $d{:}D$. For example, consider the LPEs $X(b{:}\mathit{Bool}) \approx a \cdot X(\neg b)$ and $Y \approx a \cdot Y$. Substituting the process term Y for both $X(\mathsf{t})$ and $X(\mathsf{f})$ results in sound equations modulo $\underline{\leftrightarrow}$, so according to CL-RSP, $Y \mathrel{\underline{\leftrightarrow}} X(\mathsf{t})$ and $Y \mathrel{\underline{\leftrightarrow}} X(\mathsf{f})$. Given an invariant I, we only need to find solutions $t(d)$ for $d{:}D$ with $I(d) = \mathsf{t}$.

Theorem 3.4 (CL-RSP) Consider an LPE

$$X(d{:}D) \;\approx\; \textstyle\sum_{a \in \mathcal{A}} \sum_{e : E_a} a(f_a(d, e)) \cdot X(g_a(d, e)) \lhd h_a(d, e) \rhd \delta$$

Let $I : D \to \mathit{Bool}$ be an invariant for X. Let $t(d)$ be process terms such that, for all $d{:}D$ with $\mathcal{I}(d) = \mathsf{t}$,

$$t(d) \;\underline{\leftrightarrow}\; \textstyle\sum_{a \in \mathcal{A}} \sum_{e : E_a} a(f_a(d, e)) \cdot t(g_a(d, e)) \lhd h_a(d, e) \rhd \delta$$

Then $t(d) \mathrel{\underline{\leftrightarrow}} X(d)$ for all $d{:}D$ with $\mathcal{I}(d) = \mathsf{t}$.

The *cones and foci* method from [GrS01, FoP03] rephrases the question whether $\tau_{\mathcal{I}}(X(d))$ and $Y(d')$ are branching bisimilar in terms of data equalities, where $X(d{:}D)$ and $Y(d'{:}D')$ are LPEs, and the latter LPE does not contain actions from some set $\mathcal{I}$ of internal actions. A *state mapping* ϕ relates each state in $X(d)$ to a state in $Y(d')$. Furthermore, some $d{:}D$ are declared to be *focus points*. The *cone* of a focus point consists of the states in $X(d)$ that can reach this focus point by a string of actions from $\mathcal{I}$. It is required that each reachable state in $X(d)$ is in the cone of a focus point. If a number of *matching criteria* are satisfied, then ϕ establishes a branching bisimulation relation between terms $\tau_{\mathcal{I}}(X(d))$ and $Y(\phi(d))$.

For example, consider the LPEs $X(b{:}Bool) \approx a{\cdot}X(\neg b) \lhd b \rhd \delta + c{\cdot}X(b) \lhd \neg b \rhd \delta$ and $Y(d'{:}D') \approx a{\cdot}Y(d')$, with $\mathcal{I} = \{c\}$ and focus point t. Then for any $d'{:}D'$, the state mapping $\phi(b) = d'$ for $b{:}Bool$ satisfies the matching criteria.

Given an invariant I, only $d{:}D$ with $I(d) = \mathsf{t}$ need to be in the cone of a focus point, and we only need to satisfy the matching criteria for $d{:}D$ with $I(d) = \mathsf{t}$.

Definition 3.5 (Matching criteria) Given two LPEs:

$$X(d{:}D) \quad \approx \quad \textstyle\sum_{a\in\mathcal{A}} \sum_{e:E_a} a(f_a(d,e)){\cdot}X(g_a(d,e)) \lhd h_a(d,e) \rhd \delta$$

$$Y(d'{:}D') \quad \approx \quad \textstyle\sum_{a\in\mathcal{A}\backslash\mathcal{I}} \sum_{e:E_a} a(f'_a(d',e)){\cdot}Y(g'_a(d',e)) \lhd h'_a(d',e) \rhd \delta$$

Let a predicate FC on D designate the focus points. A state mapping $\phi : D \to D'$ satisfies the *matching criteria* for $d{:}D$ if for all $a \in \mathcal{A}\backslash\mathcal{I}$ and $c \in \mathcal{I}$:

I $\forall e{:}E_c\,(h_c(d,e) \Rightarrow \phi(d) = \phi(g_c(d,e)))$;
II $\forall e{:}E_a\,(h_a(d,e) \Rightarrow h'_a(\phi(d),e))$;
III $FC(d) \Rightarrow \forall e{:}E_a\,(h'_a(\phi(d),e) \Rightarrow h_a(d,e))$;
IV $\forall e{:}E_a\,(h_a(d,e) \Rightarrow f_a(d,e) = f'_a(\phi(d),e))$;
V $\forall e{:}E_a\,(h_a(d,e) \Rightarrow \phi(g_a(d,e)) = g'_a(\phi(d),e))$.

Matching criterion I requires that the internal transitions at d are inert, meaning that d and $g_c(d,e)$ are branching bisimilar for $c \in \mathcal{I}$. 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 3.6 (Cones and foci) Given LPEs $X(d{:}D)$ and $Y(d'{:}D')$ written as in Definition 3.5. Let $I : D \to Bool$ be an invariant for X. Suppose that for all $d{:}D$ with $I(d)$:

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 X can perform transitions $d \xrightarrow{c_1} \cdots \xrightarrow{c_k} \hat{d}$ with $c_1, \ldots, c_k \in \mathcal{I}$.

Then for all $d{:}D$ with $I(d)$, $\tau_{\mathcal{I}}(X(d)) \,\underline{\leftrightarrow}_b\, Y(\phi(d))$.

4. 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.

4.1. Booleans

We introduce the data type $Bool$ of booleans.

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

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:

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

4.2. If-then-else and Equality

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

$$\mathit{if} : \mathit{Bool} \times D \times D \to D$$

with as defining equations

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

Furthermore, for each data type D in this paper one can easily define a mapping $\mathit{eq} : D \times D \to \mathit{Bool}$ such that $\mathit{eq}(d, e)$ holds if and only if $d = e$ can be derived. For notational convenience we take the liberty to write $d = e$ instead of $\mathit{eq}(d, e)$.

4.3. Natural Numbers

We introduce the data type Nat of natural numbers.

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

Here, 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. In the proofs we will take notational liberties like omitting the sign for multiplication, and abbreviating $\neg(i = j)$ to $i \neq j$, $(k < \ell) \wedge (\ell < m)$ to $k < \ell < m$, $S(0)$ to 1, and $S(S(0))$ to 2.

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

We take as binding convention:

$$\{=, \neq\} > \{\cdot\} > \{+, \dot{-}\} > \{\leq, <, \geq, >\} > \{\neg\} > \{\wedge, \vee\} > \{\Rightarrow, \Leftrightarrow\}.$$

4.4. 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:

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

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

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

4.5. 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 cell (or sequence number) i of the buffer is occupied by datum d; cells for which no datum is specified are empty. The data type *Buf* is specified as follows, where [] denotes the empty buffer:

$$\begin{array}{l} [] :\to Buf \\ inb : \Delta \times Nat \times Buf \to Buf \end{array}$$

$q|_n$ 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 cell i in q is occupied, *retrieve*(i, q) produces the datum that resides at cell i in buffer q (if this cell is occupied),[2] and *remove*(i, q) is obtained by emptying cell i in buffer q.

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

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

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

next-empty(i, q) produces the first empty cell in q, counting upwards from sequence number i onward. *next-empty*$|_n(i, q)$ does the same modulo n.

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

Intuitively, *in-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.

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

Finally, we define an operation on buffers that is only needed in the derivation of some data equalities in Section 7.1: *max*(q) produces the greatest sequence number that is occupied in q.

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

[2] 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, with the first approach an occurrence of *retrieve*$(i, [])$ might remain undetected, and the second approach would needlessly complicate the data type Δ. We prefer to work with an underspecified version of *retrieve*, which is allowed in μCRL, since data types have a loose semantics. All operations in μCRL data models, however, are total; underspecified operations lead to the existence of multiple models.

4.6. 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.

$$\begin{array}{l} []^K :\to \mathit{MedK} \\ \mathit{inm} : \Delta \times \mathit{Nat} \times \mathit{MedK} \to \mathit{MedK} \end{array}$$

$g|_n$ denotes medium g with all sequence numbers taken modulo n.

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

$\mathit{member}(d,i,g)$ produces t if and only if the pair (d,i) is in g. $\mathit{length}(g)$ denotes the length of g. $\mathit{return\text{-}dat}(i,g)$ and $\mathit{return\text{-}seq}(i,g)$ produce the datum and the sequence number, respectively, that reside at position i in g (positions are counted from 0). For convenience, we use $\mathit{last\text{-}dat}(g)$ and $\mathit{last\text{-}seq}(g)$ to produce the datum and the sequence number, respectively, that reside at the end of g. $\mathit{delete}(i,g)$ is obtained by emptying position i in g. Similarly, $\mathit{delete\text{-}last}(g)$ is obtained by emptying the last position in g.

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

The medium containing the sequence numbers from the receiver to the sender by a data type *MedL*. Similarly, we have the following defining equations.

$$\begin{array}{l} []^L :\to \mathit{MedL} \\ \mathit{inm} : \mathit{Nat} \times \mathit{MedL} \to \mathit{MedL} \end{array}$$

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

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

4.7. 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.

$$\begin{array}{l} \langle\rangle :\to \mathit{List} \\ \mathit{inl} : \Delta \times \mathit{List} \to \mathit{List} \end{array}$$

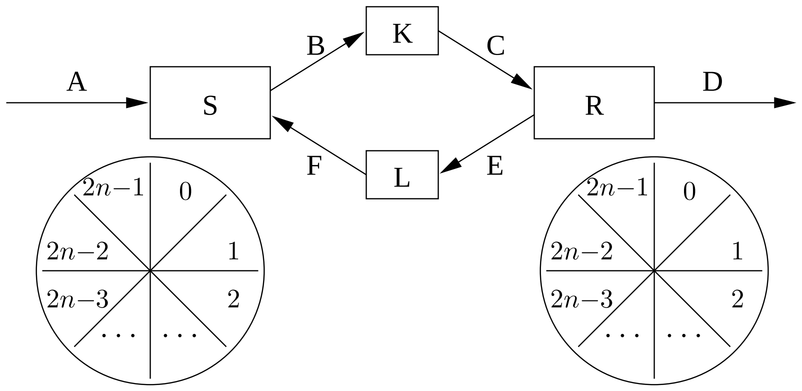

Fig. 1. Sliding window protocol

$length(\lambda)$ 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.

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

5. Sliding Window Protocol

In this section, a μCRL specification of a SWP is presented, together with its desired external behavior.

5.1. 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 cell in its window via channel D; in that case the receiving window slides forward by one cell. Furthermore, **R** can send the sequence number of the first empty cell 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 cell in the sending window, and m the first empty cell in (or just outside) the sending window. Data elements can be

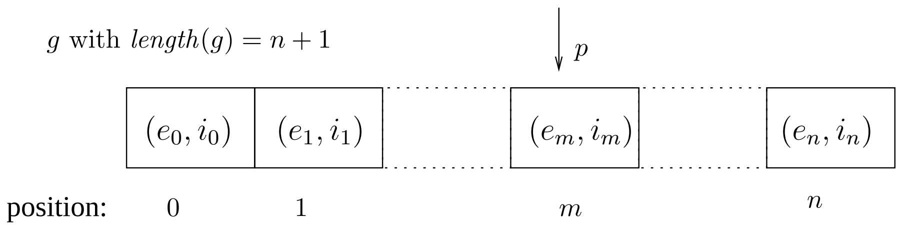

Fig. 2. The medium **K**

selected at random for transmission from (the filled part of) the sending window.

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

The receiver **R** is modeled by the process $\mathbf{R}(\ell', q')$, where q' is a buffer of size $2n$ and ℓ' the first cell in the receiving window.

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

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 frames 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 $g{:}MedK$ is a buffer with unbounded capacity, and $p{:}Nat$ a pointer indicating that the frames at positions $0, \ldots, p \dot{-} 1$ can still be lost, while the frames beyond p cannot be lost anymore and can be communicated to **R**.

K receives a frame from **S**, stores it at the front (position 0) of g, and accordingly increases p by one. It sends the last frame $(last\text{-}dat(g), last\text{-}seq(g))$ 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 ($p{:=}p \dot{-} 1$). 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 $\mathbf{L}(g', p')$.

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

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

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 $\mathcal{I} = \{c_{\mathrm{B}}, c_{\mathrm{C}}, c_{\mathrm{E}}, c_{\mathrm{F}}, j\}$.

5.2. 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

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

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 5.1 (Correctness of SWP) $\tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^{K},0) \parallel \mathbf{L}([]^{L},0))) \underline{\leftrightarrow}_b \mathbf{Z}(\langle\rangle)$.

6. 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.

6.1. Linearization

The starting point of our correctness proof is a linear specification $\mathbf{M}_{mod}$, in which no parallel composition, encapsulation and hiding 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 the linearization algorithm presented in [GPU01].

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.

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

The intuition for the LPE $\mathbf{M}_{mod}$ is as follows:

- The first summand describes that a datum d can be received by **S** through channel A if its window is not full ($in\text{-}window(\ell, m, (\ell+n)|_{2n})$). This datum is then placed in the first empty cell of the sending window ($q{:=}inb(d,m,q)$), and the next cell becomes the first empty cell ($m{:=}S(m)|_{2n}$).
- By the second summand, a frame ($retrieve(k,q),k$) can be communicated to **K** if cell k in the sending window is occupied ($test(k,q)$). This frame is then added to the buffer of **K** ($g{:=}inm(retrieve(k,q),k,g)$) and can be lost ($p{:=}S(p)$).
- The third summand describes that the first p messages in the buffer of **K** (i.e., the last p messages to enter the medium **K**) can get lost. The fourth summand describes that if $p>0$, then p can be decreased by one, meaning that the p-th frame in the buffer of **K** can no longer be lost and can be communicated to **R**.
- The fifth and sixth summand describe that the last frame ($last\text{-}dat(g), last\text{-}seq(g)$) in the buffer of **K** can be communicated to **R** if $p < length(g)$. This frame is then omitted from the buffer of **K** ($g{:=}delete\text{-}last(g)$). In the fifth summand the frame is within the receiving window ($in\text{-}window(\ell', last\text{-}seq(g), (\ell'+n)|_{2n})$), so it is included ($q'{:=}inb(last\text{-}dat(g), last\text{-}seq(g), q')$). In the sixth summand the frame is outside the receiving window, so it is omitted.
- By the seventh summand, the datum at the first cell of the receiving window ($retrieve(\ell',q')$) can be sent through channel D if this cell is occupied ($test(\ell',q')$). This cell is then emptied ($q'{:=}remove(\ell',q')$) and the first cell of the receiving window is moved forward by one ($\ell'{:=}S(\ell')|_{2n}$).
- By the eighth summand, the sequence number of the first empty cell in the receiving window can be communicated to **L**. This acknowledgement is included in the buffer of **L** ($g'{:=}inm(next\text{-}empty|_{2n}(\ell',q'),g')$) and can get lost ($p'{:=}S(p')$).
- The ninth summand describes that the first p' messages in the buffer of **L** (i.e., the last p' messages to enter the medium **L**) can get lost. The tenth summand describes that if $p'>0$, then p' can be decreased by one, meaning that the p'-th acknowledgement in the buffer of **L** can no longer be lost and can be communicated to **S**.
- By the eleventh summand, the last acknowledgement ($last\text{-}seq(g')$) in the buffer of **L** can be communicated to **S** if $p' < length(g')$. Then this acknowledgement is omitted from the buffer of **L** ($g'{:=}delete\text{-}last(g')$), the cells in the sending window before $last\text{-}seq(g')$ are emptied ($q{:=}release|_{2n}(\ell, last\text{-}seq(g'), q)$), and the sending window is moved forward ($\ell{:=}last\text{-}seq(g')$).

Proposition 6.1

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

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 modulo strong bisimilarity [GrP94]. (The details are left to the reader.) Hence, the theorem follows by CL-RSP (see Theorem 3.4). $\square$

6.2. 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.

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

Proposition 6.2

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

Proof. By a simple renaming. $\square$

6.3. 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 $\ell' \leq last\text{-}seq(g) < \ell'+n$.

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

Proposition 6.3

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

The proof of Proposition 6.3 is presented in Section 8.1. Next, in Section 8.2, we prove the correctness of $\mathbf{N}_{nonmod}$. In these proofs we will need a wide range of data equalities, which are presented in Section 7.

7. Properties of Data

7.1. 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:

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

In this section we present basic properties on modulo arithmetic, buffers, the *next-empty* operation, and lists. Unless stated otherwise (this will only happen in Lemmas 7.3.1-7.3.6 and 7.6.12) all variables that occur in a data lemma are implicitly universally quantified at the outside of the lemma.

Lemma 7.1 collects basic properties of modulo arithmetic.

Lemma 7.1 Let $n > 0$.

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

Lemma 7.2 collects basic properties of buffers.

Lemma 7.2

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

Lemma 7.3 contains properties of buffers modulo $2n$. It deals with a buffer q that has a "window" of size n, from i up to $(i + n) - 1$; in this case there is a strong correspondence between q and $q|_{2n}$.

Lemma 7.3

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

Lemma 7.4 relates $\mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i + n)|_{2n})$ to inequalities between integers without modulo arithmetic.

Lemma 7.4

1. $i \leq k < i + n \;\Rightarrow\; \mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i + n)|_{2n})$
2. $\mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i + n)|_{2n}) \;\Rightarrow\; k + n < i \vee i \leq k < i + n \vee k \geq i + 2n$

Lemma 7.5 collects basic properties of the *next-empty* operation, together with one result on *max*, which is needed to derive those properties.

Lemma 7.5

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

Lemmas 7.6 and 7.7 collect basic properties of unbounded buffers.

Lemma 7.6

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

10. $i < \mathit{length}(g) \;\Rightarrow\; \mathit{member}(\mathit{return\text{-}dat}(i,g), \mathit{return\text{-}seq}(i,g), g)$
11. $(i < \mathit{length}(g) \dot{-} 1 \wedge j < \mathit{length}(g))$
$\Rightarrow\; \mathit{return\text{-}seq}(i, \mathit{delete}(j,g)) = \mathit{if}(i < j, \mathit{return\text{-}seq}(i,g), \mathit{return\text{-}seq}(S(i),g))$
12. $\mathit{member}(d,i,g) \;\Rightarrow\; \exists j{:}\mathit{Nat}\ (j < \mathit{length}(g) \wedge \mathit{return\text{-}seq}(j,g) = i \wedge \mathit{return\text{-}dat}(j,g) = d)$

Lemma 7.7

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

Finally, Lemma 7.8 collects basic properties of lists.

Lemma 7.8

1. $(\lambda +\!\!+ \lambda') +\!\!+ \lambda'' = \lambda +\!\!+ (\lambda' +\!\!+ \lambda'')$
2. $\mathit{length}(\lambda +\!\!+ \lambda') = \mathit{length}(\lambda) + \mathit{length}(\lambda')$
3. $\mathit{append}(d, \lambda +\!\!+ \lambda') = \lambda +\!\!+ \mathit{append}(d, \lambda')$
4. $\mathit{length}(q[i..j\rangle) = j \dot{-} i$
5. $i \leq k \leq j \;\Rightarrow\; q[i..j\rangle = q[i..k\rangle +\!\!+ q[k..j\rangle$
6. $i \leq j \;\Rightarrow\; \mathit{append}(d, q[i..j\rangle) = \mathit{inb}(d,j,q)[i..S(j)\rangle$
7. $\mathit{test}(k,q) \;\Rightarrow\; \mathit{inb}(\mathit{retrieve}(k,q),k,q)[i..j\rangle = q[i..j\rangle$
8. $\neg(i \leq k < j) \;\Rightarrow\; \mathit{remove}(k,q)[i..j\rangle = q[i..j\rangle$
9. $\ell \leq i \;\Rightarrow\; \mathit{release}(k,\ell,q)[i..j\rangle = q[i..j\rangle$

7.2. Invariants

Invariants of a system are properties of data that are satisfied throughout the reachable state space of the system (see Definition 3.3). Lemma 7.9 collects 25 invariants of $\mathbf{N}_{nonmod}$ that are needed in the correctness proof. Occurrences of variables $i,j{:}\mathit{Nat}$ and $d,e{:}\Delta$ in an invariant are always implicitly universally quantified at the outside of the invariant.

Invariants 9, 11, 17, 18, 19, 22, 23 are only needed in the derivation of other invariants. We provide some intuition for the invariants that will be used in the proofs in Section 8. Invariants 7, 14, 15, 16 express that the sending window is filled from ℓ up to $m \dot{-} 1$, and that it has size n. Invariants 10, 13 express that the receiving window starts at ℓ' and stops at $\ell' + n$. Invariant 4 expresses that $\mathbf{S}$ cannot receive acknowledgements beyond $\mathit{next\text{-}empty}(\ell', q')$, and Invariant 12 that $\mathbf{R}$ cannot receive frames beyond $m \dot{-} 1$. Invariants 21, 24, 25 are based on the fact that the sending and receiving windows and the buffer of $\mathbf{K}$ coincide on occupied cells and frames with the same sequence number. Invariants 1 and 2 state that p and p' cannot exceed the length of g and g', respectively. Invariants 3 and 5 capture that sequence numbers of subsequent acknowledgement are non-decreasing. Invariant 8 depends on the fact that m is non-decreasing, and Invariants 6 and 20 depend on the fact that frames in the mediums cannot be reordered.

Lemma 7.9 The following invariants hold for $\mathbf{N}_{nonmod}(\ell, m, q, \ell', q', g, p, g', p')$.

1. $p \leq \mathit{length}(g)$
2. $p' \leq \mathit{length}(g')$

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. $test(i, q) \;\Rightarrow\; \ell \leq i$
15. $\ell \leq i < m \;\Rightarrow\; test(i, q)$
16. $m \leq \ell + n$
17. $i \leq j < length(g) \;\Rightarrow\; return\text{-}seq(i, g) + n > return\text{-}seq(j, g)$
18. $(member(d, i, g) \wedge test(j, q')) \;\Rightarrow\; i + n > j$
19. $member(d, i, g) \;\Rightarrow\; i + n \geq \ell'$
20. $member(d, i, g) \;\Rightarrow\; i + n \geq next\text{-}empty(\ell', q')$
21. $(member(d, i, g) \wedge test(i, q)) \;\Rightarrow\; retrieve(i, q) = d$
22. $(test(i, q) \wedge test(i, q')) \;\Rightarrow\; retrieve(i, q) = retrieve(i, q')$
23. $(member(d, i, g) \wedge member(e, i, g)) \;\Rightarrow\; d = e$
24. $(member(d, i, g) \wedge test(i, q')) \;\Rightarrow\; retrieve(i, q') = d$
25. $(\ell \leq i \leq m \wedge j \leq next\text{-}empty(i, q')) \;\Rightarrow\; q[i..j\rangle = q'[i..j\rangle$

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

In Section 8.1, we prove Proposition 6.3, which states that $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ are strongly bisimilar. Next, in Section 8.2 we prove that $\mathbf{N}_{nonmod}$ behaves like a FIFO queue of size $2n$. Theorem 5.1 is proved in Section 8.3.

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

In this section we present a proof of Proposition 6.3. It suffices to prove that for all $\ell, m, \ell'{:}Nat$, $q, q'{:}Buf$, $g{:}MedK$ and $g'{:}MedL$,

$$\mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g|_{2n}, p, g'|_{2n}, p') \;\underline{\leftrightarrow}\; \mathbf{N}_{nonmod}(\ell, m, q, \ell', q', g, p, g', p')$$

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.[3]

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

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

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

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.

[3] 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.

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

$$\begin{array}{lll} & m < \ell + n & \\ \Leftrightarrow & \ell \leq m < \ell + n & \text{(Inv. 7.9.4, 7.9.12)} \\ \Rightarrow & \mathit{in\text{-}window}(\ell|_{2n}, m|_{2n}, (\ell + n)|_{2n}) & \text{(Lem. 7.3.1)} \end{array}$$

Reversely,

$$\begin{array}{lll} & \mathit{in\text{-}window}(\ell|_{2n}, m|_{2n}, (\ell + n)|_{2n}) & \\ \Rightarrow & m + n < \ell \vee \ell \leq m < \ell + n \vee m \geq \ell + 2n & \text{(Lem. 7.3.2)} \\ \Leftrightarrow & m < \ell + n & \text{(Inv. 7.9.4, 7.9.12, 7.9.16)} \end{array}$$

Moreover, by Lemma 7.1.1, $(\ell + n)|_{2n} = (\ell|_{2n} + n)|_{2n}$.

• $S(m)|_{2n} = S(m|_{2n})|_{2n}$.
This follows from Lemma 7.1.1.

• $\mathit{inb}(d, m, q)|_{2n} = \mathit{inb}(d, m|_{2n}, q|_{2n})$.
This follows from the definition of buffers modulo $2n$.

B Below we equate the entire summand B of the two specifications. The argument $p := S(p)$ is omitted, because it is irrelevant for this derivation.

$$\begin{array}{lll}
 & \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k, q), k, g)|_{2n}) & \\
 & \lhd\ \mathit{test}(k, q)\ \rhd\ \delta & \\[1ex]
\approx & \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k, q), k|_{2n}, g|_{2n})) & \\
 & \lhd\ \mathit{test}(k, q) \wedge \ell \leq k < \ell + n\ \rhd\ \delta & \text{(Inv. 7.9.7, 7.9.14, 7.9.16)} \\[1ex]
\approx & \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k|_{2n}, q|_{2n}), k|_{2n}, g|_{2n})) & \\
 & \lhd\ \mathit{test}(k|_{2n}, q|_{2n}) \wedge \ell \leq k < \ell + n\ \rhd\ \delta & \text{(Lem. 7.3.1, 7.3.2)} \\[1ex]
\approx & \sum_{k':Nat} \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\
 & \lhd\ \mathit{test}(k', q|_{2n}) \wedge \ell \leq k < \ell + n \wedge k' = k|_{2n}\ \rhd\ \delta & \text{(Thm. 3.2)} \\[1ex]
\approx & \sum_{k':Nat} \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\
 & \lhd\ \mathit{test}(k', q|_{2n}) \wedge k = (\ell\ \mathit{div}\ 2n)2n + k' \wedge & \\
 & \ell|_{2n} \leq k' < \ell|_{2n} + n \wedge k' = k|_{2n}\ \rhd\ \delta & \\[1ex]
+ & \sum_{k':Nat} \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\
 & \lhd\ \mathit{test}(k', q|_{2n}) \wedge k = S(\ell\ \mathit{div}\ 2n)2n + k' \wedge & \\
 & k' + n < \ell|_{2n} \wedge k' = k|_{2n}\ \rhd\ \delta & \text{(Lem. 7.1.3, 7.1.4)} \\[1ex]
\approx & \sum_{k':Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\
 & \lhd\ \mathit{test}(k', q|_{2n}) \wedge \ell|_{2n} \leq k' < \ell|_{2n} + n \wedge k' = k'\ \rhd\ \delta & \\[1ex]
+ & \sum_{k':Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\
 & \lhd\ \mathit{test}(k', q|_{2n}) \wedge k' + n < \ell|_{2n} \wedge k' = k'\ \rhd\ \delta & \text{(Thm. 3.2)} \\[1ex]
\approx & \sum_{k':Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}\mathit{inm}(\mathit{retrieve}(k', q|_{2n}), k', g|_{2n})) & \\
 & \lhd\ \mathit{test}(k', q|_{2n})\ \rhd\ \delta & \text{(Lem. 7.3.6)}
\end{array}$$

C $k < p \Rightarrow \mathit{delete}(k, g)|_{2n} = \mathit{delete}(k, g|_{2n})$.
By Invariant 7.9.1, $k < p \leq \mathit{length}(g)$. So this follows from Lemma 7.6.4.

E • $\mathit{length}(g) = \mathit{length}(g|_{2n})$.
This follows from Lemma 7.6.1.

• $p < \mathit{length}(g) \Rightarrow (\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n = \mathit{in\text{-}window}(\ell'|_{2n}, \mathit{last\text{-}seq}(g)|_{2n}, (\ell'|_{2n} + n)|_{2n}))$.
Since $0 < \mathit{length}(g)$, Lemmas 7.6.5, 7.6.6, and 7.6.10 yield $\mathit{member}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), g)$. So by Invariant 7.9.20, $\mathit{next\text{-}empty}(\ell', q') \leq \mathit{last\text{-}seq}(g) + n$. Hence, by Lemma 7.5.3, $\ell' \leq \mathit{last\text{-}seq}(g) + n$. Furthermore, by Invariant 7.9.8, $\mathit{last\text{-}seq}(g) < m$, by Invariant 7.9.16, $m \leq \ell + n$, and by Invariants 7.9.4 and 7.9.13, $\ell \leq \ell' + n$. Hence, $\mathit{last\text{-}seq}(g) < \ell' + 2n$. So by Lemmas 7.3.1 and 7.3.2, $\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n = \mathit{in\text{-}window}(\ell'|_{2n}, \mathit{last\text{-}seq}(g)|_{2n}, (\ell' + n)|_{2n})$. And by Lemma 7.1.1, $(\ell' + n)|_{2n} = (\ell'|_{2n} + n)|_{2n}$.

• $p < \mathit{length}(g) \Rightarrow \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')|_{2n} = \mathit{inb}(\mathit{last\text{-}dat}(g|_{2n}), \mathit{last\text{-}seq}(g|_{2n}), q'|_{2n})$.
This follows from the definitions of buffers modulo $2n$, and Lemmas 7.6.5, 7.6.6, 7.6.2 and 7.6.3.

- $p < \mathit{length}(g) \Rightarrow \mathit{delete\text{-}last}(g)|_{2n} = \mathit{delete\text{-}last}(g|_{2n})$.
  This follows from Lemmas 7.6.7 and 7.6.4.

F
- $\neg(\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n) \Leftrightarrow \neg \mathit{in\text{-}window}(\ell'|_{2n}, \mathit{last\text{-}seq}(g)|_{2n}, (\ell'|_{2n} + n)|_{2n})$.
  This follows immediately from the second item of the previous case.
- $p < \mathit{length}(g) \Rightarrow \mathit{delete\text{-}last}(g)|_{2n} = \mathit{delete\text{-}last}(g|_{2n})$.
  This follows immediately from the fourth item of the previous case.

G
- $\mathit{test}(\ell', q') = \mathit{test}(\ell'|_{2n}, q'|_{2n})$.
  This follows from Lemma 7.3.1 together with Invariant 7.9.10.
- $\mathit{test}(\ell', q') \Rightarrow (\mathit{retrieve}(\ell', q') = \mathit{retrieve}(\ell'|_{2n}, q'|_{2n}))$.
  This follows from Lemma 7.3.2 together with Invariant 7.9.10.
- $S(\ell')|_{2n} = S(\ell'|_{2n})|_{2n}$.
  This follows from Lemma 7.1.1.
- $\mathit{remove}(\ell', q')|_{2n} = \mathit{remove}(\ell'|_{2n}, q'|_{2n})$.
  This follows from Lemma 7.3.3 together with Invariant 7.9.10.

H $\mathit{inm}(\mathit{next\text{-}empty}(\ell', q')|_{2n}, g')|_{2n} = \mathit{inm}(\mathit{next\text{-}empty}|_{2n}(\ell'|_{2n}, q'|_{2n}), g'|_{2n})$.
By Lemma 7.3.5 and Invariant 7.9.10, $\mathit{next\text{-}empty}(\ell', q')|_{2n} = \mathit{next\text{-}empty}|_{2n}(\ell'|_{2n}, q'|_{2n})$. So the desired equality follows from the definition of mediums modulo $2n$.

I $k < p' \Rightarrow \mathit{delete}(k, g')|_{2n} = \mathit{delete}(k, g'|_{2n})$.
By Invariant 7.9.2, $k < p' \leq \mathit{length}(g')$. So the desired equality follows from Lemma 7.7.3.

K
- $\mathit{length}(g') = \mathit{length}(g'|_{2n})$.
  This follows from Lemma 7.7.1.
- $p' < \mathit{length}(g') \Rightarrow \mathit{last\text{-}seq}(g')|_{2n} = \mathit{last\text{-}seq}(g'|_{2n})|_{2n}$.
  This follows from Lemmas 7.7.4, 7.7.2 and 7.1.1.
- $\mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)|_{2n} = \mathit{release}|_{2n}(\ell|_{2n}, \mathit{last\text{-}seq}(g')|_{2n}, q|_{2n})$.
  By Lemmas 7.7.4 and 7.7.8, $p' < \mathit{length}(g')$ implies $\mathit{member}(\mathit{last\text{-}seq}(g'), g')$. So by Invariant 7.9.6, $\ell \leq \mathit{last\text{-}seq}(g')$. By Invariants 7.9.3 and 7.9.12, $\mathit{last\text{-}seq}(g') \leq \mathit{next\text{-}empty}(\ell', q') \leq m$. And by Invariant 7.9.16, $m \leq \ell + n$. So $\ell \leq \mathit{last\text{-}seq}(g') \leq \ell + n$. Furthermore, by Invariants 7.9.7, 7.9.14 and 7.9.16, $\mathit{test}(i, q) \Rightarrow \ell \leq i < \ell + n$. Hence, the desired equation follows from Lemma 7.3.4.
- $p' < \mathit{length}(g') \Rightarrow \mathit{delete\text{-}last}(g')|_{2n} = \mathit{delete\text{-}last}(g'|_{2n})$.
  This follows from Lemmas 7.7.3 and 7.7.5.

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 (see Theorem 3.4), they are strongly (and thus branching) bisimilar. □

8.2. 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 5.2), using cones and foci (see Theorem 3.6).

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

$$\phi(\xi) = q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle{+\!\!+}q[\mathit{next\text{-}empty}(\ell', q')..m\rangle$$

Intuitively, ϕ collects the data elements in the sending and receiving windows, starting at the first cell in the receiving window (i.e., ℓ') until the first empty cell in this window, and then continuing in the sending window until the first empty cell 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

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

Lemma 8.1 For each $\xi{:}\Xi$ where the invariants in Lemma 7.9 hold, there is a $\hat{\xi}{:}\Xi$ with $\mathrm{FC}(\hat{\xi})$ such that $\mathbf{N}_{nonmod}(\xi) \xrightarrow{c_1} \cdots \xrightarrow{c_k} \mathbf{N}_{nonmod}(\hat{\xi})$, where $c_1, \ldots, c_k \in \mathcal{I}$.

Proof. By Invariants 7.9.12 and 7.9.13, $\textit{next-empty}(\ell', q') \le \min\{m, \ell' + n\}$. We prove by induction on $\min\{m, \ell' + n\} \dot{-} \textit{next-empty}(\ell', q')$ that for each state ξ where the invariants in Lemma 7.9 hold, a focus point can be reached by a sequence of communication actions.
Base Case: $\textit{next-empty}(\ell', q') = \min\{m, \ell' + n\}$.
Let $y = \textit{length}(g')$ and $x = \textit{next-empty}(\ell', q')$ at state ξ. By summand H, we reach a state ξ' with $g' := \textit{inm}(x, g')$. Hence, at state ξ' there exists a $0 \le k < y$ such that $\textit{return-seq}(k, g') = x$ and $\textit{return-seq}(i, g') \ne x$ for any $k < i < y$. In view of Invariant 7.9.5, $k < i < y \Rightarrow x > \textit{return-seq}(i, g')$. 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 $\textit{last-seq}(g') = x$. 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 $\ell = \textit{last-seq}(g') = x = \min\{m, \ell' + n\}$. Then $\ell = m$ or $\ell = \ell' + n$, so $FC(\hat{\xi})$.
Induction Case: $\textit{next-empty}(\ell', q') < \min\{m, \ell' + n\}$.
Let $y = \textit{length}(g)$ and $x = \textit{next-empty}(\ell', q')$ at state ξ. By Invariants 7.9.4 and 7.9.12, $\ell \le x < m$. So by Invariant 7.9.15, $\textit{test}(x, q)$. Furthermore, in view of Lemma 7.5.3, $\ell' \le x < \ell' + n$. By summand B, we perform a communication action to a state ξ' with $g{:=}\textit{inm}(d, x, g)$ (where d denotes $\textit{retrieve}(x, q)$). Hence, at state ξ' there exists a $0 \le k < y$ such that $\textit{return-seq}(k, g) = x$ and $\textit{return-seq}(i, g) \ne x$ 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 $\textit{last-dat}(g) = d$ and $\textit{last-seq}(g) = x$. During these executions of B, D, E and F, the values of m, ℓ' remain the same; and since during the executions of E and F $\textit{last-seq}(g) \ne x$, in view of Lemma 7.5.5, the value of $\textit{next-empty}(\ell', q')$ remains the same. By again performing summand E, we reach a state ξ'''' where $q' := \textit{inb}(d, x, q')$. Recall that $x = \textit{next-empty}(\ell', q')$.

$$\begin{array}{cll} & \textit{next-empty}(\ell', \textit{in}(d, \textit{next-empty}(\ell', q'), q')) & \\ = & \textit{next-empty}(S(\textit{next-empty}(\ell', q')), q') & \text{(Lem. 7.5.6)} \\ > & \textit{next-empty}(\ell', q') & \text{(Lem. 7.5.3)} \end{array}$$

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. $\square$

Proposition 8.2 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. According to the cones and foci method (see Theorem 3.6), we obtain the following matching criteria (see Definition 3.5). Trivial matching criteria are left out.
Class I:

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

$$p' < \textit{length}(g') \ \Rightarrow\ \phi(m, q, \ell', q') = \phi(m, \textit{release}(\ell, \textit{last-seq}(g'), q), \ell', q')$$

Class II:

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

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

Class III:

$$((\ell = m \vee \ell = \ell' + n) \wedge \textit{length}(\phi(m, q, \ell', q')) < 2n) \ \Rightarrow\ m < \ell + n$$

$$((\ell = m \vee \ell = \ell' + n) \wedge \textit{length}(\phi(m, q, \ell', q')) > 0) \ \Rightarrow\ \textit{test}(\ell', q')$$

Class IV:

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

Class V:

$$m < \ell + n \;\Rightarrow\; \phi(S(m), \mathit{inb}(d, m, q), \ell', q') = \mathit{append}(d, \phi(m, q, \ell', q'))$$
$$\mathit{test}(\ell', q') \;\Rightarrow\; \phi(m, q, S(\ell'), \mathit{remove}(\ell', q')) = \mathit{tail}(\phi(m, q, \ell', q'))$$

I.1 $(p < \mathit{length}(g) \wedge \ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n)$
$\Rightarrow \phi(m, q, \ell', q') = \phi(m, q, \ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'))$.
Let $p < \mathit{length}(g)$. By Lemmas 7.6.5, 7.6.6 and 7.6.10, $\mathit{member}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), g)$.
Case 1: $\mathit{last\text{-}seq}(g) \neq \mathit{next\text{-}empty}(\ell', q')$.
By Lemma 7.5.5, $\mathit{next\text{-}empty}(\ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) = \mathit{next\text{-}empty}(\ell', q')$. 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 7.5.2, $\mathit{test}(\mathit{last\text{-}seq}(g), q')$, so by Invariant 7.9.24 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 7.8.7, $\mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle = q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle$.
Case 1.2: $\neg(\ell' \leq \mathit{last\text{-}seq}(g) < \mathit{next\text{-}empty}(\ell', q'))$.
Using Lemma 7.8.8, 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 7.8.8, 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 7.9.4, $\ell \leq \mathit{last\text{-}seq}(g)$. By Invariant 7.9.8 and $\mathit{member}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), g)$, $\mathit{last\text{-}seq}(g) < m$. Thus, by Invariant 7.9.15, $\mathit{test}(\mathit{last\text{-}seq}(g), q)$. So we have $\mathit{retrieve}(\mathit{last\text{-}seq}(g), q) = \mathit{last\text{-}dat}(g)$ by Invariant 7.9.21 together with $\mathit{member}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), g)$. Since $\ell \leq S(\mathit{last\text{-}seq}(g)) \leq m$, by Invariant 7.9.25,

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

Hence,

$$\begin{array}{lll} & \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\mathit{last\text{-}seq}(g)..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle & \\ = & \mathit{inl}(\mathit{last\text{-}dat}(g), \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q') & \\ & [S(\mathit{last\text{-}seq}(g))..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle) & \\ = & \mathit{inl}(\mathit{last\text{-}dat}(g), \mathit{remove}(\mathit{last\text{-}seq}(g), \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) & \\ & [S(\mathit{last\text{-}seq}(g))..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle) & \text{(Lem. 7.8.8)} \\ = & \mathit{inl}(\mathit{last\text{-}dat}(g), \mathit{remove}(\mathit{last\text{-}seq}(g), q') & \\ & [S(\mathit{last\text{-}seq}(g))..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle) & \\ = & \mathit{inl}(\mathit{last\text{-}dat}(g), q'[S(\mathit{last\text{-}seq}(g))..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle) & \text{(Lem. 7.8.8)} \\ = & \mathit{inl}(\mathit{last\text{-}dat}(g), q[S(\mathit{last\text{-}seq}(g))..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle) & \text{(see above)} \\ = & q[\mathit{last\text{-}seq}(g)..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle & \end{array}$$

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

$$\begin{array}{cll}
& \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\ell'..\mathit{next\text{-}empty}(\ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'))\rangle & \\
& {+\!\!+} q[\mathit{next\text{-}empty}(\ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'))..m\rangle & \\
= & \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\ell'..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle & \\
& {+\!\!+} q[\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')..m\rangle & \text{(Lem. 7.5.6)} \\
= & (\mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\ell'..\mathit{last\text{-}seq}(g)\rangle & \\
& {+\!\!+} \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')[\mathit{last\text{-}seq}(g)..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle) & \\
& {+\!\!+} q[\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')..m\rangle & \text{(Lem. 7.5.3, 7.8.5)} \\
= & (q'[\ell'..\mathit{last\text{-}seq}(g)\rangle & \\
& {+\!\!+} q[\mathit{last\text{-}seq}(g)..\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')\rangle) & \\
& {+\!\!+} q[\mathit{next\text{-}empty}(S(\mathit{last\text{-}seq}(g)), q')..m\rangle & \text{(1), (2)} \\
= & q'[\ell'..\mathit{last\text{-}seq}(g)\rangle {+\!\!+} q[\mathit{last\text{-}seq}(g)..m\rangle & \text{(Lem. 7.8.1, 7.5.2, 7.8.5)}
\end{array}$$

I.2 $p' < \mathit{length}(g') \Rightarrow \phi(m, q, \ell', q') = \phi(m, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q), \ell', q')$.
Let $p' < \mathit{length}(g')$. By Lemmas 7.7.4 and 7.7.8, $\mathit{member}(\mathit{last\text{-}seq}(g'), g')$.
By Invariant 7.9.3, $\mathit{last\text{-}seq}(g') \leq \mathit{next\text{-}empty}(\ell', q')$. So by Lemma 7.8.9,

$$\mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)[\mathit{next\text{-}empty}(\ell', q')..m\rangle = q[\mathit{next\text{-}empty}(\ell', q')..m\rangle$$

II.1 $m < \ell + n \Rightarrow \mathit{length}(\phi(m, q, \ell', q')) < 2n$.
Let $m < \ell + n$.

$$\begin{array}{cll}
& \mathit{length}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle {+\!\!+} q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) & \\
= & \mathit{length}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle) + \mathit{length}(q[\mathit{next\text{-}empty}(\ell', q')..m\rangle)) & \text{(Lem. 7.8.2)} \\
= & (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q')) & \text{(Lem. 7.8.4)} \\
\leq & n + (m \dot{-} \ell) & \text{(Inv. 7.9.13, 7.9.4)} \\
< & 2n &
\end{array}$$

II.2 $\mathit{test}(\ell', q') \Rightarrow \mathit{length}(\phi(m, q, \ell', q')) > 0$.
$\mathit{test}(\ell', q')$ together with Lemma 7.5.3 yields $\mathit{next\text{-}empty}(\ell', q') = \mathit{next\text{-}empty}(S(\ell'), q') \geq S(\ell')$. Hence, by Lemmas 7.8.2 and 7.8.4,

$$\begin{array}{cl}
& \mathit{length}(\phi(m, q, \ell', q')) \\
= & (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q')) \\
> & 0
\end{array}$$

III.1 $((\ell = m \vee \ell = \ell' + n) \wedge \mathit{length}(\phi(m, q, \ell', q')) < 2n) \Rightarrow m < \ell + n$.
Case 1: $\ell = m$.
Then $m < \ell + n$ holds trivially.
Case 2: $\ell = \ell' + n$.

$$\begin{array}{cll}
& \mathit{length}(\phi(m, q, \ell', q')) & \\
= & (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q')) & \text{(Lem. 7.8.2, 7.8.4)} \\
\leq & ((\ell' + n) \dot{-} \ell') + (m \dot{-} \ell) & \text{(Inv. 7.9.13, 7.9.4)} \\
= & n + (m \dot{-} \ell) &
\end{array}$$

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

III.2 $((\ell = m \vee \ell = \ell' + n) \wedge \mathit{length}(\phi(m, q, \ell', q')) > 0) \Rightarrow \mathit{test}(\ell', q')$.
Case 1: $\ell = m$.
Then $m \dot{-} \mathit{next\text{-}empty}(\ell', q') \leq m \dot{-} \ell$ (Inv. 7.9.4) $= 0$, so

$$\begin{array}{cll}
& \mathit{length}(\phi(m, q, \ell', q')) & \\
= & (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q')) & \text{(Lem. 7.8.2, 7.8.4)} \\
= & \mathit{next\text{-}empty}(\ell', q') \dot{-} \ell' &
\end{array}$$

Hence, $\mathit{length}(\phi(m, q, \ell', q')) > 0$ yields $\mathit{next\text{-}empty}(\ell', q') > \ell'$, which implies $\mathit{test}(\ell', q')$.
Case 2: $\ell = \ell' + n$.
Then by Invariant 7.9.4, $\mathit{next\text{-}empty}(\ell', q') \geq \ell' + n$, which implies $\mathit{test}(\ell', q')$.

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

$\mathit{test}(\ell', q')$ implies $\mathit{next\text{-}empty}(\ell', q') = \mathit{next\text{-}empty}(S(\ell'), q') \geq S(\ell')$ (Lem. 7.5.3). Hence,

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

So

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

V.1 $m < \ell + n \Rightarrow \phi(S(m), \mathit{inb}(d, m, q), \ell', q') = \mathit{append}(d, \phi(m, q, \ell', q'))$.

$$\begin{array}{lll} & q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ & \\ & \mathit{inb}(d, m, q)[\mathit{next\text{-}empty}(\ell', q')..S(m)\rangle & \\ = & q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ & \\ & \mathit{append}(d, q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) & \text{(Lem. 7.8.6, Inv. 7.9.12)} \\ = & \mathit{append}(d, q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ & \\ & q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) & \text{(Lem. 7.8.3)} \end{array}$$

V.2 $\mathit{test}(\ell', q') \Rightarrow \phi(m, q, S(\ell'), \mathit{remove}(\ell', q')) = \mathit{tail}(\phi(m, q, \ell', q'))$.
$\mathit{test}(\ell', q')$ and Lemma 7.5.3 imply $\mathit{next\text{-}empty}(\ell', q') = \mathit{next\text{-}empty}(S(\ell'), q') \geq S(\ell')$. Hence,

$$\begin{array}{lll} & \mathit{remove}(\ell', q')[S(\ell')..\mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q'))\rangle & \\ & +\!\!+ q[\mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q'))..m\rangle & \\ = & \mathit{remove}(\ell', q')[S(\ell')..\mathit{next\text{-}empty}(S(\ell'), q')\rangle & \\ & +\!\!+ q[\mathit{next\text{-}empty}(S(\ell'), q')..m\rangle & \text{(Lem. 7.5.7)} \\ = & \mathit{remove}(\ell', q')[S(\ell')..\mathit{next\text{-}empty}(\ell', q')\rangle & \\ & +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle & \\ = & q'[S(\ell')..\mathit{next\text{-}empty}(\ell', q')\rangle & \\ & +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle & \text{(Lem. 7.8.8)} \\ = & \mathit{tail}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle & \\ & +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) & \end{array}$$

□

8.3. Correctness of the Sliding Window Protocol

Finally, we can prove Theorem 5.1.

Proof.

$$\begin{array}{lll} & \tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K}([]^K,0) \parallel \mathbf{L}([]^L,0))) & \\ \leftrightarrow & \tau_{\mathcal{I}}(\mathbf{M}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0)) & \text{(Prop. 6.1)} \\ \leftrightarrow & \tau_{\{c,j\}}(\mathbf{N}_{mod}(0,0,[],0,[],[]^K,0,[]^L,0)) & \text{(Prop. 6.2)} \\ \leftrightarrow & \tau_{\{c,j\}}(\mathbf{N}_{nonmod}(0,0,[],0,[],[]^K,0,[]^L,0)) & \text{(Prop. 6.3)} \\ \leftrightarrow_b & \mathbf{Z}(\langle\rangle) & \text{(Prop. 8.2)} \end{array}$$

□

9. Formalization and Verification in PVS

In this section we report on the formalization of the correctness proof for SWP using the theorem prover PVS [ORR+96]. Using a theorem prover without a good idea of the proof structure is generally difficult and inefficient, so it is important that we had this manual μCRL proof to start with.

There are several reasons why a computer checked proof besides a manual proof is useful. First of all, the manual μCRL proof of the SWP that we presented in the previous sections is so complicated that we decided to use a theorem prover to search for omissions and errors in this proof. The theorem proving exercise led to the detection of several omissions in the manual proof; we will report on some of these omissions in this section. After the formalization, we can be sure (modulo bugs in the prover software) that the proof is complete and correct, including the smallest detail.

Interestingly, the use of PVS also led to a more precise formalization of the data specification (see Section 9.1). Our conclusion is that the data specification in PVS is more precise, and can reveal potential errors that go undetected in a μCRL specification.

Besides providing more confidence in the proof, the use of a theorem prover can also be more efficient. In particular, tedious small proof steps concerning standard data types can be automated. But more importantly, the proofs can be *rerun* after small changes in the specification, in order to detect which parts are affected by the change. In this way only the problematic parts need attention. We followed this procedure in order to generalize our results in [FGP$^+$04] to the current setting with unbounded buffers. Recently, we also reused considerable parts of the proof for the verification of a bi-directional SWP, in which the acknowledgements are *piggy backed* on the data stream of the opposite direction. A full review of the hand-written proof would have been much more work than an interactive modification with PVS. In particular, PVS interaction helped us in updating the list of invariants.

Finally, our formalization efforts provided and tested a library of protocol-independent verification theory, in particular on linear processes and the cones and foci method [FPP05]. This library can be reused for completely different protocols as well.

We selected PVS for several reasons. First, the specification language of PVS is based on simply typed higher-order logics. PVS provides a rich set of types and the ability to define subtypes and dependent types. Second, PVS provides a powerful, extensible system for verifying obligations. It has a tool set consisting of a type checker, an interactive theorem prover, and a model checker. Third, PVS has high-level proof strategies and decision procedures that take care of many low-level details associated with computer-aided theorem proving. In addition, PVS has useful proof management facilities, such as a graphical display of the proof tree, and proof stepping and editing.

The PVS specification language is based on simply typed higher-order logics. Its type system contains basic types such as *boolean, nat, integer, real,* etc. and type constructors such as *set, tuple, record,* and *function.* Tuple types have the form `[T1,...,Tn]`, where `Ti` are type expressions. A record is a finite list of fields of the form `R:TYPE=[# E1:T1, ...,En:Tn #]`, where `Ei` are *record accessor* functions. A function constructor has the form `F:TYPE=[T1,...,Tn->R]`, where `F` is a function with domain `D=T1`$\times$`...`$\times$`Tn` and range `R`.

A PVS specification can be structured through a hierarchy of *theories.* Each theory consists of a *signature* for the type names, constants introduced in the theory, axioms, definitions, and theorems associated with the signature. A PVS theory can be parametric in certain specified types and values, which are placed between `[ ]` after the theory name.

In Section 9.1 we show examples of the original specification of some data functions, and then we introduce the modified forms of them. Moreover, we show how measure functions are used to detect termination of recursive definitions. In Section 9.2 and 9.3 we represent the LPEs and invariants of the SWP in PVS. Section 9.4 presents the equality of μCRL specification of the SWP with and without modulo arithmetic. Section 9.5 explains how the cones and foci method is used to formalize the main theorem, that is, the μCRL specification of the SWP is branching bisimilar to a FIFO queue of size $2n$. Finally, Section 9.6 is dedicated to some remarks on the verification in PVS.

We try to motivate our formalization in PVS as much as possible, and also explain some erroneous attempts, to bring across some valuable experience with PVS in such a formalization exercise.

9.1. Data Specifications in PVS

In μCRL, the (loose) semantics of a data specification is the class of all its models. Incomplete data specifications may have multiple models. Even worse, it is possible to have inconsistent data specifications for which no models exist. Here the necessity of specification with PVS emerges, because of this probable incompleteness and inconsistency which exists when working with μCRL.

In PVS, all the definitions are first type checked, which generates some *proof obligations.* Proving all these obligations ascertains that our data specification is complete and consistent.

To achieve this, having total definitions is required. So in the first place, underspecified functions need to be extended to total ones. Below there are some examples of partial definitions in the original data specification of the SWP, which we changed into total ones. Second, to guarantee totality of recursive definitions, PVS requires the user to define a so-called *measure function*. Doing this usually requires time and effort, but the advantage is that recursive definitions are guaranteed to be well-founded. PVS enabled us to find non-terminating definitions in the original data specification of the SWP, which were not detected within the framework of μCRL. Below we show some of the most interesting examples.

Example 9.1 We wanted to have $\textit{next-empty}|_n(i,q)$ as a function which produces the first empty cell in q modulo n, so it was reasonable to define it as:

$$\textit{next-empty}|_n(i,q) \quad = \quad \textit{if}\,(\textit{test}(i,q), \textit{next-empty}|_n(S(i)|_n, q), i)$$

Although the definition looks total and well-founded, this was one of the undetected potential errors that PVS detected during the type checking process. Below we bring an example to show what happens:

$$q = [(d_0, 0), (d_1, 1), (d_2, 2), (d_3, 3), (d_5, 5)],\ n = 4,\ i = 5$$

Then

$$\begin{aligned}\textit{next-empty}|_4(5,q) &= \textit{next-empty}|_4(6|_4, q)\\ &= \textit{next-empty}|_4(2,q)\\ &= \textit{next-empty}|_4(3,q)\\ &= \textit{next-empty}|_4(0,q)\\ &= \textit{next-empty}|_4(1,q)\\ &= \textit{next-empty}|_4(2,q)\\ &= \ldots\end{aligned}$$

which will never terminate. At the end we replaced it with the following definition, which is terminating and operates the way we expect.

$$\textit{next-empty}|_n(i,q) \quad = \quad \textit{if}\,(\textit{next-empty}(i|_n, q|_n) < n, \textit{next-empty}(i|_n, q|_n), \textit{next-empty}(0, q|_n))$$

Example 9.2 $\textit{release}(i,j,q)$ is obtained by emptying cells $i, \ldots, j \dot{-} 1$, as defined in Section 4.5. The original definition was the one below which we modified, because PVS detected non-termination on it.

$$\textit{release}(i,j,q) \quad = \quad \textit{if}\,(i = j, q, \textit{release}(S(i), j, \textit{remove}(i,q)))$$

It is non-terminating when $i > j$. Therefore we replaced $i = j$ with $i \geq j$ in the case distinction above.

Example 9.3 $\textit{release}|_n(i,j,q)$ behaves similar to $\textit{release}(i,j,q)$ modulo n. The previous flaw in the definition of $\textit{release}(i,j,q)$ does not apply here, since $i|_n$ will not grow beyond $n \dot{-} 1$. First, we defined it as follows:

$$\textit{release}|_n(i,j,q) \quad = \quad \textit{if}\,(i = j, q, \textit{release}(S(i)|_n, j, \textit{remove}(i,q)))$$

This definition met our expectations, except there was a undetected problem within it, that can cause a non-termination, which occurs if $i = S(j)$ and $j > n$. Thus we modified the above definition to:

$$\textit{release}|_n(i,j,q) \quad = \quad \textit{if}\,(i|_n = j|_n, q, \textit{release}|_n(S(i), j, \textit{remove}(i,q)))$$

This new definition works properly and is terminating.

We represented the μCRL abstract data types directly by PVS types. This enabled us to reuse the PVS library for definitions and theorems of "standard" data types. As an illustration, Figure 3 shows part of a PVS theory defining $\textit{release}|_n$. There D is an unspecified but non-empty type which represents the set of all data that can be communicated between the sender and the receiver. *Buf* is a list of pairs of type $D \times \textit{Nat}$ defined as $\textit{list}[[D, \textit{nat}]]$. Here we used *list* to identify the type of the set of lists, which is defined in the prelude in PVS. Therefore we simply use it without any need to define it explicitly. This figure also represents $\textit{release}|_n(i,j,q)$ in PVS. Since it is defined recursively, in order to establish its termination (or totality), it is required by PVS to have a measure function. This new function itself needs to be terminating (or decreasing) in the recursion calls. The simplest function which provided us with this property is the

```
...
D:nonempty_type
Buf:type=list[[D,nat]]
x,i,j,k,l,n: VAR nat
...
dm(i,j,n): nat =
  IF mod(i,2*n)<=mod(j,2*n)
  THEN mod(j,2*n)-mod(i,2*n)
  ELSE 2*n+mod(j,2*n)-mod(i,2*n)
  ENDIF
dist(i,j,n): RECURSIVE nat=
  IF mod(i,2*n)=mod(j,2*n) THEN 0
  ELSE S(dist(mod(S(i), 2*n), j, n))
  ENDIF
  measure(dm(i,j,n))
...
release(n)(i,j,q): RECURSIVE Buf=
  IF mod(i,2*n)=mod(j,2*n) THEN q
  ELSE release(n)(mod(S(i),2*n),j,remove(mod(i,2*n),q))
  ENDIF
  measure dist(i,j,n)
...
```

Fig. 3. An example of a data specification in PVS

so-called *dist* function, which counts the steps from $i|_{2n}$ to $j|_{2n}$, and is a recursive function itself. So a new measure function was necessary to prove termination; for this purpose we defined a function called *dm*, which is decreasing and non-recursive.

PVS does not allow to skip the proofs of basic properties of the operations on *Nat* and *Bool*, which were mentioned in Section 7.1. Below we list all auxiliary lemmas for *Nat* and *Bool* that PVS requires to be defined and proved literally, while in the μCRL proof we considered them as trivial facts. For the proofs, the reader is referred to the dump file at `http://homepages.cwi.nl/~vdpol/swp.html`.

Lemma 9.4 The followings hold for $n > 0$:

1. $i > 0 \;\Rightarrow\; i{\cdot}n \geq n$
2. $i > 0 \;\Rightarrow\; i \dot{-} n < i$
3. $i|_n \leq i$
4. $S(i)|_n \leq S(i|_n)$
5. $i|_n \neq n \dot{-} 1 \;\Rightarrow\; i|_n < S(i)|_n$
6. $i \leq j \;\Rightarrow (i\; div\; n) \leq (j\; div\; n)$
7. $i \leq j \leq i + n \;\Rightarrow (j\; div\; n) = (i\; div\; n) \vee (j\; div\; n) = S(i\; div\; n)$
8. $test(i, q|_n) \;\Rightarrow\; i < n$
9. $i + n \leq j < i + 2n \;\Rightarrow\; \neg in\text{-}window(i|_{2n}, j|_{2n}, (i + n)|_{2n})$
10. $(q|_n)|_n = q|_n$
11. $\lambda{+\!\!+}\langle\rangle = \lambda$
12. $test(i, q) \;\Rightarrow\; test(i|_n, q|_n)$

In the μCRL proof, several data lemmas contain many back and forth steps in their proof strategies, which are complicated to mimick in PVS. Therefore some of the proofs were restructured or modified in PVS, in such a way that they can be obtained without any detour. For example, Lemma 7.3.1 is proved by using Lemmas 9.4.6 and 9.4.7 above (see the dump file).

```
LPE[Act,State,Local:TYPE,n:nat]: THEORY BEGIN
 SUMMAND:TYPE= [State,Local-> [#act:Act,guard:bool,next:State#] ]
 LPE:TYPE= [#init:State,sums:[below(n)->SUMMAND]#]
END LPE
```

Fig. 4. Definition of LPE in PVS

9.2. Representing LPEs

We now show how the μCRL specification of the SWP (an LPE) can be represented in PVS (cf. [FPP05]). The main distinction will be that we have assumed so far that LPEs are *clustered*. This means that each action label occurs in at most one summand, so that the set of summands could be indexed by the set of action labels. This is no limitation, because any LPE can be transformed in clustered form, basically by replacing $+$ by $\sum$ over finite types. Clustered LPEs enable a notationally smoother presentation of the theory. However, when working with concrete LPEs this restriction is not convenient, so we avoid it in the PVS framework: an arbitrarily sized index set $\{0,\ldots,n-1\}$ will be used, represented by the PVS type `below(n)`. A second deviation is that we will assume from now on that every summand has the same set of local variables. Again this is no limitation, because void summations can always be added (i.e., $p=\sum_{d:D}p$, when d does not occur in p). This restriction is needed to avoid the use of polymorphism, which does not exist in PVS. The third deviation is that we do not distinguish action labels from action data parameters. We simply work with one type of expressions for actions. Note that this is a real extension, because one summand may now generate steps with various action labels, possibly visible as well as invisible.

So an LPE is parameterized by sets of actions (`Act`), global parameters (`State`) and local variables (`Local`), and by the size of its index set (`n`). Note that the guard, action and next-state of a summand depend on the global parameters d:`State` and on local variables e:`Local`. This dependency is represented in the definition `SUMMAND` by a PVS function type. An LPE (see Figure 4) consists of an initial state and a list of summands indexed by `below(n)`.

Figure 5 illustrates the definition of LPE by a fragment of the linear specification $\mathbf{N}_{mod}$ of SWP in PVS. It is introduced as an `lpe` of a set of actions: `Nnonmod_act`, states: `State`, local variables: `Local`, and a digit: `11` referring to the number of summands. The LPE is identified as a pair, called `init` and `sums`, where `init` is introducing the initial state of $\mathbf{N}_{mod}$ and `sums` the summands. The first `LAMBDA` maps each number to the corresponding summand in $\mathbf{N}_{mod}$. The second `LAMBDA` is representing the summands as functions over `State` and `Local`. Here, `State` is the set of states and `Local` is the data type $D\times Nat$ of all pairs (d,k) of the summation variables, which is considered as a global variable regarding the property $p=\sum_{(d,k):\texttt{Local}}p$, which was mentioned before.

9.3. Representing Invariants

By Figure 6, we explain how to represent an invariant of the μCRL specification in PVS. Invariants are boolean functions over the set of states. We present Invariant 7.9.4 from Section 7.2 as an example.

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

Figure 7 is devoted to verifying the strong bisimilarity of $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ (Proposition 6.3). `state_f` and `local_f` are introduced to construct the state mapping between $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$.

Here is one more point (next to deviations regarding data lemmas, as explained at the end of Section 9.1) where the PVS proof deviates from the manual proof. In the manual proof we used the proof principle CL-RSP [BeG94b] to derive this equivalence, by showing 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 equation of $\mathbf{N}_{nonmod}(\ell,m,q,\ell',q',g,p,g',p')$; see Section 8.1. However, in the PVS proof we introduced the state mapping (`state_f, local_f`) from the set of states of $\mathbf{N}_{nonmod}$ to those of $\mathbf{N}_{mod}$. Then we used the corresponding relation to this state mapping, and we showed that this relation is a bisimulation relation between $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$.

```
...
State: TYPE+ = [nat,nat,Buf,nat,Buf,nat,D,nat,nat,nat]
Local: TYPE+ = [D,nat]
n: posnat
e: D
...
Nmod:lpe[Nnonmod_act, State, Local, 11] =
(# init := (0,0,null,0,null,0,0,0,0),
 sums :=
LAMBDA (i:below(11)) :
LAMBDA (state:State, local:Local) :
LET (l,m,q,l1,q1,g,p,g1,p1) = state,
 (d,k) = local IN
COND
i=0 -> (#
 act := rA(d),
 guard := in-window(l,m,mod(l+n,2*n)),
 next := (l,mod(S(m),2*n),inb(d,m,q),l1,q1,g,p,g1,p1)
#),
...
i=10 -> (#
 act := tau,
 guard := p1<length(g1),
 next := (last-seq(g1),m,release2(n)(l,last-seq(g1),q),
   l1,q1,g,p,delete-last(g1),p1)
#)
ENDCOND #)
...
```

Fig. 5. The formalization of $\mathbf{N}_{mod}$ of SWP in PVS

```
...
l,m,l1,g,p,g1,p1: var nat
q, q1: var Buf
e: var D
...
inv7.9.4(l,m,q,l1,q1,g,p,g1,p1): bool= l<=next-empty(l1,q1)
...
```

Fig. 6. An example of representing invariants in PVS

The reason why we chose not to formalize CL-RSP in PVS is that it depends on recursive process equations. Therefore it would require a deep embedding of μCRL in PVS, which would complicate the formalization too much. In PVS we defined an LPO as a list of summands (not as a recursive equation), equipped with the standard LTS semantics. It could be proved directly that state mappings preserve strong bisimulation. Still, the manual proof is based on CL-RSP, mainly for algebraic reasons: By using algebraic principles only, the stated equivalence still holds in non-standard models for process algebra + CL-RSP.

```
...
state_f(l,m:nat,q:Buf,l1:nat,q1:Buf,g,p,g1,p1): State=
 (mod(l,2*n),mod(m,2*n),modulo(q,2*n),mod(l1,2*n),
  modulo(q1,2*n),g,mod(p,2*n),g1,mod(p1,2*n))
local_f(l:Local,i:below(11)): Local=
LET (e,k)=l IN
  IF i=1 THEN (e,mod(k,2*n)) ELSE (e,k) ENDIF
...
Proposition 6.3: THEOREM bisimilar (lpe2lts(Nnonmod),lpe2lts(Nmod))
...
```

Fig. 7. Equality of $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ in PVS

```
...
fc(l,m,q,l1,q1,g,p,g1,p1): bool = (l=m OR l=l1+n)
k(i): below(2)= IF i=0 THEN 0 ELSE 1 ENDIF
h(l,m,q,l1,q1,g,p,g1,p1): List_=
 concat(ql(q1,l1,next-empty(l1,q1)),ql(q,next-empty(l1,q1),m))
mc: THEOREM FORALL d: reachable(Nnonmod)(d) IMPLIES MC(Nnonmod,Z,k,h,fc)(d)
wn: LEMMA FORALL S: reachable(Nnonmod)(S) IMPLIES WN(Nnonmod,fc)(S)
main: THEOREM brbisimilar(lpe2lts(Nmod),lpe2lts(Z))
...
```

Fig. 8. Correctness of $\mathbf{N}_{mod}$ in PVS

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

Figure 8 is devoted to verifying the branching bisimilarity of $\mathbf{N}_{mod}$ and $\mathbf{Z}$ (Theorem 5.1). `ql(q,i,j)` is used to describe the function $q[i..j\rangle$, which is defined as an application on triples. `fc(l,m,q,l1,q1,g,p,g1,p1)` defines the focus condition for $\mathbf{N}_{nonmod}(l,m,q,l',q',g,p,g',p')$ as a boolean function on the set of states. The state mapping `h` maps states of $\mathbf{N}_{nonmod}$ to states of $\mathbf{Z}$, which is called $\phi : \Xi \Rightarrow \mathit{List}$ in Section 8. `k` is a Boolean function which is used to match each external action of $\mathbf{N}_{nonmod}$ to the corresponding one of $\mathbf{Z}$. This is done by relating the numbers of the summands. As PVS requires, this function must be total, therefore without loss of generality we map all summands with an internal action from $\mathbf{N}_{nonmod}$'s specification to the second summand of $\mathbf{Z}$'s specification.

According to the cones and foci proof method [FPP05], to derive that $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$ are branching bisimilar, it is enough to check the matching criteria and the reachability of focus points. The two conditions of the cones and foci proof method are represented by `mc` and `WN`, namely matching criteria and the reachability of focus points, respectively. `mc` establishes that all the matching criteria (see Section 3) hold for every reachable state `d` in `Nnonmod`, with the aforementioned `h, k` and `fc` functions. `WN` represents the fact that from all reachable states `S` in `Nnonmod`, a focus point can be reached by a finite series of internal actions. The function `lpe2lts` provides the LTS semantics of an LPE (see [FPP05]).

9.6. Remarks on the Verification in PVS

We used PVS to find omissions and undetected potential errors that have been ignored in the manual μCRL proofs. Some of them have been shown as examples in Section 9.1. PVS formalization and verification can be reused, and PVS guided us to find some important invariants. We affirmed the termination of recursive definitions by means of various measure functions. We represented LPEs in PVS and then introduced $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ as LPEs. We verified the bisimilarity of $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$. Finally we used the cones and foci proof method [FoP03], to prove that $\mathbf{N}_{mod}$ and the external behavior of the SWP, represented by $\mathbf{Z}$, are branching bisimilar.

10. Conclusions

In this paper we have proved the correctness of a SWP with an arbitrary finite window size n and sequence numbers modulo $2n$. We showed that the SWP is branching bisimilar to a queue of capacity $2n$. This proof is entirely based on the axiomatic theory underlying μCRL and the axioms characterizing the data types, and was checked with the help of PVS. It implies both safety and liveness of the protocol. Liveness depends on the fairness assumption underlying branching bisimilarity that in each infinite trace a transition that is enabled infinitely often is performed infinitely often. Of course it is still possible to make an implementation of the SWP that complies with our specification but that does not live up to this fairness assumption, so that such an implementation does violate liveness.

Acknowledgments This research is partly supported by the Dutch Technology Foundation STW under the project *Improving the Quality of Embedded Systems using Formal Design and Systematic Testing*, and by the Netherlands Organisation for Scientific Research NWO under the project *Integrating Techniques for the Verification of Distributed Systems*.

References

[BeK84] Bergstra JA, Klop JW (1984) Process algebra for synchronous communication. *Information and Computation*, 60:109–137.

[BeK86] Bergstra JA, Klop JW (1986) 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.

[BeG94a] Bezem MA, Groote JF (1994) A correctness proof of a one bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307.

[BeG94b] Bezem MA, Groote JF (1994) Invariants in process algebra with data. In: *Proc. 5th Conference on Concurrency Theory*, LNCS 836, pp. 401–416. Springer.

[Bru93] Brunekreef JJ (1993) Sliding window protocols. In *Algebraic Specification of Protocols.* Cambridge Tracts in Theoretical Computer Science 36, pp. 71–112. Cambridge University Press.

[Car91] Cardell-Oliver R (1991) Using 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.

[CeK74] Cerf VG, Kahn RE (1974) A protocol for packet network intercommunication. *IEEE Transactions on Communications*, COM-22:637–648.

[CHV03] Chkliaev D, Hooman J, de Vink E (2003) 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.

[FGP+04] Fokkink WJ, Groote JF, Pang J, Badban B, van de Pol JC (2004) Verifying a sliding window protocol in CRL. In *Proc. 10th Conference on Algebraic Methodology and Software Technology*, LNCS 3116, pp. 148-163. Springer.

[FoP03] Fokkink WJ, Pang J (2003) 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.

[FPP05] Fokkink WJ, Pang J, van de Pol JC (2005) Cones and foci: A mechanical framework for protocol verification. *Formal Methods in System Design*, To appear.

[Gla94] van Glabbeek, RJ (1994) What is branching time and why to use it? *The Concurrency Column, Bulletin of the EATCS*, 53:190-198, 1994.

[GlW96] van Glabbeek RJ, Weijland WP (1996) Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:555–600.

[GoL99] Godefroid P, Long DE (1999) Symbolic protocol verification with Queue BDDs. *Formal Methods and System Design*, 14(3):257–271.

[Gro87] Groenveld RA (1987) Verification of a sliding window protocol by means of process algebra. Report P8701, University of Amsterdam.

[Gro91] Groote JF (1991) *Process Algebra and Structured Operational Semantics.* PhD thesis, University of Amsterdam.

[GrK95] Groote JF, Korver HP (1995) 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.

[GrP94] Groote JF, Ponse A (1994) 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.

[GrP95] Groote JF, Ponse A (1995) The syntax and semantics of μCRL. In: *Proc. 1st Workshop on the Algebra of Communicating Processes*, Workshops in Computing Series, pp. 26–62. Springer.

[GPU01] Groote JF, Ponse A, Usenko YS (2001) Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1/2):39–72.
[GrR01] Groote JF, Reniers MA (2001) Algebraic process verification. In: Bergstra JA, Ponse A, Smolka SA (eds) *Handbook of Process Algebra*, pp. 1151–1208. Elsevier.
[GrS01] Groote JF, Springintveld J (2001) Focus points and convergent process operators. A proof strategy for protocol verification. *Journal of Logic and Algebraic Programming*, 49(1/2):31–60.
[Hai82] Hailpern BT (1982) *Verifying Concurrent Processes Using Temporal Logic.* LNCS 129, Springer.
[Hol91] Holzmann GJ (1991) *Design and Validation of Computer Protocols.* Prentice Hall.
[Hol97] Holzmann GJ (1997) The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279-295.
[Jon87] Jonsson B (1987) *Compositional Verification of Distributed Systems.* PhD thesis, Department of Computer Science, Uppsala University.
[JoN00] Jonsson B, Nilsson M (2000) 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.
[Kai97] Kaivola R (1997) Using compositional preorders in the verification of sliding window protocol. In: *Proc. 9th Conference on Computer Aided Verification*, LNCS 1254, pp. 48–59. Springer.
[Knu81] Knuth DE (1981) Verification of link-level protocols. *BIT*, 21:21–36.
[Lat01] Latvala T (2001) 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.
[LEW96] Loeckx J, Ehrich HD, Wolf M (1996) *Specification of Abstract Data Types.* Wiley/Teubner.
[MaV91] Madelaine E, Vergamini D (1991) 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.
[MaV90] Mauw S, Veltink GJ (1990) A process specification formalism. *Fundamenta Informaticae*, 13(2):85–139.
[Mid86] Middeldorp A (1986) Specification of a sliding window protocol within the framework of process algebra. Report FVI 86-19, University of Amsterdam.
[ORR$^+$96] Owre S, Rajan R, Rushby JM, Shankar N, Srivas MK (1996) PVS: Combining specification, proof checking, and model checking. In: *Proc. 8th Conference on Computer-Aided Verification*, LNCS 1102, pp. 411–414. Springer.
[PaS91] Paliwoda K, Sanders JM (1991) An incremental specification of the sliding-window protocol. *Distributed Computing*, 5:83–94.
[Par81] Park DMR (1981) Concurrency and automata on infinite sequences. In: *Proc. 5th GI-Conference on Theoretical Computer Science*, LNCS 104, pp. 167–183. Springer.
[RRS$^+$87] Richier JL, Rodriguez C, Sifakis J, Voiron J (1987) Verification in Xesar of the sliding window protocol. In: *Proc. 7th Conference on Protocol Specification, Testing and Verification*, pp. 235–248. North-Holland.
[RoE99] Röckl C, Esparza J (1999) Proof-checking protocols using bisimulations. In: *Proc. 10th Conference on Concurrency Theory*, LNCS 1664, pp. 525–540. Springer.
[Rus01] Rusu V (2001) Verifying a sliding-window protocol using PVS. In: *Proc. 21st Conference on Formal Techniques for Networked and Distributed Systems*, IFIP Conference Proceedings 197, pp. 251-268. Kluwer.
[Sch91] Schoone AA (1991) *Assertional Verification in Distributed Computing.* PhD thesis, Utrecht University.
[SmK00] Smith MA, Klarlund N (2000) Verification of a sliding window protocol using IOA and MONA. In: *Proc. 20th Conference on Formal Techniques for Distributed System Development*, IFIP Conference Proceedings 183, pp. 19–34. Kluwer.
[Sne95] van de Snepscheut JLA (1995) The sliding window protocol revisited. *Formal Aspects of Computing*, 7(1):3–17.
[SBL$^+$99] Stahl K, Baukus K, Lakhnech Y, Steffen M (1999) Divide, abstract, and model-check. In: *Proc. 6th International SPIN Workshop*, LNCS 1680, pp. 57–76. Springer.
[Ste76] Stenning NV (1976) A data transfer protocol. *Computer Networks*, 1(2):99–110.
[Tan81] Tanenbaum AS (1981) *Computer Networks.* Prentice Hall.
[Vaa86] Vaandrager FW (1986) Verification of two communication protocols by means of process algebra. Report CS-R8608, CWI, Amsterdam.
[Wam92] van Wamel JJ (1992) A study of a one bit sliding window protocol in ACP. Report P9212, University of Amsterdam.

A. Proofs on Properties of Data

This appendix contains proofs of the lemmas in Section 7.

A.1. Modulo Arithmetic

We present a proof of Lemma 7.1

Proof.

1. By induction on i.
   - $i < n$.

Then $i|_n = i$.

- $i \geq n$.

$$\begin{array}{lll} & (i|_n + j)|_n & \\ = & ((i \dot{-} n)|_n + j)|_n & \\ = & ((i \dot{-} n) + j)|_n & \text{(by induction, } i, n > 0) \\ = & ((i + j) \dot{-} n)|_n & (i \geq n) \\ = & (i + j)|_n & \end{array}$$

2. Trivial, by induction on i.
3. By induction on i.

- $i < n$.
  Then $i \mathit{div}\, n = 0$ and $i|_n = i$. Clearly, $i = 0{\cdot}n + i$.
- $i \geq n$.
  Then $i \mathit{div}\, n = S((i \dot{-} n) \mathit{div}\, n)$ and $i|_n = (i \dot{-} n)|_n$. Hence,

$$\begin{array}{lll} & i & \\ = & (i \dot{-} n) + n & \text{(because } i \geq n) \\ = & ((i \dot{-} n) \mathit{div}\, n){\cdot}n + (i \dot{-} n)|_n + n & \text{(by induction, } i, n > 0) \\ = & S((i \dot{-} n) \mathit{div}\, n){\cdot}n + (i \dot{-} n)|_n & \\ = & (i \mathit{div}\, n){\cdot}n + i|_n & \end{array}$$

4. Let $i \leq j < i + n$.
   CASE 1: $j \mathit{div}\, 2n < i \mathit{div}\, 2n$. This leads to a contradiction.

$$\begin{array}{lll} & i \dot{-} j & \\ = & (i \mathit{div}\, 2n){\cdot}2n + i|_{2n} \dot{-} ((j \mathit{div}\, 2n){\cdot}2n + j|_{2n}) & \text{(Lem. 7.1.3)} \\ = & (i \mathit{div}\, 2n \dot{-} j \mathit{div}\, 2n){\cdot}2n + (i|_{2n} \dot{-} j|_{2n}) & \\ \geq & 2n + (i|_{2n} \dot{-} j|_{2n}) & (j \mathit{div}\, 2n < i \mathit{div}\, 2n) \\ > & 2n \dot{-} 2n & \text{(Lem. 7.1.2)} \\ = & 0 & \text{(contradict } i \leq j) \end{array}$$

   CASE 2: $j \mathit{div}\, 2n = i \mathit{div}\, 2n$. We need to show $i|_{2n} \leq j|_{2n} < i|_{2n} + n$.

$$\begin{array}{lll} & i \leq j < i + n & \\ = & (i \mathit{div}\, 2n){\cdot}2n + i|_{2n} \leq (j \mathit{div}\, 2n){\cdot}2n + j|_{2n} < (i \mathit{div}\, 2n){\cdot}2n + i|_{2n} + n & \text{(Lem. 7.1.3)} \\ = & i|_{2n} \leq j|_{2n} < i|_{2n} + n & (j \mathit{div}\, 2n = i \mathit{div}\, 2n) \end{array}$$

   CASE 3: $j \mathit{div}\, 2n = S(i \mathit{div}\, 2n)$. We need to show $j|_{2n} + n < i|_{2n}$.

$$\begin{array}{lll} & j < i + n & \\ = & (j \mathit{div}\, 2n){\cdot}2n + j|_{2n} < (i \mathit{div}\, 2n){\cdot}2n + i|_{2n} + n & \text{(Lem. 7.1.3)} \\ = & (i \mathit{div}\, 2n){\cdot}2n + 2n + j|_{2n} < (i \mathit{div}\, 2n){\cdot}2n + i|_{2n} + n & (j \mathit{div}\, 2n = S(i \mathit{div}\, 2n)) \\ = & j|_{2n} + n < i|_{2n} & \end{array}$$

   CASE 4: $j \mathit{div}\, 2n > S(i \mathit{div}\, 2n)$. This leads to a contradiction.

$$\begin{array}{lll} & j \dot{-} (i + n) & \\ = & (j \mathit{div}\, 2n){\cdot}2n + j|_{2n} \dot{-} ((i \mathit{div}\, 2n){\cdot}2n + i|_{2n}) \dot{-} n & \text{(Lem. 7.1.3)} \\ \geq & (i \mathit{div}\, 2n){\cdot}2n + 4n + j|_{2n} \dot{-} (i \mathit{div}\, 2n){\cdot}2n \dot{-} i|_{2n} \dot{-} n & (j \mathit{div}\, 2n > S(i \mathit{div}\, 2n)) \\ = & 3n + j|_{2n} \dot{-} i|_{2n} & \\ > & 3n \dot{-} 2n & \text{(Lem. 7.1.2)} \\ > & 0 & \text{(contradict } j < i + n) \end{array}$$

5. By induction on i.

- $i < n$.
  Then $i \mathit{div}\, n = 0$.

- $i \geq n$.

$$\begin{array}{lll} & i \mathit{\ div\ } n & \\ = & S((i \dot{-} n) \mathit{\ div\ } n) & \\ \leq & S((j \dot{-} n) \mathit{\ div\ } n) & \text{(by induction, because } i \leq j, n > 0) \\ = & j \mathit{\ div\ } n & \text{(because } n \leq i \leq j) \end{array}$$

□

A.2. Buffers

We present a proof of Lemma 7.2

Proof.

1. By induction on the structure of q.
   - $q = []$.
     $\mathit{test}(i, \mathit{remove}(j, [])) = \mathit{test}(i, []) = \mathsf{f} = \mathit{test}(i, []) \wedge i \neq j$.
   - $q = \mathit{inb}(d, k, q')$.
     CASE 1: $j = k$.

$$\begin{array}{lll} & \mathit{test}(i, \mathit{remove}(j, \mathit{inb}(d, k, q'))) & \\ = & \mathit{test}(i, \mathit{remove}(j, q')) & \\ = & \mathit{test}(i, q') \wedge i \neq j & \text{(by induction)} \\ = & \mathit{test}(i, \mathit{inb}(d, k, q')) \wedge i \neq j & \text{(because } j = k) \end{array}$$

     CASE 2: $j \neq k$.
     CASE 2.1: $i = k$. Then $i \neq j$.

$$\begin{array}{ll} & \mathit{test}(i, \mathit{remove}(j, \mathit{inb}(d, k, q'))) \\ = & \mathit{test}(i, \mathit{inb}(d, k, \mathit{remove}(j, q'))) \\ = & \mathsf{t} \\ = & \mathit{test}(i, \mathit{inb}(d, k, q')) \wedge i \neq j \end{array}$$

     CASE 2.2: $i \neq k$.

$$\begin{array}{lll} & \mathit{test}(i, \mathit{remove}(j, \mathit{inb}(d, k, q'))) & \\ = & \mathit{test}(i, \mathit{inb}(d, k, \mathit{remove}(j, q'))) & \\ = & \mathit{test}(i, \mathit{remove}(j, q')) & \\ = & \mathit{test}(i, q') \wedge i \neq j & \text{(by induction)} \\ = & \mathit{test}(i, \mathit{inb}(d, k, q')) \wedge i \neq j & \end{array}$$

2. By induction on the structure of q.
   - $q = []$.
     $\mathit{remove}(j, []) = []$.
   - $q = \mathit{inb}(d, k, q')$.
     CASE 1: $j = k$.

$$\begin{array}{lll} & \mathit{retrieve}(i, \mathit{remove}(j, \mathit{inb}(d, k, q'))) & \\ = & \mathit{retrieve}(i, \mathit{remove}(j, q')) & \\ = & \mathit{retrieve}(i, q') & \text{(by induction)} \\ = & \mathit{retrieve}(i, \mathit{inb}(d, k, q')) & \end{array}$$

     CASE 2: $j \neq k$.
     CASE 2.1: $i = k$.

$$\begin{array}{ll} & \mathit{retrieve}(i, \mathit{remove}(j, \mathit{inb}(d, k, q'))) \\ = & \mathit{retrieve}(i, \mathit{inb}(d, k, \mathit{remove}(j, q'))) \\ = & d \\ = & \mathit{retrieve}(i, \mathit{inb}(d, k, q')) \end{array}$$

Case 2.2: $i \neq k$.

$$\begin{array}{rll} & \mathit{retrieve}(i, \mathit{remove}(j, \mathit{inb}(d,k,q'))) & \\ = & \mathit{retrieve}(i, \mathit{inb}(d,k,\mathit{remove}(j,q'))) & \\ = & \mathit{retrieve}(i, \mathit{remove}(j,q')) & \\ = & \mathit{retrieve}(i,q') & \text{(by induction)} \\ = & \mathit{retrieve}(i, \mathit{inb}(d,k,q')) & \end{array}$$

3. By induction on $k \dot{-} j$.

- $j \geq k$.
  Then $\mathit{test}(i, \mathit{release}(j,k,q)) = \mathit{test}(i,q)$ and $\neg(j \leq i < k) = \mathsf{t}$.
- $j < k$.

$$\begin{array}{rll} & \mathit{test}(i, \mathit{release}(j,k,q)) & \\ = & \mathit{test}(i, \mathit{release}(S(j),k,\mathit{remove}(j,q))) & \\ = & \mathit{test}(i, \mathit{remove}(j,q)) \wedge \neg(S(j) \leq i < k) & \text{(by induction)} \\ = & \mathit{test}(i,q) \wedge \neg(j \leq i < k) & \text{(Lem. 7.2.1)} \end{array}$$

4. By induction on $k \dot{-} j$.

- $j \geq k$.
  Then $\mathit{retrieve}(i, \mathit{release}(j,k,q)) = \mathit{retrieve}(i,q)$.
- $j < k$.
  Then $\neg(j \leq i < k)$ implies $i \neq j$. Hence,

$$\begin{array}{rll} & \mathit{retrieve}(i, \mathit{release}(j,k,q)) & \\ = & \mathit{retrieve}(i, \mathit{release}(S(j),k,\mathit{remove}(j,q))) & \\ = & \mathit{retrieve}(i, \mathit{remove}(j,q)) & \text{(by induction)} \\ = & \mathit{retrieve}(i,q) & \text{(Lem. 7.2.2, because } i \neq j\text{)} \end{array}$$

5. By induction on the structure of q.

- $g = []$.
  This case is trivial.
- $q = \mathit{inb}(d,k,q')$.
  By definition, $\mathit{max}(\mathit{inb}(d,k,q')) = \mathit{if}(k \geq \mathit{max}(q'), k, \mathit{max}(q'))$.
  Case 1: $k \geq \mathit{max}(q')$. Then $\mathit{max}(\mathit{inb}(d,k,q')) = k$.
  Clearly, $\mathit{test}(k, \mathit{inb}(d,k,q'))$.
  Case 2: $k < \mathit{max}(q')$. Then $\mathit{max}(\mathit{inb}(d,k,q')) = \mathit{max}(q')$.
  $\mathit{test}(\mathit{max}(q'), \mathit{inb}(d,k,q')) = \mathit{test}(\mathit{max}(q'), q')$. By induction, $\mathit{test}(\mathit{max}(q'), q')$ holds.

□

A.3. Buffers with Modulo Arithmetic

We present a proof of Lemma 7.3.

Proof.

1. By induction on the structure of q.

- $q = []$.
  $\mathit{test}(k, []) = \mathsf{f} = \mathit{test}(k|_{2n}, []|_{2n})$.
- $q = \mathit{inb}(d,\ell,q')$.
  Let $\mathit{test}(j,q) \Rightarrow i \leq j < i+n$ and $i \leq k \leq i+n$.
  Case 1: $k|_{2n} = \ell|_{2n}$.
  $\mathit{test}(\ell,q)$, so $i \leq \ell < i+n$. In combination with $i \leq k \leq i+n$ and $k|_{2n} = \ell|_{2n}$, this implies $k = \ell$.
  Hence, $\mathit{test}(k,q)$. Furthermore, $k|_{2n} = \ell|_{2n}$ implies $\mathit{test}(k|_{2n}, q|_{2n})$.

CASE 2: $k|_{2n} \neq \ell|_{2n}$. Then also $k \neq \ell$.
$\mathit{test}(j, q') \Rightarrow \mathit{test}(j, q) \Rightarrow i \leq j < i + n$, so induction can be applied with respect to q'.

$$\begin{array}{ll} & \mathit{test}(k, \mathit{inb}(d, \ell, q')) \\ = & \mathit{test}(k, q') \\ = & \mathit{test}(k|_{2n}, q'|_{2n}) \qquad \text{(by induction)} \\ = & \mathit{test}(k|_{2n}, \mathit{inb}(d, \ell, q')|_{2n}) \end{array}$$

2. By induction on the structure of q.

- $q = []$.
  $\mathit{test}(k, []) = \mathsf{f}$.
- $q = \mathit{inb}(d, \ell, q')$.
  Let $\mathit{test}(j, q) \Rightarrow i \leq j < i + n$ and $\mathit{test}(k, q)$.
  CASE 1: $k = \ell$. Then also $k|_{2n} = \ell|_{2n}$.
  Hence, $\mathit{retrieve}(k, q) = d = \mathit{retrieve}(k|_{2n}, q|_{2n})$.
  CASE 2: $k \neq \ell$.
  $\mathit{test}(j, q') \Rightarrow \mathit{test}(j, q) \Rightarrow i \leq j < i + n$, and $\mathit{test}(k, q)$ together with $k \neq \ell$ implies $\mathit{test}(k, q')$, so induction can be applied with respect to q'.
  $\mathit{test}(k, q)$ and $\mathit{test}(\ell, q)$, so $i \leq k < i + n$ and $i \leq \ell < i + n$. In combination with $k \neq \ell$, this implies $k|_{2n} \neq \ell|_{2n}$. Hence,

$$\begin{array}{ll} & \mathit{retrieve}(k, q) \\ = & \mathit{retrieve}(k, q') \\ = & \mathit{retrieve}(k|_{2n}, q'|_{2n}) \qquad \text{(by induction)} \\ = & \mathit{retrieve}(k|_{2n}, q|_{2n}) \end{array}$$

3. By induction on the structure of q.

- $q = []$.
  $\mathit{remove}(k, [])|_{2n} = [] = \mathit{remove}(k|_{2n}, []|_{2n})$.
- $q = \mathit{inb}(d, \ell, q')$.
  Let $\mathit{test}(j, q) \Rightarrow i \leq j < i + n$ and $i \leq k \leq i + n$.
  CASE 1: $k = \ell$. Then also $k|_{2n} = \ell|_{2n}$.

$$\begin{array}{ll} & \mathit{remove}(k, q)|_{2n} \\ = & \mathit{remove}(k, q')|_{2n} \\ = & \mathit{remove}(k|_{2n}, q'|_{2n}) \qquad \text{(by induction)} \\ = & \mathit{remove}(k|_{2n}, q|_{2n}) \end{array}$$

  CASE 2: $k \neq \ell$.
  $\mathit{test}(\ell, q)$, so $i \leq \ell < i + n$. In combination with $i \leq k \leq i + n$ and $k \neq \ell$, this implies $k|_{2n} \neq \ell|_{2n}$. Hence,

$$\begin{array}{ll} & \mathit{remove}(k, q)|_{2n} \\ = & \mathit{inb}(d, \ell, \mathit{remove}(k, q'))|_{2n} \\ = & \mathit{inb}(d, \ell|_{2n}, \mathit{remove}(k, q')|_{2n}) \\ = & \mathit{inb}(d, \ell|_{2n}, \mathit{remove}(k|_{2n}, q'|_{2n})) \qquad \text{(by induction)} \\ = & \mathit{remove}(k|_{2n}, q|_{2n}) \end{array}$$

4. By induction on $k \dot{-} i$. Let $\mathit{test}(j, q) \Rightarrow i \leq j < i + n$.

- $i = k$. Then also $i|_{2n} = k|_{2n}$.
  Hence, $\mathit{release}(i, k, q)|_{2n} = q|_{2n} = \mathit{release}|_{2n}(i, k, q|_{2n})$.
- $i < k \leq i + n$.

Then $i|_{2n} \neq k|_{2n}$. Hence,

$$\begin{array}{rll} & \mathit{release}(i,k,q)|_{2n} & \\ = & \mathit{release}(S(i),k,\mathit{remove}(i,q))|_{2n} & \\ = & \mathit{release}|_{2n}(S(i),k,\mathit{remove}(i,q)|_{2n}) & \text{(by induction)} \\ = & \mathit{release}|_{2n}(S(i),k,\mathit{remove}(i|_{2n},q|_{2n})) & \text{(Lem. 7.3.3)} \\ = & \mathit{release}|_{2n}(i,k,q|_{2n}) & \end{array}$$

5. By induction on $(i+n) \dot{-} k$. Let $\mathit{test}(j,q) \Rightarrow i \leq j < i+n$.

- $k = i+n$.
  $\neg\mathit{test}(i+n,q)$, so by Lemma 7.3.1, $\neg\mathit{test}((i+n)|_{2n},q|_{2n})$. By Lemma 7.1.2, $(i+n)|_{2n} < 2n$. Hence,

$$\begin{array}{rl} & \mathit{next\text{-}empty}(i+n,q)|_{2n} \\ = & (i+n)|_{2n} \\ = & \mathit{next\text{-}empty}((i+n)|_{2n},q|_{2n}) \\ = & \mathit{next\text{-}empty}|_{2n}((i+n)|_{2n},q|_{2n}) \end{array}$$

- $i \leq k \leq i+n$.
  Case 1: $\neg\mathit{test}(k,q)$. By Lemma 7.3.1, also $\neg\mathit{test}(k|_{2n},q|_{2n})$.
  By Lemma 7.1.2, $k|_{2n} < 2n$. Hence,

$$\begin{array}{rl} & \mathit{next\text{-}empty}(k,q)|_{2n} \\ = & k|_{2n} \\ = & \mathit{next\text{-}empty}(k|_{2n},q|_{2n}) \\ = & \mathit{next\text{-}empty}|_{2n}(k|_{2n},q|_{2n}) \end{array}$$

  Case 2: $\mathit{test}(k,q)$. By Lemma 7.3.1, also $\mathit{test}(k|_{2n},q|_{2n})$.
  We prove $\mathit{next\text{-}empty}|_{2n}(k|_{2n},q|_{2n}) = \mathit{next\text{-}empty}|_{2n}(S(k)|_{2n},q|_{2n})$.
  Case 2.1: $k|_{2n} = 2n \dot{-} 1$.
  By Lemma 7.5.3,

$$\begin{array}{rl} & \mathit{next\text{-}empty}(k|_{2n},q|_{2n}) \\ = & \mathit{next\text{-}empty}(S(k|_{2n}),q|_{2n}) \\ = & \mathit{next\text{-}empty}(2n,q|_{2n}) \\ \geq & 2n \end{array}$$

  Hence,

$$\begin{array}{rl} & \mathit{next\text{-}empty}|_{2n}(k|_{2n},q|_{2n}) \\ = & \mathit{next\text{-}empty}(0,q|_{2n}) \\ = & \mathit{next\text{-}empty}|_{2n}(S(k)|_{2n},q|_{2n}) \end{array}$$

  Case 2.2: $k|_{2n} < 2n \dot{-} 1$.
  Using Lemma 7.1.1, we can derive $S(k)|_{2n} = S(k|_{2n})$. Since

$$\begin{array}{rl} & \mathit{next\text{-}empty}(k|_{2n},q|_{2n}) \\ = & \mathit{next\text{-}empty}(S(k|_{2n}),q|_{2n}) \\ = & \mathit{next\text{-}empty}(S(k)|_{2n},q|_{2n}) \end{array}$$

  it follows that

$$\begin{array}{rl} & \mathit{next\text{-}empty}|_{2n}(k|_{2n},q|_{2n}) \\ = & \mathit{next\text{-}empty}|_{2n}(S(k)|_{2n},q|_{2n}) \end{array}$$

  Concluding,

$$\begin{array}{rll} & \mathit{next\text{-}empty}(k,q)|_{2n} & \\ = & \mathit{next\text{-}empty}(S(k),q)|_{2n} & \\ = & \mathit{next\text{-}empty}|_{2n}(S(k)|_{2n},q|_{2n}) & \text{(by induction)} \\ = & \mathit{next\text{-}empty}|_{2n}(k|_{2n},q|_{2n}) & \end{array}$$

6. By induction on the structure of q.

- $q = []$.
  $test(k, []|_{2n}) = \mathsf{f}$.
- $q = inb(d, \ell, q')$.
  Then $test(\ell, q)$, so $i \leq \ell < i+n$. Thus, by Lemma 7.1.4, $i|_{2n} \leq \ell|_{2n} < i|_{2n} + n \vee \ell|_{2n} + n < i|_{2n}$. Hence,

$$\begin{array}{ll} & test(k, inb(d, \ell, q')|_{2n}) \\ \Leftrightarrow & k = \ell|_{2n} \vee test(k, q'|_{2n}) \\ \Rightarrow & k = \ell|_{2n} \vee i|_{2n} \leq k < i|_{2n} + n \vee k + n < i|_{2n} \quad \text{(by induction)} \\ \Leftrightarrow & i|_{2n} \leq k < i|_{2n} + n \vee k + n < i|_{2n} \end{array}$$

□

A.4. *in-window*

We present a proof of Lemma 7.4.

Proof.

1. Let $i \leq k < i + n$.
   CASE 1: $S(i\ div\ 2n) \cdot 2n \leq k$.
   Then $S(i\ div\ 2n) \cdot 2n \leq k < i + n < S(i\ div\ 2n) \cdot 2n + n$ (by Lem. 7.1.3). It follows that $k\ div\ 2n = (i + n)\ div\ 2n = S(i\ div\ 2n)$. Hence, in view of Lemma 7.1.3, $k|_{2n} < (i + n)|_{2n} < i|_{2n}$.
   CASE 2: $k < S(i\ div\ 2n) \cdot 2n \leq i + n$.
   Then $(i\ div\ 2n) \cdot 2n \leq i \leq k < (i\ div\ 2n) \cdot 2n + 2n$, so by Lemma 7.1.5 $k\ div\ 2n = i\ div\ 2n$. Furthermore, $S(i\ div\ 2n) \cdot 2n \leq i+n < S(i\ div\ 2n) \cdot 2n+n$, so $(i + n)\ div\ 2n = S(i\ div\ 2n)$. Hence, $(i + n)|_{2n} < i|_{2n} \leq k|_{2n}$.
   CASE 3: $i + n < S(i\ div\ 2n) \cdot 2n$.
   Then $(i\ div\ 2n) \cdot 2n \leq i \leq k < i + n < (i\ div\ 2n) \cdot 2n + 2n$, so by Lemma 7.1.5 $k\ div\ 2n = (i + n)\ div\ 2n = i\ div\ 2n$. Hence, $i|_{2n} \leq k|_{2n} < (i + n)|_{2n}$.
   By definition,

$$\begin{array}{ll} & in\text{-}window(i|_{2n}, k|_{2n}, (i + n)|_{2n}) \\ = & i|_{2n} \leq k|_{2n} < (i + n)|_{2n} \vee \\ & (i + n)|_{2n} < i|_{2n} \leq k|_{2n} \vee \\ & k|_{2n} < (i + n)|_{2n} < i|_{2n} \end{array}$$

   so in all three cases we conclude $in\text{-}window(i|_{2n}, k|_{2n}, (i + n)|_{2n})$.
2. We prove

$$(i + n \leq k < i + 2n \vee i \leq k + n < i + n) \ \Rightarrow \ \neg in\text{-}window(i|_{2n}, k|_{2n}, (i + n)|_{2n}).$$

   - $i + n \leq k < i + 2n$.
     Then $i\ div\ 2n \leq (i + n)\ div\ 2n \leq k\ div\ 2n \leq S(i\ div\ 2n)$. We distinguish three cases, in which we repeatedly apply Lemma 7.1.3.
     CASE 1: $i\ div\ 2n = (i + n)\ div\ 2n = k\ div\ 2n$.
     Then $i < i + n$ yields $i|_{2n} < (i + n)|_{2n}$ and $i + n \leq k$ yields $(i + n)|_{2n} \leq k|_{2n}$.
     CASE 2: $S(i\ div\ 2n) = S((i + n)\ div\ 2n) = k\ div\ 2n$.
     Then $i < i + n$ yields $i|_{2n} < (i + n)|_{2n}$ and $k < i + 2n$ yields $k|_{2n} < i|_{2n}$.
     CASE 3: $S(i\ div\ 2n) = (i + n)\ div\ 2n = k\ div\ 2n$.
     Then $i + n \leq k$ yields $(i + n)|_{2n} \leq k|_{2n}$ and $k < i + 2n$ yields $k|_{2n} < i|_{2n}$.
     In all three cases we can conclude $\neg in\text{-}window(i|_{2n}, k|_{2n}, (i + n)|_{2n})$.
   - $i \leq k + n < i + n$.
     Then $i + n \leq k + 2n < i + 2n$, so by CASE 1, $\neg in\text{-}window(i|_{2n}, (k + 2n)|_{2n}, (i + n)|_{2n})$. Hence, $\neg in\text{-}window(i|_{2n}, k|_{2n}, (i + n)|_{2n})$.

□

A.5. Buffers and *next-empty*

We present a proof of Lemma 7.5

Proof.

1. By induction on the structure of q.
   - $q = []$.
     $\mathit{test}(i, []) = \mathsf{f}$.
   - $q = \mathit{inb}(d, j, q')$.
     CASE 1: $i = j$.
     Then clearly $i \leq \mathit{max}(\mathit{inb}(d, j, q'))$.
     CASE 2: $i \neq j$.
     Then $\mathit{test}(i, \mathit{inb}(d, j, q'))$ implies $\mathit{test}(i, q')$, so
     $$i \leq \mathit{max}(q') \text{ (by induction) } \leq \mathit{max}(\mathit{inb}(d, j, q')).$$
2. By induction on $j \dot{-} i$.
   - $i = j$.
     $\neg\mathit{test}(i, q)$ implies $\mathit{next\text{-}empty}(i, q) = i = j$.
   - $i < j$.
     CASE 1: $\neg\mathit{test}(i, q)$.
     Then $\mathit{next\text{-}empty}(i, q) = i < j$.
     CASE 2: $\mathit{test}(i, q)$.
     Then $i < j < \mathit{next\text{-}empty}(i, q) \Leftrightarrow S(i) \leq j < \mathit{next\text{-}empty}(S(i), q) \Rightarrow \mathit{test}(j, q)$ (by induction).
3. By induction on $S(\mathit{max}(q)) \dot{-} i$.
   - $\neg\mathit{test}(i, q)$. (This includes the base case $S(\mathit{max}(q)) \leq i$.)
     Then $\mathit{next\text{-}empty}(i, q) = i$.
   - $\mathit{test}(i, q)$.
     According to Lemma 7.5.1, $i \leq \mathit{max}(q)$, so $S(\mathit{max}(q)) \dot{-} S(i) < S(\mathit{max}(q)) \dot{-} i$. Hence, by induction, $\mathit{next\text{-}empty}(i, q) = \mathit{next\text{-}empty}(S(i), q) > i$.
4. By induction on $S(\mathit{max}(q)) \dot{-} i$.
   - $\neg\mathit{test}(i, q)$.
     Then $\mathit{next\text{-}empty}(i, \mathit{inb}(d, j, q)) \geq i$ (Lem. 7.5.3) $= \mathit{next\text{-}empty}(i, q)$.
   - $\mathit{test}(i, q)$. Then also $\mathit{test}(i, \mathit{inb}(d, j, q))$.
     By Lemma 7.5.1, $i \leq \mathit{max}(q)$, so $S(\mathit{max}(q)) \dot{-} S(i) < S(\mathit{max}(q)) \dot{-} i$. Hence,
     $$\begin{array}{ll} & \mathit{next\text{-}empty}(i, \mathit{inb}(d, j, q)) \\ = & \mathit{next\text{-}empty}(S(i), \mathit{inb}(d, j, q)) \\ \geq & \mathit{next\text{-}empty}(S(i), q) \qquad \text{(by induction)} \\ = & \mathit{next\text{-}empty}(i, q) \end{array}$$
5. By induction on $S(\mathit{max}(q)) \dot{-} i$. Let $j \neq \mathit{next\text{-}empty}(i, q)$.
   - $\neg\mathit{test}(i, q)$.
     Then $\mathit{next\text{-}empty}(i, q) = i$, so $j \neq i$, and so $\neg\mathit{test}(i, \mathit{inb}(d, j, q))$. Hence, $\mathit{next\text{-}empty}(i, \mathit{inb}(d, j, q)) = i$.
   - $\mathit{test}(i, q)$. Then also $\mathit{test}(i, \mathit{inb}(d, j, q))$.
     By Lemma 7.5.1, $i \leq \mathit{max}(q)$, so $S(\mathit{max}(q)) \dot{-} S(i) < S(\mathit{max}(q)) \dot{-} i$. Furthermore, $\mathit{test}(i, q)$ implies $j \neq \mathit{next\text{-}empty}(S(i), q)$. Hence,
     $$\begin{array}{ll} & \mathit{next\text{-}empty}(i, \mathit{inb}(d, j, q)) \\ = & \mathit{next\text{-}empty}(S(i), \mathit{inb}(d, j, q)) \\ = & \mathit{next\text{-}empty}(S(i), q) \qquad \text{(by induction)} \\ = & \mathit{next\text{-}empty}(i, q) \end{array}$$
6. By induction on $S(\mathit{max}(q)) \dot{-} i$.
   - $\neg\mathit{test}(i, q)$.

Then $\mathit{next\text{-}empty}(i,q)=i$. By Lemma 7.5.3, $\mathit{next\text{-}empty}(S(i),q)\neq i$. Hence,

$$\begin{array}{rll} & \mathit{next\text{-}empty}(i,\mathit{inb}(d,\mathit{next\text{-}empty}(i,q),q)) & \\ = & \mathit{next\text{-}empty}(i,\mathit{inb}(d,i,q)) & \\ = & \mathit{next\text{-}empty}(S(i),\mathit{inb}(d,i,q)) & \\ = & \mathit{next\text{-}empty}(S(i),q) & \text{(Lem. 7.5.5)} \\ = & \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(i,q)),q) & \end{array}$$

- $\mathit{test}(i,q)$.
  By Lemma 7.5.1, $i\le \mathit{max}(q)$, so the induction hypothesis can be applied with respect to $S(i)$.

$$\begin{array}{rll} & \mathit{next\text{-}empty}(i,\mathit{inb}(d,\mathit{next\text{-}empty}(i,q),q)) & \\ = & \mathit{next\text{-}empty}(S(i),\mathit{inb}(d,\mathit{next\text{-}empty}(S(i),q),q)) & \\ = & \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(S(i),q)),q) & \text{(by induction)} \\ = & \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(i,q)),q) & \end{array}$$

7. We apply induction on $S(\mathit{max}(q))\dot{-}i$.

- $\neg\mathit{test}(i,q)$.
  By Lemma 7.2.1, $\neg\mathit{test}(i,\mathit{remove}(j,q))$. Hence, $\mathit{next\text{-}empty}(i,\mathit{remove}(j,q))=i=\mathit{next\text{-}empty}(i,q)$.
- $\mathit{test}(i,q)$.
  Let $\neg(i\le j<\mathit{next\text{-}empty}(i,q))$. $\mathit{test}(i,q)$ implies $\neg(S(i)\le j<\mathit{next\text{-}empty}(S(i),q))$. Furthermore, by Lemma 7.5.1, $i\le \mathit{max}(q)$, so the induction hypothesis can be applied with respect to $S(i)$. Since $\mathit{next\text{-}empty}(i,q)=\mathit{next\text{-}empty}(S(i),q)\ge S(i)$ (Lem. 7.5.3), $\neg(i\le j<\mathit{next\text{-}empty}(i,q))$ implies $j\neq i$. Then, by Lemma 7.2.1, $\mathit{test}(i,\mathit{remove}(j,q))$. Hence,

$$\begin{array}{rll} & \mathit{next\text{-}empty}(i,\mathit{remove}(j,q)) & \\ = & \mathit{next\text{-}empty}(S(i),\mathit{remove}(j,q)) & \\ = & \mathit{next\text{-}empty}(S(i),q) & \text{(by induction)} \\ = & \mathit{next\text{-}empty}(i,q) & \end{array}$$

□

A.6. Unbounded Buffers

We present a proof of Lemma 7.6.11. The proofs of the other parts of Lemma 7.6 are straightforward, by induction on g, and left to reader.

Proof. We apply induction on the structure of g.

- $g=[]^K$.
  Then $\mathit{length}(g)=0$, so this case is trivial.
- . $g=\mathit{inm}(e,k,g_1)$.
  Let $i<\mathit{length}(g_1)$ and $j\le \mathit{length}(g_1)$.
  CASE 1: $j=0$.
  Then $\neg(i<j)$ and $\mathit{return\text{-}seq}(i,\mathit{delete}(j,g))$=$\mathit{return\text{-}seq}(i,g_1)$=$\mathit{return\text{-}seq}(S(i),g)$.
  CASE 2: $j>0$.
  If $i=0$, then $i<j$ and $\mathit{return\text{-}seq}(i,\mathit{delete}(j,g))=k=\mathit{return\text{-}seq}(i,g)$.
  If $i>0$, then

$$\begin{array}{rll} & \mathit{return\text{-}seq}(i,\mathit{delete}(j,g)) & \\ = & \mathit{return\text{-}seq}(i\dot{-}1,\mathit{delete}(j\dot{-}1,g_1)) & \\ = & \mathit{if}(i\dot{-}1<j\dot{-}1,\mathit{return\text{-}seq}(i\dot{-}1,g_1),\mathit{return\text{-}seq}(i,g_1)) & \text{(by induction)} \\ = & \mathit{if}(i<j,\mathit{return\text{-}seq}(i,g),\mathit{return\text{-}seq}(S(i),g)) & \end{array}$$

□

The proof of Lemma 7.7.9 is similar to the proof of Lemma 7.6.11. The other parts of Lemma 7.7 are straightforward, by induction on g'.

A.7. Lists

The proofs of the nine facts on lists in Lemma 7.8 are straightforward and left to the reader. We restrict to a listing of the induction bases.

Proof.

1. By induction on the length of λ.
2. By induction on the length of λ.
3. By induction on the length of λ.
4. By induction on $j \dot{-} i$.
5. By induction on $k \dot{-} i$.
6. By induction on $j \dot{-} i$.
7. By induction on $j \dot{-} i$.
8. By induction on $j \dot{-} i$, together with Lemmas 7.2.1 and 7.2.2.
9. By induction on $j \dot{-} i$, together with Lemmas 7.2.3 and 7.2.4.

$\square$

A.8. Invariants

We present a proof of Lemma 7.9

Proof. It is easy to verify that all invariants hold in the initial state (where the buffers and mediums are empty, the parameters in the natural numbers equal zero). In case 1-27 we show that the invariant is preserved by each of the summands A-K in the specification of $\mathbf{N}_{nonmod}$. For each of these invariants we only treat the summands in which one or more values of parameters occurring in the invariant are updated. In each of these proof obligations, we list the new values of these parameters together with those conjuncts in the condition of the summand under consideration that play a role in the proof.

1. $p \leq \mathit{length}(g)$.
   Summands B, C, D, E and F need to be checked. F is the same as E.
   B: $g := \mathit{inm}(\mathit{retrieve}(k,q),k,g)$, $p := S(p)$;
   $\mathit{length}(\mathit{inm}(\mathit{retrieve}(k,q),k,g)) = S(\mathit{length}(g)) \geq S(p)$.
   C: $g := \mathit{delete}(k,g)$, $p := p \dot{-} 1$; under condition $k < p$;
   Since $k < p \leq \mathit{length}(g)$, by Lemma 7.6.9, $\mathit{length}(\mathit{delete}(k,g))=\mathit{length}(g) \dot{-} 1 \geq p \dot{-} 1$.
   D: $p := p \dot{-} 1$; under condition $p > 0$;
   $p \dot{-} 1 < p \leq \mathit{length}(g)$.
   E: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
   Since $0 < \mathit{length}(g)$, by Lemmas 7.6.7 and 7.6.9, $\mathit{length}(\mathit{delete\text{-}last}(g))=\mathit{length}(g) \dot{-} 1 \geq p$.
2. $p' \leq \mathit{length}(g')$.
   Summands H, I, J and K need to be checked.
   H: $g' := \mathit{inm}(\mathit{next\text{-}empty}(\ell',q'),g')$, $p' := S(p')$;
   $\mathit{length}(\mathit{inm}(\mathit{next\text{-}empty}(\ell',q'),g')) = S(\mathit{length}(g')) \geq S(p')$.
   I: $g' := \mathit{delete}(k,g')$, $p' := p' \dot{-} 1$; under condition $k < p'$;
   Since $k < p' \leq \mathit{length}(g')$, by Lemma 7.7.7, $\mathit{length}(\mathit{delete}(k,g'))=\mathit{length}(g') \dot{-} 1 \geq p' \dot{-} 1$.
   J: $p' := p' \dot{-} 1$; under condition $p' > 0$;
   $p' \dot{-} 1 < p' \leq \mathit{length}(g')$.
   K: $g' := \mathit{delete\text{-}last}(g')$; under condition $p' < \mathit{length}(g')$;
   Since $0 < \mathit{length}(g')$, by Lemmas 7.7.5 and 7.7.7, $\mathit{length}(\mathit{delete\text{-}last}(g'))=\mathit{length}(g') \dot{-} 1 \geq p$.
3. $\mathit{member}(i,g') \;\Rightarrow\; i \leq \mathit{next\text{-}empty}(\ell',q')$.
   Summands E, G, H, I and K need to be checked.
   E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q')$;
   Let $\mathit{member}(i,g')$. Then $i \leq \mathit{next\text{-}empty}(\ell',q') \leq \mathit{next\text{-}empty}(\ell',\mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q'))$ (Lem. 7.5.4).

G: $\ell' := S(\ell')$, $q' := \mathit{remove}(\ell', q')$; under condition $\mathit{test}(\ell', q')$;
Let $\mathit{member}(i, g')$. Then,

$$\begin{array}{ll} & i \\ \leq & \mathit{next\text{-}empty}(\ell', q') \\ = & \mathit{next\text{-}empty}(S(\ell'), q') \\ = & \mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q')) \quad \text{(Lem. 7.5.7)} \end{array}$$

H: $g' := \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')$;
Let $\mathit{member}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g'))$.
Case 1: $i = \mathit{next\text{-}empty}(\ell', q')$.
$\mathit{next\text{-}empty}(\ell', q') \leq \mathit{next\text{-}empty}(\ell', q')$.
Case 2: $i \neq \mathit{next\text{-}empty}(\ell', q')$.
$\mathit{member}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) = \mathit{member}(i, g') \Rightarrow i \leq \mathit{next\text{-}empty}(\ell', q')$.
I: $g' := \mathit{delete}(k, g')$; under condition $k < p'$;
Let $\mathit{member}(i, \mathit{delete}(k, g'))$. By Invariant 7.9.2, $k < p' \leq \mathit{length}(g')$. By Lemma 7.7.6, $\mathit{member}(i, \mathit{delete}(k, g')) \Rightarrow \mathit{member}(i, g') \Rightarrow i \leq \mathit{next\text{-}empty}(\ell', q')$.
K: $g' := \mathit{delete\text{-}last}(g')$; under condition $p' < \mathit{length}(g')$;
Let $\mathit{member}(i, \mathit{delete\text{-}last}(g'))$. By Lemmas 7.7.5 and 7.7.6, $\mathit{member}(i, \mathit{delete\text{-}last}(g')) \Rightarrow \mathit{member}(i, g') \Rightarrow i \leq \mathit{next\text{-}empty}(\ell', q')$.

4. $\ell \leq \mathit{next\text{-}empty}(\ell', q')$.
Summands E, G and K need to be checked.
E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')$;
$\ell \leq \mathit{next\text{-}empty}(\ell', q') \leq \mathit{next\text{-}empty}(\ell', \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q'))$ (Lem. 7.5.4).
G: $\ell' := S(\ell')$, $q' := \mathit{remove}(\ell', q')$; under condition $\mathit{test}(\ell', q')$;

$$\begin{array}{ll} & \ell \\ \leq & \mathit{next\text{-}empty}(\ell', q') \\ = & \mathit{next\text{-}empty}(S(\ell'), q') \\ = & \mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q')) \quad \text{(Lem. 7.5.7)} \end{array}$$

K: $\ell := \mathit{last\text{-}seq}(g')$; under condition $p' < \mathit{length}(g')$.
Since $0 < \mathit{length}(g')$, by Lemmas 7.7.4 and 7.7.8, $\mathit{member}(\mathit{last\text{-}seq}(g'), g)$. Hence, by Invariant 7.9.3, $\mathit{last\text{-}seq}(g') \leq \mathit{next\text{-}empty}(\ell', q')$.

5. $i < j < \mathit{length}(g') \Rightarrow \mathit{return\text{-}seq}(i, g') \geq \mathit{return\text{-}seq}(j, g')$.
Summands H, I and K need to be checked.
H: $g' := \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')$;
Let $i < j < S(\mathit{length}(g'))$.
Case 1: $i > 0$. Then $i \dot{-} 1 < j \dot{-} 1 < \mathit{length}(g')$, so

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) \\ = & \mathit{return\text{-}seq}(i \dot{-} 1, g') \\ \geq & \mathit{return\text{-}seq}(j \dot{-} 1, g') \\ = & \mathit{return\text{-}seq}(j, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) \end{array}$$

Case 2: $i = 0$.
Since $j > 0$, $\mathit{return\text{-}seq}(j, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) = \mathit{return\text{-}seq}(j \dot{-} 1, g')$. Since $j \dot{-} 1 < \mathit{length}(g')$, by Lemma 7.7.8, $\mathit{member}(\mathit{return\text{-}seq}(j \dot{-} 1, g'), g')$. By Invariant 7.9.3,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(j \dot{-} 1, g') \\ \leq & \mathit{next\text{-}empty}(\ell', q') \\ = & \mathit{return\text{-}seq}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) \quad \text{(because } i = 0\text{)} \end{array}$$

I: $g' := \mathit{delete}(k, g')$; under condition $k < p'$;
Let $i < j < \mathit{length}(\mathit{delete}(k, g'))$. By Invariant 7.9.2, $k < p' \leq \mathit{length}(g')$. So by Lemma 7.7.7, $\mathit{length}(\mathit{delete}(k, g')) = \mathit{length}(g') \dot{-} 1$. Since $i < S(i) \leq j < S(j) < \mathit{length}(g')$, $\mathit{return\text{-}seq}(i, g') \geq$

$\mathit{return\text{-}seq}(S(i), g') \geq \mathit{return\text{-}seq}(j, g') \geq \mathit{return\text{-}seq}(S(j), g')$. So by Lemma 7.7.9,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete}(k, g')) \\ \geq & \mathit{return\text{-}seq}(S(i), g') \\ \geq & \mathit{return\text{-}seq}(j, g') \\ \geq & \mathit{return\text{-}seq}(j, \mathit{delete}(k, g')) \end{array}$$

K: $g' := \mathit{delete\text{-}last}(g')$; under condition $p' < \mathit{length}(g')$;
Let $i < j < \mathit{length}(\mathit{delete\text{-}last}(g'))$. Since $0 < \mathit{length}(g')$, by Lemmas 7.7.5 and 7.7.7, $\mathit{length}(\mathit{delete\text{-}last}(g')) = \mathit{length}(g') \dot{-} 1$. By Lemmas 7.7.5 and 7.7.9,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete\text{-}last}(g')) \\ = & \mathit{return\text{-}seq}(i, g') \\ \geq & \mathit{return\text{-}seq}(j, g') \\ = & \mathit{return\text{-}seq}(i, \mathit{delete\text{-}last}(g')) \end{array}$$

6. $\mathit{member}(i, g') \Rightarrow \ell \leq i$.
Summands H, I and K need to be checked.
H: $g' := \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')$;
Let $\mathit{member}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g'))$.
Case 1: $i = \mathit{next\text{-}empty}(\ell', q')$.
By Invariant 7.9.4, $\ell \leq \mathit{next\text{-}empty}(\ell', q')$.
Case 2: $i \neq \mathit{next\text{-}empty}(\ell', q')$.
$\mathit{member}(i, \mathit{inm}(\mathit{next\text{-}empty}(\ell', q'), g')) \Rightarrow \mathit{member}(i, g') \Rightarrow \ell \leq i$.
I: $g' := \mathit{delete}(k, g')$; under condition $k < p'$;
By Invariant 7.9.2, $k < p' \leq \mathit{length}(g')$. So by Lemma 7.7.6, $\mathit{member}(i, \mathit{delete}(k, g')) \Rightarrow \mathit{member}(i, g') \Rightarrow \ell \leq i$.
K: $g' := \mathit{delete\text{-}last}(g')$; under condition $p' < \mathit{length}(g')$;
Since $0 < \mathit{length}(g')$, by Lemmas 7.7.5 and 7.7.6, $\mathit{member}(i, \mathit{delete\text{-}last}(g')) \Rightarrow \mathit{member}(i, g') \Rightarrow \ell \leq i$.
7. $\mathit{test}(i, q) \Rightarrow i < m$.
Summands A and K need to be checked.
A: $m := S(m)$, $q := \mathit{inb}(d, m, q)$;
$\mathit{test}(i, \mathit{inb}(d, m, q)) \Leftrightarrow (i = m \vee \mathit{test}(i, q)) \Rightarrow (i = m \vee i < m) \Leftrightarrow i < S(m)$.
K: $q := \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)$;
$\mathit{test}(i, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)) \Rightarrow \mathit{test}(i, q)$ (Lem. 7.2.3) $\Rightarrow i < m$.
8. $\mathit{member}(d, i, g) \Rightarrow i < m$.
Summands A, B, C, E and F need to be checked. F is the same as E.
A: $m := S(m)$;
$\mathit{member}(d, i, g) \Rightarrow i < m < S(m)$.
B: $g := \mathit{inm}(\mathit{retrieve}(k, q), k, g)$; under condition $\mathit{test}(k, q)$;
Let $\mathit{member}(d, i, \mathit{inm}(\mathit{retrieve}(k, q), k, g))$.
Case 1: $i = k$.
Since $\mathit{test}(k, q)$, by Invariant 7.9.7, $k < m$.
Case 2: $i \neq k$.
$\mathit{member}(d, i, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) = \mathit{member}(d, i, g) \Rightarrow i < m$.
C: $g := \mathit{delete}(k, g)$; under condition $k < p$;
By Invariant 7.9.1, $k < p \leq \mathit{length}(g)$. So by Lemma 7.6.8, $\mathit{member}(d, i, \mathit{delete}(k, g)) \Rightarrow \mathit{member}(d, i, g) \Rightarrow i < m$.
E: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
Since $0 < \mathit{length}(g)$, by Lemmas 7.6.7 and 7.6.8, $\mathit{member}(d, i, \mathit{delete\text{-}last}(g)) \Rightarrow \mathit{member}(d, i, g) \Rightarrow i < m$.
9. $\mathit{test}(i, q') \Rightarrow i < m$.
Summands A, E and G need to be checked.
A: $m := S(m)$;
$\mathit{test}(i, q') \Rightarrow i < m < S(m)$.
E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')$; under condition $p < \mathit{length}(g)$;
Since $0 < \mathit{length}(g)$, by Lemmas 7.6.5, 7.6.6 and 7.6.10, $\mathit{member}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), g)$. By Invariant

7.9.8, $\mathit{last\text{-}seq}(g) < m$. Hence,

$$\begin{array}{ll} & \mathit{test}(i, \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) \\ \Leftrightarrow & (i = \mathit{last\text{-}seq}(g) \vee \mathit{test}(i, q')) \\ \Rightarrow & (i = \mathit{last\text{-}seq}(g) \vee i < m) \\ \Leftrightarrow & i < m \end{array}$$

G: $q' := \mathit{remove}(\ell', q')$;
$\mathit{test}(i, \mathit{remove}(\ell', q')) \Rightarrow \mathit{test}(i, q')$ (Lem. 7.2.1) $\Rightarrow i < m$.

10. $\mathit{test}(i, q') \Rightarrow \ell' \leq i < \ell' + n$.
Summands E and G need to be checked.
E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')$; under condition $\ell' \leq \mathit{last\text{-}seq}(g) < \ell' + n$;

$$\begin{array}{ll} & \mathit{test}(i, \mathit{inb}(\mathit{last\text{-}dat}(g), \mathit{last\text{-}seq}(g), q')) \\ \Leftrightarrow & (i = \mathit{last\text{-}seq}(g) \vee \mathit{test}(i, q')) \\ \Rightarrow & (i = \mathit{last\text{-}seq}(g) \vee \ell' \leq i < \ell' + n) \\ \Leftrightarrow & \ell' \leq i < \ell' + n \end{array}$$

G: $\ell' := S(\ell')$, $q' := \mathit{remove}(\ell', q')$;

$$\begin{array}{lll} & \mathit{test}(i, \mathit{remove}(\ell', q')) & \\ \Leftrightarrow & (\mathit{test}(i, q') \wedge i \neq \ell') & \text{(Lem. 7.2.1)} \\ \Rightarrow & (\ell' \leq i < \ell' + n \wedge i \neq \ell') & \\ \Rightarrow & S(\ell') \leq i < S(\ell') + n & \end{array}$$

11. $\ell' \leq m$.
Summands A and G need to be checked.
A: $m := S(m)$;
$\ell' \leq m < S(m)$.
G: $\ell' := S(\ell')$; under condition $\mathit{test}(\ell', q')$;
By Invariant 7.9.9, $\mathit{test}(\ell', q') \Rightarrow \ell' < m$. Hence, $S(\ell') \leq m$.

12. $\mathit{next\text{-}empty}(\ell', q') \leq m$.
By Invariant 7.9.11, $\ell' \leq m$. By Invariant 7.9.9, $\neg \mathit{test}(m, q')$. Hence, by Lemma 7.5.2, $\mathit{next\text{-}empty}(\ell', q') \leq m$.

13. $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$.
By Invariant 7.9.10, $\neg \mathit{test}(\ell' + n, q')$. Hence, by Lemma 7.5.2, $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$.

14. $\mathit{test}(i, q) \Rightarrow \ell \leq i$.
Summands A and K need to be checked.
A: $q := \mathit{inb}(d, m, q)$;
By Invariants 7.9.4 and 7.9.12, $\ell \leq m$. Hence,

$$\begin{array}{ll} & \mathit{test}(i, \mathit{inb}(d, m, q)) \\ \Leftrightarrow & (i = m \vee \mathit{test}(i, q)) \\ \Rightarrow & (i = m \vee \ell \leq i) \\ \Leftrightarrow & \ell \leq i \end{array}$$

K: $\ell := \mathit{last\text{-}seq}(g')$, $q := \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)$;

$$\begin{array}{lll} & \mathit{test}(i, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)) & \\ \Leftrightarrow & (\mathit{test}(i, q) \wedge \neg(\ell \leq i < \mathit{last\text{-}seq}(g'))) & \text{(Lem. 7.2.3)} \\ \Rightarrow & (\ell \leq i \wedge \neg(\ell \leq i < \mathit{last\text{-}seq}(g'))) & \\ \Rightarrow & \mathit{last\text{-}seq}(g') \leq i & \end{array}$$

15. $\ell \leq i < m \Rightarrow \mathit{test}(i, q)$.
Summands A and K need to be checked.

A: $m := S(m)$, $q := \mathit{inb}(d, m, q)$;

$$\begin{array}{ll} & \ell \leq i < S(m) \\ \Rightarrow & (i = m \vee \ell \leq i < m) \\ \Rightarrow & (i = m \vee \mathit{test}(i, q)) \\ \Leftrightarrow & \mathit{test}(i, \mathit{inb}(d, m, q)) \end{array}$$

K: $\ell := \mathit{last\text{-}seq}(g')$, $q := \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)$; under condition $p' < \mathit{length}(g')$;
Since $0 < \mathit{length}(g')$, by Lemmas 7.7.4 and 7.7.8, $\mathit{member}(\mathit{last\text{-}seq}(g'), g')$. Then by Invariant 7.9.6, $\ell \leq \mathit{last\text{-}seq}(g')$. Hence,

$$\begin{array}{lll} & \mathit{last\text{-}seq}(g') \leq i < m & \\ \Leftrightarrow & (\ell \leq i < m \wedge \neg(\ell \leq i < \mathit{last\text{-}seq}(g'))) & \\ \Rightarrow & (\mathit{test}(i, q) \wedge \neg(\ell \leq i < \mathit{last\text{-}seq}(g'))) & \\ \Leftrightarrow & \mathit{test}(i, \mathit{release}(\ell, \mathit{last\text{-}seq}(g'), q)) & \text{(Lem. 7.2.3)} \end{array}$$

16. $m \leq \ell + n$.
Summands A and K need to be checked.
A: $m := S(m)$; under condition $m < \ell + n$;
Then $S(m) \leq \ell + n$.
K: $\ell := \mathit{last\text{-}seq}(g')$; under condition $p' < \mathit{length}(g')$;
Since $0 < \mathit{length}(g')$, by Lemmas 7.7.4 and 7.7.8, $\mathit{member}(\mathit{last\text{-}seq}(g'), g')$. Then by Invariant 7.9.6, $\ell \leq \mathit{last\text{-}seq}(g')$. Hence, $m \leq \ell + n \leq \mathit{last\text{-}seq}(g') + n$.

17. $i \leq j < \mathit{length}(g) \;\Rightarrow\; \mathit{return\text{-}seq}(i, g) + n > \mathit{return\text{-}seq}(j, g)$.
Summands B, C, E and F need to be checked. F is the same as E.
B: $g := \mathit{inm}(\mathit{retrieve}(k, q), k, g)$; under condition $\mathit{test}(k, q)$;
Case 1: $i > 0$. Let $i \leq j < S(\mathit{length}(g))$.

$$\begin{array}{ll} & \mathit{return\text{-}seq}(j, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) \\ = & \mathit{return\text{-}seq}(j \dot{-} 1, g) \\ < & \mathit{return\text{-}seq}(i \dot{-} 1, g) + n \\ = & \mathit{return\text{-}seq}(i, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) + n \end{array}$$

Case 2: $i = 0$.
The case $j = 0$ is trivial. So suppose that $j > 0$. Lemma 7.6.10 yields
$\mathit{member}(\mathit{return\text{-}dat}(j \dot{-} 1, g), \mathit{return\text{-}seq}(j \dot{-} 1, g), g)$. By Invariant 7.9.8, $\mathit{return\text{-}seq}(j \dot{-} 1, g) < m$. By Invariant 7.9.14, $\mathit{test}(k, q) \Rightarrow \ell \leq k$.

$$\begin{array}{lll} & \mathit{return\text{-}seq}(j, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) & \\ = & \mathit{return\text{-}seq}(j \dot{-} 1, g) & \\ < & m & \\ \leq & \ell + n & \text{(Lem. 7.9.16)} \\ \leq & k + n & \\ = & \mathit{return\text{-}seq}(i, \mathit{inm}(\mathit{retrieve}(k, q), k, g)) + n & \text{(because } i = 0\text{)} \end{array}$$

C: $g := \mathit{delete}(k, g)$; under condition $k < p$;
Let $i \leq j < \mathit{length}(\mathit{delete}(k, g))$. By Invariant 7.9.1, $k < p \leq \mathit{length}(g)$. By Lemma 7.6.9, $\mathit{length}(\mathit{delete}(k, g)) = \mathit{length}(g) \dot{-} 1$.
Case 1: $k \leq i$.
Since $S(i) \leq S(j) < \mathit{length}(g)$, by Lemma 7.6.11,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete}(k, g)) + n \\ = & \mathit{return\text{-}seq}(S(i), g) + n \\ > & \mathit{return\text{-}seq}(S(j), g) \\ = & \mathit{return\text{-}seq}(j, \mathit{delete}(k, g)) \end{array}$$

Case 2: $i < k \leq j$.

Since $i < S(j) < \mathit{length}(g)$, by Lemma 7.6.11,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete}(k,g)) + n \\ = & \mathit{return\text{-}seq}(i,g) + n \\ > & \mathit{return\text{-}seq}(S(j),g) \\ = & \mathit{return\text{-}seq}(j, \mathit{delete}(k,g)) \end{array}$$

Case 3: $j < k$.
Since $i \leq j < \mathit{length}(g)$, by Lemma 7.6.11,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete}(k,g)) + n \\ = & \mathit{return\text{-}seq}(i,g) + n \\ > & \mathit{return\text{-}seq}(j,g) \\ = & \mathit{return\text{-}seq}(j, \mathit{delete}(k,g)) \end{array}$$

E: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
Let $i \leq j < \mathit{length}(\mathit{delete\text{-}last}(g))$. Since $0 < \mathit{length}(g)$, by Lemmas 7.6.6 and 7.6.9, $\mathit{length}(\mathit{delete\text{-}last}(g)) = \mathit{length}(g) \dot{-} 1$. Since $i \leq j < \mathit{length}(g)$, by Lemma 7.6.11,

$$\begin{array}{ll} & \mathit{return\text{-}seq}(i, \mathit{delete\text{-}last}(g)) + n \\ = & \mathit{return\text{-}seq}(i,g) + n \\ > & \mathit{return\text{-}seq}(j,g) \\ = & \mathit{return\text{-}seq}(j, \mathit{delete\text{-}last}(g)) \end{array}$$

18. $(\mathit{member}(d,i,g) \wedge \mathit{test}(j,q')) \;\Rightarrow\; i+n > j$.
Summands B, C, E, F and G need to be checked.
B: $g := \mathit{inm}(\mathit{retrieve}(k,q),k,g)$; under condition $\mathit{test}(k,q)$;
Let $\mathit{member}(d,i,\mathit{inm}(\mathit{retrieve}(k,q),k,g))$ and $\mathit{test}(j,q')$.
Case 1: $i = k$.
By Invariant 7.9.14, $\mathit{test}(k,q)$ yields $\ell \leq k$, and by Invariant 7.9.9, $\mathit{test}(j,q')$ yields $j < m$. Hence, $k+n \geq \ell+n \geq m$ (Inv. 7.9.16) $> j$.
Case 2: $i \neq k$.
$\mathit{member}(d,i,\mathit{inm}(\mathit{retrieve}(k,q),k,g)) = \mathit{member}(d,i,g)$. Hence, $i+n > j$.
C: $g := \mathit{delete}(k,g)$; under condition $k < p$;
Let $\mathit{member}(d,i,\mathit{delete}(k,g))$ and $\mathit{test}(j,q')$. By Invariant 7.9.1, $k < p \leq \mathit{length}(g)$. So by Lemma 7.6.8, $\mathit{member}(d,i,\mathit{delete}(k,g)) \Rightarrow \mathit{member}(d,i,g)$. Hence, $i+n > j$.
E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q')$, $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$ and $\ell' \leq \mathit{last\text{-}seq}(g) < \ell'+n$.
Let $\mathit{member}(d,i,\mathit{delete\text{-}last}(g))$ and $\mathit{test}(j,\mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q'))$. Since $0 < \mathit{length}(g)$, by Lemmas 7.6.7 and 7.6.8, $\mathit{member}(d,i,\mathit{delete\text{-}last}(g)) \Rightarrow \mathit{member}(d,i,g)$.
Case 1: $j = \mathit{last\text{-}seq}(g)$.
The case $i = \mathit{last\text{-}seq}(g)$. So suppose that $i \neq \mathit{last\text{-}seq}(g)$. Since $0 < \mathit{length}(g)$, by Lemma 7.6.6, $\mathit{last\text{-}seq}(g) = \mathit{return\text{-}seq}(\mathit{length}(g) \dot{-} 1, g)$. Since $\mathit{member}(d,i,g)$, by Lemma 7.6.12, there exists a $k < \mathit{length}(g)$ such that $\mathit{return\text{-}seq}(k,g) = i$. By Invariant 7.9.17, $i+n > \mathit{return\text{-}seq}(\mathit{length}(g) \dot{-} 1, g) = \mathit{last\text{-}seq}(g)$.
Case 2: $j \neq \mathit{last\text{-}seq}(g)$.
$\mathit{test}(j,\mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q')) = \mathit{test}(j,q')$. Hence, $i+n > j$.
F: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
Let $\mathit{member}(d,i,\mathit{delete\text{-}last}(g))$ and $\mathit{test}(j,q')$. Since $0 < \mathit{length}(g)$, by Lemmas 7.6.7 and 7.6.8, $\mathit{member}(d,i,\mathit{delete\text{-}last}(g)) \Rightarrow \mathit{member}(d,i,g)$. Hence, $i+n > j$.
G: $q' := \mathit{remove}(\ell',q')$;
Let $\mathit{member}(d,i,g)$ and $\mathit{test}(j,\mathit{remove}(\ell',q'))$. By Lemma 7.2.1, $\mathit{test}(j,\mathit{remove}(\ell',q')) \Rightarrow \mathit{test}(j,q')$. Hence, $i+n > j$.

19. $\mathit{member}(d,i,g) \;\Rightarrow\; i+n \geq \ell'$.
Summands B, C, E, F and G need to be checked. F is the same as E.
B: $g := \mathit{inm}(\mathit{retrieve}(k,q),k,g)$; under condition $\mathit{test}(k,q)$;
Let $\mathit{member}(d,i,\mathit{inm}(\mathit{retrieve}(k,q),k,g))$.

CASE 1: $i = k$.
By Invariant 7.9.14, $\mathit{test}(k,q)$ yields $\ell \leq k$. Hence, $k+n \geq \ell+n \geq m$ (Inv. 7.9.16) $\geq \ell'$ (Inv. 7.9.11).
CASE 2: $i \neq k$.
$\mathit{member}(d,i,\mathit{inm}(\mathit{retrieve}(k,q),k,g)) = \mathit{member}(d,i,g) \Rightarrow i+n \geq \ell'$.
C: $g := \mathit{delete}(k,g)$; under condition $k < p$;
Let $\mathit{member}(d,i,\mathit{delete}(k,g))$. By Invariant 7.9.1, $k < p \leq \mathit{length}(g)$. By Lemma 7.6.8,
$\mathit{member}(d,i,\mathit{delete}(k,g)) \Rightarrow \mathit{member}(d,i,g) \Rightarrow i+n \geq \ell'$.
E: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
Let $\mathit{member}(d,i,\mathit{delete\text{-}last}(g))$. Since $0 < \mathit{length}(g)$, by Lemmas 7.6.7 and 7.6.8,
$\mathit{member}(d,i,\mathit{delete\text{-}last}(g)) \Rightarrow \mathit{member}(d,i,g) \Rightarrow i+n \geq \ell'$.
G: $\ell' = S(\ell')$; under condition $\mathit{test}(\ell',q')$;
Let $\mathit{member}(d,i,g)$. By Invariant 7.9.18, $\mathit{test}(\ell',q')$ implies $i+n > \ell'$. Hence, $i+n \geq S(\ell')$.

20. $\mathit{member}(d,i,g) \;\Rightarrow\; i+n \geq \mathit{next\text{-}empty}(\ell',q')$.
We distinguish two cases.
CASE 1: $q' = []$.
Then $\mathit{next\text{-}empty}(\ell',q') = \ell'$. By Invariant 7.9.19, $\mathit{member}(d,i,g) \Rightarrow i+n \geq \ell'$.
CASE 2: $q' \neq []$.
By Lemma 7.2.5, $\mathit{test}(\mathit{max}(q'),q')$. So Invariant 7.9.18 yields $\mathit{member}(d,i,g) \Rightarrow i+n > \mathit{max}(q')$. By Lemmas 7.5.1 and 7.5.2, $\mathit{next\text{-}empty}(\ell',q') \leq S(\mathit{max}(q'))$. Hence, $\mathit{member}(d,i,g) \Rightarrow i+n \geq \mathit{next\text{-}empty}(\ell',q')$.

21. $(\mathit{member}(d,i,g) \wedge \mathit{test}(i,q)) \;\Rightarrow\; \mathit{retrieve}(i,q) = d$.
Summands A, B, C, E, F and K need to be checked. F is the same as E.
A: $q := \mathit{inb}(e,m,q)$;
By Invariant 7.9.8, $\mathit{member}(d,i,g) \Rightarrow i < m$. So $\mathit{retrieve}(i,\mathit{inb}(e,m,q)) = \mathit{retrieve}(i,q) = d$.
B: $g := \mathit{inm}(\mathit{retrieve}(k,q),k,g)$;
Let $\mathit{member}(d,i,\mathit{inm}(\mathit{retrieve}(k,q),k,g))$ and $\mathit{test}(i,q)$. The case $d = \mathit{retrieve}(k,q) \wedge i = k$ is trivial. And otherwise, $\mathit{member}(d,i,\mathit{inm}(\mathit{retrieve}(k,q),k,g)) = \mathit{member}(d,i,g)$, so since $\mathit{test}(i,q)$, $\mathit{retrieve}(i,q) = d$.
C: $g := \mathit{delete}(k,g)$; under condition $k < p$;
Let $\mathit{member}(d,i,\mathit{delete}(k,g))$ and $\mathit{test}(i,q)$. By Invariant 7.9.1, $k < p \leq \mathit{length}(g)$. Then by Lemma 7.6.8, $\mathit{member}(d,i,\mathit{delete}(k,g)) \Rightarrow \mathit{member}(d,i,g)$. Since $\mathit{test}(i,q)$, $\mathit{retrieve}(i,q) = d$.
E: $g := \mathit{delete\text{-}last}(g)$; under condition $p < \mathit{length}(g)$;
Let $\mathit{member}(d,i,\mathit{delete\text{-}last}(g))$ and $\mathit{test}(i,q)$. Since $0 < \mathit{length}(g)$, by Lemmas 7.6.7 and 7.6.8, $\mathit{member}(d,i,\mathit{delete\text{-}last}(g)) \Rightarrow \mathit{member}(d,i,g)$. Since $\mathit{test}(i,q)$, $\mathit{retrieve}(i,q) = d$.
K: $q := \mathit{release}(\ell,\mathit{last\text{-}seq}(g'),q)$;
Let $\mathit{member}(d,i,\mathit{delete\text{-}last}(g))$ and $\mathit{test}(i,\mathit{release}(\ell,\mathit{last\text{-}seq}(g'),q))$. By Lemma 7.2.3, $\mathit{test}(i,q)$ and $\neg(\ell \leq i < \mathit{last\text{-}seq}(g'))$. By Lemma 7.2.4, $\mathit{retrieve}(i,\mathit{release}(\ell,\mathit{last\text{-}seq}(g'),q)) = \mathit{retrieve}(i,q) = d$.

22. $(\mathit{test}(i,q) \wedge \mathit{test}(i,q')) \;\Rightarrow\; \mathit{retrieve}(i,q) = \mathit{retrieve}(i,q')$.
Summands A, E, G and K must be checked.
A: $q := \mathit{inb}(d,m,q)$;
By Invariant 7.9.9, $\mathit{test}(i,q')$ implies $i \neq m$. So

$$\begin{array}{ll} & \mathit{test}(i,\mathit{inb}(d,m,q)) \wedge \mathit{test}(i,q') \\ \Leftrightarrow & \mathit{test}(i,q) \wedge \mathit{test}(i,q') \\ \Rightarrow & \mathit{retrieve}(i,\mathit{inb}(d,m,q)) = \mathit{retrieve}(i,q) = \mathit{retrieve}(i,q') \end{array}$$

E: $q' := \mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q')$; under condition $p < \mathit{length}(g)$;
Let $\mathit{test}(i,q)$ and $\mathit{test}(i,\mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q'))$.
CASE 1: $i \neq \mathit{last\text{-}seq}(g)$.

$$\begin{array}{ll} & \mathit{test}(i,q) \wedge \mathit{test}(i,\mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q')) \\ \Rightarrow & \mathit{test}(i,q) \wedge \mathit{test}(i,q') \\ \Rightarrow & \mathit{retrieve}(i,q) = \mathit{retrieve}(i,q') = \mathit{retrieve}(i,\mathit{inb}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),q')) \end{array}$$

CASE 2: $i = \mathit{last\text{-}seq}(g)$.
Since $0 < \mathit{length}(g)$, by Lemmas 7.6.5, 7.6.6 and 7.6.10, $\mathit{member}(\mathit{last\text{-}dat}(g),\mathit{last\text{-}seq}(g),g)$. Since

test(*last-seq*(*g*), *q*),

$$\begin{array}{ll} & retrieve(last\text{-}seq(g), q) \\ = & last\text{-}dat(g) \qquad\qquad\qquad\qquad \text{(Inv. 7.9.21)} \\ = & retrieve(last\text{-}dat(g), inb(last\text{-}dat(g), last\text{-}seq(g), q')) \end{array}$$

G: $q' := remove(\ell', q')$;

$$\begin{array}{lll} & test(i,q) \wedge test(i, remove(\ell', q')) & \\ \Leftrightarrow & test(i,q) \wedge test(i,q') \wedge i \neq \ell' & \text{(Lem. 7.2.1)} \\ \Rightarrow & retrieve(i,q) = retrieve(i,q') & \\ & = retrieve(i, remove(\ell', q')) & \text{(Lem. 7.2.2)} \end{array}$$

K: $q := release(\ell, last\text{-}seq(g'), q)$;

$$\begin{array}{lll} & test(i, release(\ell, last\text{-}seq(g'), q)) \wedge test(i,q') & \\ \Leftrightarrow & test(i,q) \wedge test(i,q') \wedge \neg(\ell \leq i < last\text{-}seq(g')) & \text{(Lem. 7.2.3)} \\ \Rightarrow & retrieve(i,q') = retrieve(i,q) & \\ & = retrieve(i, release(\ell, h', q)) & \text{(Lem. 7.2.4)} \end{array}$$

23. $(member(d,i,g) \wedge member(e,i,g)) \;\Rightarrow\; d = e$.
Summands B, C, E and F need to be checked. F is the same as E.
B: $g := inm(retrieve(k,q), k, g)$; under condition $test(k,q)$;
Let $member(d, i, inm(retrieve(k,q), k, g))$ and $member(e, i, inm(retrieve(k,q), k, g))$.
Case 1: $i = k$.
By Invariant 7.9.21, $test(k,q)$ implies $d = retrieve(k,q) = e$.
Case 2: $i \neq k$.
$member(d, i, inm(retrieve(k,q), k, g)) \Rightarrow member(d,i,g)$ and
$member(e, i, inm(retrieve(k,q), k, g)) \Rightarrow member(e,i,g)$. Hence, $d = e$.
C: $g := delete(k,g)$; under condition $k < p$;
By Invariant 7.9.1, $k < p \leq length(g)$. By Lemma 7.6.8,

$$\begin{array}{ll} & member(d,i,delete(k,g)) \wedge member(e,i,delete(k,g)) \\ \Rightarrow & member(d,i,g) \wedge member(e,i,g) \\ \Rightarrow & d = e \end{array}$$

E: $g := delete\text{-}last(g)$; under condition $p < length(g)$;
Since $0 < length(g)$, by Lemmas 7.6.7 and 7.6.8,

$$\begin{array}{ll} & member(d,i,delete\text{-}last(g)) \wedge member(e,i,delete\text{-}last(g)) \\ \Rightarrow & member(d,i,g) \wedge member(e,i,g) \\ \Rightarrow & d = e \end{array}$$

24. $(member(d,i,g) \wedge test(i,q')) \;\Rightarrow\; retrieve(i,q') = d$.
Summands B, C, E, F and G need to be checked.
B: $g := inm(retrieve(k,q), k, g)$; under condition $test(k,q)$;
Let $member(d, i, inm(retrieve(k,q), k, g))$ and $test(i,q')$.
Case 1: $d = retrieve(k,q)$ and $i = k$.
Since $test(k,q)$ and $test(k,q')$, by Invariant 7.9.22, $retrieve(k,q') = d = retrieve(k,q)$.
Case 2: Otherwise, $member(d, i, inm(retrieve(k,q), k, g)) = member(d,i,g)$.
Since $test(i,q')$, $retrieve(i,q') = d$.
C: $g := delete(k,g)$; under condition $k < p$;
Let $member(d,i,delete(k,g))$ and $test(i,q')$. By Invariant 7.9.1, $k < p \leq length(g)$. By Lemma 7.6.8, $member(d,i,delete(k,g)) \Rightarrow member(d,i,g)$. Since $test(i,q')$, $retrieve(i,q') = d$.
E: $q' := inb(last\text{-}dat(g), last\text{-}seq(g), q')$, $g := delete\text{-}last(g)$; under condition $p < length(g)$;
Let $member(d,i,delete\text{-}last(g))$ and $test(i, inb(last\text{-}dat(g), last\text{-}seq(g), q'))$. Since $0 < length(g)$, by Lemmas 7.6.7 and 7.6.8, $member(d,i,delete\text{-}last(g)) \Rightarrow member(d,i,g)$.
Case 1: $i = last\text{-}seq(g)$.
Since $0 < length(g)$, by Lemmas 7.6.5, 7.6.6 and 7.6.10, $member(last\text{-}dat(g), last\text{-}seq(g), g)$. Since

$member(d, last\text{-}seq(g), delete\text{-}last(g))$, by Invariant 7.9.23,
$d = last\text{-}dat(g) = retrieve(last\text{-}seq(g), inb(last\text{-}dat(g), last\text{-}seq(g), q'))$.
CASE 2: $i \neq last\text{-}seq(g)$.
Then $test(i, inb(last\text{-}dat(g), last\text{-}seq(g), q')) \Rightarrow test(i, q')$. Since $member(d, i, g)$, $retrieve(i, q') = d$.
F: $g := delete\text{-}last(g)$; under condition $p < length(g)$;
Let $member(d, i, delete\text{-}last(g))$ and $test(i, q')$. Since $0 < length(g)$, by Lemmas 7.6.7 and 7.6.8,
$member(d, i, delete\text{-}last(g)) \Rightarrow member(d, i, g)$. Since $test(i, q')$, $retrieve(i, q') = d$
G: $q' := remove(\ell', q')$;
By Lemma 7.2.1, $test(i, remove(\ell', q'))$ implies $test(i, q')$ and $i \neq \ell'$. Hence,
$member(d, i, g) \Rightarrow retrieve(i, remove(\ell', q')) = retrieve(i, q')$ (Lem. 7.2.2)$= d$.

25. $(\ell \leq i \leq m \wedge j \leq next\text{-}empty(i, q')) \;\Rightarrow\; q[i..j\rangle = q'[i..j\rangle$.
Let $\ell \leq i \leq m$ and $j \leq next\text{-}empty(i, q')$). We apply induction on $j \dot{-} i$.
If $i \geq j$, then $q[i..j\rangle = \langle\rangle = q'[i..j\rangle$.
Let $i < j$.
CASE 1: $i = m$.
By Invariant 7.9.9, $j \leq next\text{-}empty(i, q') = m$. Hence, $q[i..j\rangle = \langle\rangle = q'[i..j\rangle$.
CASE 2: $\ell \leq i < m$.
Then by Invariant 7.9.15, $test(i, q)$. Furthermore, by Lemma 7.5.2, $i < j \leq next\text{-}empty(i, q')$ implies $test(i, q')$. Hence,

$$\begin{array}{cll} & q[i..j\rangle & \\ = & inb(retrieve(i, q), q[S(i)..j\rangle) & \\ = & inb(retrieve(i, q), q'[S(i)..j\rangle) & \text{(by induction)} \\ = & inb(retrieve(i, q'), q'[S(i)..j\rangle) & \text{(Inv. 7.9.22)} \\ = & q'[i..j\rangle. & \end{array}$$

□