Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Verifying a Sliding Window Protocol in mCRL

W.J. Fokkink, J.F. Groote, J. Pang, B. Badban,
J.C. van de Pol

REPORT SEN-R0308 September 30, 2003

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Verifying a Sliding Window Protocol in μCRL

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

ABSTRACT

We prove the correctness of a sliding window protocol with an arbitrary finite window size n and sequence numbers modulo $2n$. The correctness consists of showing that the sliding window protocol is branching bisimilar to a queue of capacity $2n$. The proof is given entirely on the basis of an axiomatic theory.

2000 Mathematics Subject Classification: 68N30 [Specification and verification]; 68Q85 [Process algebras and bisimulation]
1998 ACM Computing Classification System: D.2.1 [Requirements/Specifications]; C.2.2 [Protocol verification]; F.3.1[Specifying, verifying and reasoning about programs]
Keywords and Phrases: μCRL, branching bisimulation, process algebra, sliding window protocols, specification, verification techniques
Note: This research is partly supported by the Dutch Technology Foundation STW under the project CES5008: Improving the quality of embedded systems using formal design and systematic testing.

1. Introduction

Sliding window protocols [9] (SWPs) ensure successful transmission of messages from a sender to a receiver through a lossy channel. The main characteristic of these protocols is that the sender does not wait for an incoming acknowledgement before sending next messages, for optimal use of available 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, which 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 for a nice argument [48]). The messages at the sender are numbered from i to $i+n$ (modulo $2n$); this is called a *window*. When an acknowledgement reaches the sender, indicating that k messages have arrived correctly, the window *slides* forward, so that the sending buffer can contain messages with sequence numbers $i+k$ to $i+k+n$ (modulo $2n$). The window of the receiver slides forward when the first element in this window is passed on to the environment.

Within the process algebraic community, SWPs have attracted much 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 8, and restrict here to the context in which this paper was written. After the advent of process algebra in the early eighties, 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 [35] and Brunekreef [7] gave specifications in ACP [3] and PSF [34], respectively. Vaandrager [49], Groenveld [16], van Wamel [50] and Bezem and Groote [6] 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 [48] (not taking into account additional features such as duplex message passing and piggybacking) correct using μCRL [20], which is a suitable process algebraic formalism for such purposes. This turned out to be unexpectedly hard, which is shown by the 13 year it took to finish the current paper. We therefore consider the current paper as a true milestone in process algebraic verification.

Our first observation was that the external behaviour of the protocol, as given in [48], 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 [48] contained a deadlock [17, 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 [48]. 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 [18], and to the development of the *cones and foci* proof method [23, 13]. 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 [43], 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 channels are lossy queues of capacity one. We manually prove that the external behaviour of this protocol is branching bisimilar [14] to a FIFO queue of capacity $2n$. This proof is entirely based on the axiomatic theory underlying μCRL and the axioms characterising the data types. It implies both safety and liveness of the protocol (the latter under the assumption of fairness). First, we linearise the specification, meaning that we get rid of parallel operators. Moreover, communication actions are stripped from their data parameters. Then we eliminate modulo arithmetic, using the proof principle CL-RSP [5]. Finally, we apply the cones and foci technique, to prove that the linear specification without modulo arithmetic is branching bisimilar to a FIFO queue of capacity $2n$. All lemmas for the data types, all invariants and all correctness proofs have been checked using PVS. The PVS files are available via `http://www.cwi.nl/~badban/vmcai.html`. Ongoing research is to extend the current verification to a setting with channels of unbounded capacity.

A concise overview of other verifications of SWPs is presented in Section 8. 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 introduces the process part of μCRL. In Section 3, the data types needed for specifying the SWP and its external behaviour are presented. Section 4 features the μCRL specifications of the SWP and its external behaviour. In Section 5, three consecutive transformations are applied to the specification of the SWP, to linearise the specification, eliminate arguments of communication actions, and get rid of modulo arithmetic. In Section 6, properties of the data types and invariants of the transformed specification are proved. In Section 7, it is proved that the three transformations preserve branching bisimulation, and that the transformed specification behaves like a FIFO queue. Finally, Section 8 gives an overview of related work on verifying SWPs.

2. μCRL

μCRL [20] (see also [22]) is a language for specifying distributed systems and protocols in an algebraic style. It is based on the process algebra ACP [3] extended with equational abstract data types [32]. In a μCRL specification, one part specifies the data types by means of equations $d = e$, while a second part specifies the process behaviour. We assume the data sort of booleans *Bool* with constants t and f, and the usual connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$ and $\leftrightarrow$. For a boolean b, we abbreviate $b = \mathsf{t}$ to b and $b = \mathsf{f}$ to $\neg b$.

The data types needed for our μCRL specification of a SWP are presented in Section 3. In this section we focus on the process part of μCRL. Processes are represented by process terms, which describe the order in which the actions from a set $\mathcal{A}$ may happen. A process term consists of action names and recursion variables combined by process algebraic operators. Actions and recursion variables may carry data parameters. There are two predefined actions outside $\mathcal{A}$: δ represents deadlock, and τ a hidden action. These two actions never carry data parameters. $p{\cdot}q$ denotes sequential composition and $p + q$ non-deterministic choice. Summation $\sum_{d:D} p(d)$ provides the possibly infinite choice over a data type D, and the conditional construct $p \lhd b \rhd q$ with b a data term of sort *Bool* behaves as p if b and as q if $\neg b$. Parallel composition $p \parallel q$ interleaves the actions of p and q; moreover, actions from p and q may also synchronise to a communication action, when this is explicitly allowed by a predefined communication function. Two actions can only synchronise if their data parameters are equal. Encapsulation $\partial_{\mathcal{H}}(p)$, which renames all occurrences in p of actions from the set $\mathcal{H}$ into δ, can be used to force actions into communication. Hiding $\tau_{\mathcal{I}}(p)$ renames all occurrences in p of actions from the set $\mathcal{I}$ into τ. Finally, processes can be specified by means of recursive equations

$$X(d_1{:}D_1, \ldots, d_n{:}D_n) \approx p$$

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

$$X(d_1{:}D_1, \ldots, d_n{:}D_n) \approx \sum_{i=1}^{\ell} \sum_{z_i:Z_i} a_i(e^i_1, \ldots, e^i_{m_i}){\cdot}X(d^i_1, \ldots, d^i_n) \lhd b_i \rhd \delta.$$

To each μCRL specification belongs a directed graph, called a labelled transition system, which is defined by the structural operational semantics of μCRL (see [20]). In this labelled transition system, the states are process terms, and the edges are labelled with parameterised actions. Branching bisimulation $\underline{\leftrightarrow}_b$ [14] and strong bisimulation $\underline{\leftrightarrow}$ [39] are two well-established equivalence relations on the states in labelled transition systems. Conveniently, strong bisimulation equivalence implies branching bisimulation equivalence. The proof theory of μCRL from [19] is sound modulo branching bisimulation equivalence, meaning that if $p \approx q$ can be derived from it then $p \underline{\leftrightarrow}_b q$.

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

- *Sum elimination* [18] states that a summation over a data type from which only one element can be selected can be removed. To be more precise,

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

- *CL-RSP* [5] states that the solutions of a linear μCRL specification that does not contain any infinite τ sequence are all strongly bisimilar. This proof principle basically extends RSP [4] to a setting with data. The reader is referred to [5] for more details regarding CL-RSP.

- The *cones and foci* method from [13, 23] rephrases the question whether two linear μCRL specifications $\tau_{\mathcal{I}}(S_1)$ and S_2 are branching bisimilar, where S_2 does not contain actions from some set $\mathcal{I}$ of internal actions, in terms of data equalities. A *state mapping* ϕ relates each state in S_1 to a state in S_2. Furthermore, some states in S_1 are declared to be *focus points*, by means of a predicate *FC*. The *cone* of a focus point consists of the states in S_1 that can reach this focus point by a string of actions from $\mathcal{I}$. It is required that each reachable state in S_1 is in the cone of a focus point. If a number of *matching criteria* are satisfied, then $\tau_{\mathcal{I}}(S_1)$ and S_2 are branching bisimilar. The reader is referred to [13] for the technical details of the cones and foci technique.

3. Data Types

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

3.1 Booleans

We introduce the data type *Bool* of booleans.

$$\begin{array}{l} \mathsf{t}, \mathsf{f} :\rightarrow \mathit{Bool} \\ \wedge, \vee : \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} \\ \neg : \mathit{Bool} \rightarrow \mathit{Bool} \\ \rightarrow, \leftrightarrow : \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} \end{array}$$

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

$$\begin{array}{lcllcl} b \wedge \mathsf{t} & = & b & \neg\mathsf{t} & = & \mathsf{f} \\ b \wedge \mathsf{f} & = & \mathsf{f} & \neg\mathsf{f} & = & \mathsf{t} \\ b \vee \mathsf{t} & = & \mathsf{t} & b \rightarrow b' & = & b' \vee \neg b \\ b \vee \mathsf{f} & = & b & b \leftrightarrow b' & = & (b \rightarrow b') \wedge (b' \rightarrow b) \end{array}$$

Unless otherwise stated, data parameters in boolean formulas are universally quantified.

3.2 If-then-else and Equality

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

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

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 $eq : D \times D \rightarrow \mathit{Bool}$ such that $eq(d, e)$ holds if and only if $d = e$ can be derived. For notational convenience we take the liberty to write $d = e$ instead of $eq(d, e)$.

3.3 Natural Numbers

We introduce the data type *Nat* of natural numbers.

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

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

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

We take as binding convention:

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

3.4 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:

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

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

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

3.5 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 modelled as a list of pairs (d, i) with $d \in \Delta$ and $i \in \mathit{Nat}$, representing that position (or sequence number) i of the buffer is occupied by datum d. The data type Buf is specified as follows, where $[]$ denotes the empty buffer:

$$\begin{array}{l}
[] :\rightarrow \mathit{Buf} \\
\mathit{in} : \Delta \times \mathit{Nat} \times \mathit{Buf} \rightarrow \mathit{Buf}
\end{array}$$

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

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

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

[1] Note that $\mathit{retrieve}(i, [])$ is undefined. One could choose to equate it to a default value in Δ, or to a fresh error element in Δ. However, the first approach could cover up flaws in the μCRL specification of the SWP, and the second

position i in buffer q.

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

Note that $\mathit{retrieve}(i, [])$ cannot be equated to an element $\mathit{release}(i, j, q)$ is obtained by emptying positions i up to j in q. $\mathit{release}|_n(i, j, q)$ does the same modulo n.

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

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

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

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

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

Finally, we define an operation on buffers that is only needed in the derivation of some data equalities in Section 6.1: $\mathit{max}(q)$ produces the greatest sequence number that is occupied in q.

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

3.6 Lists

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

$$\begin{array}{l} \langle\rangle :\rightarrow \mathit{List} \\ \mathit{in} : \Delta \times \mathit{List} \rightarrow \mathit{List} \end{array}$$

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

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

approach would needlessly complicate the data type Δ. We prefer to work with a partially defined version of *retrieve*, which is allowed in μCRL. All operations in μCRL models, however, are total; partially specified operations just lead to the existence of multiple models.

Furthermore, $q[i..j\rangle$ is the list containing the elements in buffer q at positions i up to but not including j. An empty position in q, in between i and j, gives rise to an occurrence of the default datum d_0 in $q[i..j\rangle$.

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

4. Sliding Window Protocol

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

4.1 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 lossy channel, represented by the medium **K** and the channels B and C. Upon reception, **R** may store the datum in its buffer, where its position in the buffer is dictated by the attached sequence number. In order to avoid a possible overlap between the sequence numbers of different data elements in the buffers of **S** and **R**, no more than one half of the buffers of **S** and **R** may be occupied at any time; these halves are called the sending and the receiving window, respectively. **R** can pass on a datum that resides at the first position in its window via channel D; in that case the receiving window slides forward by one position. Furthermore, **R** can send the sequence number of the first empty position in (or just outside) its window as an acknowledgement to **S** via a lossy channel, represented by the medium **L** and the channels E and F. If **S** receives this acknowledgement, its window slides accordingly.

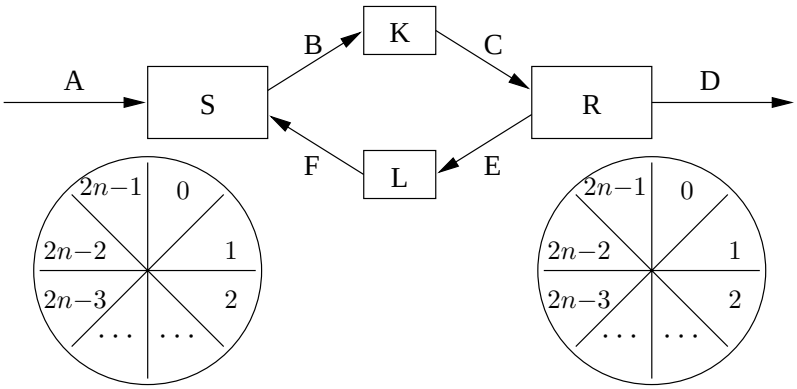

Figure 1: Sliding window protocol

The sender **S** is modelled by the process $\mathbf{S}(\ell, m, q)$, where q is a buffer of size $2n$, ℓ the first position in the sending window, and m the first empty position in (or just outside) the sending window.

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

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

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

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.

Finally, we specify the mediums $\mathbf{K}$ and $\mathbf{L}$, which have capacity one and may lose frames between $\mathbf{S}$ and $\mathbf{R}$, and vice versa. The action j expresses the nondeterministic choice whether or not a frame is lost.

$$\begin{aligned}\mathbf{K} \;&\approx\; \textstyle\sum_{d:\Delta}\sum_{k:Nat} r_{\mathrm{B}}(d,k)\cdot(j\cdot s_{\mathrm{C}}(d,k) + j)\cdot\mathbf{K}\\ \mathbf{L} \;&\approx\; \textstyle\sum_{k:Nat} r_{\mathrm{E}}(k)\cdot(j\cdot s_{\mathrm{F}}(k) + j)\cdot\mathbf{L}\end{aligned}$$

The initial state of the SWP is expressed by

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

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

4.2 External Behaviour

Data elements that are read from channel A by $\mathbf{S}$ should be sent into channel D by $\mathbf{R}$ in the same order, and no data elements should be lost. In other words, the SWP is intended to be a solution for the linear specification

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

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, $s_{\mathrm{D}}(top(\lambda))$ 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 behaviour of our μCRL specification of a SWP corresponds to a FIFO queue of size $2n$.

Theorem 4.1 $\tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K} \parallel \mathbf{L})) \leftrightarrows_b \mathbf{Z}(\langle\rangle)$.

5. 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.

5.1 Linearisation

The starting point of our correctness proof is a linear specification $\mathbf{M}_{mod}$, in which no parallel operators occur. $\mathbf{M}_{mod}$ can be obtained from the μCRL specification of the SWP without the hiding operator, i.e.,

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

by means of a linearisation algorithm presented in [21]. $\mathbf{M}_{mod}$ contains five extra parameters: $e{:}D$ and $g, g', h, h'{:}Nat$. Intuitively, g (resp. g') equals zero when medium K (resp. L) is inactive, equals one when K (resp. L) just received a data packet, and equals two if K (resp. L) decides to pass on this datum. Furthermore, e (resp. h) equals the datum that is being sent from **S** to **R** (resp. the position of this datum in the sending window) while $g \neq 0$, and equals the dummy value d_0 (resp. 0) while $g = 0$. Finally h' equals the first empty position in the receiving window while $g' \neq 0$ and equals 0 while $g' = 0$.

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.

$$
\begin{aligned}
& \mathbf{M}_{mod}(\ell{:}Nat, m{:}Nat, q{:}Buf, \ell'{:}Nat, q'{:}Buf, g{:}Nat, e{:}D, h{:}Nat, g'{:}Nat, h'{:}Nat)) \\
\approx\ & \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d){\cdot}\mathbf{M}_{mod}(m{:=}S(m)|_{2n}, q{:=}in(d,m,q)) \lhd \mathit{in\text{-}window}(\ell, m, (\ell+n)|_{2n}) \rhd \delta \\
+\ & \textstyle\sum_{k:Nat} c_{\mathrm{B}}(\mathit{retrieve}(k,q),k){\cdot}\mathbf{M}_{mod}(g{:=}1, e{:=}\mathit{retrieve}(k,q), h{:=}k) \lhd \mathit{test}(k,q) \wedge g = 0 \rhd \delta \\
+\ & j{\cdot}\mathbf{M}_{mod}(g{:=}0, e{:=}d_0, h{:=}0) \lhd g = 1 \rhd \delta \\
+\ & j{\cdot}\mathbf{M}_{mod}(g{:=}2) \lhd g = 1 \rhd \delta \\
+\ & c_{\mathrm{C}}(e,h){\cdot}\mathbf{M}_{mod}(q'{:=}in(e,h,q'), g{:=}0, e{:=}d_0, h{:=}0) \lhd \mathit{in\text{-}window}(\ell', h, (\ell'+n)|_{2n}) \wedge g = 2 \rhd \delta \\
+\ & c_{\mathrm{C}}(e,h){\cdot}\mathbf{M}_{mod}(g{:=}0, e{:=}d_0, h{:=}0) \lhd \neg\mathit{in\text{-}window}(\ell', h, (\ell'+n)|_{2n}) \wedge g = 2 \rhd \delta \\
+\ & s_{\mathrm{D}}(\mathit{retrieve}(\ell', q')){\cdot}\mathbf{M}_{mod}(\ell'{:=}S(\ell')|_{2n}, q'{:=}\mathit{remove}(\ell', q')) \lhd \mathit{test}(\ell', q') \rhd \delta \\
+\ & c_{\mathrm{E}}(\mathit{next\text{-}empty}|_{2n}(\ell', q')){\cdot}\mathbf{M}_{mod}(g'{:=}1, h'{:=}\mathit{next\text{-}empty}|_{2n}(\ell', q')) \lhd g' = 0 \rhd \delta \\
+\ & j{\cdot}\mathbf{M}_{mod}(g'{:=}0, h'{:=}0) \lhd g' = 1 \rhd \delta \\
+\ & j{\cdot}\mathbf{M}_{mod}(g'{:=}2) \lhd g' = 1 \rhd \delta \\
+\ & c_{\mathrm{F}}(h'){\cdot}\mathbf{M}_{mod}(\ell{:=}h', q{:=}\mathit{release}|_{2n}(\ell, h', q), g'{:=}0, h'{:=}0) \lhd g' = 2 \rhd \delta
\end{aligned}
$$

Theorem 5.1 $\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K} \parallel \mathbf{L}) \leftrightarrows \mathbf{M}_{mod}(0,0,[],0,[],0,d_0,0,0,0)$.

Proof. It is not hard to see that replacing $\mathbf{M}_{mod}(\ell, m, q, \ell', q', g, e, h, g', h')$ by $\partial_{\mathcal{H}}(\mathbf{S}(\ell, m, q) \parallel \mathbf{R}(\ell', q') \parallel \mathbf{K} \parallel \mathbf{L})$ is a solution for the recursive equation above, using the axioms of μCRL [19]. (The details are left to the reader.) Hence, the theorem follows by CL-RSP [5]. ☒

5.2 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.

$$
\begin{aligned}
& \mathbf{N}_{mod}(\ell{:}Nat, m{:}Nat, q{:}Buf, \ell'{:}Nat, q'{:}Buf, g{:}Nat, e{:}D, h{:}Nat, g'{:}Nat, h'{:}Nat)) \\
\approx\ & \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d){\cdot}\mathbf{N}_{mod}(m{:=}S(m)|_{2n}, q{:=}in(d,m,q)) \lhd \mathit{in\text{-}window}(\ell, m, (\ell+n)|_{2n}) \rhd \delta \\
+\ & \textstyle\sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g{:=}1, e{:=}\mathit{retrieve}(k,q), h{:=}k) \lhd \mathit{test}(k,q) \wedge g = 0 \rhd \delta \\
+\ & j{\cdot}\mathbf{N}_{mod}(g{:=}0, e{:=}d_0, h{:=}0) \lhd g = 1 \rhd \delta \\
+\ & j{\cdot}\mathbf{N}_{mod}(g{:=}2) \lhd g = 1 \rhd \delta \\
+\ & c{\cdot}\mathbf{N}_{mod}(q'{:=}in(e,h,q'), g{:=}0, e{:=}d_0, h{:=}0) \lhd \mathit{in\text{-}window}(\ell', h, (\ell'+n)|_{2n}) \wedge g = 2 \rhd \delta \\
+\ & c{\cdot}\mathbf{N}_{mod}(g{:=}0, e{:=}d_0, h{:=}0) \lhd \neg\mathit{in\text{-}window}(\ell', h, (\ell'+n)|_{2n}) \wedge g = 2 \rhd \delta \\
+\ & s_{\mathrm{D}}(\mathit{retrieve}(\ell', q')){\cdot}\mathbf{N}_{mod}(\ell'{:=}S(\ell')|_{2n}, q'{:=}\mathit{remove}(\ell', q')) \lhd \mathit{test}(\ell', q') \rhd \delta
\end{aligned}
$$

$$
\begin{aligned}
&+ \quad c{\cdot}\mathbf{N}_{mod}(g'{:=}1, h'{:=}\mathit{next\text{-}empty}|_{2n}(\ell', q')) \lhd\ g' = 0 \rhd \delta \\
&+ \quad j{\cdot}\mathbf{N}_{mod}(g'{:=}0, h'{:=}0) \lhd\ g' = 1 \rhd \delta \\
&+ \quad j{\cdot}\mathbf{N}_{mod}(g'{:=}2) \lhd\ g' = 1 \rhd \delta \\
&+ \quad c{\cdot}\mathbf{N}_{mod}(\ell{:=}h', q{:=}\mathit{release}|_{2n}(\ell, h', q), g'{:=}0, h'{:=}0) \lhd\ g' = 2 \rhd \delta
\end{aligned}
$$

Theorem 5.2 $\tau_{\mathcal{I}}(\mathbf{M}_{mod}(0, 0, [], 0, [], 0, d_0, 0, 0, 0)) \leftrightarrows \tau_{\{c,j\}}(\mathbf{N}_{mod}(0, 0, [], 0, [], 0, d_0, 0, 0, 0))$.

Proof. We define

$$\tau_{\mathcal{I}}(\mathbf{M}_{mod}(\ell, m, q, \ell', q', g, e, h, g', h'))\ B\ \tau_{\{c,j\}}(\mathbf{N}_{mod}(\ell, m, q, \ell', q', g, e, h, g', h')).$$

It is not hard to see that B is a strong bisimulation relation [39]. ⊠

5.3 Getting Rid of Modulo Arithmetic

$\mathbf{N}_{nonmod}$ is obtained by eliminating all occurrences of $|_{2n}$ from the specification of $\mathbf{N}_{mod}$.

$$
\begin{aligned}
&\mathbf{N}_{nonmod}(\ell{:}\mathit{Nat}, m{:}\mathit{Nat}, q{:}\mathit{Buf}, \ell'{:}\mathit{Nat}, q'{:}\mathit{Buf}, g{:}\mathit{Nat}, e{:}D, h{:}\mathit{Nat}, g'{:}\mathit{Nat}, h'{:}\mathit{Nat}) \\
\approx\ & \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d){\cdot}\mathbf{N}_{nonmod}(m{:=}S(m), q{:=}\mathit{in}(d, m, q)) \lhd\ m < \ell + n \rhd \delta && (A) \\
+\ & \textstyle\sum_{k:Nat} c{\cdot}\mathbf{N}_{nonmod}(g{:=}1, e{:=}\mathit{retrieve}(k, q), h{:=}k) \lhd\ \mathit{test}(k, q) \wedge g = 0 \rhd \delta && (B) \\
+\ & j{\cdot}\mathbf{N}_{nonmod}(g{:=}0, e{:=}d_0, h{:=}0) \lhd\ g = 1 \rhd \delta && (C) \\
+\ & j{\cdot}\mathbf{N}_{nonmod}(g{:=}2) \lhd\ g = 1 \rhd \delta && (D) \\
+\ & c{\cdot}\mathbf{N}_{nonmod}(q'{:=}\mathit{in}(e, h, q'), g{:=}0, e{:=}d_0, h{:=}0) \lhd\ \ell' \leq h < \ell' + n \wedge g = 2 \rhd \delta && (E) \\
+\ & c{\cdot}\mathbf{N}_{nonmod}(g{:=}0, e{:=}d_0, h{:=}0) \lhd \neg(\ell' \leq h < \ell' + n) \wedge g = 2 \rhd \delta && (F) \\
+\ & s_{\mathrm{D}}(\mathit{retrieve}(\ell', q')){\cdot}\mathbf{N}_{nonmod}(\ell'{:=}S(\ell'), q'{:=}\mathit{remove}(\ell', q')) \lhd\ \mathit{test}(\ell', q') \rhd \delta && (G) \\
+\ & c{\cdot}\mathbf{N}_{nonmod}(g'{:=}1, h'{:=}\mathit{next\text{-}empty}(\ell', q')) \lhd\ g' = 0 \rhd \delta && (H) \\
+\ & j{\cdot}\mathbf{N}_{nonmod}(g'{:=}0, h'{:=}0) \lhd\ g' = 1 \rhd \delta && (I) \\
+\ & j{\cdot}\mathbf{N}_{nonmod}(g'{:=}2) \lhd\ g' = 1 \rhd \delta && (J) \\
+\ & c{\cdot}\mathbf{N}_{nonmod}(\ell{:=}h', q{:=}\mathit{release}(\ell, h', q), g'{:=}0, h'{:=}0) \lhd\ g' = 2 \rhd \delta && (K)
\end{aligned}
$$

Theorem 5.3 $\mathbf{N}_{mod}(0, 0, [], 0, [], 0, d_0, 0, 0, 0) \leftrightarrows \mathbf{N}_{nonmod}(0, 0, [], 0, [], 0, d_0, 0, 0, 0)$.

The proof of Theorem 5.3 is presented in Section 7.1. Next, in Section 7.2, we prove the correctness of $\mathbf{N}_{nonmod}$. In these proofs we will need a wide range of data equalities, which we proceed to prove in Section 6.

6. Properties of Data

6.1 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:

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

Lemmas 6.1 and 6.2 collect basic facts on modulo arithmetic and on buffers, respectively. Lemma 6.3 collects some results on modulo arithmetic related to buffers. Lemma 6.4 collects some facts on the

next-empty operation, together with one result on *max*, which is needed to derive those facts. Finally, Lemma 6.5 contains basic facts on lists.

Lemma 6.1 Let $n > 0$.

1. $(i|_n + j)|_n = (i + j)|_n$
2. $i|_n < n$
3. $(i{\cdot}n)|_n = 0$
4. $i = (i \mathit{\ div\ } n){\cdot}n + i|_n$
5. $j \le i \le j+n \ \rightarrow\ (i \mathit{\ div\ } 2n = j \mathit{\ div\ } 2n \wedge j|_{2n} \le i|_{2n} \le j|_{2n}+n) \vee (i \mathit{\ div\ } 2n = S(j \mathit{\ div\ } 2n) \wedge i|_{2n}+n \le j|_{2n})$
6. $i \le j \ \rightarrow\ i \mathit{\ div\ } n \le j \mathit{\ div\ } n$

Proof.

1. By induction on i.
   - $i < n$.
     Then $i|_n = i$.
   - $i \ge n$.

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

2. Trivial, by induction on i.
3. Trivial, by induction on i.
4. By induction on i.
   - $i < n$.
     Then $i \mathit{\ div\ } n = 0$ and $i|_n = i$. Clearly, $i = 0{\cdot}n + i$.
   - $i \ge n$.
     Then $i \mathit{\ div\ } n = S((i \dot{-} n) \mathit{\ div\ } n)$ and $i|_n = (i \dot{-} n)|_n$. Hence,

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

5. Let $j \le i \le j + n$.
   Case 1: $i \mathit{\ div\ } 2n < j \mathit{\ div\ } 2n$.

   $$\begin{array}{lcll} j - i & = & (j \mathit{\ div\ } 2n){\cdot}2n + j|_{2n} - ((i \mathit{\ div\ } 2n){\cdot}2n + i|_{2n}) & \text{(Lem. 6.1.4)} \\ & = & (j \mathit{\ div\ } 2n - i \mathit{\ div\ } 2n){\cdot}2n + (j|_{2n} - i|_{2n}) & \\ & \ge & 2n + (j|_{2n} - i|_{2n}) & (i \mathit{\ div\ } 2n < j \mathit{\ div\ } 2n) \\ & > & 2n - 2n & \text{(Lem. 6.1.2)} \\ & = & 0 & \text{(Contradict with } j \le i) \end{array}$$

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

$$\begin{array}{rcll} j \leq i \leq j+n & = & (j \mathit{div} 2n)\cdot 2n + j|_{2n} \leq (i \mathit{div} 2n)\cdot 2n + i|_{2n} & \\ & & \leq (j \mathit{div} 2n)\cdot 2n + j|_{2n} + n & \text{(Lem. 6.1.4)} \\ & = & j|_{2n} \leq i|_{2n} \leq j|_{2n} + n & (i \mathit{div} 2n = j \mathit{div} 2n) \end{array}$$

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

$$\begin{array}{rcll} i \leq j+n & = & (i \mathit{div} 2n)\cdot 2n + i|_{2n} \leq (j \mathit{div} 2n)\cdot 2n + j|_{2n} + n & \text{(Lem. 6.1.4)} \\ & = & (j \mathit{div} 2n)\cdot 2n + 2n + i|_{2n} \leq (j \mathit{div} 2n)\cdot 2n + j|_{2n} + n & (i \mathit{div} 2n = S(j \mathit{div} 2n)) \\ & = & i|_{2n} + n \leq j|_{2n} & \end{array}$$

CASE 4: $i \mathit{div} 2n > S(j \mathit{div} 2n)$.

$$\begin{array}{rcll} i - (j+n) & = & (i \mathit{div} 2n)\cdot 2n + i|_{2n} - ((j \mathit{div} 2n)\cdot 2n + j|_{2n}) - n & \text{(Lem. 6.1.4)} \\ & \geq & (j \mathit{div} 2n)\cdot 2n + 4n + i|_{2n} - (j \mathit{div} 2n)\cdot 2n - j|_{2n} - n & (i \mathit{div} 2n > S(j \mathit{div} 2n)) \\ & = & 3n + i|_{2n} - j|_{2n} & \\ & > & 3n - 2n & \text{(Lem. 6.1.2)} \\ & > & 0 & \text{(Contradict with } i < j+n) \end{array}$$

6. By induction on i.

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

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

⊠

Lemma 6.2 1. $\neg \mathit{test}(i,q) \rightarrow \mathit{remove}(i,q) = q$

2. $\mathit{test}(i, \mathit{remove}(j,q)) = (\mathit{test}(i,q) \wedge i \neq j)$

3. $i \neq j \rightarrow \mathit{retrieve}(i, \mathit{remove}(j,q)) = \mathit{retrieve}(i,q)$

4. $\mathit{test}(i, \mathit{release}(j,k,q)) = (\mathit{test}(i,q) \wedge \neg(j \leq i < k))$

5. $\neg(j \leq i < k) \rightarrow \mathit{retrieve}(i, \mathit{release}(j,k,q)) = \mathit{retrieve}(i,q)$

Proof.

1. By induction on the structure of q.

   - $q = []$.
     $\mathit{remove}(i, []) = []$.
   - $q = \mathit{in}(d,j,q')$.
     $\neg \mathit{test}(i, \mathit{in}(d,j,q'))$ implies $i \neq j$ and $\neg \mathit{test}(i,q')$.

$$\begin{array}{rcll} \mathit{remove}(i, \mathit{in}(d,j,q')) & = & \mathit{in}(d,j,\mathit{remove}(i,q')) & \\ & = & \mathit{in}(d,j,q') & \text{(by induction)} \end{array}$$

2. By induction on the structure of q.

   - $q = []$.
     $\mathit{test}(i, \mathit{remove}(j, [])) = \mathit{test}(i, []) = \mathsf{f} = \mathit{test}(i, []) \wedge i \neq j$.
   - $q = \mathit{in}(d, k, q')$.
     Case 1: $j = k$.

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

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

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

     Case 2.2: $i \neq k$.

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

3. By induction on the structure of q.

   - $q = []$.
     $\mathit{remove}(j, []) = []$.
   - $q = \mathit{in}(d, k, q')$.
     Case 1: $j = k$.

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

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

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

     Case 2.2: $i \neq k$.

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

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

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

- $j < k$.

$$\begin{array}{lll} & \mathit{test}(i, \mathit{release}(j,k,q)) & \\ = & \mathit{test}(i, \mathit{release}(S(j),k,\mathit{remove}(j,q))) & \\ = & \mathit{test}(i,\mathit{remove}(j,q)) \wedge \neg(S(j) \le i < k) & \text{(by induction)} \\ = & \mathit{test}(i,q) \wedge \neg(j \le i < k) & \text{(Lem. 6.2.2)} \end{array}$$

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

- $j \ge k$.
  Then $\mathit{retrieve}(i, \mathit{release}(j,k,q)) = \mathit{retrieve}(i,q)$.
- $j < k$.
  Then $\neg(j \le i < k)$ implies $i \ne j$. Hence,

$$\begin{array}{lll} & \mathit{retrieve}(i, \mathit{release}(j,k,q)) & \\ = & \mathit{retrieve}(i, \mathit{release}(S(j),k,\mathit{remove}(j,q))) & \\ = & \mathit{retrieve}(i,\mathit{remove}(j,q)) & \text{(by induction)} \\ = & \mathit{retrieve}(i,q) & \text{(Lem. 6.2.3, because } i \ne j) \end{array}$$

⊠

Lemma 6.3 1. $\mathit{test}(k, q|_{2n}) \;\rightarrow\; k = k|_{2n}$

2. $\forall j{:}\mathit{Nat}(\mathit{test}(j,q) \rightarrow i \le j < i+n) \wedge i \le k \le i+n \;\rightarrow\; \mathit{test}(k,q) = \mathit{test}(k|_{2n}, q|_{2n})$

3. $\forall j{:}\mathit{Nat}(\mathit{test}(j,q) \rightarrow i \le j < i+n) \wedge \mathit{test}(k,q) \;\rightarrow\; \mathit{retrieve}(k,q) = \mathit{retrieve}(k|_{2n}, q|_{2n})$

4. $\forall j{:}\mathit{Nat}(\mathit{test}(j,q) \rightarrow i \le j < i+n) \wedge i \le k \le i+n \;\rightarrow\; \mathit{remove}(k,q)|_{2n} = \mathit{remove}(k|_{2n}, q|_{2n})$

5. $\forall j{:}\mathit{Nat}(\mathit{test}(j,q) \rightarrow i \le j < i+n) \wedge i \le k \le i+n \;\rightarrow\; \mathit{release}(i,k,q)|_{2n} = \mathit{release}|_{2n}(i,k,q|_{2n})$

6. $\forall j{:}\mathit{Nat}(\mathit{test}(j,q) \rightarrow i \le j < i+n) \wedge i \le k \le i+n \;\rightarrow$
   $\mathit{next\text{-}empty}(k,q)|_{2n} = \mathit{next\text{-}empty}|_{2n}(k|_{2n}, q|_{2n})$

7. $i \le k < i+n \;\rightarrow\; \mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i+n)|_{2n})$

8. $\mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i+n)|_{2n}) \;\rightarrow\; k+n < i \vee i \le k < i+n \vee k \ge i+2n$

9. $\forall j{:}\mathit{Nat}(\mathit{test}(j,q) \rightarrow i \le j < i+n) \wedge \mathit{test}(k, q|_{2n}) \;\rightarrow\; \mathit{in\text{-}window}(i|_{2n}, k, (i+n)|_{2n})$

Proof.

1. Trivial, by induction on the structure of q, using Lemma 6.1.2.

2. By induction on the structure of q.

- $q = []$.
  $\mathit{test}(k, []) = \mathsf{f} = \mathit{test}(k|_{2n}, []|_{2n})$.
- $q = \mathit{in}(d, \ell, q')$.
  Let $\mathit{test}(j,q) \rightarrow i \le j < i+n$ and $i \le k \le i+n$.
  Case 1: $k|_{2n} = \ell|_{2n}$.
  $\mathit{test}(\ell, q)$, so $i \le \ell < i+n$. In combination with $i \le k \le i+n$, $k|_{2n} = \ell|_{2n}$, Lemmas 6.1.4 and 6.1.5, 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 $k \neq \ell$.
$\mathit{test}(j, q') \to \mathit{test}(j, q) \to i \leq j < i + n$, so induction can be applied with respect to q'.

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

3. By induction on the structure of q.

- $q = []$
  $\mathit{test}(k, []) = \mathsf{f}$.
- $q = \mathit{in}(d, \ell, q')$.
  Let $\mathit{test}(j, q) \to i \leq j < i + n$ 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: $k \neq \ell$.
  $\mathit{test}(j, q') \to \mathit{test}(j, q) \to i \leq j < i + n$, 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 $k \neq \ell$, Lemmas 6.1.4 and 6.1.5, this implies $k|_{2n} \neq \ell|_{2n}$. Hence,

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

4. By induction on the structure of q.

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

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

  Case 2: $k \neq \ell$.
  $\mathit{test}(\ell, q)$, so $i \leq \ell < i + n$. In combination with $i \leq k \leq i + n$, $k \neq \ell$, Lemma 6.1.4 and 6.1.5, this implies $k|_{2n} \neq \ell|_{2n}$. Hence,

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

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

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

- $i < k \leq i + n$.
  By Lemma 6.1.4 and 6.1.5, $i|_{2n} \neq k|_{2n}$. Hence,

$$\begin{array}{lcll} \mathit{release}(i,k,q)|_{2n} & = & \mathit{release}(S(i),k,\mathit{remove}(i,q))|_{2n} & \\ & = & \mathit{release}|_{2n}(S(i),k,\mathit{remove}(i,q)|_{2n}) & \text{(by induction)} \\ & = & \mathit{release}|_{2n}(S(i),k,\mathit{remove}(i|_{2n},q|_{2n})) & \text{(Lem. 6.3.4)} \\ & = & \mathit{release}|_{2n}(i,k,q|_{2n}) & \end{array}$$

6. 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 6.3.2, $\neg\mathit{test}((i+n)|_{2n}, q|_{2n})$. By Lemma 6.1.2, $(i+n)|_{2n} < 2n$. Hence,

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

- $i \leq k \leq i + n$.
  Case 1: $\neg\mathit{test}(k,q)$. By Lemma 6.3.2, also $\neg\mathit{test}(k|_{2n}, q|_{2n})$.
  By Lemma 6.1.2, $k|_{2n} < 2n$. Hence,

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

  Case 2: $\mathit{test}(k,q)$. By Lemma 6.3.2, 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: $k|_{2n} = 2n - 1$.
  By Lemma 6.4.3,

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

  Hence,

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

  Case 2.2: $k|_{2n} < 2n - 1$.
  Using Lemma 6.1.1, we can derive $S(k)|_{2n} = S(k|_{2n})$. Since $\mathit{next\text{-}empty}(k|_{2n}, q|_{2n}) = \mathit{next\text{-}empty}(S(k|_{2n}), q|_{2n}) = \mathit{next\text{-}empty}(S(k)|_{2n}, q|_{2n})$, it follows that

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

  Concluding,

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

7. Let $i \leq k < i + n$.

   Case 1: $S(i \mathit{\ div\ } 2n){\cdot}2n \leq k$.
   Then $S(i \mathit{\ div\ } 2n){\cdot}2n \leq k < i + n < S(i \mathit{\ div\ } 2n){\cdot}2n + n$ (by Lemma 6.1.4), so using Lemmas 6.1.2, 6.1.5 and 6.1.6 it follows that $k \mathit{\ div\ } 2n = (i + n) \mathit{\ div\ } 2n = S(i \mathit{\ div\ } 2n)$. Hence, in view of Lemma 6.1.4, $k|_{2n} < (i + n)|_{2n} < i|_{2n}$.

   Case 2: $k < S(i \mathit{\ div\ } 2n){\cdot}2n \leq i + n$.
   Then $(i \mathit{\ div\ } 2n){\cdot}2n \leq i \leq k < (i \mathit{\ div\ } 2n){\cdot}2n + 2n$, so by Lemma 6.1.6 $k \mathit{\ div\ } 2n = i \mathit{\ div\ } 2n$. Furthermore, $S(i \mathit{\ div\ } 2n){\cdot}2n \leq i + n < S(i \mathit{\ div\ } 2n){\cdot}2n + n$, so $(i + n) \mathit{\ div\ } 2n = S(i \mathit{\ div\ } 2n)$. Hence, $(i + n)|_{2n} < i|_{2n} \leq k|_{2n}$.

   Case 3: $i + n < S(i \mathit{\ div\ } 2n){\cdot}2n$.
   Then $(i \mathit{\ div\ } 2n){\cdot}2n \leq i \leq k < i + n < (i \mathit{\ div\ } 2n){\cdot}2n + 2n$, so by Lemma Lemma 6.1.6 $k \mathit{\ div\ } 2n = (i + n) \mathit{\ div\ } 2n = i \mathit{\ div\ } 2n$. Hence, $i|_{2n} \leq k|_{2n} < (i + n)|_{2n}$.

   By definition,

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

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

8. We prove $i + n \leq k < i + 2n \vee i \leq k + n < i + n \ \rightarrow\ \neg \mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i + n)|_{2n})$.

   - $i + n \leq k < i + 2n$.
     Then $i \mathit{\ div\ } 2n \leq (i + n) \mathit{\ div\ } 2n \leq k \mathit{\ div\ } 2n \leq S(i \mathit{\ div\ } 2n)$. We distinguish three cases, in which we repeatedly apply Lemma 6.1.4.
     Case 1: $i \mathit{\ div\ } 2n = (i + n) \mathit{\ div\ } 2n = k \mathit{\ div\ } 2n$.
     Then $i < i + n$ yields $i|_{2n} < (i + n)|_{2n}$ and $i + n \leq k$ yields $(i + n)|_{2n} \leq k|_{2n}$.
     Case 2: $S(i \mathit{\ div\ } 2n) = S((i + n) \mathit{\ div\ } 2n) = k \mathit{\ div\ } 2n$.
     Then $i < i + n$ yields $i|_{2n} < (i + n)|_{2n}$ and $k < i + 2n$ yields $k|_{2n} < i|_{2n}$.
     Case 3: $S(i \mathit{\ div\ } 2n) = (i + n) \mathit{\ div\ } 2n = k \mathit{\ div\ } 2n$.
     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 $\neg \mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i + n)|_{2n})$.
   - $i \leq k + n < i + n$.
     Then $i+n \leq k+2n < i+2n$, so by case A, $\neg \mathit{in\text{-}window}(i|_{2n}, (k + 2n)|_{2n}, (i + n)|_{2n})$. Hence, $\neg \mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i + n)|_{2n})$.

9. By induction on the structure of q.

   - $q = []$.
     This case follows from the fact that $\mathit{test}(k, []|_{2n}) = \mathsf{f}$.
   - $q = \mathit{in}(d, \ell, q')$.
     Then $\mathit{test}(\ell, q)$, so $i \leq \ell < i + n$. Thus, by Lemma 6.3.7, $\mathit{in\text{-}window}(i|_{2n}, \ell|_{2n}, (i + n)|_{2n})$. Hence,

$$\begin{array}{lcl} \mathit{test}(k, \mathit{in}(d, \ell, q')|_{2n}) & \leftrightarrow & k = \ell|_{2n} \vee \mathit{test}(k, q'|_{2n}) \\ & \rightarrow & k = \ell|_{2n} \vee \mathit{in\text{-}window}(i|_{2n}, k, (i+n)|_{2n}) \\ & \leftrightarrow & \mathit{in\text{-}window}(i|_{2n}, k, (i+n)|_{2n}) \end{array}$$

$\boxtimes$

Lemma 6.4 1. $\mathit{test}(i,q) \;\rightarrow\; i \leq \mathit{max}(q)$

2. $i \leq j \wedge \neg\mathit{test}(j,q) \;\rightarrow\; \mathit{next\text{-}empty}(i,q) \leq j$

3. $\mathit{next\text{-}empty}(i,q) \geq i$

4. $\mathit{next\text{-}empty}(i, \mathit{in}(d,j,q)) \geq \mathit{next\text{-}empty}(i,q)$

5. $j \neq \mathit{next\text{-}empty}(i,q) \;\rightarrow\; \mathit{next\text{-}empty}(i, \mathit{in}(d,j,q)) = \mathit{next\text{-}empty}(i,q)$

6. $\mathit{next\text{-}empty}(i, \mathit{in}(d, \mathit{next\text{-}empty}(i,q), q)) = \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(i,q)), q)$

7. $\neg(i \leq j < \mathit{next\text{-}empty}(i,q)) \;\rightarrow\; \mathit{next\text{-}empty}(i, \mathit{remove}(j,q)) = \mathit{next\text{-}empty}(i,q)$

Proof.

1. By induction on the structure of q.
   - $q = []$.
     $\mathit{test}(i, []) = \mathsf{f}$.
   - $q = \mathit{in}(d,j,q')$.
     Case 1: $i = j$.
     Then clearly $i \leq \mathit{max}(\mathit{in}(d,j,q'))$.
     Case 2: $i \neq j$.
     Then $\mathit{test}(i, \mathit{in}(d,j,q'))$ implies $\mathit{test}(i,q')$, so $i \leq \mathit{max}(q')$ (by induction) $\leq \mathit{max}(\mathit{in}(d,j,q'))$.

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: $\mathit{test}(i,q)$.
     If $\neg\mathit{test}(j,q)$, then $\mathit{next\text{-}empty}(i,q) = \mathit{next\text{-}empty}(S(i),q) \leq j$ (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 $\mathit{next\text{-}empty}(i,q) = i$.
   - $\mathit{test}(i,q)$.
     By Lemma 6.4.1, $i \leq \mathit{max}(q)$, 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 $\mathit{next\text{-}empty}(i, \mathit{in}(d,j,q)) \geq i$ (Lem. 6.4.3) $= \mathit{next\text{-}empty}(i,q)$.
   - $\mathit{test}(i,q)$. Then also $\mathit{test}(i, \mathit{in}(d,j,q))$.
     By Lemma 6.4.1, $i \leq \mathit{max}(q)$, so $S(\mathit{max}(q)) \dot{-} S(i) < S(\mathit{max}(q)) \dot{-} i$. Hence,

$$\begin{array}{lll} & \mathit{next\text{-}empty}(i, \mathit{in}(d,j,q)) & \\ = & \mathit{next\text{-}empty}(S(i), \mathit{in}(d,j,q)) & \\ \geq & \mathit{next\text{-}empty}(S(i),q) & \text{(by induction)} \\ = & \mathit{next\text{-}empty}(i,q) & \end{array}$$

5. By induction on $S(max(q)) \dot{-} i$. Let $j \neq next\text{-}empty(i, q)$.

   - $\neg test(i, q)$.
     Then $next\text{-}empty(i, q) = i$. This implies $j \neq i$, so we have $\neg test(i, in(d, j, q))$. Hence, $next\text{-}empty(i, in(d, j, q)) = i$.
   - $test(i, q)$. Then also $test(i, in(d, j, q))$.
     By Lemma 6.4.1, $i \leq max(q)$, so $S(max(q)) \dot{-} S(i) < S(max(q)) \dot{-} i$. Furthermore, $test(i, q)$ implies $j \neq next\text{-}empty(S(i), q)$. Hence,

$$\begin{array}{lll} & next\text{-}empty(i, in(d, j, q)) & \\ = & next\text{-}empty(S(i), in(d, j, q)) & \\ = & next\text{-}empty(S(i), q) & \text{(by induction)} \\ = & next\text{-}empty(i, q) & \end{array}$$

6. By induction on $S(max(q)) \dot{-} i$.

   - $\neg test(i, q)$.
     Then $next\text{-}empty(i, q) = i$. Furthermore, by Lemma 6.4.3, $next\text{-}empty(S(i), q) \neq i$. Hence,

$$\begin{array}{lll} & next\text{-}empty(i, in(d, next\text{-}empty(i, q), q)) & \\ = & next\text{-}empty(i, in(d, i, q)) & \\ = & next\text{-}empty(S(i), in(d, i, q)) & \\ = & next\text{-}empty(S(i), q) & \text{(Lem. 6.4.5)} \\ = & next\text{-}empty(S(next\text{-}empty(i, q)), q) & \end{array}$$

   - $test(i, q)$.
     By Lemma 6.4.1, $i \leq max(q)$, so the induction hypothesis can be applied with respect to $S(i)$.

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

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

   - $\neg test(i, q)$.
     Then, by Lemma 6.2.2, $\neg test(i, remove(j, q))$. So we have $next\text{-}empty(i, remove(j, q)) = i = next\text{-}empty(i, q)$.
   - $test(i, q)$.
     Let $\neg(i \leq j < next\text{-}empty(i, q))$. $test(i, q)$, implies $\neg(S(i) \leq j < next\text{-}empty(S(i), q))$. Furthermore, by Lemma 6.4.1, $i \leq max(q)$, so the induction hypothesis can be applied with respect to $S(i)$. Since $next\text{-}empty(i, q) = next\text{-}empty(S(i), q) \geq S(i)$ (Lem. 6.4.3), $\neg(i \leq j < next\text{-}empty(i, q))$ implies $j \neq i$. Then, by Lemma 6.2.2, $test(i, remove(j, q))$. Hence,

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

$\boxtimes$

Lemma 6.5 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. $i \leq j \;\rightarrow\; \mathit{append}(d, q[i..j\rangle) = \mathit{in}(d, j, q)[i..S(j)\rangle$

7. $\mathit{test}(k, q) \;\rightarrow\; \mathit{in}(\mathit{retrieve}(k, q), k, q)[i..j\rangle = q[i..j\rangle$

8. $\neg(i \leq k < j) \;\rightarrow\; \mathit{remove}(k, q)[i..j\rangle = q[i..j\rangle$

9. $\ell \leq i \;\rightarrow\; \mathit{release}(k, \ell, q)[i..j\rangle = q[i..j\rangle$

Proof. The proofs of these nine facts are straightforward and left to the reader. We restrict to a listing of the induction bases.

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 Lemma 6.2.2, 6.2.3.
9. By induction on $j \dot{-} i$, together with Lemma 6.2.4, 6.2.5.

⊠

6.2 Invariants

Invariants of a system are properties of data that are satisfied throughout the reachable state space of the system. Lemma 6.6 collects 22 invariants of $\mathbf{N}_{nonmod}$ that are needed in the correctness proof.

Lemma 6.6 The following invariants hold for $\mathbf{N}_{nonmod}(\ell, m, q, \ell', q', g, e, h, g', h')$.

1. $h' \leq \mathit{next\text{-}empty}(\ell', q')$
2. $\ell \leq \mathit{next\text{-}empty}(\ell', q')$
3. $g' \neq 0 \;\rightarrow\; \ell \leq h'$
4. $\mathit{test}(i, q) \;\rightarrow\; i < m$
5. $g \neq 0 \;\rightarrow\; h < m$
6. $\mathit{test}(i, q') \;\rightarrow\; i < m$
7. $\mathit{test}(i, q') \;\rightarrow\; \ell' \leq i < \ell' + n$

8. $\ell' \leq m$
9. $\mathit{next\text{-}empty}(\ell', q') \leq m$
10. $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$
11. $\ell \leq m$
12. $\mathit{test}(i, q) \;\rightarrow\; \ell \leq i$
13. $\ell \leq i < m \;\rightarrow\; \mathit{test}(i, q)$
14. $\ell \leq \ell' + n$
15. $m \leq \ell + n$
16. $g \neq 0 \;\rightarrow\; \mathit{next\text{-}empty}(\ell', q') \leq h + n$
17. $\ell' \leq i < h' \;\rightarrow\; \mathit{test}(i, q')$
18. $\ell' \leq i < \ell \;\rightarrow\; \mathit{test}(i, q')$
19. $g \neq 0 \wedge \mathit{test}(h, q) \;\rightarrow\; \mathit{retrieve}(h, q) = e$
20. $\mathit{test}(i, q) \wedge \mathit{test}(i, q') \;\rightarrow\; \mathit{retrieve}(i, q) = \mathit{retrieve}(i, q')$
21. $g \neq 0 \wedge \mathit{test}(h, q') \;\rightarrow\; \mathit{retrieve}(h, q') = e$
22. $\ell \leq i \wedge j \leq \mathit{next\text{-}empty}(i, q') \;\rightarrow\; q[i..j\rangle = q'[i..j\rangle$

Proof. It is easy to verify that all invariants hold in the initial state (where the buffers are empty, the parameters in the natural numbers equal zero, and e has the default value d_0). In case 1-21 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. $h' \leq \mathit{next\text{-}empty}(\ell', q')$.
   Summands E, G, H, I and K need to be checked. Summands, H, I and K are trivial, because in these cases $h' := \mathit{next\text{-}empty}(\ell', q')$ or $h' := 0$.

   E: $q' := \mathit{in}(e, h, q')$;
   $h' \leq \mathit{next\text{-}empty}(\ell', q') \leq \mathit{next\text{-}empty}(\ell', \mathit{in}(e, h, q'))$ (Lem. 6.4.4).

   G: $\ell' := S(\ell')$, $q' := \mathit{remove}(\ell', q')$; under condition $\mathit{test}(\ell', q')$;
   $h' \leq \mathit{next\text{-}empty}(\ell', q') = \mathit{next\text{-}empty}(S(\ell'), q') = \mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q'))$ (Lem. 6.4.7).

2. $\ell \leq \mathit{next\text{-}empty}(\ell', q')$.
   Summands E, G and K need to be checked.

   E: $q' := \mathit{in}(e, h, q')$;
   $\ell \leq \mathit{next\text{-}empty}(\ell', q') \leq \mathit{next\text{-}empty}(\ell', \mathit{in}(e, h, q'))$ (Lem. 6.4.4).

   G: $\ell' := S(\ell')$, $q' := \mathit{remove}(\ell', q')$; under condition $\mathit{test}(\ell', q')$;
   $\ell \leq \mathit{next\text{-}empty}(\ell', q') = \mathit{next\text{-}empty}(S(\ell'), q') = \mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q'))$ (Lem. 6.4.7).

   K: $\ell := h'$;
   $h' \leq \mathit{next\text{-}empty}(\ell', q')$ by Invariant 6.6.1.

3. $g' \neq 0 \to \ell \leq h'$.
   Summands H, I, J and K need to be checked. Summands I and K are trivial, because in these cases $g' := 0$.

   H: $g' := 1$, $h' := \mathit{next\text{-}empty}(\ell', q')$;
   By Invariant 6.6.2, $\ell \leq \mathit{next\text{-}empty}(\ell', q')$.

   J: $g' := 2$; under condition $g' = 1$;
   $g' = 1$ implies $\ell \leq h'$.

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

   A: $m := S(m)$, $q := \mathit{in}(d, m, q)$;
   $\mathit{test}(i, \mathit{in}(d, m, q)) \leftrightarrow i = m \vee \mathit{test}(i, q) \to i = m \vee i < m \leftrightarrow i < S(m)$.

   K: $q := \mathit{release}(\ell, h', q)$;
   $\mathit{test}(i, \mathit{release}(\ell, h', q)) \to \mathit{test}(i, q)$ (Lem. 6.2.4) $\to i < m$.

5. $g \neq 0 \to h < m$.
   Summands A-F need to be checked. Summands C, E and F are trivial, because in these cases $g := 0$.

   A: $m := S(m)$;
   If $g \neq 0$, then $h < m < S(m)$.

   B: $g := 1$, $h := k$; under condition $\mathit{test}(k, q)$;
   By Invariant 6.6.4, $\mathit{test}(k, q)$ implies $k < m$.

   D: $g := 2$; under condition $g = 1$;
   $g = 1$ implies $h < m$.

6. $\mathit{test}(i, q') \to i < m$.
   Summands A, E and G need to be checked.

   A: $m := S(m)$;
   $\mathit{test}(i, q')$ implies $i < m < S(m)$.

   E: $q' := \mathit{in}(e, h, q')$; under condition $g = 2$;
   $g = 2$, so by Invariant 6.6.5, $h < m$. Hence,

   $$\begin{array}{lcl} \mathit{test}(i, \mathit{in}(e, h, q')) & \leftrightarrow & (i = h \vee \mathit{test}(i, q')) \\ & \to & (i = h \vee i < m) \\ & \leftrightarrow & i < m \end{array}$$

   G: $q' := \mathit{remove}(\ell', q')$;
   $\mathit{test}(i, \mathit{remove}(\ell', q')) \to \mathit{test}(i, q')$ (Lem. 6.2.2) $\to i < m$.

7. $\mathit{test}(i, q') \to \ell' \leq i < \ell' + n$.
   Summands E and G need to be checked.

   E: $q' := \mathit{in}(e, h, q')$; under condition $\ell' \leq h < \ell + n$;

   $$\begin{array}{lcl} \mathit{test}(i, \mathit{in}(e, h, q')) & \leftrightarrow & i = h \vee \mathit{test}(i, q') \\ & \to & i = h \vee \ell' \leq i < \ell' + n \\ & \leftrightarrow & \ell' \leq i < \ell' + n \end{array}$$

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

   $$\begin{array}{lcll} \mathit{test}(i, \mathit{remove}(\ell', q')) & \leftrightarrow & \mathit{test}(i, q') \wedge i \neq \ell' & \text{(Lem. 6.2.2)} \\ & \to & \ell' \leq i < \ell' + n \wedge i \neq \ell' & \\ & \to & S(\ell') \leq i < S(\ell') + n & \end{array}$$

8. $\ell' \leq m$.
   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 6.6.6, $\mathit{test}(\ell', q')$ implies $\ell' < m$. So $S(\ell') \leq m$.

9. $\mathit{next\text{-}empty}(\ell', q') \leq m$.
   By Invariant 6.6.8, $\ell' \leq m$. Furthermore, by Invariant 6.6.6, $\neg\mathit{test}(m, q')$. Hence, by Lemma 6.4.2, $\mathit{next\text{-}empty}(\ell', q') \leq m$.

10. $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$.
    By Invariant 6.6.7, $\neg\mathit{test}(\ell' + n, q')$. Hence, by Lemma 6.4.2, $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$.

11. By Invariants 6.6.2 and 6.6.9.

12. $\mathit{test}(i, q) \rightarrow \ell \leq i$.
    Summands A and K need to be checked.

    A: $q := \mathit{in}(d, m, q)$;
    By Invariant 6.6.11, $\ell \leq m$. So $\mathit{test}(i, \mathit{in}(d, m, q)) \leftrightarrow i = m \vee \mathit{test}(i, q) \rightarrow i = m \vee \ell \leq i \rightarrow \ell \leq i$.

    K: $\ell := h'$, $q := \mathit{release}(\ell, h', q)$;
    $\mathit{test}(i, \mathit{release}(\ell, h', q)) \rightarrow \mathit{test}(i, q)$ (Lem. 6.2.4) $\rightarrow \ell \leq i$.

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

    A: $m := S(m)$, $q := \mathit{in}(d, m, q)$;
    $\ell \leq i < S(m) \rightarrow i = m \vee \ell \leq i < m \rightarrow i = m \vee \mathit{test}(i, q) \leftrightarrow \mathit{test}(i, \mathit{in}(d, m, q))$.

    K: $\ell := h'$, $q := \mathit{release}(\ell, h', q)$; under condition $g' = 2$;
    $g' = 2$, so by Invariant 6.6.3, $\ell \leq h'$. Hence,

    $$\begin{array}{lll} h' \leq i < m & \leftrightarrow & \ell \leq i < m \wedge \neg(\ell \leq i < h') \\ & \rightarrow & \mathit{test}(i, q) \wedge \neg(\ell \leq i < h') \\ & \leftrightarrow & \mathit{test}(i, \mathit{release}(\ell, h', q)) \qquad \text{(Lem. 6.2.4)} \end{array}$$

14. By Invariants 6.6.2 and 6.6.10.

15. $m \leq \ell + n$.
    Summands A and K need to be checked.

    A: $m := S(m)$; under condition $m < \ell + n$;
    Then $S(m) \leq \ell + n$.

    K: $\ell := h'$; under condition $g' = 2$;
    $g' = 2$, so by Invariant 6.6.3, $\ell \leq h'$. Hence, $m \leq \ell + n \leq h' + n$.

16. $g \neq 0 \rightarrow \mathit{next\text{-}empty}(\ell', q') \leq h + n$.
    Summands B-G need to be checked. Summands C, E and F are trivial, because in these cases $g := 0$.

    B: $g := 1$, $h := k$; under condition $\mathit{test}(k, q)$;
    By Invariant 6.6.9, $\mathit{next\text{-}empty}(\ell', q') \leq m$. By Invariant 6.6.15, $m \leq \ell + n$. Since $\mathit{test}(k, q)$, Invariant 6.6.12 yields $\ell \leq k$. So $\mathit{next\text{-}empty}(\ell', q') \leq m \leq \ell + n \leq k + n$.

    D: $g := 2$; under condition $g = 1$;
    $g = 1$ implies $\mathit{next\text{-}empty}(\ell', q') \leq h + n$.

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

$$\begin{array}{lcll} \mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q')) & = & \mathit{next\text{-}empty}(S(\ell'), q') & \text{(Lem. 6.4.7)} \\ & = & \mathit{next\text{-}empty}(\ell', q') & \\ & \leq & h + n & \end{array}$$

17. $\ell' \leq i < h' \rightarrow \mathit{test}(i, q')$.
Summands E, G, H, I and K need to be checked. Summands I and K are trivial, because in these cases $h' := 0$.

E: $q' := \mathit{in}(e, h, q')$;
$\ell' \leq i < h' \rightarrow \mathit{test}(i, q') \rightarrow \mathit{test}(i, \mathit{in}(e, h, q'))$.

G: $\ell' := S(\ell')$, $q' := \mathit{remove}(\ell', q')$;
$S(\ell') \leq i < h' \leftrightarrow \ell' \leq i < h' \wedge i \neq \ell' \rightarrow \mathit{test}(i, q') \wedge i \neq \ell' \leftrightarrow \mathit{test}(i, \mathit{remove}(\ell', q'))$ (Lem. 6.2.2).

H: $h' := \mathit{next\text{-}empty}(\ell', q')$;
By Lemma 6.4.2, $\ell' \leq i < \mathit{next\text{-}empty}(\ell', q') \rightarrow \mathit{test}(i, q')$.

18. $\ell' \leq i < \ell \rightarrow \mathit{test}(i, q')$.
Summands E, G and K need to be checked.

E: $q' := \mathit{in}(e, h, q')$;
$\ell' \leq i < \ell \rightarrow \mathit{test}(i, q') \rightarrow \mathit{test}(i, \mathit{in}(e, h, q'))$.

G: $\ell' := S(\ell')$, $q' := \mathit{remove}(\ell', q')$;
$S(\ell') \leq i < \ell \leftrightarrow \ell' \leq i < \ell \wedge i \neq \ell' \rightarrow \mathit{test}(i, q') \wedge i \neq \ell' \leftrightarrow \mathit{test}(i, \mathit{remove}(\ell', q'))$ (Lem. 6.2.2).

K: $\ell := h'$;
By Invariant 6.6.17, $\ell' \leq i < h' \rightarrow \mathit{test}(i, q')$.

19. $g \neq 0 \wedge \mathit{test}(h, q) \rightarrow \mathit{retrieve}(h, q) = e$.
Summands A-F and K need to be checked. Summands C, E and F are trivial, because in these cases $g := 0$.

A: $q := \mathit{in}(d, m, q)$;
By Invariant 6.6.5, $g \neq 0$ implies $h < m$. Hence, $\mathit{retrieve}(h, \mathit{in}(d, m, q)) = \mathit{retrieve}(h, q) = e$.

B: $g := 1$, $e := \mathit{retrieve}(k, q)$, $h := k$;
$\mathit{retrieve}(k, q) = \mathit{retrieve}(k, q)$ holds trivially.

D: $g := 2$; under condition $g = 1$;
If $\mathit{test}(h, q)$, then in view of $g = 1$, $\mathit{retrieve}(h, q) = e$.

K: $q := \mathit{release}(\ell, h', q)$;
Let $g \neq 0 \wedge \mathit{test}(h, \mathit{release}(\ell, h', q))$. By Lemma 6.2.4, $\mathit{test}(h, q)$ and $\neg(\ell \leq h < h')$. Hence, by Lemma 6.2.5, $\mathit{retrieve}(h, \mathit{release}(\ell, h', q)) = \mathit{retrieve}(h, q) = e$.

20. $\mathit{test}(i, q) \wedge \mathit{test}(i, q') \rightarrow \mathit{retrieve}(i, q) = \mathit{retrieve}(i, q')$.
Summands A, E, G and K must be checked.

A: $q := \mathit{in}(d, m, q)$;
By Invariant 6.6.6, $\mathit{test}(i, q')$ implies $i \neq m$.

$$\begin{array}{ll} & \mathit{test}(i, \mathit{in}(d, m, q)) \wedge \mathit{test}(i, q') \\ \leftrightarrow & \mathit{test}(i, q) \wedge \mathit{test}(i, q') \\ \rightarrow & \mathit{retrieve}(i, \mathit{in}(d, m, q)) = \mathit{retrieve}(i, q) = \mathit{retrieve}(i, q') \end{array}$$

E: $q' := \mathit{in}(e, h, q')$; under condition $g = 2$;
Let $\mathit{test}(i, q) \wedge \mathit{test}(i, \mathit{in}(e, h, q'))$.

Case 1: $i \neq h$.

$$\begin{array}{ll} & \mathit{test}(i,q) \wedge \mathit{test}(i, \mathit{in}(e,h,q')) \\ \rightarrow & \mathit{test}(i,q) \wedge \mathit{test}(i,q') \\ \rightarrow & \mathit{retrieve}(i,q) = \mathit{retrieve}(i,q') = \mathit{retrieve}(i, \mathit{in}(e,h,q')) \end{array}$$

Case 2: $i = h$.
Then $\mathit{test}(h,q)$, so Invariant 6.6.19 together with $g = 2$ yields $\mathit{retrieve}(h, \mathit{in}(e,h,q')) = e = \mathit{retrieve}(h,q)$.

G: $q' := \mathit{remove}(\ell', q')$;
By Lemma 6.2.2, $\mathit{test}(i, \mathit{remove}(\ell', q'))$ implies $i \neq \ell'$.

$$\begin{array}{lll} & \mathit{test}(i,q) \wedge \mathit{test}(i, \mathit{remove}(\ell', q')) & \\ \rightarrow & \mathit{test}(i,q) \wedge \mathit{test}(i,q') & \text{(Lem. 6.2.2)} \\ \rightarrow & \mathit{retrieve}(i,q) = \mathit{retrieve}(i,q') = \mathit{retrieve}(i, \mathit{remove}(\ell', q')) & \text{(Lem. 6.2.3)} \end{array}$$

K: $q := \mathit{release}(\ell, h', q)$;
By Lemma 6.2.4, $\mathit{test}(i, \mathit{release}(\ell, h', q))$ implies $\mathit{test}(i,q) \wedge \neg(\ell \leq i < h')$. Hence,

$$\begin{array}{lll} & \mathit{test}(i, \mathit{release}(\ell, h', q)) \wedge \mathit{test}(i,q') & \\ \rightarrow & \mathit{test}(i,q) \wedge \mathit{test}(i,q') & \text{(Lem. 6.2.4)} \\ \rightarrow & \mathit{retrieve}(i,q') = \mathit{retrieve}(i,q) = \mathit{retrieve}(i, \mathit{release}(\ell, h', q)) & \text{(Lem. 6.2.5)} \end{array}$$

21. $g \neq 0 \wedge \mathit{test}(h,q') \rightarrow \mathit{retrieve}(h,q') = e$.
Summands B-G need to be checked. Summands C, E and F are trivial, because in these cases $g := 0$.

B: $g := 1$, $e := \mathit{retrieve}(k,q)$, $h := k$; under condition $\mathit{test}(k,q)$;
If $\mathit{test}(k,q')$, then by Invariant 6.6.20, $\mathit{retrieve}(k,q') = \mathit{retrieve}(k,q)$.

D: $g := 2$; under condition $g = 1$;
If $\mathit{test}(h,q')$, then in view of $g = 1$, $\mathit{retrieve}(h,q') = e$.

G: $q' := \mathit{remove}(\ell', q')$;
Let $g \neq 0$ and $\mathit{test}(h, \mathit{remove}(\ell', q'))$. By Lemma 6.2.2, $\mathit{test}(h,q')$ and $h \neq \ell'$. Hence, by Lemma 6.2.3, $\mathit{retrieve}(h, \mathit{remove}(\ell', q')) = \mathit{retrieve}(h,q') = e$.

22. $\ell \leq i \wedge j \leq \mathit{next\text{-}empty}(i,q') \;\rightarrow\; q[i..j\rangle = q'[i..j\rangle$.
We apply induction on $j \dot{-} i$.

If $i \geq j$, then $q[i..j\rangle = \langle\rangle = q'[i..j\rangle$.

If $i < j$, then we distinguish two cases.

Case 1: $i \geq m$.
Then by Invariant 6.6.4, $\neg\mathit{test}(i,q)$, and by Invariant 6.6.6, $\neg\mathit{test}(i,q')$.

$$\begin{array}{lll} q[i..j\rangle = \mathit{in}(d_0, q[S(i)..j\rangle) & = & \mathit{in}(d_0, q'[S(i)..j\rangle) \text{ (by induction)} \\ & = & q'[i..j\rangle \end{array}$$

Case 2: $i < m$.
Since $\ell \leq i < m$, by Invariant 6.6.13, $\mathit{test}(i,q)$. Furthermore, $i < j \leq \mathit{next\text{-}empty}(i,q')$, so $\mathit{test}(i,q')$. Hence,

$$\begin{array}{lll} q[i..j\rangle & = & \mathit{in}(\mathit{retrieve}(i,q), q[S(i)..j\rangle) \\ & = & \mathit{in}(\mathit{retrieve}(i,q), q'[S(i)..j\rangle) \quad \text{(by induction)} \\ & = & \mathit{in}(\mathit{retrieve}(i,q'), q'[S(i)..j\rangle) \quad \text{(Inv. 6.6.20)} \\ & = & q'[i..j\rangle. \end{array}$$

$\boxtimes$

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

In this section, we prove Theorem 5.3, which states that $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ are strongly bisimilar. Next, we prove that $\mathbf{N}_{nonmod}$ behaves like a FIFO queue of size $2n$.

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

In this section we present a proof of Theorem 5.3. It suffices to prove that for all $\ell, m, \ell', h, h' : Nat$, $q, q' : Buf$, $e : \Delta$ and $g, g' \leq 2$,

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

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

$$\begin{array}{lll} & \mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g, e, h|_{2n}, g', h'|_{2n}) & \\ \approx & \sum_{d:\Delta} r_{\mathrm{A}}(d) \cdot \mathbf{N}_{mod}(m{:=}S(m)|_{2n}, q{:=}in(d, m, q)|_{2n}) \lhd\ m < \ell + n \rhd \delta & (A) \\ + & \sum_{k:Nat} c \cdot \mathbf{N}_{mod}(g{:=}1, e{:=}retrieve(k, q), h{:=}k|_{2n}) \lhd\ test(k, q) \wedge g = 0 \rhd \delta & (B) \\ + & j \cdot \mathbf{N}_{mod}(g{:=}0, e{:=}d_0, h{:=}0) \lhd\ g = 1 \rhd \delta & (C) \\ + & j \cdot \mathbf{N}_{mod}(g{:=}2) \lhd\ g = 1 \rhd \delta & (D) \\ + & c \cdot \mathbf{N}_{mod}(q'{:=}in(e, h, q')|_{2n}, g{:=}0, e{:=}d_0, h{:=}0) \lhd\ \ell' \leq h < \ell' + n \wedge g = 2 \rhd \delta & (E) \\ + & c \cdot \mathbf{N}_{mod}(g{:=}0, e{:=}d_0, h{:=}0) \lhd\ \neg(\ell' \leq h < \ell' + n) \wedge g = 2 \rhd \delta & (F) \\ + & s_{\mathrm{D}}(retrieve(\ell', q')) \cdot \mathbf{N}_{mod}(\ell'{:=}S(\ell')|_{2n}, q'{:=}remove(\ell', q')|_{2n}) \lhd\ test(\ell', q') \rhd \delta & (G) \\ + & c \cdot \mathbf{N}_{mod}(g'{:=}1, h'{:=}next\text{-}empty(\ell', q')|_{2n}) \lhd\ g' = 0 \rhd \delta & (H) \\ + & j \cdot \mathbf{N}_{mod}(g'{:=}0, h'{:=}0) \lhd\ g' = 1 \rhd \delta & (I) \\ + & j \cdot \mathbf{N}_{mod}(g'{:=}2) \lhd\ g' = 1 \rhd \delta & (J) \\ + & c \cdot \mathbf{N}_{mod}(\ell{:=}h'|_{2n}, q{:=}release(\ell, h', q)|_{2n}, g'{:=}0, h'{:=}0) \lhd\ g' = 2 \rhd \delta & (K) \end{array}$$

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

$$\begin{array}{lll} & \mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g, e, h|_{2n}, g', h'|_{2n}) & \\ \approx & \sum_{d:\Delta} r_{\mathrm{A}}(d) \cdot \mathbf{N}_{mod}(m{:=}S(m|_{2n})|_{2n}, q{:=}in(d, m|_{2n}, q|_{2n})) & \\ & \qquad \lhd\ in\text{-}window(\ell|_{2n}, m|_{2n}, (\ell|_{2n} + n)|_{2n}) \rhd \delta & \\ + & \sum_{k:Nat} c \cdot \mathbf{N}_{mod}(g{:=}1, e{:=}retrieve(k, q|_{2n}), h{:=}k) \lhd\ test(k, q|_{2n}) \wedge g = 0 \rhd \delta & \\ + & j \cdot \mathbf{N}_{mod}(g{:=}0, e{:=}d_0, h{:=}0) \lhd\ g = 1 \rhd \delta & \\ + & j \cdot \mathbf{N}_{mod}(g{:=}2) \lhd\ g = 1 \rhd \delta & \\ + & c \cdot \mathbf{N}_{mod}(q'{:=}in(e, h|_{2n}, q'|_{2n}), g{:=}0, e{:=}d_0, h{:=}0) & \\ & \qquad \lhd\ in\text{-}window(\ell'|_{2n}, h|_{2n}, (\ell'|_{2n} + n)|_{2n}) \wedge g = 2 \rhd \delta & \\ + & c \cdot \mathbf{N}_{mod}(g{:=}0, e{:=}d_0, h{:=}0) \lhd\ \neg in\text{-}window(\ell'|_{2n}, h|_{2n}, (\ell'|_{2n} + n)|_{2n}) \wedge g = 2 \rhd \delta & \\ + & s_{\mathrm{D}}(retrieve(\ell'|_{2n}, q'|_{2n})) \cdot \mathbf{N}_{mod}(\ell'{:=}S(\ell'|_{2n})|_{2n}, q'{:=}remove(\ell'|_{2n}, q'|_{2n})) \lhd\ test(\ell'|_{2n}, q'|_{2n}) \rhd \delta & \end{array}$$

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

$+ \quad c{\cdot}\mathbf{N}_{mod}(g'{:=}1, h'{:=}\mathit{next\text{-}empty}|_{2n}(\ell'|_{2n}, q'|_{2n})) \lhd\ g' = 0 \rhd \delta$

$+ \quad j{\cdot}\mathbf{N}_{mod}(g'{:=}0, h'{:=}0) \lhd\ g' = 1 \rhd \delta$

$+ \quad j{\cdot}\mathbf{N}_{mod}(g'{:=}2) \lhd\ g' = 1 \rhd \delta$

$+ \quad c{\cdot}\mathbf{N}_{mod}(\ell{:=}h'|_{2n}, q{:=}\mathit{release}|_{2n}(\ell|_{2n}, h'|_{2n}, q|_{2n}), g'{:=}0, h'{:=}0) \lhd\ g' = 2 \rhd \delta$

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

A

- $m < \ell + n = \mathit{in\text{-}window}(\ell|_{2n}, m|_{2n}, (\ell|_{2n} + n)|_{2n})$.
  $m < \ell + n \leftrightarrow \ell \leq m < \ell + n$ (Inv. 6.6.11) $\rightarrow \mathit{in\text{-}window}(\ell|_{2n}, m|_{2n}, (\ell + n)|_{2n})$ (Lem. 6.3.7). Reversely, $\mathit{in\text{-}window}(\ell|_{2n}, m|_{2n}, (\ell + n)|_{2n}) \rightarrow m + n < \ell \vee \ell \leq m < \ell + n \vee m \geq \ell + 2n$ (Lem. 6.3.8) $\leftrightarrow m < \ell + n$ (Inv. 6.6.11 and 6.6.15). Furthermore, by Lemma 6.1.1, $(\ell + n)|_{2n} = (\ell|_{2n} + n)|_{2n}$. So concluding, $m < \ell + n = \mathit{in\text{-}window}(\ell|_{2n}, m|_{2n}, (\ell|_{2n} + n)|_{2n})$.
- $S(m)|_{2n} = S(m|_{2n})|_{2n}$.
  This follows from Lemma 6.1.1.
- $\mathit{in}(d, m, q)|_{2n} = \mathit{in}(d, m|_{2n}, q|_{2n})$.
  This follows from the definition of buffers modulo $2n$.

B Below we equate the entire summand B of the two specifications. The conjunct $g = 0$ and the argument $g{:=}1$ of summand B are omitted, because they are irrelevant for this derivation.

By Invariants 6.6.4, 6.6.12 and 6.6.15, $\mathit{test}(j, q) \rightarrow \ell \leq j < \ell + n$. So by Lemma 6.3.9, $\mathit{test}(k', q|_{2n})$ implies $\mathit{in\text{-}window}(\ell|_{2n}, k', (\ell + n)|_{2n})$. By Lemma 6.3.1, $k' = k'|_{2n}$, so by Lemma 6.3.8 this implies $k' + n < \ell|_{2n} \vee \ell|_{2n} \leq k' < \ell|_{2n} + n \vee k' \geq \ell + 2n$. By Lemmas 6.1.2 and, 6.3.1, $k' = k'|_{2n} < 2n$, so this implies $k' + n < \ell|_{2n} \vee \ell|_{2n} \leq k' < \ell|_{2n} + n$.

$$
\begin{array}{lll}
 & \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(e{:=}\mathit{retrieve}(k, q), h{:=}k|_{2n}) & \\
 & \lhd\ \mathit{test}(k, q) \rhd \delta & \\
\approx & \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(e{:=}\mathit{retrieve}(k, q), h{:=}k|_{2n}) & \\
 & \lhd\ \mathit{test}(k, q) \wedge \ell \leq k < \ell + n \rhd \delta & \text{(Inv. 6.6.4, 6.6.12, 6.6.15)} \\
\approx & \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(e{:=}\mathit{retrieve}(k|_{2n}, q|_{2n}), h{:=}k|_{2n}) & \\
 & \lhd\ \mathit{test}(k|_{2n}, q|_{2n}) \wedge \ell \leq k < \ell + n \rhd \delta & \text{(Lem. 6.3.2, 6.3.3)} \\
\approx & \sum_{k':Nat} \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(e{:=}\mathit{retrieve}(k', q|_{2n}), h{:=}k') & \\
 & \lhd\ \mathit{test}(k', q|_{2n}) \wedge \ell \leq k < \ell + n \wedge k' = k|_{2n} \rhd \delta & \text{(sum elim.)} \\
\approx & \sum_{k':Nat} \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(e{:=}\mathit{retrieve}(k', q|_{2n}), h{:=}k') & \\
 & \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 & \\
+ & \sum_{k':Nat} \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(e{:=}\mathit{retrieve}(k', q|_{2n}), h{:=}k') & \\
 & \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. 6.1.4, 6.1.5)} \\
\approx & \sum_{k':Nat} c{\cdot}\mathbf{N}_{mod}(e{:=}\mathit{retrieve}(k', q|_{2n}), h{:=}k') & \\
 & \lhd\ \mathit{test}(k', q|_{2n}) \wedge \ell|_{2n} \leq k' < \ell|_{2n} + n \wedge k' = k' \rhd \delta & \\
+ & \sum_{k':Nat} c{\cdot}\mathbf{N}_{mod}(e{:=}\mathit{retrieve}(k', q|_{2n}), h{:=}k') & \\
 & \lhd\ \mathit{test}(k', q|_{2n}) \wedge k' + n < \ell|_{2n} \wedge k' = k' \rhd \delta & \text{(sum elim., Lem. 6.1.3)} \\
\approx & \sum_{k':Nat} c{\cdot}\mathbf{N}_{mod}(e{:=}\mathit{retrieve}(k', q|_{2n}), h{:=}k') & \\
 & \lhd\ \mathit{test}(k', q|_{2n}) \rhd \delta & \text{(see above)}
\end{array}
$$

E
- $g = 2 \rightarrow \ell' \leq h < \ell' + n = \mathit{in\text{-}window}(\ell'|_{2n}, h|_{2n}, (\ell' + n)|_{2n})$.
  Let $g = 2$. By Lemma 6.4.3, $\ell' \leq \mathit{next\text{-}empty}(\ell', q')$, and by Invariant 6.6.16 together with $g = 2$, $\mathit{next\text{-}empty}(\ell', q') \leq h + n$. Hence, $\ell' \leq h + n$. Furthermore, by Invariant 6.6.5 together with $g = 2$, $h < m$, by Invariant 6.6.15, $m \leq \ell + n$, and by Invariant 6.6.14, $\ell \leq \ell' + n$. Hence, $h < \ell' + 2n$. So using Lemmas 6.3.7 and 6.3.8, it follows that $\ell' \leq h < \ell' + n = \mathit{in\text{-}window}(\ell'|_{2n}, h|_{2n}, (\ell' + n)|_{2n})$.
- $\mathit{in}(e, h, q')|_{2n} = \mathit{in}(e, h|_{2n}, q'|_{2n})$.
  This follows from the definition of buffers modulo $2n$.

F $g = 2 \rightarrow \neg(\ell' \leq h < \ell' + n) = \neg\mathit{in\text{-}window}(\ell'|_{2n}, h|_{2n}, (\ell' + n)|_{2n})$.
This follows immediately from the first item of the previous case.

G
- $\mathit{test}(\ell', q') = \mathit{test}(\ell'|_{2n}, q'|_{2n})$.
  This follows from Lemma 6.3.2 together with Invariant 6.6.7.
- $\mathit{test}(\ell', q') \rightarrow (\mathit{retrieve}(\ell', q') = \mathit{retrieve}(\ell'|_{2n}, q'|_{2n}))$.
  This follows from Lemma 6.3.3 together with Invariant 6.6.7.
- $S(\ell')|_{2n} = S(\ell'|_{2n})|_{2n}$.
  This follows from Lemma 6.1.1.
- $\mathit{remove}(\ell', q')|_{2n} = \mathit{remove}(\ell'|_{2n}, q'|_{2n})$.
  This follows from Lemma 6.3.4 together with Invariant 6.6.7.

H $\mathit{next\text{-}empty}(\ell', q')|_{2n} = \mathit{next\text{-}empty}|_{2n}(\ell'|_{2n}, q'|_{2n})$.
This follows from Lemma 6.3.6 together with Invariant 6.6.7.

K $g' = 2 \rightarrow \mathit{release}(\ell, h', q)|_{2n} = \mathit{release}|_{2n}(\ell|_{2n}, h'|_{2n}, q|_{2n})$.
Let $g' = 2$. By Invariant 6.6.3 together with $g' = 2$, $\ell \leq h'$. By Invariant 6.6.1, $h' \leq \mathit{next\text{-}empty}(\ell', q')$. By Invariant 6.6.9, $\mathit{next\text{-}empty}(\ell', q') \leq m$. By Invariant 6.6.15, $m \leq \ell + n$. So $\ell \leq h' \leq \ell + n$. Hence, the desired equation follows from Lemma 6.3.5 together with Invariant 6.6.4, 6.6.12 and 6.6.15.

Hence, $\mathbf{N}_{mod}(\ell|_{2n}, m|_{2n}, q|_{2n}, \ell'|_{2n}, q'|_{2n}, g, e, h|_{2n}, g', h'|_{2n})$ is a solution for the defining equation of $\mathbf{N}_{nonmod}(\ell, m, q, \ell', q', g, e, h, g', h')$. So by CL-RSP, they are strongly (and thus branching) bisimilar. ⊠

7.2 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 4.2), using the cones and foci method [13].

Let Ξ abbreviate $\mathit{Nat} \times \mathit{Nat} \times \mathit{Buf} \times \mathit{Nat} \times \mathit{Buf} \times \mathit{Nat} \times \Delta \times \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat}$. Furthermore, let $\xi{:}\Xi$ denote $(\ell, m, q, \ell', q', g, e, h, g', h')$. 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 position of the receiving window (i.e., ℓ') until the first empty position in this window, and then continuing in the sending window until the first empty position in that window (i.e., m). Note that ϕ is independent of e, g, ℓ, h, g', h'; we therefore write $\phi(m, q, \ell', q')$.

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

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

Lemma 7.1 For each $\xi{:}\Xi$ where the invariants in Lemma 6.6 hold, there is a $\hat{\xi}{:}\Xi$ with $\mathrm{FC}(\hat{\xi})$ such that $\mathbf{N}_{nonmod}(\xi) \stackrel{c_1}{\rightarrow} \cdots \stackrel{c_n}{\rightarrow} \mathbf{N}_{nonmod}(\hat{\xi})$, where $c_1, \ldots, c_n \in \mathcal{I}$.

Proof. In case $g \neq 0$ in ξ, by summands C, E and F, we can perform one or two communication actions to a state where $g = 0$. By Invariants 6.6.9 and 6.6.10, $\mathit{next\text{-}empty}(\ell', q') \leq \min\{m, \ell'+n\}$. We prove by induction on $\min\{m, \ell'+n\} - \mathit{next\text{-}empty}(\ell', q')$ that for each state ξ' where $g = 0$ and the invariants in Lemma 6.6 hold, a focus point can be reached by a sequence of communication actions.

BASE CASE: $\mathit{next\text{-}empty}(\ell', q') = \min\{m, \ell'+n\}$.

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

INDUCTION CASE: $\mathit{next\text{-}empty}(\ell', q') < \min\{m, \ell'+n\}$.

By Invariant 6.6.12, $\ell \leq \mathit{next\text{-}empty}(\ell', q') < m$. Recall that $g = 0$, so by Invariant 6.6.13, $\mathit{test}(\mathit{next\text{-}empty}(\ell', q'), q)$. Furthermore, in view of Lemma 6.4.3, $\ell' \leq \mathit{next\text{-}empty}(\ell, q') < \ell'+n$. Hence, by summands B, D and E from ξ' we can perform three communication actions to a state ξ'' where $g = 0$, and in comparison to ξ' the values of m and ℓ' remain the same where $q' := \mathit{in}(d, \mathit{next\text{-}empty}(\ell', q'), q')$ (where d denotes $\mathit{retrieve}(\mathit{next\text{-}empty}(\ell', q'), q)$). By Lemmas 6.4.6 and 6.4.3, $\mathit{next\text{-}empty}(\ell', \mathit{in}(d, \mathit{next\text{-}empty}(\ell', q'), q')) = \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(\ell', q')), q') > \mathit{next\text{-}empty}(\ell', q')$. So we can apply the induction hypothesis to conclude that from ξ'' a focus point can be reached by a sequence of communication actions.

⊠

Theorem 7.2 For all $e{:}\Delta$,

$$\tau_{\{c,j\}}(\mathbf{N}_{nonmod}(0, 0, [], 0, [], 0, e, 0, 0, 0)) \leftrightarrows_b \mathbf{Z}(\langle\rangle).$$

Proof. By the cones and foci method we obtain the following matching criteria (cf. [13]). Trivial matching criteria are left out.
Class I:

$$\ell' \leq h < \ell'+n \wedge g = 2 \;\rightarrow\; \phi(m, q, \ell', q') = \phi(m, q, \ell', \mathit{in}(e, h, q'))$$

$$g' = 2 \;\rightarrow\; \phi(m, q, \ell', q') = \phi(m, \mathit{release}(\ell, h', q), \ell', q')$$

Class II:

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

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

Class III:

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

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

Class IV:

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

Class V:

$$m < \ell + n \;\rightarrow\; \phi(S(m), \mathit{in}(d, m, q), \ell', q') = \mathit{append}(d, \phi(m, q, \ell', q'))$$

$$\mathit{test}(\ell', q') \;\rightarrow\; \phi(m, q, S(\ell'), \mathit{remove}(\ell', q')) = \mathit{tail}(\phi(m, q, \ell', q'))$$

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

Case 1: $h \neq \mathit{next\text{-}empty}(\ell', q'))$.
Let $g = 2$. By Lemma 6.4.5, $\mathit{next\text{-}empty}(\ell', \mathit{in}(e, h, q')) = \mathit{next\text{-}empty}(\ell', q')$. Hence,

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

Case 1.1: $\ell' \leq h < \mathit{next\text{-}empty}(\ell', q'))$.
By Lemma 6.4.2, $\mathit{test}(h, q')$, and by Invariant 6.6.21 together with $g = 2$, $\mathit{retrieve}(h, q') = e$. So by Lemma 6.5.7, $\mathit{in}(e, h, q')[\ell'..\mathit{next\text{-}empty}(\ell', q'))\rangle = q'[\ell'..\mathit{next\text{-}empty}(\ell', q'))\rangle$.

Case 1.2: $\neg(\ell' \leq h < \mathit{next\text{-}empty}(\ell', q'))$.
Using Lemma 6.5.8, we have $\mathit{in}(e, h, q')[\ell'..\mathit{next\text{-}empty}(\ell', q'))\rangle = q'[\ell'..\mathit{next\text{-}empty}(\ell', q'))\rangle$.

Case 2: $h = \mathit{next\text{-}empty}(\ell', q')$.
Let $g = 2$. The derivation splits into two parts.

(1) Using Lemma 6.5.8, it follows that $\mathit{in}(e, h, q')[\ell'..h\rangle = q'[\ell'..h\rangle$.

(2) By Invariant 6.6.2, $\ell \leq h$, and by Invariant 6.6.5 together with $g = 2$, $h < m$. Thus, by Invariant 6.6.13, $\mathit{test}(h, q)$. So by Invariant 6.6.19 together with $g = 2$, $\mathit{retrieve}(h, q) = e$. Hence,

$$\begin{array}{lll}
& \mathit{in}(e, h, q')[h..\mathit{next\text{-}empty}(S(h), q')\rangle & \\
= & \mathit{in}(e, \mathit{in}(e, h, q')[S(h)..\mathit{next\text{-}empty}(S(h), q')\rangle) & \\
= & \mathit{in}(e, q'[S(h)..\mathit{next\text{-}empty}(S(h), q')\rangle) & \text{(Lem. 6.5.8)} \\
= & \mathit{in}(e, q[S(h)..\mathit{next\text{-}empty}(S(h), q')\rangle) & \text{(Inv. 6.6.22)} \\
= & q[h..\mathit{next\text{-}empty}(S(h), q')\rangle &
\end{array}$$

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

$$\begin{array}{lll}
& \mathit{in}(e, h, q')[\ell'..\mathit{next\text{-}empty}(\ell', \mathit{in}(e, h, q'))\rangle & \\
& {+}{+}q[\mathit{next\text{-}empty}(\ell', \mathit{in}(e, h, q'))..m\rangle & \\
= & \mathit{in}(e, h, q')[\ell'..\mathit{next\text{-}empty}(S(h), q')\rangle & \\
& {+}{+}q[\mathit{next\text{-}empty}(S(h), q')..m\rangle & \text{(Lem. 6.4.6)} \\
= & \mathit{in}(e, h, q')[\ell'..h\rangle{+}{+}\mathit{in}(e, h, q')[h..\mathit{next\text{-}empty}(S(h), q')\rangle & \\
& {+}{+}q[\mathit{next\text{-}empty}(S(h), q')..m\rangle & \text{(Lem. 6.5.5)} \\
= & q'[\ell'..h\rangle{+}{+}q[h..\mathit{next\text{-}empty}(S(h), q')\rangle & \\
& {+}{+}q[\mathit{next\text{-}empty}(S(h), q')..m\rangle & \text{(1), (2)} \\
= & q'[\ell'..h\rangle{+}{+}q[h..m\rangle & \text{(Lem. 6.5.1, 6.4.2, 6.5.5, Inv. 6.6.6)}
\end{array}$$

I.2 $g' = 2 \rightarrow \phi(m, q, \ell', q') = \phi(m, \mathit{release}(\ell, h', q), \ell', q')$.

By Invariant 6.6.1, $h' \leq \mathit{next\text{-}empty}(\ell', q')$. So by Lemma 6.5.9,

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

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

Let $m < \ell + n$. By Invariant 6.6.10, $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$. Hence,

$$\begin{array}{lll}
& \mathit{length}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle{+}{+}q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) & \\
= & \mathit{length}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle) + \mathit{length}(q[\mathit{next\text{-}empty}(\ell', q')..m\rangle)) & \text{(Lem. 6.5.2)} \\
= & (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q')) & \text{(Lem. 6.5.4)} \\
\leq & n + (m \dot{-} \ell) & \text{(Inv. 6.6.2)} \\
< & 2n &
\end{array}$$

II.2 $\mathit{test}(\ell', q') \rightarrow \mathit{length}(\phi(m, q, \ell', q')) > 0$.

$\mathit{test}(\ell', q')$ together with Lemma 6.4.3 yields $\mathit{next\text{-}empty}(\ell', q') = \mathit{next\text{-}empty}(S(\ell'), q') \geq S(\ell')$. Hence, by Lemmas 6.5.2 and 6.5.4, $\mathit{length}(\phi(m, q, \ell', q')) = (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q')) > 0$.

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: $\ell = \ell' + n$.
By Invariant 6.6.10, $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$. Hence,

$$\begin{array}{lll} & \mathit{length}(\phi(m, q, \ell', q')) & \\ = & (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q')) & \\ \leq & ((\ell' + n) \dot{-} \ell') + (m \dot{-} \ell) & \text{(Inv. 6.6.2)} \\ = & n + (m \dot{-} \ell) & \end{array}$$

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. 6.6.2) $= 0$, so we have $\mathit{length}(\phi(m, q, \ell', q')) = \mathit{next\text{-}empty}(\ell', q') \dot{-} \ell'$. 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 6.6.2, $\mathit{next\text{-}empty}(\ell', q') \geq \ell' + n$, 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. 6.4.3). Hence, we have $q'[\ell' .. \mathit{next\text{-}empty}(\ell', q')\rangle = \mathit{in}(\mathit{retrieve}(\ell', q'), q'[S(\ell') .. \mathit{next\text{-}empty}(\ell', q')\rangle)$. It is easy to see that $\mathit{top}(\phi(m, q, \ell', q')) = \mathit{retrieve}(\ell', q')$.

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

$$\begin{array}{lll} & q'[\ell' .. \mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ \mathit{in}(d, m, q)[\mathit{next\text{-}empty}(\ell', q') .. S(m)\rangle & \\ = & q'[\ell' .. \mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ \mathit{append}(d, q[\mathit{next\text{-}empty}(\ell', q') .. m\rangle) & \text{(Lem. 6.5.6, Inv. 6.6.9)} \\ = & \mathit{append}(d, q'[\ell' .. \mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q') .. m\rangle) & \text{(Lem. 6.5.3)} \end{array}$$

V.2 $\mathit{test}(\ell', q') \rightarrow \phi(m, q, S(\ell'), \mathit{remove}(\ell', q')) = \mathit{tail}(\phi(m, q, \ell', q'))$.

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

$$\begin{array}{lll} & \mathit{remove}(\ell', q')[S(\ell') .. \mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q'))\rangle & \\ & +\!\!+ q[\mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q')) .. m\rangle & \\ = & \mathit{remove}(\ell', q')[S(\ell') .. \mathit{next\text{-}empty}(S(\ell'), q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(S(\ell'), q') .. m\rangle & \text{(Lem. 6.4.7)} \\ = & \mathit{remove}(\ell', q')[S(\ell') .. \mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q') .. m\rangle & \\ = & q'[S(\ell') .. \mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q') .. m\rangle & \text{(Lem. 6.5.8)} \\ = & \mathit{tail}(q'[\ell' .. \mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q') .. m\rangle) & \end{array}$$

$\boxtimes$

7.3 Correctness of the Sliding Window Protocol
Finally, we can prove Theorem 4.1.

Proof.

$$
\begin{array}{lll}
 & \tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}(0,0,[]) \parallel \mathbf{R}(0,[]) \parallel \mathbf{K} \parallel \mathbf{L})) & \\
\underline{\leftrightarrow} & \tau_{\mathcal{I}}(\mathbf{M}_{mod}(0,0,[],0,[],0,d_0,0,0,0)) & \text{(Thm. 5.1)} \\
\underline{\leftrightarrow} & \tau_{\{c,j\}}(\mathbf{N}_{mod}(0,0,[],0,[],0,d_0,0,0,0)) & \text{(Thm. 5.2)} \\
\underline{\leftrightarrow} & \tau_{\{c,j\}}(\mathbf{N}_{nonmod}(0,0,[],0,[],0,d_0,0,0,0)) & \text{(Thm. 5.3)} \\
\underline{\leftrightarrow}_b & \mathbf{Z}(\langle\rangle) & \text{(Thm. 7.2)}
\end{array}
$$

⊠

8. Related Work

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

Infinite window size Stenning [47] 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 [30] examined more general principals behind Stenning's protocol, and manually verified some safety properties. Hailpern [24] used temporal logic to formulate safety and liveness properties for Stenning's protocol, and established their validity by informal reasoning. Jonsson [27] also verified both safety and liveness properties of the protocol, using temporal logic and a manual compositional verification technique.

Fixed finite window size Richier *et al.* [40] 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 [33] specified a SWP in Lotos, with the help of the simulation environment Lite, and proved some safety properties for window size six. Holzmann [25, 26] used the Spin model checker to verify both safety and liveness properties of a SWP with sequence numbers up to five. Kaivola [29] verified safety and liveness properties using model checking for a SWP with window size up to seven. Godefroid and Long [15] 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.* [46] 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 [44] specified a SWP in the high-level language IOA, and used the theorem prover MONA to verify a safety property for unbounded sequence numbers with window size up to 256. Latvala [31] 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 [8] specified a SWP using higher order logic, and manually proved and mechanically checked safety properties using HOL. (Van de Snepscheut [45] noted that what Cardell-Oliver claims to be a liveness property is in fact a safety property.) Schoone [43] manually proved safety properties for several SWPs using assertional verification. Van de Snepscheut [45] gave a correctness proof of a SWP as a sequence of correctness preserving transformations of a sequential program. Paliwoda and Sanders [38] specified a reduced version of what they call a SWP (but which is in fact very similar to the bakery protocol from [18]) in the process algebra CSP, and verified a safety property modulo trace semantics. Röckl and Esparza [41] verified the correctness of

this bakery protocol modulo weak bisimulation using Isabelle/HOL, by explicitly checking a bisimulation relation. Jonsson and Nilsson [28] used an automated reachability analysis to verify safety properties for a SWP with arbitrary sending window size and receiving window size one. Rusu [42] used the theorem prover PVS to verify both safety and liveness properties for a SWP with unbounded sequence numbers. Chkliaev *et al.* [10] used a timed state machine in PVS to specify a SWP in which messages can be lost, duplicated or reordered, and proved some safety properties with the mechanical support of PVS.

References

1. R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, 1969.
3. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.
4. J.A. Bergstra and J.W. Klop. Verification of an alternating bit protocol by means of process algebra. In *Proc. Spring School on Mathematical Methods of Specification and Synthesis of Software Systems*, LNCS 215, pp. 9–23. Springer, 1986.
5. M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In *Proc. CONCUR'94*, LNCS 836, pp. 401–416. Springer, 1994.
6. M.A. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.
7. J.J. Brunekreef. Sliding window protocols. In S. Mauw and G. Veltink, eds, *Algebraic Specification of Protocols*. Cambridge Tracts in Theoretical Computer Science 36, pp. 71–112. Cambridge University Press, 1993.
8. R. Cardell-Oliver. Using higher order logic for modelling real-time protocols. In *Proc. TAPSOFT'91*, LNCS 494, pp. 259–282. Springer, 1991.
9. V.G. Cerf and R.E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, COM-22:637–648, 1974.
10. D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In *Proc. TACAS'03*, LNCS 2619, pp. 113–127. Springer, 2003.
11. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.
12. E.W. Dijkstra, editor. *Formal Development of Programs and Proofs*. Addison-Wesley, 1989
13. W.J. Fokkink and J. Pang. Cones and foci for protocol verification revisited. In *Proc. FOSSACS'03*, LNCS 2620, pp. 267–281. Springer, 2003.
14. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

15. P. Godefroid and D.E. Long. Symbolic protocol verification with Queue BDDs. *Formal Methods and System Design*, 14(3):257–271, 1999.
16. R.A. Groenveld. Verification of a sliding window protocol by means of process algebra. Report P8701, University of Amsterdam, 1987.
17. J.F. Groote. *Process Algebra and Structured Operational Semantics.* PhD thesis, University of Amsterdam, 1991.
18. J.F. Groote and H.P. Korver. Correctness proof of the bakery protocol in μCRL. In *Proc. ACP'94*, Workshops in Computing, pp. 63–86. Springer, 1995.
19. J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In *Proc. SoSL'93*, Workshops in Computing, pp. 232–251. Springer, 1994.
20. J.F. Groote and A. Ponse. Syntax and semantics of μCRL. In *Proc. ACP'94*, Workshops in Computing, pp. 26–62. Springer, 1995.
21. J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization of parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1/2):39–72, 2001.
22. J.F. Groote and M. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, eds, *Handbook of Process Algebra*, pp. 1151–1208. Elsevier, 2001.
23. J.F. Groote and J. Springintveld. Focus points and convergent process operators: A proof strategy for protocol verification. *Journal of Logic and Algebraic Programming*, 49(1/2):31–60, 2001.
24. B.T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic.* LNCS 129, Springer, 1982.
25. G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
26. G.J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279-295, 1997.
27. B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Uppsala University, 1987.
28. B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Proc. TACAS'00*, LNCS 1785, pp. 220–234. Springer, 2000
29. R. Kaivola. Using compositional preorders in the verification of sliding window protocol. In *Proc. CAV'97*, LNCS 1254, pp. 48–59. Springer, 1997.
30. D.E. Knuth. Verification of link-level protocols. *BIT*, 21:21–36, 1981.
31. T. Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In *Proc. APN'01*, LNCS 2075, pp. 242–262. Springer, 2001.
32. J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types.* Wiley/Teubner, 1996.
33. E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol in Lotos. In *Proc. FORTE'91*, IFIP Transactions, pp. 495-510. North-Holland, 1991.
34. S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, 13(2):85–139, 1990.
35. A. Middeldorp. Specification of a sliding window protocol within the framework of process algebra. Report FVI 86-19, University of Amsterdam, 1986.
36. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
37. S. Owre, J.M. Rushby, and N. Shankar. PVS: a prototype verification system. In *Proc. CADE'92*, LNCS 607, pp. 748–752. Springer, 1992
38. K. Paliwoda and J.W. Sanders. An incremental specification of the sliding-window protocol. *Distributed Computing*, 5:83–94, 1991.

39. D.M.R. Park. Concurrency and automata on infinite sequences. In *Proc. 5th GI Conference*, LNCS 104, pp. 167–183. Springer, 1981.

40. J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in Xesar of the sliding window protocol. In *Proc. PSTV'87*, pp. 235–248. North-Holland, 1987.

41. C. Röckl and J. Esparza. Proof-checking protocols using bisimulations. In *Proc. CONCUR'99*, LNCS 1664, pp. 525–540. Springer, 1999.

42. V. Rusu. Verifying a sliding-window protocol using PVS. In *Proc. FORTE'01*, Conference Proceedings 197, pp. 251-268. Kluwer, 2001.

43. A.A. Schoone. *Assertional Verification in Distributed Computing.* PhD thesis, Utrecht University, 1991.

44. M.A. Smith and N. Klarlund. Verification of a sliding window protocol using IOA and MONA. In *Proc. FORTE/PSTV'00*, pp. 19–34. Kluwer, 2000.

45. J.L.A. van de Snepscheut. The sliding window protocol revisited. *Formal Aspects of Computing*, 7(1):3–170, 1995.

46. K. Stahl, K. Baukus, Y. Lakhnech, and M. Steffen. Divide, abstract, and model-check. In *Proc. SPIN'99*, LNCS 1680, pp. 57–76. Springer, 1999.

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

48. A.S. Tanenbaum. *Computer Networks.* Prentice Hall, 1981.

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

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