HAL
archives-ouvertes.fr

Verification of a Sliding Window Protocol Using IOA and MONA

Mark Smith, Nils Klarlund

▶ **To cite this version:**

Mark Smith, Nils Klarlund. Verification of a Sliding Window Protocol Using IOA and MONA. [Research Report] RR-3959, INRIA. 2000. <inria-00072689>

HAL Id: inria-00072689
https://hal.inria.fr/inria-00072689

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Verification of a Sliding Window Protocol Using IOA and MONA

Mark A. Smith , Nils Klarlund

N°3959

July 2000

THÈME 4

ISSN 0249-6399 ISRN INRIA/RR--3959--FR+ENG

Verification of a Sliding Window Protocol Using IOA and MONA

Mark A. Smith* , Nils Klarlund†

Thème 4 — Simulation et optimisation
de systèmes complexes
Projets Sigma2

Rapport de recherche n°3959 — July 2000 — 23 pages

Abstract: We show how to use a decision procedure for WS1S (the MONA tool) to give automated correctness proofs of a sliding window protocol under assumptions of unbounded window sizes, buffer sizes, and channel capacities. We also verify a version of the protocol where the window size is fixed. Since our mechanized target logic is WS1S, not the finite structures of traditional model checking, our method employs only two easy reductions outside the decidable framework. Additionally, we formulate invariants that describe the reachable global states, but the bulk of the detailed reasoning is left to the decision procedure. Because the notation of WS1S is too low-level to describe complicated protocols at a reasonable level of abstraction, we use a higher level language for the protocol description, and then build a tool that automatically translates this language to the MONA syntax. The higher level language we use is IOA. It is a language for distributed programming and is based on Input/Output Automata.

Key-words: automated verification, formal methods, sliding window protocol, MONA, I/O Automata.

(Résumé : tsvp)

Most of this work was done while the first author was at AT&T Labs Research, Florham Park, NJ. A preliminary version of this paper appears in FORTE/PSTV2000.

* IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France. Mark.Smith@irisa.fr
† AT&T Labs Research, Florham Park, NJ, U.S.A.. klarlund@research.att.com

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Verification of a Sliding Window Protocol Using IOA and MONA

Résumé : Nous montrons comment utiliser une procédure de décision pour WS1S (l'outil MONA) afin de fournir une preuve automatisée de la correction d'un protocole à fenêtre avec l'hypothèse de tailles de fenêtre, tampons et canaux non bornées. La vérification est aussi effectuée pour une version avec fenêtre de taille fixe. Puisque la logique employée est WS1S, par opposition à l'utilisation de structures finies comme dans le "model-checking" classique, notre méthode utilise deux réductions simples situées en dehors du cadre décidable. De plus, nous formulons des invariants qui décrivent les états globaux accessibles, mais l'essentiel du raisonnement détaillé est laissé à la charge de la procédure de décision. La notation WS1S est de trop bas niveau pour décrire des protocoles compliqués. Aussi nous utilisons un langage de description de plus haut niveau (IOA) et avons construit un outil qui le traduit dans la syntaxe de MONA. IOA est dédié à la programmation répartie et est fondé sur le modèle des "Input/Output Automata".

Mots-clé : vérification automatissé, methodes formelles, protocole à fenêtre, MONA, I/O Automata.

1 Introduction

Network protocols are complicated. Protocol developers often determine the validity of their protocols by running simulations. While simulations can be useful, simulations generally cannot show that a protocol is correct for all possible cases. Mechanical formal verification is another way to check the validity of protocols. The traditional benchmark used for testing mechanical verification of protocols is the alternating bit protocol [2]. For this protocol, many papers have shown that mechanical verification is viable (see for example [22, 5, 18, 11, 17, 1]).

In this article we demonstrate that an abstract algorithm description language can be effectively linked to the MONA tool [15, 12] for verification purposes. In particular, with only a couple of simple abstraction steps, the verification of invariants can be fully automated for network protocols that are substantially more complicated than the alternating bit protocol. We also argue that the state machine description IOA language [8] allows us to express the protocol at a convincing level of abstraction, and we discuss a translator from IOA to MONA.

We use these tools to verify a sliding window protocol. The protocol allows a dynamic and unbounded window size; and it assumes unbounded, lossy channels, and unbounded buffers. Our automated tools also allow us to verify the protocol with a fixed window size, and we verify the protocol with a window size as large as 256. The proof depends on two simple, meta-logical simplifications in the spirit of abstract interpretation. The proof itself consists of an invariant whose correctness is established automatically by MONA.

The main contribution of our work is to effectively demonstrate that the MONA tool can be used to verify non-trivial protocols with unbounded data structures as well as bounded data structures. Unbounded data structures cannot be handled by traditional Binary Decision Diagrams (BDDs) [4] based tools. On the other hand, theorem provers (for example [10, 19, 7]) can also handle unbounded data structures. However, given the complexity of the protocol we verify in this work, we believe our verification process requires a lot less user interaction than is typical with theorem provers. By linking the MONA tool with IOA via the translation tool, we also allow users to describe protocols in a reasonably intuitive manner.

1.1 The MONA Tool

The MONA tool is an efficient implementation of the decision procedures for *WS1S (Weak Second-order Logic of One Successor)* and *WS2S (Weak Second-order Logic of Two Successor)*. WS1S is a variation of first-order logic with quantifiers and boolean connectives. Its interpretation is tied to a somewhat weakened version of arithmetic. The WS1S first order variables denote natural numbers, which can be compared and subjected to addition by constants. The logic also allows second-order variables which can be interpreted as a finite set of numbers. WS2S is a generalization of WS1S interpreted over the infinite binary tree.

A MONA program consists of a number of declarations and formulas. While the WS1S/WS2S logics have a simple and natural notation, the notation is too low-level to define protocols in a readable manner. Thus, we decided to use a high level language for describing the protocols. The protocol written in the high level language would then be translated to an equivalent MONA program.

1.2 The IOA Language

We decided to use the IOA language of Garland and Lynch [8] as the high level language. The IOA language is a precise language for describing Input/Output (I/O) Automata and stating their properties. I/O Automata [16], models components in asynchronous concurrent systems as labeled transitions systems. The model provides a mathematical framework for describing and reasoning about system components that interact with each other in an asynchronous manner. An I/O automaton consists of five components, a set of states, a nonempty set of start states, a set of actions, a set of steps, and a optional set of tasks. The set of actions can be partitioned into three disjoint sets, input, output, and internal. The set of tasks is a partition of the output and internal actions of the automaton.

1.3 Related Work

The *sliding window* or *go back n* protocol is used in various standard data link control (DLC) protocols such as HDLC (high level DCL) and SDLC (synchronous DCL). It also forms the basis of reliable transport layer protocols like TCP. The sliding window protocol has been studied extensively and there are several works that present manual verification of the protocol. For example see [23, 13].

There has been a lot less work on mechanical verification of the sliding window protocol. An early example of this work is by Brand and Joyner [3]. This work is similar to the work presented here in that only safety properties are verified. Their verification method is based on symbolic execution. Another way in which this work is similar to ours is that the verification is based on proving a number of invariants. However, the verification of the invariants is not fully automatic. In order to prove them, the system in [3] generates over a thousand theorems. Of these theorems only about 80% are proven automatically. The other 20% are proven by hand.

In [21] Richier *et al.* use temporal logic and model checking to verify safety properties of the sliding window protocol. Because of the state explosion problem, verification is only possible for a fairly restricted set of values of parameters. Another paper in which a mechanical verification of the sliding window protocol is carried out is [14] by Kaivola. In this paper both safety and liveness proofs are performed. In order to alleviate the state explosion problem, Kaivola uses compositional property preserving equivalences and preorders which allows one to replace components of a system with smaller ones. The verification can then be carried out on the smaller systems. Using *non-divergent failures divergences* (NDFD) preorders, safety and liveness properties of the sliding window protocol for arbitrary channel lengths and "realistic" parameter values are verified semi-automatically. The largest parameter value presented in Kaivola's paper is a window size of seven, which is used with two other smaller fixed values that represent buffer sizes. Godefroid and Long [9] verify a full duplex sliding window protocol with the queue size as large as 11. Queue BDDs which they introduce and use in their verification of the protocol are very similar to BDD-represented automata used by the MONA tool.

One way in which our work here differs from the other papers on mechanical verification of the sliding window protocol is that our model of the protocol is different from the others. First, we assume that the sender and receiver have unbounded buffers and that the sender uses an unbounded

set of sequence numbers. Because we assume unbounded sequence numbers, we do not require that the channels in our model be FIFO. While the assumption of unbounded buffers and sequence numbers is not a realistic one, in the situation where the sliding window protocol is used in the transport layer of the Internet (a la TCP [20]), the assumption of FIFO channels is not realistic either. TCP does not assume FIFO channels, instead it uses timeout mechanisms to simulate unbounded sequence numbers. This timeout mechanism is quite complicated, so we decide to just assume unbounded sequence numbers. Nevertheless, we believe our model more closely resembles the actual workings of TCP that the models used in the related work discussed above. Another way in which our work differs is that we do not have to assume a fixed window size. Our model allows the window size to be set nondeterministically. However, if we assume a fixed window size, we have verified the protocol with a window size of 256.

1.4 Organization of the Paper

In Section 2 we briefly describe the program that translates IOA to the MONA syntax. Section 3 and 4 contain descriptions of two versions of the sliding window protocol with unbounded window sizes, and also describes the abstractions we use. The proof of safety is discussed in Section 5. In Section 6 we discuss the case where we fix the window size, and we also present some experimental results. Section 7 contains some concluding remarks. Finally, we have two appendices with the IOA code for the various models of the protocol we present.

2 The Translator Program

The IOA language is Turing complete. Therefore, we cannot translate every IOA program into a valid equivalent WS1S/WS2S formula. However, we are able to translate a subset of the IOA language that is sufficiently expressive to allow the formulation of reasonable protocol descriptions.

We treat the simple built in types of the IOA language in a straightforward manner. IOA types boolean and (non-negative) integer translate to boolean and integer in WS1S. However, we cannot translate reals, and we do not translate IOA types character or string.

The IOA language also has several built in "compound" types. These types are, `Array[I, E]`, `Map[I, E]`, `Seq[E]`, `Set[E]` and `Mset[E]`. For these types, `I` is the index type, and `E` is the type of the elements. Each compound type has a set of built in operators. We are able to translate these compound types when the size of `E` has a known bound. However, for type `Set[E]`, where `E` is the set of integers, we translate directly to the MONA representation of sets of integers. The other types we represent in MONA as a group of sets where one set contains the valid indices, when the data type has indices, and the other set(s) are used to represent the elements. The IOA language also has an enumeration type. We represent this type in MONA using boolean valued variables, where the bit value of the variables taken together as binary number corresponds to the enumerated values.

We illustrate the translation schema with a small example. If we have an IOA type `Animals` which is an enumeration of `dog, cat, rat`, and `bat`, it is translated to MONA boolean (zero'th-order) variables, say `S0` and `S1`, where the values `S0 = false` and `S1 = false` represent `dog`,

`S0 = false` and `S1 = true` represent `cat`, `S0 = true` and `S1 = false` represent `rat`, and `S0 = true` and `S1 = true` represent `bat`. A compound IOA variable `animalBuf` that has type `Seq[Animals]` is translated into three second order MONA variables, say `S2, S3`, and `S4`. The first set `S2` holds the set of valid indices for the sequence. The other two sets encode the values in these positions in the following manner: for each valid index in `S2` the presence or absence of that index in sets `S3` and `S4` encodes the values in the position. Thus, if `dog` is the value of position 0 in the sequence, then sets `S3` and `S4` do not contain the value 0. However, if `cat` is the value in position 0, then set `S3` does not contain 0 while set `S4` does. Thus, `animalBuf = [cat, rat, dog, bat, rat]` where `cat` is at index position 0 is represented by the MONA sets `S2` = $\{0, 1, 2, 3, 4\}$, `S3` = $\{1, 3, 4\}$, and `S4` = $\{0, 3\}$. IOA operators on sequences are translated accordingly. We give two examples here, `head` and `tail`. The MONA translation of the `head` operator checks if the value 0 is present in the first set representing the sequence, This is a test for emptiness, and if the sequence is empty `head` is undefined. If the sequence is not empty, the two boolean values returned by the MONA operations `0 in S3` and `0 in S4` gives the value of the head of the sequence. In this example the values returned would be `false` and `true`, which is our MONA representation for `cat`. Obtaining the value at each indexed position i in the sequence is translated in the same manner by replacing 0 with i. To get the `tail` of a sequence, the maximum valued element is removed from the first set in the representation. This gives the new set of indices. For our example, the MONA syntax is `S2` $\setminus\{\max$ `S2`$\}$. To update the values in the other sets we first remove the element 0 from each of these sets and then subtract 1 from each remaining element. The MONA syntax for these operations are: `S3` $\setminus\{0\} - 1$ and `S4` $\setminus\{0\} - 1$. Thus, for our example after a `tail` operation, `S2` = $\{0, 1, 2, 3\}$, `S3` = $\{0, 2\}$, and `S4` = $\{2\}$.

IOA transitions are translated to MONA predicates. We used unprimed and primed MONA variables to represent the pre and post states of the IOA variables. IOA transitions are atomic, but the assignments in a transition happen sequentially. Thus, in our MONA translation we add temporary variables to hold intermediate values. The translator also generates a MONA predicate that represents the assignment of initial values to the IOA variables.

To facilitate the verification process we add a predicate, **preds**, section to the IOA language constructs. The **preds** section of an IOA program contains named predicates on the state of the program variables. We translate the predicates to unprimed and primed MONA predicates of the same name. The unprimed state represents the values before a transition and the primed states represent the values after the transition. Two MONA predicates `Inv` and `Inv`$'$ that respectively are the conjunction of the unprimed and primed predicates in the **preds** section are automatically generated. `Inv` is the Invariant we wish to verify holds for the IOA protocol. The translator also creates a separate file for each transition in the IOA program. Each file contains a MONA predicate of the form `Inv` $\wedge$ *step* $\Rightarrow$ `Inv`$'$, where step is the name of the IOA transition. Additionally, a MONA file which contains the MONA formula to test that `Inv` holds in the initial state is generated. These are the necessary files for the MONA tool to automatically verify the invariant. To verify that the invariant holds, we run MONA on each of the files. If MONA tells us that all the formulas are valid, then the invariant holds.

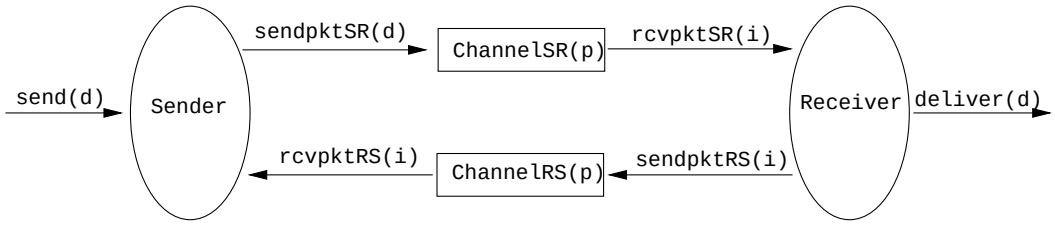

Figure 1: *The basic structure and components of a sliding window protocol.*

3 Sliding Window Protocol Description

In this section we describe our general IOA model of a sliding window protocol. The sliding window protocol is a mechanism for reliable message delivery in a distributed setting where a sender and a receiver communicate over lossy channels. Any message delivery protocol has the same basic four components, a sender, a receiver, a channel from the sender to the receiver, and a channel from the receiver to the sender. The basic structure of these protocols is shown in Figure 1. In addition to the four components we model, there are two external *users*. The user on the sender side inputs the data to be sent by the sender, and the user on the receiver side gets the data that the receiver delivers.

The basic idea of the sliding protocol is that a *window* of size $n \geq 0$ determines how many successive packets of data can be sent in the absence of a new acknowledgment. The window size may be fixed or may vary depending on the conditions of the different components of the protocol. In our model we will assume that n varies nondeterministically. Each packet of data is sequentially numbered, so the sender is not allowed to send packet $i + n$ before packet i has been acknowledged. Thus, if i is the sequence number of the packet most recently acknowledged by the receiver, there is a window of data numbered $i + 1$ to $i + n$ which the sender can transmit. As successively higher-numbered acknowledgments are received, the window slides forward. The acknowledgment mechanism is cumulative in that if the receiver acknowledges packet k, where $k \geq i + 1$, it means it has successfully received all packets up to k. Packet k is acknowledged by sending a request for packet $k + 1$. Typically, transmitted data is kept on a retransmission buffer until it has been acknowledged. Thus, when k is acknowledged, packets with sequence number less than or equal to k are removed from the retransmission buffer. Packets that are not acknowledged are eventually retransmitted. For our modeling of the sliding window protocol we assume that sequence numbers are unbounded, we do not assume that the channels are FIFO.

Below we present two IOA versions of the sliding window protocol. The first version is a general formulation, where the data domain is unspecified. The second version of the protocol is a parameterized transition system expressible in WS1S; it relies on two additional assumptions: that the data domain consists of two values and that the channels are approximated (in the sense to be explained).

3.1 A General Sliding Window Protocol Model

In this section we define a sliding window protocol as an I/O Automaton. The I/O automaton is defined using the IOA language. The general IOA model is shown in Figures 2, 3, and 4 in Appendix A.

3.1.1 Types

The type `D` represents a data domain whose internal structure is unspecified. The type `A` is an acknowledgment type, and has only one possible value, `ack`.

3.1.2 Signature and States

The **signature** is a partition of the actions of the automaton into input, output and internal actions. The actions `send(d), sendpktSR(d), prepareNewSeg(d), prepareRetranSeg(d)` and `rcvpktRS(i)` all happen on the sender side. The actions `rcvpktSR(i), sendpktRS(i)`, and `deliver(d)` happen on the receiver side, and the actions `dropSR(i)` and `dropRS(i)` happen in the channel from the sender to the receiver and the channel from the receiver to the sender respectively. We explain the functionality of each action when we describe the transitions below.

The **states** part of the IOA program represents the set of states and the set of initial values. In the listing, variables `SendBuf` through `hRetranBuf` are variables of the sender, variables `RcvBuf` through `temp` are receiver variables, variable `transitSR` is the state variable of the channel from the sender to the receiver, and variable `transitRS` is the state variable of the channel from the receiver to the sender. The variable `SendBuf` is a sequence that holds elements of type `D` and is initially empty. This buffer holds the input data as it is received from the external user. The position of the element in the sequence corresponds to the sequence number of that piece of data. The buffer has an associated pointer called `hSendBuf`; the section from `hSendBuf` to `len(SendBuf)` is called the *unsent* part of `SendBuf`, which is empty to begin with as denoted by the initial value of `hSendBuf`.

The variables `RetranBuf` and `hRetranBuf` represent the retransmission buffer in the same way the send buffer is represented. The variable `W` represents the current edge of the window, that is, the highest index allocated for use by the retransmission buffer. The current size of the sliding window is `W - hRetranBuf + 1`. Variable `readyToSend` is a boolean flag. It is set whenever a datum in the `segment` variable is ready to be transmitted; `seqNum` is then the sequence number of that segment.

On the receiver side, the variable `RcvBuf` holds received data before it is passed to the external user. Variable `hRcvBuf` is the index of the first element of the *undelivered* part of this buffer. The variable `sendAck` is a boolean flag that enables the sending of an acknowledgment packet, and `temp` is a data variable that is used to hold a value received from the channel before it is added to `RcvBuf`.

The variables `transitSR` and `transitRS` are map types that represent packets in transit from the sender to the receiver and from the receiver to the sender respectively. Both maps are

indexed by integers. The `transitSR` map holds elements that are multisets with elements of type `D`. These elements are multisets because there may be duplicates of data packets on the channel. The index of the defined elements in `transitSR` represents the sequence number of the packet(s). For `transitRS` the element type is a multiset of type `A`. The index of an element of `transitRS` represents the acknowledgment number of the packet.

3.1.3 Transitions

The `send(d)` input causes the data item `d` from the external user to be appended to the end of `SendBuf`. The internal action `prepareNewSeg(d)` prepares the data item `d` to be sent. This is data that has not been sent before. The precondition for this action requires `readyToSend = false`, which means the sending of no other data is currently enabled. The condition texttthSendBuf $\leq$ `len(SendBuf)` means the active part of the send buffer, `SendBuf`, is not empty. The data item `d` is the element index by `hSendBuf`. There must also be space available in the allowable window. If `hSendBuf` $<$ `W` then there is space in the window. `W` is initially set non-deterministically to a value greater than zero, and it is non-deterministically updated when acknowledgments are received. The effect of the `prepareNewSeg(d)` action is to append `d` to the retransmission buffer, `RetranBuf`. The sending of this data item in a packet is also enabled by assigning `seqNum` to `hSendBuf`, setting `readyToSend` to true, and assigning `segment` to `d`. `hSendBuf` is also incremented so that it is the sequence number of the next new packet to be sent.

The internal action `sendpktSR(d)` sends the data item `d` if `readyToSend` is true and `d = segment`. This action assigns `readyToSend` to false so that no more data is sent until some preparation occurs. An element is placed on the channel by inserting it into the multiset in the position in the `transitSR` map indexed by `seqNum`.

The internal action `rcvpktRS(i)` passes an acknowledgment packet from the receiver to the sender. The position of the element in the `transitRS` map represents the acknowledgment number. When the packet is received it is deleted from the multiset. An acknowledgment is only valid if it is not an old acknowledgment (`hRetranBuf` $<$ `i`). If the acknowledgment packet is valid, then `hRetranBuf` is assigned to `i`, which effectively removes the acknowledged packets from the active part of `RetranBuf`. When packets are acknowledged, there is more room in the available window for new packets. Therefore, for valid acknowledgments we also non-deterministically update the value of `W`, the edge of the window. This new value of `W` must of course be no less than the previous value, and it must also be no less than the head of the retransmission buffer.

The `prepareRetranSeg(d)` action prepares the data item at the head of the retransmission buffer for retransmission. The preconditions and effects of this action are similar to those of the `prepareNewSeg(d)` action except that the data item comes from `RetranBuf` and not `SendBuf`.

The `rcvpktSR(i)` action happens on the receiver side. This action passes a data item to the receiver. The element in the `transitSR` map must be defined, and it must be a non-empty multiset. If a packet is received, then the receiver must send an acknowledgment; that is why the `sendAck` flag is set to true. In our model the receiver only accepts the next piece of data it is expecting, so it adds data to `RcvBuf` if its sequence number is the next one expected by the receiver. That is, if `i`

`= len(RcvBuf) + 1`. The variable `temp` is used to hold a data value from the multiset that is extracted non-deterministically. That data value is then appended to the receive buffer.

The output action `deliver(d)` passes the data item at the head of `RcvBuf` to the external user. There is an item to be passed only if `hRcvB` is less than or equal to `len(RcvBuf)`. Naturally, `hRcvB` is incremented after the data is passed.

The `sendpktRS(i)` action sends an acknowledgment packet. The acknowledgment number `i` is `len(RcvBuf) + 1`. The variables `sendAck` and `transitSR` are updated accordingly.

The actions `dropSR(i)` and `dropRS(i)` are internal actions of the channel from the sender to the receiver and the channel from the receiver to the sender respectively. These actions drop a copy of a packet from the respective channels by removing elements from the respective multisets at index position `i`.

4 A WS1S Sliding Window Model

We desire to prove that what comes out is what goes in; that is, for all reachable states:

Predicate omega
For all i if $i \leq$ `len(RcvBuf)` then `SendBuf[i] = rcvBuf[i]`.

We establish this safety property by exhibiting an invariant that implies `omega`. However, we are not able to carry out any automated proof of invariance for the general model. Instead, we will bend the general model into an *abstract model* whose transitions can be represented in WS1S. The abstract model is shown in Figures 5, 6, 7 and 8 in Appendix B.

4.1 First simplifying assumption

The general model is parameterized with the `D` type. Our first simplifying assumption is based on idea of data-independence:

Proposition 1 *[24] The* `omega` *property holds for all reachable states of all abstract programs if and only if it holds for* `D` $= \{$`white, red`$\}$.

Proof: The program itself contains no comparisons of data. Thus, if there is some `D` and some reachable state that violates the `omega` property by virtue of `SendBuf[i]` $= d'$ and `rcvBuf[i]` $= d''$ with $d' \neq d''$, then the same program on the same domain `D` but with all `send (`d`)` actions replaced by `send(`$\hat{d}$`)`, where $\hat{d} = d$ if $d \in \{d', d''\}$ and, arbitrarily, $\hat{d} = d'$ if $d \notin \{d', d''\}$, will result in a reachable state violating the `omega` property. Thus, a two-value data domain suffices to establish `omega` for all domains. (This proof is due to Wolper [24].) ■

The *concrete model* is the general model, where `D` $= \{$`white, red`$\}$.

4.2 Second Simplifying Assumption

Our second simplifying assumption is that the channel structure of the concrete model can be simplified so that each sequence number is associated not with a multiset, but with an element in a fixed,

finite set. The method is that of abstract interpretation [6], where the computation is modeled in an abstract domain, which approximates the concrete domain.

4.2.1 The Abstract Domain

The abstract domain `multiD` models a multiset over `D` by four different values: `D_empty` represents the empty multiset; `D_r` represents a multiset that contains only `red` occurrences; `D_w` represents a multiset that contain only `white` occurrences; and `D_rw` represents a multiset that have occurrences of both `red` or `white`. This correspondence can be expressed mathematically by an *abstraction function* $\alpha_{\texttt{multiD}}$: `Mset`[{`red, white`}] $\rightarrow$ `MultiD` that maps a multiset to an abstract value. Similarly, the correspondence can be expressed by a *concretization function* $\gamma_{\texttt{multiD}}$: `MultiD` $\rightarrow$ `Set`[`Mset`[{`red, white`}]] that maps an abstract value to the set of corresponding multisets. Note that if a is not `D_empty`, then $\gamma_{\texttt{multiD}}(a)$ is an infinite set of multisets.

Also for any multiset M, $M \in \gamma_{\texttt{multiD}} \circ \alpha_{\texttt{multiD}}(M)$. In this sense, the abstract domain approximates the concrete values of the multisets. The approximation is not exact, of course, since generally there will be many other multisets in $\gamma_{\texttt{multiD}} \circ \alpha_{\texttt{multiD}}(M)$ than M.

The multisets of the `transitRS` channel are modeled by the domain `multiA` whose values are `A_empty`, denoting the empty multiset, and `A_pres`, denoting a non-empty multiset.

In the second IOA model, we call a global state a an abstract state. It contains two maps modeling the channels, and each may contain an unbounded number of abstract values denoting multisets.

The concretization functions above define for an abstract state a a set $\gamma(a)$ of states c of the concrete IOA model.

All operations involving the multisets are then modified to work on the abstract values. For example, the dropping of a message as specified in `dropSR(i)`, where `i` is a sequence number, involves the deletion of an element `v` from the multiset `transitSR(i)`. In the WS1S version, the deletion is modeled by changing the `multiD` value non-deterministically. For example, `D_rw` is changed to either `D_r` (there is only one occurrence of a white message, which is dropped), `D_w` (there are only white occurrences after dropping the single red message), or `D_rw` (there are still occurrences of both red and white messages after a message has been dropped).

This treatment of the abstractions entails that for any computation in the concrete model $c_0, c_1, \ldots$, we can find a corresponding computation in the abstract domain $a_0, a_1, \ldots$, whose states include those of the concrete computation in the sense that $c_i \in \gamma(a_i)$. In particular, if c_0 is the initial state of the concrete model, then we can let a_0 be the initial state of the abstract model.

Proposition 2 *Let ϕ be a formula that does not refer to channels. If ϕ holds for all reachable states of the abstract model, then ϕ holds for all reachable states of the concrete model.*

Proof:

1. We omit the details which establish on a transition by transition basis that the abstract operations are such that any concrete computation corresponds to an abstract one.
2. By the restriction on ϕ, all variables ϕ mentions have the same value in c as in $\psi(c)$. Therefore, ϕ holds about c if and only if it holds about $\psi(c)$.

The statement of the proposition is implied by 1. and 2. ■

4.3 The Abstract Model

The abstract model (Figures 5, 6, 7, and 8) is obtained from a transformation of the concrete model.

The **preds** section are predicates on the state of the automaton. We prove in Section 5, where these predicates are discussed further, that their conjunction is an invariant of the IOA `Sliding-Window`.

5 Proof of Safety

We use the standard inductive meta-argument for proving safety properties through invariants. We find a formula ϕ_I that implies the safety property, and we show that ϕ_I holds for the initial state, and that for every step (s, a, s') if ϕ_I holds in state s then it also holds in state s'. All such proofs are completely automated.

Our main task at hand is to find ϕ_I.

5.1 The Predicates

Predicate omega asserts the safety property. It corresponds to predicate `omega` in the IOA and MONA descriptions of the sliding window protocol. The IOA version of the predicate is shown in Figure 8. By itself this predicate is not invariant, so a stronger condition is needed. The other predicates are found by a combination of intuition and interaction with the MONA tool. When the MONA tool is run on a predicate that is not an invariant, the counter example provided by MONA gives some indication of what is needed to strengthen the predicate. Typically, the counter example involves some state that intuition tells one is not reachable from any start state. A predicate is then written to verify this intuition. The verification of this new predicate may in turn lead to the formulation of other predicates. Below we list the other predicates we need and give some intuition as to why they are needed.

Predicate alpha asserts basic properties about the length of the various buffers in the automaton `SlidingWindow`. This predicate corresponds to predicates `alpha1` through `alpha6` in the IOA and MONA versions of the sliding window protocol. The IOA verison is shown in Figure 8.

Predicate alpha

1. `hSendBuf` $\leq$ `(len(SendBuf) + 1)`.
2. `hSendBuf` $>$ `seqNum`.
3. `hSendBuf = len(RetranBuf) + 1`.
4. `hRetranBuf` $> 0 \wedge$ (`readyToSend` $\Rightarrow$ `seqNum` > 0).
5. For all i if `define(transitSR, i)` then (`hSendBuf` $> i$).
6. `hSendBuf` $>$ `len(RcvBuf)` $\wedge$ `len(SendBuf)` $\geq$ `len(RcvBuf)`.

Predicate beta, shown below, corresponds to predicate `beta` in Figure 8. It states that the retransmission buffer is a prefix of the send buffer.

Predicate beta
For all i if $i \leq$ `len(RetranBuf)` then `SendBuf[i] = RetranBuf[i]`.

Predicate gamma, shown below, corresponds to predicate `gamma` in Figure 8. It states that when a segment with a given sequence number is to be placed on the channel, the value of the element in the sequence number position in the send buffer is the same as the value of the segment.

Predicate gamma
For all `d` if `segment = d` $\wedge$ `readyToSend` then `SendBuf[seqNum] = d`.

Predicate delta corresponds to predicates `delta1` and `delta2` in Figure 8. It states that for all reachable states of the protocol, if there is an abstraction for a data type in particular position in the `transitSR` map, then that actual element is in the same position in `SendBuf`.

Predicate delta

1. For all i if `transitSR[i] = D_r` then `SendBuf[i] = red`.
2. For all i if `transitSR[i] = D_w` then `SendBuf[i] = white`.

Shown below is Predicate epsilon. It corresponds to predicates `epsilon1` and `epsilon2` in Figure 8. It asserts that if there is an element of a particular value at an index i in the retransmission buffer and that index is also defined in `transitSR`, then either the abstract representation of the element in `transitSR` corresponds to the value of the element at index i in the retransmission buffer, or the abstract representation of the element in `transitSR` corresponds to the element being absent.

Predicate epsilon

1. For all i if `defined(transitSR, i)` $\wedge$ `RetranBuf[i] = red` then `transitSR[i] = D_r` $\vee$ `transitSR[i] = D_empty`.
2. For all i if `defined(transitSR, i)` $\wedge$ `RetranBuf[i] = white` then `transitSR[i] = D_w` $\vee$ `transitSR[i] = D_empty`.

The final predicate is Predicate zeta. It corresponds to predicates `zeta1`, `zeta2`, and `zeta3` in Figure 8. Parts one and two assert that if there is a packet with the current sequence number in transit, and the current segment has a particular color value, then the afore mentioned packet in transit never has a value that corresponds to a different color from the color of the current segment. The third part says that in our model of the protocol there is never a packet in transit from the sender to the receiver that has a value that corresponds to it having both the values `red` and `white`.

Predicate zeta

1. If `defined(transitSR, seqNum)` $\wedge$ `segment = red` then `transitSR[seqNum]` $\neq$ `D_w`.

2. If `defined(transitSR, seqNum)` $\wedge$ `segment = white` then `transitSR[seqNum]` $\neq$ `D_r`.
3. For all i if `defined(transitSR, i)` then `transitSR[i]` $\neq$ `D_rw`.

We claim the conjunction of all the predicates above is invariant for `SlidingWindow`. The claim is proved by using MONA to automatically verify the following:

$$\mathbf{omega} \wedge \mathbf{alpha} \wedge \mathbf{beta} \wedge \mathbf{gamma} \wedge \mathbf{delta} \wedge \mathbf{epsilon} \wedge \mathbf{zeta}.$$

This invariant clearly implies the safety property. The verification of the protocol with unbounded window sizes took 68.26 seconds. The largest number of states generated by the MONA tool during the process was 18886, and the largest number of BDD nodes was 277859. We ran the tools on a SUNW,Ultra-5_10.

6 Fixed Window Size

In this section we discuss our results for the protocol where the window size is fixed. Figure 9 in Appendix B shows the two changes we make in the protocol to use a fixed window size. For the `prepareNewSed(d)` transition the test to check if there is available space in the allowable window is changed to `hSendBuf` $\leq$ `(hRetranBuf + 256)`, where 256 is the fixed window size. For the `rcvpktRS(i)` transition, the change is removing the update of the variable `W`.

Table 1 lists our experimental results. In the table, the states and BDD nodes columns holds the largest number of states and the largest number of BDD nodes generated by the MONA tool during the verification respectively. The results show that the processing time, the largest number of states, and the maximum number of BDD nodes generated by the MONA tool grows linearly as a function of the window size. The linear growth is to be expected: the amount of information that flows across positions i to $i + 1$ in the unbounded queues is bounded, since it is encoded in a finite-state automaton. The automaton in the bounded case must, in addition, count up to the window size, and that makes the resulting automaton size linear in the window size. In our experiments we note that the size of the automaton and the running time in the bounded case surpasses that of the unbounded case for window sizes greater than 44.

window size	time (sec's)	states	BDD nodes
16	30.98	4838	147223
32	46.67	5983	258583
64	86.63	11103	481303
128	342.01	21343	926743
256	1286.45	41823	1817623

Table 1: *The results of our experiments verifying the protocol with different fi xed window sizes.*

7 Conclusion and Future Work

In this paper we have demonstrated that a decision procedure for WS1S can be used to verify safety properties of a non-trivial network protocol.

Although our work indicates that for network protocols a good deal of nitty-gritty reasoning about queues and indices can be carried out automatically, more research into the following problems is needed:

- handling of sequence numbers that wrap around modulo the length of the window,
- handling of a more concrete presentation of queues, such as those actually found in a real programming language,
- automated support for abstraction steps,
- handling of liveness properties.

Acknowledgments
Thanks to Dennis Dams and Richard Trefler for helpful comments.

References

[1] P.A. Abdulla, A. Aniinichini, S. Bensalem, A. Bouajjani, P.Habermehl, and Y. Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In *Computer Aided Verification. 11th International Conference, CAV '99*, volume 1633 of *LNCS*. Springer-Verlag, July 1999.

[2] K. A. Barlett, R. A. Scantlebury, and P. C. Wilkinson. A note on reliable transmission over half duplex links. *Communications of the ACM*, 12, 1969.

[3] Daniel Brand and Jr William H. Joyner. Verification of HDLC. *IEEE Transactions on Communications*, com-30(5):1136–1142, May 1982.

[4] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[5] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of LNCS. Springer-Verlag, 1981.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[7] S.J. Garland and J.V. Guttag. A guide to the larch prover. Technical Report 82, DEC, Systems Research Center, December 1991. Updated version available as `http://larch.lcs.mit.edu:8001/larch/LP/overview.html`.

[8] Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for designing, analyzing, and builiding distributed systems. Technical Report MIT/LCS/TR-762, M.I.T., August 1998.

[9] P. Godefroid and D.E. Long. Symbolic protocol verification with Queue BDDs. *Formal Methods in System Design*, 14(13):257–271, may 1999.

[10] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[11] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe (FME)*, volume 1051 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1996.

[12] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order login in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95 LCNS 1019*, 1995.

[13] T. Higashino, M. Mori, Y. Sugiyama, K. Taniguchi, and T. Kasami. An algebraic specification of HDLC procedures and its verification. *IEEE Transactions on Software Engineering*, SE-10(6):825–836, 1984.

[14] Roope Kaivola. Using compositional preorders in the verification of sliding window protocol. In *Computer Aided Verification. 9th International Conference, CAV'97*, volume 1254 of *LNCS*, pages 48–59, Haifa, Israel, June 1997. Springer-Verlag.

[15] Nils Klarlund. Mona & Fido: The logic-automaton connection in practice. In *CSL '97 Proceedings*, volume 1414 of *LNCS*. Springer-Verlag, 1998.

[16] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI Quarterly*, 3(2), September 1989.

[17] Panagiotis Manolios, Kedar Namjoshi, and Robert Sumners. Linking theorem proving and model-checking with well-founded bisimulations. In *Computer Aided Verification. 11th International Conference, CAV '99*, volume 1633 of *LNCS*. Springer-Verlag, July 1999.

[18] R. Milner. *Communication and Concurrency*. Prentice Hall, 1990.

[19] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[20] Jon Postel. Transmission Control Protocol - DARPA Internet Program Specification (Internet Standard STC-007). Internet RFC-793, September 1981.

[21] J. L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in Xesar of the sliding window protocol. In *Protocol Specification, Testing and Verification VII*, pages 235–248. North-Holland, 1987.

[22] Krishnan Sabnani. An algorithmic technique for protocol verification. *IEEE Transactions on Communications*, 36(8), August 1988.

[23] A. Udaya Shankar and S. S. Lam. An HDLC protocol specification and verification using image protocols. *ACM Transactions on Computer Systems*, 1(4):331–368, 1983.

[24] P. Wolper. Synthesis of communication processes from temporal logic specifications. In *Proceedings 13th ACM Symposium on POPL*, pages 184–193, January 1986.

A The General IOA Model

```
type D  % data types
type A = enumeration of ack
automaton SlidingWindow(D: type)
signature
  input send(d: D)
  internal sendpktSR(d: D), prepareNewSeg(d: D), prepareRetranSeg(d: D),
           rcvpktRS(i: Int), rcvpktSR(i: Int), sendpktRS(i: Int),
           dropSR(i: Int), dropRS(i: Int)
  output deliver(d: D)
states
  SendBuf: Seq[D] := {},
  hSendBuf: Int := 1,
  W: Int := choose n where (n > 0),
  RetranBuf: Seq[D] := {},
  hRetranBuf: Int := 1,
  readyToSend: Bool := false,
  segment: D,
  seqNum: Int := 0,
  RcvBuf: Seq[D] := {},
  hRcvB: Int := 1,
  sendAck: Bool := false,
  temp: D,
  transitSR: Map[Int, Mset[D]] := empty,
  transitRS: Map[Int, Mset[A]] := empty
```

Figure 2: *Type definitions, the signature, and set of states of the general model of the protocol.*

```
transitions
input send(d)
eff SendBuf := SendBuf ⊢ d

internal prepareNewSeg(d)
pre readyToSend = false  ∧ hSendBuf ≤ len(SendBuf) ∧ d = SendBuf[hSendBuf]
    ∧ len(RetranBuf) < W
eff RetranBuf := RetranBuf ⊢ d;
    seqNum := hSendBuf;
    hSendBuf := hSendBuf + 1;
    readyToSend := true;
    segment := d

internal sendpktSR(d)
pre readyToSend = true ∧ d = segment
eff readyToSend := false;
    transitSR := update(transitSR, seqNum, insert(d, transitSR[seqNum]))

internal rcvpktRS(i)
pre defined(transitRS, i) ∧ transitRS[i] ≠ {}
eff transitRS :=  update(transitRS, i, delete(ack, transitRS[i]));
    if hRetranBuf < i then
        hRetranBuf := i;
        W := choose n where (n ≥ W ∧ n ≥ hRetranBuf)
    fi

internal prepareRetranSeg(d)
pre readyToSend = false ∧ d = RetranBuf[hRetranBuf]
    ∧ hRetranBuf ≤ len(RetranBuf)
eff readyToSend := true;
    seqNum := hRetranBuf;
    segment := d

internal rcvpktSR(i)
pre defined(transitSR, i) ∧ transitSR[i] ≠ {}
eff sendAck := true;
    if i = len(RcvBuf) + 1  then
       temp := choose v where (v ∈ transitSR[i]);
       RcvBuf := RcvBuf ⊢ temp;
       transitSR :=  update(transitSR, i, delete(temp, transitSR[i]))
    fi
```

Figure 3: *Transitions for the general IOA model of the protocol.*

```
output deliver(d)
pre hRcvB ≤ len(RcvBuf)  ∧ d = RcvBuf[hRcvB]
eff hRcvB := hRcvB + 1
internal sendpktRS(i)
pre sendAck = true ∧ i = len(RcvBuf) + 1
eff sendAck := false;
    transitRS := update(transitRS, i, insert(ack, transitRS[i]))

internal dropSR(i)
pre defined(transitSR, i) ∧ transitSR[i] ≠ {}
eff temp := choose v where (v ∈ transitSR[i]);
    transitSR := update(transitSR, i, delete(temp, transitSR[i]))

internal dropRS(i)
pre transitRS[i] ≠ {}
eff transitRS := update(transitRS, i, delete(ack, transitRS[i]))
```

Figure 4: *The final group of transitions for the general IOA model of the protocol.*

B The WS1S IOA Model

```
type D = enumeration of red, white    % data types
type MultiD = enumeration of D_empty, D_r, D_w, D_rw  % data packet types
type MultiA = enumeration of A_pres, A_empty  % ack packets types

automaton SlidingWindow
signature
  input send(d: D)
  internal sendpktSR(p: D), prepareNewSeg(d: D), prepareRetranSeg(d: D),
           rcvpktRS(i: Int),  rcvpktSR(i: Int), sendpktRS(i: Int),
           dropSR(i: Int), dropRS(i: Int)
  output deliver(d: D)
```

Figure 5: *Type definitions and the signature of the WS1S IOA model of the protocol.*

```
states
  SendBuf: Seq[D] := {},
  hSendBuf: Int := 1,
  W: Int := choose n where (n > 0),
  readyToSend: Bool := false,
  seqNum: Int := 0,
  segment: D,
  RetranBuf: Seq[D] := {},
  hRetranBuf: Int := 1,
  RcvBuf: Seq[D] := {},
  hRcvBuf: Int := 1,
  sendAck: Bool := false,
  transitSR: Map[Int, MultiD] := empty,
  transitRS: Map[Int, MultiA] := empty

transitions
input send(d)
eff SendBuf := SendBuf ⊢ d

internal prepareNewSeg(d)
pre readyToSend = false  ∧ hSendBuf ≤ len(SendBuf) ∧ d = SendBuf[hSendBuf]
     ∧ len(RetranBuf) < W
eff RetranBuf := RetranBuf ⊢ d;
   seqNum := hSendBuf;
   hSendBuf := hSendBuf + 1;
   readyToSend := true;
   segment := d

internal prepareRetranSeg(d)
pre readyToSend = false ∧ d = RetranBuf[hRetranBuf]
     ∧ hRetranBuf ≤ len(RetranBuf)
eff readyToSend := true;
   seqNum := hRetranBuf;
   segment := d

internal rcvpktRS(i)
pre defined(transitRS, i) ∧ transitRS[i] = A_pres
eff if hRetranBuf < i then
       hRetranBuf := i;
       W := choose n where (n ≥ W ∧ n ≥ hRetranBuf)
   fi;
   transitRS := choose v
      where (v = update(transitRS, i, A_empty)) ∨ (v = transitRS)
```

Figure 6: *The set of states and transitions of the WS1S IOA model of the protocol.*

```
internal sendpktSR(d)
pre readyToSend = true ∧ d = segment
eff readyToSend := false;
   if d = red then
      if (¬ defined(transitSR, seqNum) ∨ (transitSR[seqNum] = D_empty)
         ∨ transitSR[seqNum] = D_r) then
         transitSR := update(transitSR, seqNum, D_r)
      elseif transitSR[seqNum] = D_w then
         transitSR := update(transitSR, seqNum, D_rw) fi
   elseif d = white then
      if (¬ defined(transitSR, seqNum) ∨ (transitSR[seqNum] = D_empty)
         ∨ transitSR[seqNum] =  D_w) then
         transitSR := update(transitSR, seqNum, D_w)
      elseif transitSR[seqNum] = D_r then
         transitSR := update(transitSR, seqNum, D_rw) fi
   fi

internal rcvpktSR(i)
pre defined(transitSR, i) ∧ transitSR[i] ≠ D_empty
eff sendAck := true;
   if i = len(RcvBuf) + 1 then
      if transitSR[i] = D_r then
         RcvBuf := RcvBuf ⊢ red;
         transitSR :=  choose v
            where (v = update(transitSR, i, D_empty)) ∨ (v = transitSR)
      elseif transitSR[i] = D_w then
         RcvBuf := RcvBuf ⊢ white;
         transitSR :=  choose v
            where (v = update(transitSR, i, D_empty)) ∨ (v = transitSR)
      elseif transitSR[i] = D_rw then
         RcvBuf := choose b
            where (b = RcvBuf ⊢ red) ∨ (b = RcvBuf ⊢ white);
         if last(RcvBuf) = red then
            transitSR :=  choose v
               where (v = update(transitSR, i, D_w)) ∨ (v = transitSR)
         else transitSR :=  choose v
            where (v = update(transitSR, i, D_r)) ∨ (v = transitSR) fi
      fi
   fi
```

Figure 7: *More transitions of the WS1S IOA model of the protocol.*

```
output deliver(d)
pre hRcvBuf ≤ len(RcvBuf) ∧ d = RcvBuf[hRcvBuf]
eff hRcvBuf := hRcvBuf + 1
internal sendpktRS(i)
pre sendAck = true ∧ i = len(RcvBuf) + 1
eff sendAck := false;
   transitRS := update(transitRS, i, A_pres)

internal dropSR(i)
pre defined(transitSR, i) ∧ transitSR[i] ≠ D_empty
eff if transitSR[i] = D_r ∨ transitSR[i] = D_w then
       transitSR :=  choose v
          where (v = update(transitSR, i, D_empty)) ∨ (v = transitSR)
    else transitSR :=  choose v
       where (v = update(transitSR, i, D_r)) ∨
             (v = update(transitSR, i, D_w)) ∨ (v = transitSR)
    fi

internal dropRS(i)
pre defined(transitRS, i) ∧  transitRS[i] = A_pres
eff transitRS := choose v
       where (v = update(transitRS, i, A_empty)) ∨ (v = transitRS)

preds
alpha1 hSendBuf ≤ (len(SendBuf) + 1);
alpha2 hSendBuf > seqNum;
alpha3 hSendBuf = len(RetranBuf) + 1;
alpha4 (hRetranBuf > 0) ∧ (readyToSend ⇒ seqNum > 0);
alpha5 ∀ i: Int (defined(transitSR,i) ⇒ (hSendBuf > i ));
alpha6 hSendBuf > len(RcvBuf)  ∧ len(SendBuf) ≥ len(RcvBuf);
beta  ∀ i: Int ((i ≤ len(RetranBuf)) ⇒ (SendBuf[i] = RetranBuf[i]));
gamma ∀ d: D ((segment = d ∧ readyToSend) ⇒ SendBuf[seqNum] = d);
delta1 ∀ i: Int ((transitSR[i] = D_r) ⇒ SendBuf[i] = red);
delta2 ∀ i: Int ((transitSR[i] = D_w) ⇒ SendBuf[i] = white);
epsilon1 ∀ i: Int ((defined(transitSR, i) ∧ (RetranBuf[i] = red)) ⇒
(transitSR[i] = D_r ∨ transitSR[i] = D_empty));
epsilon2 ∀ i: Int ((defined(transitSR, i) ∧ (RetranBuf[i] = white)) ⇒
(transitSR[i] = D_w ∨ transitSR[i] = D_empty));
zeta1 (defined(transitSR, seqNum) ∧ segment = red) ⇒
transitSR[seqNum] ≠ D_w;
zeta2 (defined(transitSR, seqNum) ∧ segment = white) ⇒
transitSR[seqNum] ≠ D_r;
zeta3  ∀ i: Int (defined(transitSR, i) ⇒ transitSR[i] ≠ D_rw);
omega  ∀ i: Int (i ≤ len(RcvBuf)  ⇒ (SendBuf[i] = RcvBuf[i]))
```

Figure 8: *The final set of transitions, and a series of predicates the conjunction of which is invariant on the reachable states of the protocol.*

```
internal prepareNewSeg(d)
pre readyToSend = false  ∧ hSendBuf ≤ len(SendBuf) ∧ d = SendBuf[hSendBuf]
    ∧ hSendBuf ≤ (hRetranBuf + 256)
eff RetranBuf := RetranBuf ⊢ d;
   seqNum := hSendBuf;
   hSendBuf := hSendBuf + 1;
   readyToSend := true;
   segment := d

internal rcvpktRS(i)
pre defined(transitRS, i) ∧ transitRS[i] = A_pres
eff if hRetranBuf < i then hRetranBuf := i fi;
   transitRS := choose v
      where (v = update(transitRS, i, A_empty)) ∨ (v = transitRS)
```

Figure 9: *The transitions of the protocol with a fixed window size that are different from the transitions in the protocol with unbounded and dynamic window sizes.*

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399