Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Mechanical Verification of a Two-Way Sliding Window Protocol

Bahareh BADBAN [a,1], Wan FOKKINK [b] and Jaco VAN DE POL [c]
[a] *University of Konstanz, Department of Computer and Information Science*
badban@inf.uni-konstanz.de
[b] *Vrije Universiteit Amsterdam, Department of Computer Science*
wanf@cs.vu.nl
[c] *University of Twente, Department of EEMCS, Formal Methods and Tools*
j.c.vandepol@ewi.utwente.nl

Abstract. We prove the correctness of a two-way sliding window protocol with piggybacking, where the acknowledgements of the latest received data are attached to the next data transmitted back into the channel. The window sizes of both parties are considered to be finite, though they can be different. We show that this protocol is equivalent (branching bisimilar) to a *pair of FIFO* queues of finite capacities. The protocol is first modeled and manually proved for its correctness in the process algebraic language of μCRL. We use the theorem prover PVS to formalize and mechanically prove the correctness of the protocol. This implies both safety and liveness (under the assumption of fairness).

Keywords. μCRL, FIFO queues, PVS, sliding window protocols.

Introduction

A sliding window protocol [7] (SWP) ensures successful transmission of messages from a sender to a receiver through a medium in which messages may get lost. Its main characteristic is that the sender does not wait for incoming acknowledgments before sending next messages, for optimal use of bandwidth. Many data communication systems include a SWP, in one of its many variations.

In SWPs, both the sender and the receiver maintain a buffer. We consider a *two-way* SWP, in which both parties can both send and receive data elements from each other. One way of achieving full-duplex data transmission is to have two separate communication channels and use each one for simplex data traffic (in different directions). Then there are two separate physical circuits, each with a forward channel (for data) and a reverse channel (for acknowledgments). In both cases the bandwidth of the reverse channel is almost entirely wasted. In effect, the user is paying for two circuits but using the capacity of one. A better idea is to use the same circuit in both directions. Each party maintains two buffers, for storing the two opposite data streams. In this two-way version of the SWP, an acknowledgment that is sent from one party to the other may get a free ride by attaching it to a data element. This method for efficiently passing acknowledgments and data elements through a channel in the same direction, which is known as *piggybacking*, is used broadly in transmission control protocols,

[1] Corresponding Author: *Bahareh Badban, University of Konstanz, Department of Computer and Information Science, Universitätsstr. 10, P.O. Box D67, 78457 Konstanz, Germany*. Tel.: +49 (0)7531 88 4079; Fax: +49 (0)7531 88 3577; E-mail: badban@inf.uni-konstanz.de.

see [39]. The main advantage of piggybacking is a better use of available bandwidth. The extra acknowledgment field in the data frame costs only a few bits, whereas a separate acknowledgment would need a header and a checksum. In addition, fewer frames sent means fewer 'frame arrived' interrupts.

The current paper builds on a verification of a one-way version of the SWP in [1, 9]. The protocol is specified in μCRL [15], which is a language based on process algebra and abstract data types. The verification is formalized in the theorem prover PVS [29]. The correctness proof is based on the so-called *cones and foci* method [10, 18], which is a symbolic approach towards establishing a branching bisimulation relation. The starting point of the cones and foci method are two μCRL specifications, expressing the implementation and the desired external behavior of a system. A *state mapping* ϕ relates each state of the implementation to a state of the desired external behavior. Furthermore, the user must declare which states in the implementation are *focus points*, whereby each reachable state of the implementation should be able to get to a focus point by a sequence of hidden transitions, carrying the label τ. If a number of *matching criteria* are met, consisting of equations between data objects, then states s and $\phi(s)$ are branching bisimilar. Roughly, the matching criteria are: (1) if $s \xrightarrow{\tau} s'$ then $\phi(s) = \phi(s')$, (2) each transition $s \xrightarrow{a} s'$ with $a \neq \tau$ must be matched by a transition $\phi(s) \xrightarrow{a} \phi(s')$, and (3) if s is a focus point, then each transition of $\phi(s)$ must be matched by a transition of s.

The crux of the cones and foci method is that the matching criteria are formulated syntactically, in terms of relations between data terms. Thus, one obtains clear proof obligations, which can be verified with a theorem prover. The cones and foci method provides a general verification approach, which can be applied to a wide range of communication protocols and distributed algorithms.

The main motivations for the current research is to provide a mechanized correctness proof of the most complicated version of the SWP in [39], including the piggybacking mechanism. Here we model buffers (more realistically) as ordered lists, without multiple occurrences of the same index. Therefore two buffers are equal only if they are identical. That is, any swapping or repetition of elements results in a different buffer. It was mainly this shift to ordered lists without duplications (i.e. each buffer is uniquely represented with no more that once occurrence of each index), that made this verification exercise hard work. Proving that each reachable state can get to a focus point by a sequence of τ-transitions appeared to be considerably hard (mainly because communication steps of the two data streams can happen simultaneously).

The medium between the sender and the receiver is modeled as a lossy queue of capacity one. With buffers of sizes $2n_1$ and $2n_2$, and windows of sizes n_1 and n_2, respectively, we manually (paper-and-pencil) prove that the external behavior of this protocol is branching bisimilar [43] to a pair of FIFO queues of capacity $2n_1$ and $2n_2$. This implies both safety and liveness of the protocol (the latter under the assumption of fairness, which intuitively states that no message gets lost infinitely often).

The structure of the proof is as follows. First, we linearize the specification, meaning that we get rid of parallel operators. Moreover, communication actions are stripped from their data parameters. Then we eliminate modulo arithmetic, using an idea from Schoone [35]. Finally, we apply the cones and foci technique, to prove that the linear specification without modulo arithmetic is branching bisimilar to a pair of FIFO queues of capacity $2n_1$ and $2n_2$. The lemmas for the data types, the invariants, the transformations and the matching criteria have all been checked using PVS 2.3. The PVS files are available via `http://www.cs.utwente.nl/~vdpol/piggybacking.html`.

The remainder of this paper is set up as follows. In Section 1 the μCRL language is explained. In Section 2 the data types needed for specifying the protocol are presented. Section

3 features the μCRL specifications of the two-way SWP with piggybacking, and its external behavior. In Section 4, three consecutive transformations are applied to the specification of the SWP, to linearize the specification, eliminate arguments of communication actions, and get rid of modulo arithmetic. In Section 5, properties of the data types and invariants of the transformed specification are formulated; their proofs are in [2]. In Section 6, it is proved that the three transformations preserve branching bisimilarity, and that the transformed specification behaves as a pair of FIFO queues. In Section 7, we present the formalization of the verification of the SWP in PVS. We conclude the paper in Section 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 buffer and window size at the sender and the receiver. Case studies that do treat arbitrary finite buffer and window sizes mostly restrict to safety properties.

Unbounded sequence numbers Stenning [38] 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 [25] examined more general principles behind Stenning's protocol, and manually verified some safety properties. Hailpern [19] used temporal logic to formulate safety and liveness properties for Stenning's protocol, and established their validity by informal reasoning. Jonsson [22] also verified safety and liveness properties of the protocol, using temporal logic and a manual compositional verification technique. Rusu [34] used the theorem prover PVS to verify safety and liveness properties for a SWP with unbounded sequence numbers.

Fixed finite window size Vaandrager [40], Groenveld [12], van Wamel [44] and Bezem and Groote [4] manually verified in process algebra a SWP with window size one. Richier *et al.* [32] 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 [28] specified a SWP in Lotos, with the help of the simulation environment Lite, and proved some safety properties for window size six. Holzmann [20, 21] used the Spin model checker to verify safety and liveness properties of a SWP with sequence numbers up to five. Kaivola [24] verified safety and liveness properties using model checking for a SWP with window size up to seven. Godefroid and Long [11] specified a full duplex SWP in a guarded command language, and verified the protocol for window size two using a model checker based on Queue BDDs. Stahl *et al.* [37] 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 [36] specified a SWP in the high-level language IOA, and used the theorem prover MONA to verify a safety property for unbounded sequence numbers with window size up to 256. Jonsson and Nilsson [23] used an automated reachability analysis to verify safety properties for a SWP with a receiving window of size one. Latvala [26] modeled a SWP using Coloured Petri nets. A liveness property was model checked with fairness constraints for window size up to eleven.

Arbitrary finite window size Cardell-Oliver [6] specified a SWP using higher order logic, and manually proved and mechanically checked safety properties using HOL. (Van de Snepscheut [41] noted that what Cardell-Oliver claims to be a liveness property is in fact a safety property.) Schoone [35] manually proved safety properties for several SWPs using assertional verification. Van de Snepscheut [41] gave a correctness proof of a SWP as a sequence of correctness preserving transformations of a sequential program. Paliwoda and Sanders [30] specified a reduced version of what they call a SWP (but which is in fact very similar to the bakery protocol from [13]) in the process algebra CSP, and verified a safety property modulo trace semantics. Röckl and Esparza [33] verified the correctness of this bakery protocol modulo weak bisimilarity using Isabelle/HOL, by explicitly checking a bisimulation relation. Chkliaev *et al.* [8] used a timed state machine in PVS to specify a SWP with a timeout mechanism and proved some safety properties with the mechanical support of PVS; correctness is based on the timeout mechanism, which allows messages in the mediums to be reordered.

1. μCRL

μCRL [15] (see also [17]) 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 [27]. We will use $\approx$ for equality between process terms and $=$ for equality between data terms.

A μCRL specification of data types consists of two parts: a signature of function symbols from which one can build data terms, and axioms that induce an equality relation on data terms of the same type. They provide a loose semantics, meaning that it is allowed to have multiple models. The data types needed for our μCRL specification of a SWP are presented in Section 2. In particular we have the data sort of booleans $Bool$ with constants $\texttt{true}$ and $\texttt{false}$, and the usual connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$ and $\leftrightarrow$. For a boolean b, we abbreviate $b = \texttt{true}$ to b and $b = \texttt{false}$ to $\neg b$.

The process part of μCRL is specified using a number of pre-defined process algebraic operators, which we will present below. From these operators one can build process terms, which describe the order in which the atomic actions from a set $\mathcal{A}$ may happen. A process term consists of actions and recursion variables combined by the process algebraic operators. Actions and recursion variables may carry data parameters. There are two predefined actions outside $\mathcal{A}$: δ represents deadlock, and τ a hidden action. These two actions never carry data parameters.

Two elementary operators to construct processes are *sequential composition*, written $p{\cdot}q$, and *alternative composition*, written $p+q$. The process $p{\cdot}q$ first executes p, until p terminates, and then continues with executing q. The process $p+q$ non-deterministically behaves as either p or q. *Summation* $\sum_{d:D} p(d)$ provides the possibly infinite non-deterministic choice over a data type D. For example, $\sum_{n:\mathbb{N}} a(n)$ can perform the action $a(n)$ for all natural numbers n. The *conditional* construct $p \lhd b \rhd q$, with b a data term of sort $Bool$, behaves as p if b and as q if $\neg b$. *Parallel composition* $p \parallel q$ performs the processes p and q in parallel; in other words, it consists of the arbitrary interleaving of actions of the processes p and q. For example, if there is no communication possible between actions a and b, then $a \parallel b$ behaves as $(a{\cdot}b) + (b{\cdot}a)$. Moreover, actions from p and q may also synchronize to a communication action, when this is explicitly allowed by a predefined *communication function*; two actions can only synchronize if their data parameters are equal. *Encapsulation* $\partial_{\mathcal{H}}(p)$, which renames all occurrences in p of actions from the set $\mathcal{H}$ into δ, can be used to force actions into communication. For example, if actions a and b communicate to c, then $\partial_{\{a,b\}}(a \parallel b) \approx c$. *Hiding* $\tau_{\mathcal{I}}(p)$ renames all occurrences in p of actions from the set $\mathcal{I}$ into τ. Finally, processes can be specified by means of recursive equations

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

where X is a recursion variable, d_i a data parameter of type D_i for $i = 1, \ldots, n$, and p a process term (possibly containing recursion variables and the parameters d_i). For example, let $X(n{:}\mathbb{N}) \approx a(n){\cdot}X(n + 1)$; then $X(0)$ can execute the infinite sequence of actions $a(0){\cdot}a(1){\cdot}a(2) \cdot \cdots$.

Definition 1 (Linear process equation) *A recursive specification is a linear process equation (LPE) if it is of the form*

$$X(d{:}D) \approx \sum_{j\in J} \sum_{e_j:E_j} a_j(f_j(d,e_j))\cdot X(g_j(d,e_j)) \lhd h_j(d,e_j) \rhd \delta$$

with J a finite index set, $f_j : D \times E_j \to D_j$, $g_j : D \times E_j \to D$, and $h_j : D \times E_j \to Bool$.

Note that an LPE does not contain parallel composition, encapsulation and hiding, and uses only one recursion variable. Groote, Ponse and Usenko [16] presented a linearization algorithm that transforms μCRL specifications into LPEs.

To each μCRL specification belongs a directed graph, called a labeled transition system. In this labeled transition system, the states are process terms, and the edges are labeled with parameterized actions. For example, given the μCRL specification $X(n{:}\mathbb{N}) \approx a(n)\cdot X(n{+}1)$, we have transitions $X(n) \xrightarrow{a(n)} X(n+1)$. Branching bisimilarity $\underline{\leftrightarrow}_b$ [43] and strong bisimilarity $\underline{\leftrightarrow}$ [31] are two well-established equivalence relations on states in labeled transition systems.[1] Conveniently, strong bisimilarity implies branching bisimilarity. The proof theory of μCRL from [14] is sound with respect to branching bisimilarity, meaning that if $p \approx q$ can be derived from it then $p \ \underline{\leftrightarrow}_b\ q$.

Definition 2 (Branching bisimulation) *Given a labeled transition system. A* strong bisimulation relation *$\mathcal{B}$ is a symmetric binary relation on states such that if $s\,\mathcal{B}\,t$ and $s \xrightarrow{\ell} s'$, then there exists t' such that $t \xrightarrow{\ell} t'$ and $s'\,\mathcal{B}\,t'$. Two states s and t are* strongly bisimilar, *denoted by $s \ \underline{\leftrightarrow}\ t$, if there is a strong bisimulation relation $\mathcal{B}$ such that $s\,\mathcal{B}\,t$.*

A strong and branching bisimulation relation *$\mathcal{B}$ is a symmetric binary relation on states such that if $s\,\mathcal{B}\,t$ and $s \xrightarrow{\ell} s'$, then*

- *either $\ell = \tau$ and $s'\,\mathcal{B}\,t$;*
- *or there is a sequence of (zero or more) τ-transitions $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} \hat{t}$ such that $s\,\mathcal{B}\,\hat{t}$ and $\hat{t} \xrightarrow{\ell} t'$ with $s'\,\mathcal{B}\,t'$.*

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

See [42] for a lucid exposition on why branching bisimilarity constitutes a sensible equivalence relation for concurrent processes.

The goal of this section is to prove that the initial state of the forthcoming μCRL specification of a two-way SWP is branching bisimilar to a pair of FIFO queues. In the proof of this fact, in Section 6, we will use three proof techniques to derive that two μCRL specifications are branching (or even strongly) bisimilar: invariants, bisimulation criteria, and cones and foci. An *invariant* $I : D \to Bool$ [5] characterizes the set of reachable states of an LPE $X(d{:}D)$. That is, if $I(d) = \texttt{true}$ and X can evolve from d to d' in zero or more transitions, then $I(d') = \texttt{true}$.

Definition 3 (Invariant) *$I : D \to Bool$ is an invariant for an LPE in Definition 1 if for all $d{:}D$, $j \in J$ and $e_j{:}E_j$. $(I(d) \wedge h_j(d,e_j)) \ \to\ I(g_j(d,e_j))$.*

If I holds in a state d and $X(d)$ can perform a transition, meaning that $h_j(d,e_j) = \texttt{true}$ for some $e_j{:}E$, then it is ensured by the definition above that I holds in the resulting state $g_j(d,e_j)$.

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

Bisimulation criteria rephrase the question whether $X(d)$ and $Y(d')$ are strongly bisimilar in terms of data equalities, where $X(d{:}D)$ and $Y(d'{:}D')$ are LPEs. A *state mapping* ϕ relates each state in $X(d)$ to a state in $Y(d')$. If a number of bisimulation criteria are satisfied, then ϕ establishes a strong bisimulation relation between terms $X(d)$ and $Y(\phi(d))$.

Definition 4 (Bisimulation criteria) *Given two LPEs,*

$$\begin{aligned} X(d{:}D) &\approx \textstyle\sum_{j\in J} \sum_{e_j:E_j} a_j(f_j(d,e_j)){\cdot}X(g_j(d,e_j)) \lhd h_j(d,e_j) \rhd \delta \\ Y(d'{:}D') &\approx \textstyle\sum_{j\in J} \sum_{e'_j:E'_j} a_j(f'_j(d',e'_j)){\cdot}X(g'_j(d',e'_j)) \lhd h'_j(d',e'_j) \rhd \delta \end{aligned}$$

and an invariant $I : D \to Bool$ for X. A state mapping $\phi : D \to D'$ *and* local mappings $\psi_j : E_j \to E'_j$ *for $j \in J$ satisfy the* bisimulation criteria *if for all states $d \in D$ in which invariant I holds:*

I $\forall j{\in}J\, \forall e_j{:}E_j\ (h_j(d,e_j) \leftrightarrow h'_j(\phi(d),\psi_j(e_j)))$,
II $\forall j{\in}J\, \forall e_j{:}E_j\ (h_j(d,e_j) \wedge I(d)) \to (a_j(f_j(d,e_j)) = a_j(f'_j(\phi(d),\psi_j(e_j))))$,
III $\forall j{\in}J\, \forall e_j{:}E_j\ (h_j(d,e_j) \wedge I(d)) \to (\phi(g_j(d,e_j)) = g'_j(\phi(d),\psi_j(e_j)))$.

Criterion I expresses that at each summand i, the corresponding guard of X holds if and only if the corresponding guard of Y holds with parameters $(\phi(d),\psi_j(e_j))$. Criterion II (III) states that at any summand i, the corresponding action (next state, after applying ϕ on it) of X could be equated to the corresponding action (next state) of Y with parameters $(\phi(d),\psi_j(e_j))$.

Theorem 5 (Bisimulation criteria) *Given two LPEs $X(d{:}D)$ and $Y(d'{:}D')$ written as in Definition 4, and $I : D \to Bool$ an invariant for X. Let $\phi : D \to D'$ and $\psi_j : E_j \to E'_j$ for $j \in J$ satisfy the bisimulation criteria in Definition 4. Then $X(d) \underline{\leftrightarrow} Y(\phi(d))$ for all $d \in D$ in which I holds.*

This theorem has been proved in PVS. The proof is available at `http://www.cs.utwente.nl/~vdpol/piggybacking.html`.

The *cones and foci* method from [10, 18] rephrases the question whether $\tau_{\mathcal{I}}(X(d))$ and $Y(d')$ are branching bisimilar in terms of data equalities, where $X(d{:}D)$ and $Y(d'{:}D')$ are LPEs, and the latter LPE does not contain actions from some set $\mathcal{I}$ of internal actions. A *state mapping* ϕ relates each state in $X(d)$ to a state in $Y(d')$. Furthermore, some $d{:}D$ are declared to be *focus points*. The *cone* of a focus point consists of the states in $X(d)$ that can reach this focus point by a string of actions from $\mathcal{I}$. It is required that each reachable state in $X(d)$ is in the cone of a focus point. If a number of *matching criteria* are satisfied, then ϕ establishes a branching bisimulation relation between terms $\tau_{\mathcal{I}}(X(d))$ and $Y(\phi(d))$.

Definition 6 (Matching criteria) *Given two LPEs:*

$$\begin{aligned} X(d{:}D) &\approx \textstyle\sum_{j\in J} \sum_{e_j:E_j} a_j(f_j(d,e_j)){\cdot}X(g_j(d,e_j)) \lhd h_j(d,e_j) \rhd \delta \\ Y(d'{:}D') &\approx \textstyle\sum_{\{j\in J \mid a_j \notin \mathcal{I}\}} \sum_{e_j:E_j} a_j(f'_j(d',e_j)){\cdot}Y(g'_j(d',e_j)) \lhd h'_j(d',e_j) \rhd \delta \end{aligned}$$

Let FC: $D \to Bool$ be a predicate which designates the focus points, and $\mathcal{I} \subset \{a_j \mid j \in J\}$. A state mapping $\phi : D \to D'$ satisfies the matching criteria *for $d{:}D$ if for all $j \in J$ with $a_j \notin \mathcal{I}$ and all $k \in J$ with $a_k \in \mathcal{I}$:*

I. $\forall e_k{:}E_k\ (h_k(d,e_k) \to \phi(d) = \phi(g_k(d,e_k)))$;
II. $\forall e_j{:}E_j\ (h_j(d,e_j) \to h'_j(\phi(d),e_j))$;
III $FC(d) \to \forall e_j{:}E_j\ (h'_j(\phi(d),e_j) \to h_j(d,e_j))$;
IV $\forall e_j{:}E_j\ (h_j(d,e_j) \to f_j(d,e_j) = f'_j(\phi(d),e_j))$;
V $\forall e_j{:}E_j\ (h_j(d,e_j) \to \phi(g_j(d,e_j)) = g'_j(\phi(d),e_j))$.

Matching criterion I requires that the internal transitions at d are inert, meaning that d and $g_k(d, e_k)$ are branching bisimilar. Criteria II, IV and V express that each external transition of d can be simulated by $\phi(d)$. Finally, criterion III expresses that if d is a focus point, then each external transition of $\phi(d)$ can be simulated by d.

Theorem 7 (Cones and foci) *Given LPEs $X(d{:}D)$ and $Y(d'{:}D')$ written as in Definition 6. Let $I : D \to Bool$ be an invariant for X. Suppose that for all $d{:}D$ with $I(d)$:*

1. *$\phi : D \to D'$ satisfies the matching criteria for d; and*
2. *there is a $\hat{d}{:}D$ such that $FC(\hat{d})$ and X can perform transitions $d \xrightarrow{c_1} \cdots \xrightarrow{c_k} \hat{d}$ with $c_1, \ldots, c_k \in \mathcal{I}$.*

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

PVS proof of this is in [10]. For example, consider the LPEs $X(b{:}Bool) \approx a{\cdot}X(b) \lhd b \rhd \delta + c{\cdot}X(\neg b) \lhd \neg b \rhd \delta$ and $Y(d'{:}D') \approx a{\cdot}Y(d')$, with $\mathcal{I} = \{c\}$ and focus point `true`. Moreover, $X(\texttt{false}) \xrightarrow{c} X(\texttt{true})$, i.e., `false` can reach the focus point in a single c-transition. For any $d'{:}D'$, the state mapping $\phi(b) = d'$ for $b{:}Bool$ satisfies the matching criteria.

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

2. Data Types

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

Booleans. We introduce constant functions `true`, `false` of type $Bool$. $\wedge$ and $\vee$ both of type $Bool \times Bool \to Bool$ represent conjunction and disjunction operators, also $\to$ and $\leftrightarrow$ of the same exact type, denote implication and bi-implication, and $\neg : Bool \to Bool$ denotes negation. For any given sort D we consider a function *if* : $Bool \times D \times D \to D$ which functions an If-Then-Else operation, and also a mapping $eq : D \times D \to Bool$ such that $eq(d, e)$ holds if and only if $d = e$. For notational convenience we take the liberty to write $d = e$ instead of $eq(d, e)$.

Natural Numbers. 0: $\to \mathbb{N}$ denotes zero and $S{:}\mathbb{N} \to \mathbb{N}$ the successor function. The infix operations $+$, $\dot{-}$ and $\cdot$ of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ represent addition, monus (also called cut-off subtraction) and multiplication, respectively. The infix operations $\leq$, $<$, $\geq$ and $>$ of type $\mathbb{N} \times \mathbb{N} \to Bool$ are the less-than(-or-equal) and greater-than(-or-equal) operations. $|$, div of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ are modulo (some natural number) and dividing functions respectively. The rewrite rules applied over this data type, are explained in detail in Section 2 in [2].

Since the buffers at the sender and the receiver in the SWP are of finite size, modulo calculations will play an important role. $i|_n$ denotes i modulo n, while $i \; div \; n$ denotes i integer divided by n. In the proofs we will take notational liberties like omitting the sign for multiplication, and abbreviating $\neg(i = j)$ to $i{\neq}j$, $(k{<}\ell){\wedge}(\ell{<}m)$ to $k{<}\ell{<}m$, $S(0)$ to 1, and $S(S(0))$ to 2. We will also use the standard induction rule to prove some properties.

Buffers. Each party in the two-way SWP will both maintain two buffers containing the sending and the receiving window (outside these windows both buffers will be empty).

$[] :\to Buf$; *inb*, *add* : $\Delta \times \mathbb{N} \times Buf \to Buf$;
$|$, $\|$: $Buf \times \mathbb{N} \to Buf$;
smaller, *test* : $\mathbb{N} \times Buf \to Bool$; *sorted* : $Buf \to Bool$;
retrieve : $\mathbb{N} \times Buf \to \Delta$; *remove* : $\mathbb{N} \times Buf \to Buf$;

$$\begin{array}{l}
\textit{release},\ \textit{release}|_n : \mathbb{N}\times\mathbb{N}\times \textit{Buf} \to \textit{Buf};\\
\textit{next-empty},\ \textit{next-empty}|_n : \mathbb{N}\times \textit{Buf} \to \mathbb{N};\\
\textit{in-window} : \mathbb{N}\times\mathbb{N}\times\mathbb{N} \to \textit{Bool} \text{ and}\\
\textit{max} : \textit{Buf} \to \mathbb{N}
\end{array}$$

are the functions we use for buffers. And, the rewrite rules are:

$$\begin{array}{l}
\textit{add}(d,i,[]) = \textit{inb}(d,i,[])\\
\textit{add}(d,i,\textit{inb}(e,j,q)) = \textit{if}(i{>}j, \textit{inb}(e,j,\textit{add}(d,i,q)),\\
\qquad\qquad\qquad\qquad\qquad\quad \textit{inb}(d,i,\textit{remove}(i,\textit{inb}(e,j,q))))\\
[]|_n = [] \ \text{ and }\ \textit{inb}(d,i,q)|_n = \textit{inb}(d,i|_n,q|_n)\\
[]\|_n = [] \ \text{ and }\ \textit{inb}(d,i,q)\|_n = \textit{add}(d,i|_n,q\|_n)\\
\textit{smaller}(i,[]) = \texttt{true} \ \text{ and }\ \textit{smaller}(i,\textit{inb}(d,j,q)) = i<j \wedge \textit{smaller}(i,q)\\
\textit{sorted}([]) = \texttt{true} \ \text{ and }\ \textit{sorted}(\textit{inb}(d,j,q)) = \textit{smaller}(j,q) \wedge \textit{sorted}(q)\\
\textit{test}(i,[]) = \texttt{false} \ \text{ and }\ \textit{test}(i,\textit{inb}(d,j,q)) = i{=}j \vee \textit{test}(i,q)\\
\textit{retrieve}(i,\textit{inb}(d,j,q)) = \textit{if}(i{=}j, d, \textit{retrieve}(i,q))\\
\textit{remove}(i,[]) = []\\
\textit{remove}(i,\textit{inb}(d,j,q)) = \textit{if}(i{=}j, \textit{remove}(i,q), \textit{inb}(d,j,\textit{remove}(i,q)))\\
\textit{release}(i,j,q) = \textit{if}(i \geq j, q, \textit{release}(S(i),j,\textit{remove}(i,q)))\\
\textit{release}|_n(i,j,q) = \textit{if}(i|_n{=}j|_n, q, \textit{release}|_n(S(i),j,\textit{remove}(i|_n,q)))\\
\textit{next-empty}(i,q) = \textit{if}(\textit{test}(i,q), \textit{next-empty}(S(i),q), i)\\
\textit{next-empty}|_n(i,q) = \textit{if}(\textit{next-empty}(i|_n,q) < n, \textit{next-empty}(i|_n,q),\\
\qquad\qquad\qquad\qquad\quad \textit{if}(\textit{next-empty}(0,q) < n, \textit{next-empty}(0,q), n))\\
\textit{in-window}(i,j,k) = i \leq j < k \vee k < i \leq j \vee j < k < i\\
\textit{max}([]){=}0 \ \text{ and }\ \textit{max}(\textit{inb}(d,i,q)){=}\textit{if}(i \geq \textit{max}(q), i, \textit{max}(q))
\end{array}$$

More explanation on this is in [2] Section 2.

Δ represents the set of data elements that can be communicated between the two parties. The buffers are modeled as a list of pairs (d,i) with $d{:}\Delta$ and $i{:}\mathbb{N}$, representing that cell (or sequence number) i of the buffer is occupied by datum d; cells for which no datum is specified are empty. The empty buffer is denoted by $[]$, and $\textit{inb}(d,i,q)$ is the buffer that is obtained from q by simply putting (d,i) on top of the buffer q.

add inserts data into the queue, while keeping it sorted (if the queue itself is so) and avoiding duplications. $q|_n$ is taking the sequence numbers in q of modulo n, and With $q\|_n$ the resulting buffer is further sorted out. *sorted* announces whether or not a buffer is sorted. *smaller* makes sure that the first data in the queue is having the smallest index number.

$\textit{test}(i,q)$ is `true` if and only if the ith location in q is occupied. $\textit{retrieve}(i,q)$ reveals q's ith element [2] $\textit{remove}(i,q)$ wipes the ith element out. $\textit{release}(i,j,q)$ empties ith to jth locations, where $\textit{release}|_n(i,j,q)$ does the analogous modulo n. $\textit{next-empty}(i,q)$ reveals the first empty cell in q as of i, where $\textit{next-empty}|_n(i,q)$ operates the same modulo n. $\textit{in-window}(i,j,k)$ is `true` if and only if $i \leq j \leq k \dot{-} 1$, modulo n. Finally, $\textit{max}(q)$ reports the greatest occupied place in q.

Lists. $List$ is used for the specification of the external behavior of the protocol. $\langle\rangle :\to List$, $\textit{inl} : \Delta \times List \to List$, $\textit{length} : List \to \mathbb{N}$, $top : List \to \Delta$, $tail : List \to List$, $\textit{append} : \Delta \times List \to List$, $+\!\!+ : List \times List \to List$ and $\lambda,\ \lambda' : List$ represent the functions, where $\langle\rangle$ denotes the empty list, and $\textit{inl}(d,\lambda)$ adds datum d at the top of list λ. A special datum d_0 is specified to serve as a dummy value for data parameters. $\textit{length}(\lambda)$ denotes the length of λ, $top(\lambda)$ produces the datum that resides at the top of λ, $tail(\lambda)$ is obtained by removing

[2] Note that $\textit{retrieve}(i,[])$ is undefined. One could choose to equate it to a default value in Δ, or to a fresh error element in Δ. However, with the first approach an occurrence of $\textit{retrieve}(i,[])$ might remain undetected, and the second approach would needlessly complicate the data type Δ. We prefer to work with an under-specified version of *retrieve*, which is allowed in μCRL, since data types have a loose semantics. All operations in μCRL data models, however, are total; partial operations lead to the existence of multiple models.

the top position in λ, $\mathit{append}(d, \lambda)$ adds datum d at the end of λ, and $\lambda{+\!\!+}\lambda'$ represents list concatenation. Finally, $q[i..j\rangle$ is the list containing the elements in buffer q at positions i up to but not including j. The rewrite rules which are being used are:

$$\begin{array}{l}
\mathit{length}(\langle\rangle) = 0 \text{ and } \mathit{length}(\mathit{inl}(d, \lambda)) = S(\mathit{length}(\lambda)) \\
top(\mathit{inl}(d, \lambda)) = d, \quad tail(\mathit{inl}(d, \lambda)) = \lambda \\
\mathit{append}(d, \langle\rangle) = \mathit{inl}(d, \langle\rangle) \\
\mathit{append}(d, \mathit{inl}(e, \lambda)) = \mathit{inl}(e, \mathit{append}(d, \lambda)) \\
\langle\rangle{+\!\!+}\lambda = \lambda \text{ and } \mathit{inl}(d, \lambda){+\!\!+}\lambda' = \mathit{inl}(d, \lambda{+\!\!+}\lambda') \\
q[i..j\rangle = \mathit{if}(i \geq j, \langle\rangle, \mathit{inl}(\mathit{retrieve}(i, q), q[S(i)..j\rangle))
\end{array}$$

Detailed description on this data type is written in [2] Section 2.

3. Two-Way SWP with Piggybacking

This section contains the specification of the protocol in μCRL. Figure 1 illustrates the the protocol we work on (i.e. a two-way SWP with piggybacking). In this protocol sender $(\mathbf{S}/\mathbf{R})$ stores data elements that it receives via channel A in a buffer of size $2n$, in the order in which they are received. It can send a datum, together with its sequence number in the buffer, to a receiver $\mathbf{R}/\mathbf{S}$ via a medium that behaves as a lossy queue of capacity one, represented by the medium $\mathbf{K}$ and the channels B and C. Upon receipt, the receiver 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 sender and receiver, no more than one half of each of these two buffers may be occupied at any time; these halves are called the sending and the receiving window, respectively. The receiver can pass on a datum that is located at the first cell in its window via channel D; in that case the receiving window slides forward by one cell. Furthermore, the receiver can send the sequence number of the first empty cell in (or just outside) its window as an acknowledgment to the sender via a medium that behaves as a lossy queue of capacity one, represented by the medium $\mathbf{L}$ and the channels E and F. If the sender receives this acknowledgment, its window slides forward accordingly. In a two-way SWP, data streams are in both directions, meaning that $\mathbf{S}/\mathbf{R}$ and $\mathbf{R}/\mathbf{S}$ both act as sender and receiver at the same time. In addition to this, in our protocol when a datum arrives, the receiver may either send an acknowledgment back to the channel or it might instead wait until the network layer passes on the next datum. In latter case, once this new datum is to be sent into the channel, the awaited acknowledgment can be attached to it, and hence get a free ride. This technique is known as *piggybacking*.

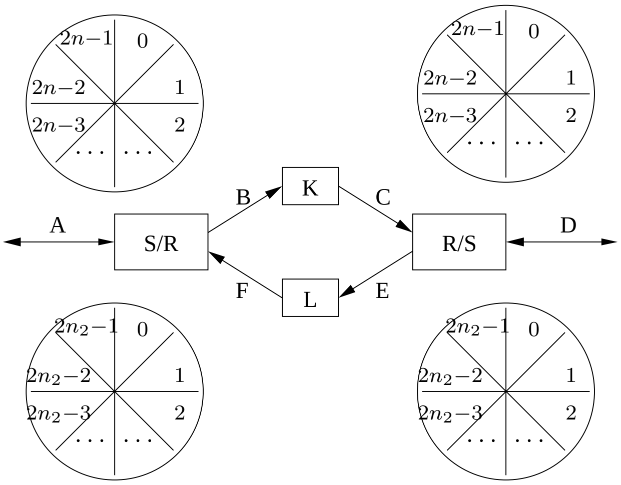

Figure 1. A two sided Sliding window protocol

3.1. Specification

The sender/receiver $\mathbf{S}/\mathbf{R}$ is modeled by the process $\mathbf{S}/\mathbf{R}(\ell, m, q, q_2', \ell_2')$, where q is its sending buffer of size $2n$, ℓ is the first cell in the window of q, and m the first empty cell in (or just outside) this window. Furthermore, q_2' is the receiving buffer of size $2n_2$, and ℓ_2' is the first cell in the window of q_2.

The μCRL specification of $\mathbf{S}/\mathbf{R}$ consists of seven clauses. The first clause of the specification expresses that $\mathbf{S}/\mathbf{R}$ can receive a datum via channel A and place it in its sending window, under the condition that this window is not yet full. The next two clauses specify that $\mathbf{S}/\mathbf{R}$ can receive a datum/acknowledgment pair via channel F; the data part is either added to q_2 if it is within the receiving window (second clause), or ignored if it is outside this window (third clause). In both clauses, q is emptied from ℓ up to but not including the received acknowledgment. The fourth clause specifies the reception of a single (i.e., non-piggybacked) acknowledgment. According to the fifth clause, data elements for transmission via channel B are taken (at random) from the filled part of the sending window; the first empty position in (or just outside) the receiving window is attached to this datum as an acknowledgment. In the sixth clause, $\mathbf{S}/\mathbf{R}$ sends a single acknowledgment. Finally, clause seven expresses that if the first cell in the receiving window is occupied, then $\mathbf{S}/\mathbf{R}$ can send this datum into channel A, after which the cell is emptied.

$$
\begin{aligned}
&\mathbf{S}/\mathbf{R}(\ell{:}\mathbb{N}, m{:}\mathbb{N}, n{:}\mathbb{N}, n_2{:}\mathbb{N}, q{:}\mathit{Buf}, q_2'{:}\mathit{Buf}, \ell_2'{:}\mathbb{N})\\
&\qquad \approx \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{S}/\mathbf{R}(\ell, S(m)|_{2n}, \mathit{add}(d,m,q), q_2', \ell_2') \lhd \mathit{in\text{-}window}(\ell, m, (\ell+n)|_{2n}) \rhd \delta\\
&\qquad + \textstyle\sum_{d:\Delta}\sum_{i:\mathbb{N}}\sum_{k:\mathbb{N}} r_{\mathrm{F}}(d,i,k)\cdot\mathbf{S}/\mathbf{R}(k, m, \mathit{release}|_{2n}(\ell,k,q), \mathit{add}(d,i,q_2'), \ell_2')\\
&\qquad\qquad\qquad\qquad \lhd \mathit{in\text{-}window}(\ell_2', i, (\ell_2'+n_2)|_{2n_2}) \rhd \delta\\
&\qquad + \textstyle\sum_{d:\Delta}\sum_{i:\mathbb{N}}\sum_{k:\mathbb{N}} r_{\mathrm{F}}(d,i,k)\cdot\mathbf{S}/\mathbf{R}(k, m, \mathit{release}|_{2n}(\ell,k,q), q_2', \ell_2')\\
&\qquad\qquad\qquad\qquad \lhd \neg\mathit{in\text{-}window}(\ell_2', i, (\ell_2'+n_2)|_{2n_2}) \rhd \delta\\
&\qquad + \textstyle\sum_{k:\mathbb{N}} r_{\mathrm{F}}(k)\cdot\mathbf{S}/\mathbf{R}(k, m, \mathit{release}|_{2n}(\ell,k,q), q_2', \ell_2')\\
&\qquad + \textstyle\sum_{k:\mathbb{N}} s_{\mathrm{B}}(\mathit{retrieve}(k,q), k, \mathit{next\text{-}empty}|_{2n_2}(\ell_2', q_2'))\cdot\mathbf{S}/\mathbf{R}(\ell, m, q, q_2', \ell_2') \lhd \mathit{test}(k,q) \rhd \delta\\
&\qquad + s_{\mathrm{B}}(\mathit{next\text{-}empty}|_{2n_2}(\ell_2', q_2'))\cdot\mathbf{S}/\mathbf{R}(\ell, m, q, q_2', \ell_2')\\
&\qquad + s_{\mathrm{A}}(\mathit{retrieve}(\ell_2', q_2'))\cdot\mathbf{S}/\mathbf{R}(\ell, m, q, \mathit{remove}(\ell_2', q_2'), S(\ell_2')|_{2n_2}) \lhd \mathit{test}(\ell_2', q_2') \rhd \delta
\end{aligned}
$$

The μCRL specification of $\mathbf{R}/\mathbf{S}$ (in [2] Appendix A) is symmetrical to the one of $\mathbf{S}/\mathbf{R}$. In the process $\mathbf{R}/\mathbf{S}(\ell_2, m_2, q_2, q', \ell')$, q' is the receiving buffer of size $2n$, and ℓ' is the first position in the window of q. Furthermore, q_2 is the sending buffer of size $2n_2$, ℓ_2 is the first position in the window of q_2, and m_2 the first empty position in (or just outside) this window.

Mediums $\mathbf{K}$ and $\mathbf{L}$, introduced below, are of capacity one. These mediums are specified in a way that they may lose frames or acknowledgments:

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

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. The *initial state* of the SWP is expressed by $\tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}/\mathbf{R}(0,0,[],[],0) \parallel \mathbf{R}/\mathbf{S}(0,0,[],[],0) \parallel \mathbf{K} \parallel \mathbf{L}))$ where the set $\mathcal{H}$ consists of the read and send actions over the internal channels B, C, E, and F, namely $\mathcal{H}{=}\{s_{\mathrm{B}}, r_{\mathrm{B}}, s_{\mathrm{C}}, r_{\mathrm{C}}, s_{\mathrm{E}}, r_{\mathrm{E}}, s_{\mathrm{F}}, r_{\mathrm{F}}\}$ while the set $\mathcal{I}$ consists of the communication actions over these internal channels together with j, namely $\mathcal{I}{=}\{c_{\mathrm{B}}, c_{\mathrm{C}}, c_{\mathrm{E}}, c_{\mathrm{F}}, j\}$.

3.2. External Behavior

Data elements that are read from channel A should be sent into channel D in the same order, and vice versa data elements that are read from channel D should be sent into channel A in the same order. No data elements should be lost. In other words, the SWP is intended to be a solution for the following linear μCRL specification, representing *a pair of* FIFO queues of capacity $2n$ and $2n_2$.

$$\begin{aligned}\mathbf{Z}(\lambda_1{:}List, \lambda_2{:}List) \approx{} & \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{Z}(\mathit{append}(d,\lambda_1),\lambda_2) \lhd \mathit{length}(\lambda_1) < 2n \rhd \delta \\ & + s_{\mathrm{D}}(\mathit{top}(\lambda_1))\cdot\mathbf{Z}(\mathit{tail}(\lambda_1),\lambda_2) \lhd \mathit{length}(\lambda_1) > 0 \rhd \delta \\ & + \textstyle\sum_{d:\Delta} r_{\mathrm{D}}(d)\cdot\mathbf{Z}(\lambda_1,\mathit{append}(d,\lambda_2)) \lhd \mathit{length}(\lambda_2) < 2n_2 \rhd \delta \\ & + s_{\mathrm{A}}(\mathit{top}(\lambda_2))\cdot\mathbf{Z}(\lambda_1,\mathit{tail}(\lambda_2)) \lhd \mathit{length}(\lambda_2) > 0 \rhd \delta\end{aligned}$$

Note that $r_{\mathrm{A}}(d)$ can be performed until the list λ_1 contains $2n$ elements, because in that situation the sending window of $\mathbf{S}/\mathbf{R}$ and the receiving window of $\mathbf{R}/\mathbf{S}$ will be filled. Furthermore, $s_{\mathrm{D}}(top(\lambda_1))$ can only be performed if λ_1 is not empty. Likewise, $r_{\mathrm{D}}(d)$ can be performed until the list λ_2 contains $2n_2$ elements, and $s_{\mathrm{A}}(top(\lambda_2))$ can only be performed if λ_2 is not empty.

4. Modifying 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.

Linearization. The starting point of our correctness proof is a linear specification M_{mod}, in which no parallel composition, encapsulation and hiding operators occur. M_{mod} can be obtained from the μCRL specification of the SWP without the hiding operator, i.e., $\partial_{\mathcal{H}}(\mathbf{S}/\mathbf{R}(0,0,[],[],0) \parallel \mathbf{R}/\mathbf{S}(0,0,[],[],0) \parallel \mathbf{K} \parallel \mathbf{L})$ by means of the linearization algorithm presented in [16]; and according to [16], the following result can be obtained:

Proposition 8 $\partial_{\mathcal{H}}(\mathbf{S}/\mathbf{R}(0,0,[],[],0) \parallel \mathbf{R}/\mathbf{S}(0,0,[],[],0) \parallel \mathbf{K} \parallel \mathbf{L}) \underline{\leftrightarrow}$
$\mathbf{M}_{mod}(0,0,[],[],0,5,0,d_0,0,5,0,d_0,0,0,0,[],[],0)$.

$\mathbf{M}_{mod}$ contains eight extra parameters: $e, e_2{:}D$ and $g, g', h, h', h_2, h_2'{:}\mathbb{N}$. Intuitively, g is 5 when medium $\mathbf{K}$ is inactive, is 4 or 2 when $\mathbf{K}$ just received a data frame or a single acknowledgment, respectively, and is 3 or 1 when $\mathbf{K}$ has decided to pass on this data frame or acknowledgment, respectively. The parameters e, h and h_2' represent the memory of $\mathbf{K}$, meaning that they can store the datum that is being sent from $\mathbf{S}/\mathbf{R}$ to $\mathbf{R}/\mathbf{S}$, the position of this datum in q, and the first empty position in the window of q_2', respectively. Initially, or when medium $\mathbf{K}$ is inactive, g, e, h and h_2' have the values 5, d_0, 0 and 0. Likewise, g' captures the five states of medium $\mathbf{L}$, and e_2, h_2 and h' represent the memory of $\mathbf{L}$.

The linear specification M_{mod} of the SWP, with encapsulation but without hiding, is written below. For the sake of presentation, in states that results after a transition we only present parameters whose values have changed. In this specification

- The first summand describes that a datum d can be received by $\mathbf{S}/\mathbf{R}$ through channel A, if q's window is not full ($\mathit{in\text{-}window}(\ell, m, (\ell+n)|_{2n})$). This datum is then placed in the first empty cell of q's window ($q{:=}\mathit{add}(d,m,q)$), and the next cell becomes the first empty cell of this window ($m{:=}S(m)|_{2n}$).
- By the 2nd summand, a frame $(\mathit{retrieve}(k,q), k, \mathit{next\text{-}empty}|_{2n_2}(\ell_2', q_2'))$ can be communicated to $\mathbf{K}$, if cell k in q's window is occupied ($\mathit{test}(k,q)$). And by the 19th summand, an acknowledgment $\mathit{next\text{-}empty}|_{2n_2}(\ell_2', q_2')$ can be communicated to $\mathbf{K}$.
- The fifth and third summand describe that medium $\mathbf{K}$ decides to pass on a frame or acknowledgment, respectively. The fourth summand describes that $\mathbf{K}$ decides to lose this frame or acknowledgment.
- The sixth and seventh summand describe that the frame in medium $\mathbf{K}$ is communicated to $\mathbf{R}/\mathbf{S}$. In the sixth summand the frame is within the window of q' ($\mathit{in\text{-}window}(\ell', h, (\ell'+n)|_{2n})$), so it is included ($q'{:=}\mathit{add}(e,h,q')$). In the seventh summand the frame is outside the window of q', so it is omitted. In both cases, the first cell of the window of q' is moved forward to h_2' ($\ell_2{:=}h_2'$), and the cells before h_2' are emptied ($q_2{:=}\mathit{release}|_{2n_2}(\ell_2, h_2', q_2)$).

- The twentieth and last summand describes that the acknowledgment in medium $\mathbf{K}$ is communicated to $\mathbf{R}/\mathbf{S}$. Then the first cell of the window of q' is moved forward to h'_2, and the cells before h'_2 are emptied.
- By the eighth summand, $\mathbf{R}/\mathbf{S}$ can send the datum at the first cell in the window of q' ($\mathit{retrieve}(\ell', q')$) through channel D, if this cell is occupied ($\mathit{test}(\ell', q')$). This cell is then emptied ($q'{:=}\mathit{remove}(\ell', q')$), and the first cell of the window of q' is moved forward by one ($\ell'{:=}S(\ell')|_{2n}$).
- Other summands are symmetric counterparts to the ones described above.

$$\begin{aligned}
&\mathbf{M}_{mod}(\ell, m, q, q'_2, \ell'_2, g, h, e, h'_2, g', h_2, e_2, h', \ell_2, m_2, q_2, q', \ell')\\
&\approx \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{M}_{mod}(m{:=}S(m)|_{2n}, q{:=}\mathit{add}(d,m,q)) \lhd \mathit{in\text{-}window}(\ell, m, (\ell+n)|_{2n}) \rhd \delta && (A1)\\
&+ \textstyle\sum_{k:\mathbb{N}} c_{\mathrm{B}}(\mathit{retrieve}(k,q), k, \mathit{next\text{-}empty}|_{2n_2}(\ell'_2, q'_2))\cdot\mathbf{M}_{mod}(g{:=}4, e{:=}\mathit{retrieve}(k,q), h{:=}k,\\
&\qquad\qquad h'_2{:=}\mathit{next\text{-}empty}|_{2n_2}(\ell'_2, q'_2)) \lhd \mathit{test}(k,q) \wedge g=5 \rhd \delta && (B1)\\
&+ j\cdot\mathbf{M}_{mod}(g{:=}1, e{:=}d_0, h{:=}0) \lhd g=2 \rhd \delta && (C1)\\
&+ j\cdot\mathbf{M}_{mod}(g{:=}5, e{:=}d_0, h{:=}0, h_2{:=}0) \lhd g=2 \vee g=4 \rhd \delta && (D1)\\
&+ j\cdot\mathbf{M}_{mod}(g{:=}3) \lhd g=4 \rhd \delta && (E1)\\
&+ c_{\mathrm{C}}(e,h,h'_2)\cdot\mathbf{M}_{mod}(\ell_2{:=}h'_2, q'{:=}\mathit{add}(e,h,q'), g{:=}5, e{:=}d_0, h{:=}0, h'_2{:=}0,\\
&\qquad\qquad q_2{:=}\mathit{release}|_{2n_2}(\ell_2, h'_2, q_2)) \lhd \mathit{in\text{-}window}(\ell', h, (\ell'+n)|_{2n}) \wedge g=3 \rhd \delta && (F1)\\
&+ c_{\mathrm{C}}(e,h,h'_2)\cdot\mathbf{M}_{mod}(\ell_2{:=}h'_2, g{:=}5, e{:=}d_0, h{:=}0, h'_2{:=}0, q_2{:=}\mathit{release}|_{2n_2}(\ell_2, h'_2, q_2))\\
&\qquad\qquad \lhd \neg\mathit{in\text{-}window}(\ell', h, (\ell'+n)|_{2n}) \wedge g=3 \rhd \delta && (G1)\\
&+ 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 && (H1)\\
&+ c_{\mathrm{E}}(\mathit{next\text{-}empty}|_{2n}(\ell', q'))\cdot\mathbf{M}_{mod}(g'{:=}2, h_2{:=}0, h'{:=}\mathit{next\text{-}empty}|_{2n}(\ell', q')) \lhd g'=5 \rhd \delta && (I1)\\
&+ j\cdot\mathbf{M}_{mod}(g'{:=}1, e_2{:=}d_0, h_2{:=}0) \lhd g'=2 \rhd \delta && (J1)\\
&+ j\cdot\mathbf{M}_{mod}(g'{:=}5, h_2{:=}0, e_2{:=}d_0, h'{:=}0) \lhd g'=2 \vee g'=4 \rhd \delta && (K1)\\
&+ j\cdot\mathbf{M}_{mod}(g'{:=}3) \lhd g'=4 \rhd \delta && (L1)\\
&+ c_{\mathrm{F}}(h')\cdot\mathbf{M}_{mod}(\ell{:=}h', q{:=}\mathit{release}|_{2n}(\ell, h', q), g'{:=}5, h_2{:=}0, e_2{:=}d_0, h'{:=}0) \lhd g'=1 \rhd \delta && (M1)\\
&+ \textstyle\sum_{d:\Delta} r_{\mathrm{D}}(d)\cdot\mathbf{M}_{mod}(m_2{:=}S(m_2)|_{2n}, q_2{:=}\mathit{add}(d, m_2, q_2))\\
&\qquad\qquad \lhd \mathit{in\text{-}window}(\ell_2, m_2, (\ell_2+n_2)|_{2n_2}) \rhd \delta && (N1)\\
&+ \textstyle\sum_{k:\mathbb{N}} c_{\mathrm{E}}(\mathit{retrieve}(k, q_2), k, \mathit{next\text{-}empty}|_{2n}(\ell', q'))\cdot\mathbf{M}_{mod}(g'{:=}4, e_2{:=}\mathit{retrieve}(k, q_2),\\
&\qquad\qquad h_2{:=}k, h'{:=}\mathit{next\text{-}empty}|_{2n}(\ell', q')) \lhd \mathit{test}(k, q_2) \wedge g'=5 \rhd \delta && (O1)\\
&+ c_{\mathrm{F}}(e_2, h_2, h')\cdot\mathbf{M}_{mod}(\ell{:=}h', q'_2{:=}\mathit{add}(e_2, h_2, q'_2), g'{:=}5, e_2{:=}d_0, h_2{:=}0, h'{:=}0,\\
&\qquad\qquad q{:=}\mathit{release}|_{2n}(\ell, h', q)) \lhd \mathit{in\text{-}window}(\ell'_2, h_2, (\ell'_2+n_2)|_{2n_2}) \wedge g'=3 \rhd \delta && (P1)\\
&+ c_{\mathrm{F}}(e_2, h_2, h')\cdot\mathbf{M}_{mod}(\ell{:=}h', g'{:=}5, e_2{:=}d_0, h_2{:=}0, h'{:=}0, q{:=}\mathit{release}|_{2n}(\ell, h', q))\\
&\qquad\qquad \lhd \neg\mathit{in\text{-}window}(\ell'_2, h_2, (\ell'_2+n_2)|_{2n_2}) \wedge g'=3 \rhd \delta && (Q1)\\
&+ s_{\mathrm{A}}(\mathit{retrieve}(\ell'_2, q'_2))\cdot\mathbf{M}_{mod}(\ell'_2{:=}S(\ell'_2)|_{2n_2}, q'_2{:=}\mathit{remove}(\ell'_2, q'_2)) \lhd \mathit{test}(\ell'_2, q'_2) \rhd \delta && (R1)\\
&+ c_{\mathrm{B}}(\mathit{next\text{-}empty}|_{2n_2}(\ell'_2, q'_2))\cdot\mathbf{M}_{mod}(g{:=}2, h{:=}0, h'_2{:=}\mathit{next\text{-}empty}|_{2n_2}(\ell'_2, q'_2)) \lhd g=5 \rhd \delta && (S1)\\
&+ c_{\mathrm{C}}(h'_2)\cdot\mathbf{M}_{mod}(\ell_2{:=}h'_2, q_2{:=}\mathit{release}|_{2n_2}(\ell_2, h'_2, q_2), g{:=}5, h{:=}0, e{:=}d_0, h'_2{:=}0) \lhd g=1 \rhd \delta && (T1)
\end{aligned}$$

$\mathbf{N}_{mod}$: No Communication Action's Arguments.

The linear specification $\mathbf{N}_{mod}$ (Written in [2] Appendix A) is obtained from $\mathbf{M}_{mod}$ by renaming all arguments from communication actions (e.g. $c_{\mathrm{F}}(e_2, h_2, h')$) to a fresh action c. Since we want to show that the "external" behavior of this protocol is branching bisimilar to *a pair of* FIFO queues (of capacity $2n$ and $2n_2$), the internal actions can be removed. The following proposition is then a trivial result of this renaming:

Proposition 9 $\tau_{\mathcal{I}}(\mathbf{M}_{mod}(0,0,[],[],0,5,0,d_0,0,5,0,d_0,0,0,0,[],[],0)) \underline{\leftrightarrow}$ $\tau_{\{c,j\}}(\mathbf{N}_{mod}(0,0,[],[],0,5,0,d_0,0,5,0,d_0,0,0,0,[],[],0))$.

$\mathbf{N}_{nonmod}$: **No Modulo Arithmetic.**

The specification of $\mathbf{N}_{nonmod}$ is obtained by eliminating all occurrences of $|_{2n}$ (resp. $|_{2n_2}$) from $\mathbf{N}_{mod}$, and replacing all guards of the form $\mathit{in\text{-}window}(i,j,(i+k)|_{2n})$ (respectively $\mathit{in\text{-}window}(i,j,(i+k)|_{2n_2})$) with $i{\leq}j{<}i+n$ (respectively $i{\leq}j{<}i+n_2$). According to what just mentioned, only $A1$, $F1$, $G1$, $N1$, $P1$ and $Q1$ whose guards are of this form, will be subjected to change. We name each new clause after its corresponding one by removing the index 1 from it, that is e.g. $A1$ will become A, and so forth. As an example we show this clause below, the whole specification of $\mathbf{N}_{nonmod}$ is in [2] Appendix A.

$$\textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d){\cdot}\mathbf{N}_{nonmod}(m{:=}S(m), q{:=}\mathit{add}(d,m,q)) \lhd l < m < \ell{+}n \rhd \delta \qquad (A)$$

In Section 6.1, we will prove that $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$ are strongly bisimilar. In order to demonstrate the correctness of $\mathbf{N}_{nonmod}$ (see Section 6.2) there will be a number of properties on the Data Types which should be investigated first. In the next section we list these properties, and thereafter, in its following section, we will prove the correctness.

5. Properties of Data Types

This section presents some properties of the data types and the *ordered* buffers, also some invariants of the final specification of the system; all proofs are in [2] Appendix B.

5.1. Basic Properties

These properties contain some mathematical reasoning over the functions in our specification of the system, with/without modulo arithmetic. One of them for example is: $\mathit{test}(k,q) \;\rightarrow\; \mathit{add}(\mathit{retrieve}(k,q),k,q)[i..j\rangle = q[i..j\rangle$. The entire list is in [2] Appendix B.1.

5.2. Ordered Buffers

Lemma 10 *Some properties on add(.,.) function:*

1. $\mathit{test}(i,q) \;\rightarrow\; \mathit{test}(i,\mathit{add}(d,j,q))$
2. $\mathit{next\text{-}empty}(i,\mathit{add}(d,j,q)) \geq \mathit{next\text{-}empty}(i,q)$
3. $\mathit{test}(i,\mathit{add}(d,j,q)) = (i{=}j \vee \mathit{test}(i,q))$
4. $\mathit{retrieve}(i,\mathit{add}(d,j,q)) = \mathit{if}(i{=}j,d,\mathit{retrieve}(i,q))$
5. $\mathit{remove}(i,\mathit{add}(d,i,q)) = \mathit{remove}(i,q)$
6. $j \neq \mathit{next\text{-}empty}(i,q) \rightarrow \mathit{next\text{-}empty}(i,\mathit{add}(d,j,q)) = \mathit{next\text{-}empty}(i,q)$
7. $\mathit{next\text{-}empty}(i,\mathit{add}(d,\mathit{next\text{-}empty}(i,q),q)) = \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(i,q)),q)$
8. $i < j \;\rightarrow\; \mathit{remove}(i,\mathit{add}(d,j,q)) = \mathit{add}(d,j,\mathit{remove}(i,q))$
9. $i \neq j \;\rightarrow\; \mathit{add}(e,i,\mathit{add}(d,j,q)) = \mathit{add}(d,j,\mathit{add}(e,i,q))$

Lemma 11 *Ordered buffers maintain the following properties:*

1. $\mathit{smaller}(i,q) \;\rightarrow\; \mathit{smaller}(i,\mathit{remove}(j,q))$
2. $i < j \wedge \mathit{smaller}(i,q) \;\rightarrow\; \mathit{smaller}(i,\mathit{add}(d,j,q))$
3. $\mathit{smaller}(i,q) \;\rightarrow\; \mathit{remove}(i,q) = q$
4. $i < j \wedge \mathit{smaller}(j,q) \;\rightarrow\; \mathit{smaller}(i,q)$
5. $\mathit{sorted}(q) \;\rightarrow\; \mathit{sorted}(\mathit{add}(d,i,q))$
6. $\mathit{smaller}(i,q) \;\rightarrow\; \mathit{add}(d,i,q) = \mathit{inb}(d,i,q)$
7. $\mathit{sorted}(q) \wedge j < i \;\rightarrow\; \mathit{remove}(i,\mathit{add}(d,j,q)) = \mathit{add}(d,j,\mathit{remove}(i,q))$
8. $\mathit{sorted}(q) \;\rightarrow\; \mathit{add}(d,i,q) = \mathit{add}(d,i,\mathit{remove}(i,q))$

Lemma 12 *For $n > 0$, the following results hold on $q\|_n$.*

1. $\mathit{sorted}(q\|_n)$
2. $\mathit{test}(i, q|_n) = \mathit{test}(i, q\|_n)$
3. $\mathit{retrieve}(i|_n, q|_n) = \mathit{retrieve}(i|_n, q\|_n)$
4. $j \neq i \;\rightarrow\; \mathit{remove}(i, \mathit{add}(d, j, q\|_n)) = \mathit{add}(d, j, \mathit{remove}(i, q\|_n))$
5. $\forall j{:}\mathbb{N}(\mathit{test}(j, q) \rightarrow i \leq j < i + n) \wedge i \leq k \leq i + n \rightarrow$
   $\mathit{next\text{-}empty}|_{2n}(k|_{2n}, q|_{2n}) = \mathit{next\text{-}empty}|_{2n}(k|_{2n}, q\|_{2n})$
6. $\forall j{:}\mathbb{N}(\mathit{test}(j, q) {\rightarrow} i{\leq}j{<}i + n) \wedge i{\leq}k{\leq}i + n \rightarrow \mathit{remove}(k, q)\|_{2n} = \mathit{remove}(k|_{2n}, q\|_{2n})$
7. $\forall j{:}\mathbb{N}(\mathit{test}(j, q) {\rightarrow} i{\leq}j{<}i + n) \wedge i{\leq}k{\leq}i + n \rightarrow \mathit{release}(i, k, q)\|_{2n}{=}$
   $\mathit{release}|_{2n}(i|_{2n}, k|_{2n}, q\|_{2n})$
8. $\forall j{:}\mathbb{N}(\mathit{test}(j, q) {\rightarrow} i{\leq}j{<}i + n){\wedge}i{\leq}k{\leq}i + n \rightarrow \mathit{add}(d, k, q)\|_{2n} = \mathit{add}(d, k|_{2n}, q\|_{2n})$

All the abovementioned lemmas are proved in detail in [2] Appendix B.2.

5.3. Invariants

Invariants of a system are properties of data that are satisfied throughout the reachable state space of the system (see Definition 3). Lemma 13 collects 19 invariants of $\mathbf{N}_{nonmod}$ (and their symmetric counterparts). Occurrences of variables $i, j{:}\mathbb{N}$ in an invariant are always implicitly universally quantified at the outside of the invariant.

Invariants 6, 8, 15 and 17 are only needed in the derivation of other invariants. We provide some intuition for the (first of each pair of) invariants that will be used in the correctness proofs in Section 6 and in the derivations of the data lemmas. Invariants 4, 11, 12, 13 express that the sending window of $\mathbf{S}/\mathbf{R}$ is filled from ℓ up to but not including m, and that it has size n. Invariants 7, 10 express that the receiving window of $\mathbf{R}/\mathbf{S}$ starts at ℓ' and stops at $\ell'{+}n$. Invariant 2 expresses that **S/R** cannot receive acknowledgments beyond *next-empty*(ℓ', q'), and Invariant 9 that **R/S** cannot receive frames beyond $m \dot{-} 1$. Invariants 16, 18, 19 are based on the fact that the sending window of **S/R**, the receiving window of **R/S**, and **K** (when active) coincide on occupied cells and frames with the same sequence number. Invariants 1, 3, 5 and 14 give bounds on the parameters h and h' of mediums **K** and **L**.

Lemma 13 $\mathbf{N}_{nonmod}(\ell, m, q, q'_2, \ell'_2, g, h, e, h'_2, g', h_2, e_2, h', \ell_2, m_2, q_2, q', \ell')$ *satisfies the following invariants.*

1. $h' \leq \mathit{next\text{-}empty}(\ell', q')$ $\;\texttt{and}\;$ $h'_2 \leq \mathit{next\text{-}empty}(\ell'_2, q'_2)$
2. $\ell \leq \mathit{next\text{-}empty}(\ell', q')$ $\;\texttt{and}\;$ $\ell_2 \leq \mathit{next\text{-}empty}(\ell'_2, q'_2)$
3. $g' \neq 5 \;\rightarrow\; \ell \leq h'$ $\;\texttt{and}\;$ $g \neq 5 \;\rightarrow\; \ell_2 \leq h'_2$
4. $\mathit{test}(i, q) \;\rightarrow\; i < m$ $\;\texttt{and}\;$ $\mathit{test}(i, q_2) \;\rightarrow\; i < m_2$
5. $(g = 3 \vee g = 4) \;\rightarrow\; h < m$ $\;\texttt{and}\;$ $(g' = 3 \vee g' = 4) \;\rightarrow\; h_2 < m_2$
6. $\mathit{test}(i, q') \;\rightarrow\; i < m$ $\;\texttt{and}\;$ $\mathit{test}(i, q'_2) \;\rightarrow\; i < m_2$
7. $\mathit{test}(i, q') \;\rightarrow\; \ell' \leq i < \ell' + n$ $\;\texttt{and}\;$ $\mathit{test}(i, q'_2) \;\rightarrow\; \ell'_2 \leq i < \ell'_2 + n_2$
8. $\ell' \leq m$ $\;\texttt{and}\;$ $\ell'_2 \leq m_2$
9. $\mathit{next\text{-}empty}(\ell', q') \leq m$ $\;\texttt{and}\;$ $\mathit{next\text{-}empty}(\ell'_2, q'_2) \leq m_2$
10. $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$ $\;\texttt{and}\;$ $\mathit{next\text{-}empty}(\ell'_2, q'_2) \leq \ell'_2 + n_2$
11. $\mathit{test}(i, q) \;\rightarrow\; \ell \leq i$ $\;\texttt{and}\;$ $\mathit{test}(i, q_2) \;\rightarrow\; \ell_2 \leq i$
12. $\ell \leq i < m \;\rightarrow\; \mathit{test}(i, q)$ $\;\texttt{and}\;$ $\ell_2 \leq i < m_2 \;\rightarrow\; \mathit{test}(i, q_2)$
13. $m \leq \ell + n$ $\;\texttt{and}\;$ $m_2 \leq \ell_2 + n_2$
14. $(g = 3{\vee}g = 4) \rightarrow \mathit{next\text{-}empty}(\ell', q') \leq h + n$ $\;\texttt{and}$
    $(g' = 3{\vee}g' = 4) \rightarrow \mathit{next\text{-}empty}(\ell'_2, q'_2) \leq h_2 + n_2$
15. $\ell' \leq i < h' \;\rightarrow\; \mathit{test}(i, q')$ $\;\texttt{and}\;$ $\ell'_2 \leq i < h'_2 \;\rightarrow\; \mathit{test}(i, q'_2)$
16. $(g = 3 \vee g = 4) \wedge \mathit{test}(h, q) \;\rightarrow\; \mathit{retrieve}(h, q) = e$ $\;\texttt{and}$
    $(g' = 3 \vee g' = 4) \wedge \mathit{test}(h_2, q_2) \;\rightarrow\; \mathit{retrieve}(h_2, q_2) = e_2$

17. $(\mathit{test}(i,q) \wedge \mathit{test}(i,q')) \;\rightarrow\; \mathit{retrieve}(i,q) = \mathit{retrieve}(i,q')$ and
    $(\mathit{test}(i,q_2) \wedge \mathit{test}(i,q_2')) \;\rightarrow\; \mathit{retrieve}(i,q_2) = \mathit{retrieve}(i,q_2')$
18. $((g=3 \vee g=4) \wedge \mathit{test}(h,q')) \;\rightarrow\; \mathit{retrieve}(h,q') = e$ and
    $((g'=3 \vee g'=4) \wedge \mathit{test}(h_2,q_2')) \;\rightarrow\; \mathit{retrieve}(h_2,q_2') = e_2$
19. $(\ell \leq i \wedge j \leq \mathit{next\text{-}empty}(i,q')) \;\rightarrow\; q[i..j\rangle = q'[i..j\rangle$ and
    $(\ell_2 \leq i \wedge j \leq \mathit{next\text{-}empty}(i,q_2')) \;\rightarrow\; q_2[i..j\rangle = q_2'[i..j\rangle$

In the initial state $\mathrm{N}_{nonmod}(0,0,[],[],0,5,\,0,d_0,0,5,0,d_0,0,0,0,[],[],0)$ all these invariants are satisfied. Also, all invariants are preserved by all summands. So they are satisfied in all reachable states of N_{nonmod}. For a proof of this lemma see [2] Appendix B.3.

6. Correctness of N_{mod}

In Section 6.1, we establish the strong bisimilarity of N_{mod} and N_{nonmod}. In order to prove this, we show that the bisimulation criteria in Definition 4 hold. Then according to Theorem 5, proof is complete. Section 6.2 demonstrates that N_{nonmod} behaves like *a pair of* FIFO queues. Finally, the correctness of the two-way SWP is established in Section 6.3.

6.1. *Equality of* N_{mod} *and* N_{nonmod}

Proposition 14 $\mathrm{N}_{nonmod}(0,0,[],[],0,5,0,d_0,0,5,0,d_0,0,0,0,[],[],0) \;\underline{\leftrightarrow}$
$\mathrm{N}_{mod}(0,0,[],[],0,5,0,d_0,0,5,0,d_0,0,0,0,[],[],0).$

Proof. By Theorem 5, it suffices to define a state mapping ϕ and local mappings ψ_j for $j = 1,2,\ldots,20$ that satisfy the bisimulation criteria in Definition 4, with respect to the invariants in Lemma 13.

Let Ξ abbreviate $\mathbb{N}\times\mathbb{N}\times \mathit{Buf}\times \mathit{Buf}\times\mathbb{N}\times\mathbb{N}\times\mathbb{N}\times\Delta\times\mathbb{N}\times\mathbb{N}\times\mathbb{N}\times\Delta\times\mathbb{N}\times\mathbb{N}\times\mathbb{N}\times \mathit{Buf}\times \mathit{Buf}\times\mathbb{N}$. We use $\xi{:}\Xi$ to abbreviate $(\ell,m,q,q_2',\ell_2',g,h,e,h_2',g',h_2,e_2,h',\ell_2,m_2,q_2,q',\ell')$, then we define $\phi:\Xi\rightarrow\Xi$ by:

$$\begin{aligned}\phi(\xi) = (&\ell|_{2n}, m|_{2n}, q\|_{2n}, q_2'\|_{2n_2}, \ell_2'|_{2n_2}, g, h|_{2n}, e, h_2'|_{2n_2},\\ &g', h_2|_{2n_2}, e_2, h'|_{2n}, \ell_2|_{2n_2}, m_2|_{2n_2}, q_2\|_{2n_2}, q'\|_{2n}, \ell'|_{2n})\end{aligned}$$

Furthermore, $\psi_2:\mathbb{N}\rightarrow\mathbb{N}$ maps k to $k|_{2n}$, and $\psi_{15}:\mathbb{N}\rightarrow\mathbb{N}$ maps k to $k|_{2n_2}$; the other 18 local mappings are simply the identity. We show that ϕ and the ψ_j satisfy the bisimulation criteria. For each summand, we list (and prove) the non-trivial bisimulation criteria that it induces. For a detailed proof, see [2] Appendix C. ■

6.2. *Correctness of* N_{nonmod}

We prove that N_{nonmod} is branching bisimilar to the pair of FIFO queues **Z** (see Section 3.2), using cones and foci (see Theorem 7)

The state mapping $\phi:\Xi\rightarrow \mathit{List}\times\mathit{List}$, which maps states of N_{nonmod} to states of **Z**, is defined by:

$$\phi(\xi) = (\phi_1(m,q,\ell',q'), \phi_2(m_2,q_2,\ell_2',q_2'))$$

where

$$\begin{aligned}\phi_1(m,q,\ell',q') &= q'[\ell'..\mathit{next\text{-}empty}(\ell',q')\rangle{+\!\!+}q[\mathit{next\text{-}empty}(\ell',q')..m\rangle\\ \phi_2(m_2,q_2,\ell_2',q_2') &= q_2'[\ell_2'..\mathit{next\text{-}empty}(\ell_2',q_2')\rangle{+\!\!+}q_2[\mathit{next\text{-}empty}(\ell_2',q_2')..m_2\rangle\end{aligned}$$

Intuitively, ϕ_1 collects data elements in the sending window of **S/R** and the receiving window of **R/S**, starting at the first cell in the receiving window (i.e., ℓ') until the first empty cell in this window, and then continuing in the sending window until the first empty cell in that window (i.e., m). Likewise, ϕ_2 collects data elements in the sending window of **R/S** and the receiving window of **S/R**.

The focus points are states where in the direction from **S/R** to **R/S**, either the sending window of **S/R** is empty (meaning that $\ell = m$), or the receiving window from **R/S** is full and all data elements in this receiving window have been acknowledged (meaning that $\ell = \ell' + n$). Likewise for the direction from **R/S** to **S/R**. That is, the focus condition reads

$$FC(\xi) \ := \ (\ell = m \vee \ell = \ell' + n) \wedge (\ell_2 = m_2 \vee \ell_2 = \ell'_2 + n_2)$$

Lemma 15 *For each $\xi{:}\Xi$ with $\mathbf{N}_{nonmod}(\xi)$ reachable from the initial state, there is a $\hat{\xi}{:}\Xi$ with FC($\hat{\xi}$) such that $\mathbf{N}_{nonmod}(\xi) \stackrel{c_1}{\to} \cdots \stackrel{c_n}{\to} \mathbf{N}_{nonmod}(\hat{\xi})$, where $c_1, \ldots, c_n \in \mathcal{I}$.*

Proof. We prove (see [2] Appendix C) that for each $\xi{:}\Xi$ where the invariants in Lemma 13 hold, there is a finite sequence of internal actions which ends in a state where $(\ell = m \vee \ell = \ell' + n) \wedge (\ell_2 = m_2 \vee \ell_2 = \ell'_2 + n_2)$. ■

Proposition 16 $\tau_{\{c,j\}}(\mathbf{N}_{nonmod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)) \mathrel{\underline{\leftrightarrow}_b} \mathbf{Z}(\langle\rangle, \langle\rangle)$.

Proof. We prove this using cones and foci method. See [2] Appendix C. ■

6.3. Correctness of the Two-Way Sliding Window Protocol

Finally, we can prove the main result of our specification which is:

Theorem 17 (Correctness)
$\tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S/R}(0, 0, [], [], 0) \parallel \mathbf{R/S}(0, 0, [], [], 0) \parallel \mathbf{K} \parallel \mathbf{L})) \mathrel{\underline{\leftrightarrow}_b} \mathbf{Z}(\langle\rangle, \langle\rangle)$

Proof. We combine the equivalences that have been obtained so far:

$$\begin{array}{lll}
 & \tau_I(\partial_H(\mathbf{S/R}(0, 0, [], [], 0) \parallel \mathbf{K} \parallel \mathbf{R/S}(0, 0, [], [], 0) \parallel \mathbf{L})) & \\
\underline{\leftrightarrow} & \tau_I(\mathbf{M}_{mod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)) & \text{(Proposition 8)} \\
\underline{\leftrightarrow} & \tau_{\{c,j\}}(\mathbf{N}_{mod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)) & \text{(Proposition 9)} \\
\underline{\leftrightarrow} & \tau_{\{c,j\}}(\mathbf{N}_{nonmod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)) & \text{(Proposition 14)} \\
\underline{\leftrightarrow}_b & \mathbf{Z}(\langle\rangle, \langle\rangle) & \text{(Proposition 16)}
\end{array}$$

■

7. Formalization in PVS

In this section we show the formalization and verification of the correctness proof of the SWP with piggybacking in PVS [29].

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

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

In μCRL, the semantics of a data specification is the set of all its models. Incomplete data specifications may have multiple models. Even worse, it is possible to have inconsistent data specifications for which no models exist. Here the necessity of specification with PVS emerges, because of this probable incompleteness and inconsistency which exists when working with μCRL. Moreover, PVS was used to search for omissions and errors in the manual μCRL proof of the SWP with piggybacking.

In Section 7.1 we show examples of the original specification of some data functions, then we introduce the modified forms of them. Moreover, we show how measure functions are used to detect the termination of recursive definitions. In Section 7.2 and 7.3 we represent the LPEs and invariants of the SWP with piggybacking in PVS. Section 7.4 presents the equality of μCRL specification of the SWP with piggybacking with and without modulo arithmetic. Section 7.5 explains how the cones and foci method is used to formalize the main theorem, that is the μCRL specification of the SWP with piggybacking is branching bisimilar to a FIFO queue of size $2n$. Finally, Section 7.6 is dedicated to some remarks on the verification in PVS.

7.1. Data Specifications in PVS

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

To achieve this, having total definitions is required. So in the first place, partially defined functions need to be extended to total ones. Below there are some examples of partial definitions in the original data specification of the SWP with piggybacking, which we changed into total ones. Second, to guarantee totality of recursive definitions, PVS requires the user to define a so-called *measure function*. Doing this usually requires time and effort, but the advantage is that recursive definitions are guaranteed to be well-founded. PVS enabled us to find non-terminating definitions in the original data specification of the SWP with piggybacking, which were not detected within the framework of μCRL. After finding these non-terminating definitions with PVS, we searched for new definition which can express the operation we look for. Then we replaced the old definitions with new terminating ones in our μCRL framework. Below we show some of the most interesting examples.

Example 18 *We defined a function next-empty which seeks for the first empty position in q from a given position i. This function is identified as:*

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

We also need to have next-empty$|_n(i,q)$ as a function which produces the first empty position in q modulo n, from position i. It looked reasonable to define it as:

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

Although the definition looks total and well-founded, this was one of the undetected potential errors that PVS detected during the type checking process. Below we bring an example to show what happens. Let $q = [(d_0,0),(d_1,1),(d_2,2),(d_3,3),(d_5,5)],\ n=4,\ i=5$ then

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

which will never terminate. The problem is that modulo n all the places in q are occupied, and since $0 \le i|_n < n$ hence test(i,q) will always be `true`*. Hence each position will call for its immediate next position and so on. Therefore the calls will never stop.*

At the end we replaced it with the following definition, which is terminating and operates the way as we expect.

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

```
...
D:nonempty_type
Buf:type=list[[D,nat]]
x,i,j,k,l,n: VAR nat
...
dm(i,j,n): nat =
         IF mod(i,n)<=mod(j,n)
         THEN mod(j,n)-mod(i,n)
         ELSE n+mod(j,n)-mod(i,n)
         ENDIF
...
release(n)(i,j,q): RECURSIVE Buf=
         IF mod(i,n)=mod(j,n) THEN q
         ELSE release(n)(mod(i+1,n),j,remove(mod(i,n),q))
         ENDIF
         measure dm(i,j,n)
...
```

Figure 2. An example of data specification in PVS

This function first checks whether there is any empty place after $i|_n$ (incl. $i|_n$ itself). If this is the case then that position would be the result, otherwise using next-empty$(0, q)$ it will check if there is any empty position in the buffer modulo n. If so then that position would be the value of the function since next-empty$(i|_n, q)$ will reach it. If all the buffer modulo n is full then n would be the result, because n is bigger that all the possible values for the function (i.e. $i|_n$ at most) and moreover it indicates that the buffer is full modulo n.

In [2] Appendix D there are similar examples for *release*(i, j, q) and *release*$|_n(i, j, q)$, detected errors by PVS, and also our ultimate solutions for them.

We represented the μCRL abstract data types directly by PVS types. This enables us to reuse the PVS library for definitions and theorems of "standard" data types. Figure 2 illustrates part of a PVS theory defining *release*$|_n$. There *D* is an unspecified but non-empty type which represents the set of all data elements that can be communicated between the sender and the receiver. `Buf` is list of pairs of type $D \times \mathbb{N}$ defined as $list[[D, nat]]$. Here we used *list* to identify the type of lists, which is defined in the prelude in PVS. Therefore we simply use it without any need to define it explicitly. This figure also represents *release*$|_n(i, j, q)$ in PVS. Since it is defined recursively, in order to establish its termination (or totality), it is required by PVS to have a measure function. We define a measure function called dm which is decreasing and non-recursive. Here, PVS uses its type-checker to check the validity of dm. It generates two type-check proof obligations: if $i|_n < j|_n$ then $j|_n - i|_n \geq 0$ and if $i|_n \geq j|_n$ then $n + j|_n - i|_n \geq 0$. The first proof obligation is proved in one trivial step. The second one is proved using Lemma 19.

In [2] Appendix D, we also list the extra data lemmas which had to be proved in PVS while they are considered to be trivial in the manual proof.

7.2. Representing LPEs

We now reuse [10] to show how the μCRL specification of the SWP with piggybacking (an LPE) can be represented in PVS. The main distinction will be that we have assumed so far that LPEs are *clustered*. This means that each action label occurs in at most one summand, so that the set of summands could be indexed by the set of action labels. This is no limitation, because any LPE can be transformed in clustered form, basically by replacing $+$ by $\sum$ over

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

Figure 3. Definition of LPE in PVS

finite types. Clustered LPEs enable a notationally smoother presentation of the theory. However, when working with concrete LPEs this restriction is not convenient, so we avoid it in the PVS framework: an arbitrarily sized index set $\{0, \ldots, n-1\}$ will be used, represented by the PVS type `below(n)`. A second deviation is that we will assume from now on that every summand has the same set of local variables. Again this is no limitation, because void summations can always be added (i.e. $p = \sum_{d:D} p$, when d does not occur in p). This restriction is needed to avoid the use of polymorphism, which does not exist in PVS. The third deviation is that we do not distinguish action labels from action data parameters. We simply work with one type of expressions for actions. This allows that a summand can generate transitions with various labels. This generalization makes the formalization a bit smoother, but was not really exploited.

So an LPE is parameterized by sets of actions (`Act`), global parameters (`State`) and local variables (`Local`), and by the size of its index set (`n`). Note that the guard, action and next-state of a summand depend on the global parameters $d : State$ and on local variables $e : Local$. This dependency is represented in the definition `SUMMAND` by a PVS function type. In Figure 3 an LPE consists of an initial state and a list of summands indexed by `below(n)`.

A concrete LPE by a fragment of the linear specification N_{mod} of SWP with piggybacking in PVS (see Figure 6 in Appendix D in [2]) is introduced as an `lpe` of a set of actions: `Nnonmod_act`, states: `State`, local variables: `Local`, and a digit: `20` referring to the number of summands. The LPE is identified as a pair, called `init` and `sums`, where `init` is introducing the initial state of N_{mod} and `sums` the summands. The first `LAMBDA` maps each number to the corresponding summand in N_{mod}. The second `LAMBDA` is representing the summands as functions over `State` and `Local`. Here, `State` is the set of states and `Local` is the data type $D \times \mathbb{N}$ of all pairs (d, k) of the summation variables, which is considered as a global variable regarding the property: $p = \sum_{(d,k):local} p$, which is mentioned before.

7.3. Representing Invariants

Invariants are boolean functions over the set of states. In Figure 4, we explain how to represent an invariant of the μCRL specification, in PVS. This figure illustrates the (first part of the) Invariant 13.9 from Section 5.3

7.4. Equality of N_{mod} and N_{nonmod}

Strong bisimilarity of N_{mod} and N_{nonmod} (Proposition 14) is depicted in Figure 5. `state_f` and `local_f` are introduced to construct the state mapping between N_{nonmod} and N_{mod}. In PVS we introduce the state mapping (`state_f, local_f`) from the set of states and local variables of N_{nonmod} to those of N_{mod}. Then we use the corresponding relation to this state mapping, and we show that this relation is a bisimulation between N_{nonmod} and N_{mod}.

In PVS we defined an LPE as a list of summands (not as a recursive equation), equipped with the standard LTS semantics. It could be proved directly that state mappings preserve strong bisimulation.

```
...
l,m,l12,g,h,h12,g1,h2,h1,l2,m2,l1: var nat
q,q1,q2,q12 : var Buf
e,e2: var D
...
inv(l,m,q,q12,l12,g,h,e,h12,g1,h2,e2,h1,l2,m2,q2,q1,l1): bool= next_empty(l1,q1)<=m
...
```

Figure 4. An example of representing invariants in PVS

```
...
state_f(l,m,q,q12,l12,g,h,e,h12,g1,h2,e2,h1,l2,m2,q2,q1,l1): State=
      (mod(l,2*n),mod(m,2*n),modulo2(q,2*n),modulo2(q12,2*n2),mod(l12,2*n2),
        g,mod(h,2*n),e,mod(h1,2*n2),g1,mod(h2,2*n2),e2,mod(h1,2*n),
        mod(l2,2*n2),mod(m2,2*n2),modulo2(q2,2*n2),modulo2(q1,2*n),
        mod(l1,2*n)),
local_f(l:Local,i:below(20)): Local=
      LET (e,k)=l IN
      IF i=4 THEN (e,mod(k,2*n)) ELSE (IF i=9 THEN (e,mod(k,2*n2)) ELSE(e,k)) ENDIF
...
Propsimilaosition_6_22: proposition bisimilar (lpe2lts(Nnonmod),lpe2lts(Nmod))
...
```

Figure 5. Equality of $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ in PVS

By contrast, the manual proof that $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ are strongly bisimilar is based on the proof principle CL-RSP [5], which states that each LPE has a unique solution, modulo strong bisimilarity. An advantage of this approach is that by using algebraic principles only, the stated equivalence also holds in non-standard models for process algebra + CL-RSP. We did not formalize CL-RSP in PVS because it depends on recursive process equations; this would have required a laborious embedding of μCRL in PVS, which would complicate the formalization too much.

7.5. *Correctness of* $\mathbf{N}_{mod}$

The branching bisimilarity verification of $\mathbf{N}_{mod}$ and $\mathbf{Z}$ (Theorem 17) is pictured in Figure 6. The function `fc(l,m,q,q12,l12,g,h,e,h12,g1,h2,e2,h1,l2,m2,q2,q1,l1)` defines the focus condition for $\mathbf{N}_{nonmod}(\ell, m, q, q_2', \ell_2', g, h, e, h_2', g', h_2, e_2, h', \ell_2, m_2, q_2, q', \ell')$ as a boolean function on set of states. `qlist(q,i,j)` is used to describe the function $q[i..j\rangle$, which is defined as an application on triples. The state mapping `h` maps states of $\mathbf{N}_{nonmod}$ to states of $\mathbf{Z}$, which is called $\phi : \Xi \to \mathit{List} \times \mathit{List}$ in Section 6.2. `k` is a Boolean function which is used to match each external action of $\mathbf{N}_{nonmod}$ to the corresponding one of $\mathbf{Z}$. This is done by corresponding the number of each summand of $\mathbf{N}_{nonmod}$ to one of $\mathbf{Z}$. As PVS requires, this function must be total, therefore without loss of generality we map all the summands with an internal action, from $\mathbf{N}_{nonmod}$'s specification, to the second summand of $\mathbf{Z}$'s specification.

According to cones and foci proof method [10], to derive that $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$ are branching bisimilar, it is enough to check the matching criteria and the reachability of focus points. The two conditions of the cones and foci proof method are represented by `mc` and `WN`, namely matching criteria and the reachability of focus points, respectively. `mc` establishes that

```
...
fc(l,m,q,q12,l12,g,h,e,h12,g1,h2,e2,h1,l2,m2,q2,q1,l1): bool =
         (l=m OR l=l1+n) AND (l2=m2 OR l2=l12+n2)
k(i): below(2)= IF i=18 THEN 0 ELSE
         IF i=10 THEN 1 ELSE
         IF i=11 THEN 2 ELSE 3 ENDIF ENDIF ENDIF
h(l,m,q,q12,l12,g,h,e,h12,g1,h2,e2,h1,l2,m2,q2,q1,l1): [List_,List_]=
       (concat(qlist(q1,l1,next_empty(l1,q1)),qlist(q,next_empty(l1,q1),m)),
         concat(qlist(q12,l12,next_empty(l12,q12)),qlist(q2,next_empty(l12,q12),m2))
mc: THEOREM FORALL d: reachable(Nnonmod)(d) IMPLIES MC(Nnonmod,Z,k,h,fc)(d)
WN: LEMMA FORALL S: reachable(Nnonmod)(S) IMPLIES WN(Nnonmod,fc)(S)
main: THEOREM brbisimilar(lpe2lts(Nmod),lpe2lts(Z))
...
```

Figure 6. Correctness of $\mathbf{N}_{mod}$ in PVS

all the matching criteria (see Section 1) hold for every reachable state d in Nnonmod, with the aforementioned h, k and fc functions. WN represents the fact that from all reachable states S in Nnonmod, a focus point can be reached by a finite series of internal actions. The function lpe2lts provides the Labeled Transition System semantics of an LPE (see [10]).

7.6. Remarks on the Verification in PVS

We used PVS to find the omissions and undetected potential errors that have been ignored in the manual μCRL proofs; some of them have been shown as examples in Section 7.1. PVS guided us to find some important invariants. We affirmed the termination of recursive definitions by means of various measure functions. We represented LPEs in PVS and then introduced $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ as LPEs. We verified the bisimulation of $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$. Finally we used the cones and foci proof method [10], to prove that $\mathbf{N}_{mod}$ and the external behavior of the SWP with piggybacking, represented by $\mathbf{Z}$, are branching bisimilar.

8. Conclusions

In this paper we verify a two-way sliding window protocol which has the acknowledgments piggybacked on data. This way acknowledgments take a free ride in the channel. As a result the available bandwidth is used better. We present a specification of a sliding window protocol with piggybacking in μCRL, and then verify the specification with the PVS theorem prover.

An important aim of this paper is to show how one can incrementally extend a PVS verification effort, in this case the one described in [1]. PVS verification can be reused to check modifications of the SWP nearly automatically. We benefited from the PVS formalizations and lemmas in [1], e.g. properties of data types and those invariants which are not directly working with the internal structure of buffers (i.e. ordered lists). These are also mentioned in [2]. Note that a large part of the complete formalization consists of developing the meta theory. This part is split in generic PVS files with proofs. This generic part can be reused for the correctness proof of many other protocols. In particular, the generic part consists of the definition of an LTS, various forms of bisimulation (with proofs that they form equivalence relations), the definition of LPEs, their operational semantics, the notions of state mappings between LPEs, the notion of an invariant of an LPE (and its relation with reachable states), the proof rules for tau-reachability (with a soundness proof), and the matching criteria (including the proof of the theorem, that from the cones and foci method one may conclude branching bisimilarity).

For a specific protocol verification one must formalize the used data types (or find them in PVS's prelude), define LPEs for the specification and implementation, list the invariants, the focus conditions and the state mapping. From this, all proof obligations (like invariants and matching criteria) are generated automatically. Most obligations can be discharged automatically, but still many must be proven manually. Also, tau-reachability must typically be proven manually, using the predefined proof rules. However, some steps remain protocol-specific, such as the transition from modulo to full arithmetic in the case of the Sliding Window Protocol.

Here, we model the medium between the sending and receiving window as a queue of capacity one. So a possible extension of this work would be to verify this protocol with mediums of unbounded size, i.e. we can define the mediums as lists of pairs (d, i) by:

$$\begin{array}{ll} \texttt{cons:} & [\,] :\to Medium \\ \texttt{func:} & \mathit{add} : \Delta \times \mathbb{N} \times Medium \to Medium \end{array}$$

Acknowledgements

We would like to thank Jun Pang for the helpful discussions and for his μCRL files on the one-way SWP.

References

[1] B. Badban, W. Fokkink, J. Groote, J. Pang, and J. van de Pol. Verifying a sliding window protocol in μCRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.

[2] B. Badban, W. Fokkink, and J. van de Pol. Mechanical verification of a two-way sliding window protocol (full version including proofs). Technical Report TR-CTIT-08-45, University of Twente, CTIT, July 2008.

[3] J. Bergstra and J. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

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

[5] M. Bezem and J. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proc. 5th Conference on Concurrency Theory*, LNCS 836, pages 401–416, 1994.

[6] R. Cardell-Oliver. Using higher order logic for modeling real-time protocols. In J. Diaz and F. Orejas, editors, *Proc. 4th Joint Conference on Theory and Practice of Software Development*, Lecture Notes in Computer Science 494, pages 259–282, 1991.

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

[8] D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In H. Garavel and J. Hatcliff, editors, *Proc. 9th Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 2619, pages 113–127, 2003.

[9] W. Fokkink, J. Groote, J. Pang, B. Badban, and J. van de Pol. Verifying a sliding window protocol in μCRL. In S. M. C. Rattray and C. Shankland, editors, *Proc. 10th Conference on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science 3116, pages 148–163, 2004.

[10] W. Fokkink, J. Pang, and J. van de Pol. Cones and foci: A mechanical framework for protocol verification. *Formal Methods in System Design*, 29(1):1–31, 2006.

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

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

[13] J. Groote and H. Korver. Correctness proof of the bakery protocol in μCRL. In A. Ponse, C. Verhoef, and S. v. Vlijmen, editors, *Proc. 1st Workshop on Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 63–86. Springer-Verlag, 1995.

[14] J. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D. Andrews, J. Groote, and C. Middelburg, editors, *Proc. Workshop on Semantics of Specification Languages*, Workshops in Computing Series, pages 231–250, 1994.

[15] J. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S. v. Vlijmen, editors, *Proc. 1st Workshop on Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer-Verlag, 1995.

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

[17] J. Groote and M. Reniers. Algebraic process verification. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier, 2001.

[18] J. 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.

[19] B. Hailpern. *Verifying Concurrent Processes Using Temporal Logic.* LNCS 129. Springer-Verlag, 1982.

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

[21] G. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

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

[23] B. Jonsson and M. Nilson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *Proc. 6th Conference on Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science 1785, pages 220–234. Springer-Verlag, 2000.

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

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

[26] T. Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In J. Colom and M. Koutny, editors, *Proc. 21st Conference on Application and Theory of Petri Nets*, Lecture Notes in Computer Science 2075, pages 242–262. Springer-Verlag, 2001.

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

[28] E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol in LOTOS. In E. Knuth and L. Wegner, editors, *Proc. 4th Conference on Formal Description Techniques for Distributed Systems and Communication Protocols,* IFIP Transactions (C-2), pages 495–510. North-Holland, 1991.

[29] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Proc. 8th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 411–414. Springer-Verlag, 1996.

[30] K. Paliwoda and J. Sanders. An incremental specification of the sliding-window protocol. *Distributed Computing*, 5(2):83–94, 1991.

[31] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. 5th GI-Conference on Theoretical Computer Science*, LNCS 104, pages 167–183, 1981.

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

[33] C. Röckl and J. Esparza. Proof-checking protocols using bisimulations. In J. Baeten and S. Mauw, editors, *Proc. 10th Conference on Concurrency Theory*, LNCS 1664, pages 525–540, 1999.

[34] V. Rusu. Verifying a sliding-window protocol using PVS. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proc. 21st Conference on Formal Techniques for Networked and Distributed Systems,* IFIP Conference Proceedings 197, pages 251–268. Kluwer Academic, 2001.

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

[36] M. Smith and N. Klarlund. Verification of a sliding window protocol using IOA and MONA. In T. Bolognesi and D. Latella, editors, *Proc. 20th Joint Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 19–34. Kluwer Academic Publishers, 2000.

[37] K. Stahl, K. Baukus, Y. Lakhnech, and M. Steffen. Divide, abstract, and model-check. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proc. 6th SPIN Workshop on Practical Aspects of Model Checking*, Lecture Notes in Computer Science 1680, pages 57–76. Springer-Verlag, 1999.

[38] N. Stenning. A data transfer protocol. *Computer Networks*, 1:99–110, 1976.

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

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

[41] J. van de Snepscheut. The sliding window protocol revisted. *Formal Aspects of Computing*, 7(1):3–170, 1995.

[42] R. van Glabbeek. What is branching time and why to use it? *The Concurrency Column, Bulletin of the EATCS*, 53:190–198, 1994.

[43] R. van Glabbeek and W. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

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