# Formal Verification of Communication Protocol using Type Theory

Xingyuan Zhang, Xiren Xie
PLA University of Science and Technology,
Nanjing, China
Email: {xyzhang, xiexr}@public1.ptt.js.cn

Malcolm Munro
Department of Computer Science, University of Durham,
Science Laboratories, South Road, Durhram, DH1 3LE, U.K.
Email: Malcolm.Munro@durham.ac.uk

Mark Harman, Lin Hu
Department of Information Systems and Computing,
Brunel University, Uxbridge, Middlesex, UB8 3PH, U.K.
Email: {Mark.Harman, Lin.Hu}@brunel.ac.uk

*Abstract*— **In this paper, an approach is proposed to verify communication protocol using the type theoretical proof assistant Coq. Compared with existing methods of protocol verification, this approach is based directly on the simple notion of event trace. Without the burden of embedding external concurrent languages such as process algebra, finite state machine, temporal logic, etc., this approach leads to very efficient reasoning. The approach is deliberately designed to exploit the computational mechanism intrinsic to type theory so that many cases can be proved automatically by computation.**

**Because of these advantages, even non-trivial protocols can be verified within reasonable cost.**

**This paper shows that both safeness and liveness can be formalized and verified using only finite event traces. A simplified version of the sliding window protocol is used to illustrate the approach. All the results presented in this paper have been mechanically checked in Coq. The relevant Coq scripts are accessible through Internet.**

**Keywords: Protocol Engineering, Protocol Verification**

## I. Introduction

Protocol verification can roughly be divided into two categories: Model Checking [1], [2], [3] and Theorem Proving [4], [5], [6], [7]. This paper belongs to the latter category.

Mechanical support is crucial for the theorem proving approach to scale up. However, 'which language to use' remains an open question. A lot of formalisms have been proposed, Process Algebra [8], [9], I/O Automata [10], Temporal Logic [11], [12], Statechart [13] being just a few. One problem with these formalisms is that most of them leave the definition and reasoning of data types to the 'underlying mathematics'. The consequence is that when developing mechanical support for these languages, two languages have to be dealt with – the language for protocol specification (protocol language) and the language for the 'underlying mathematics' (mathematics language). How to merge these two languages into one integrated language is not yet clear.

As a practical consequence of this problem, theorem proving based protocol verification usually begins with language embedding. The disadvantage of language embedding is that a lot of theorem proving resources are spent on the management of language being embedded, rather than on the verification of the protocol itself. This problem may be alleviated by using 'shallow embedding'.

In this paper, we go a step further – there is no embedding at all. Communication protocol is specified and verified using only standard mathematical notations. Since most general purpose proof assistant systems are initially developed to support mathematics, this approach is likely to be better supported.

The basic ideas of our approach are:

1) The semantics of concurrent systems is modelled as set of event traces. The main difference from Focus is that we deal with finite traces only. Such a simpler notion of trace leads to efficient reasoning. It is shown that both safeness and liveness properties can be modelled using finite traces only.
2) Only standard mathematical language is used. In stead of using an embedded language such as Process Algebra, I/O Automata, Statechart or Temporal Logic, this paper uses the language of Coq [14], [15], [16], which is the language of constructive mathematics. Without the burden of language embedding, the theorem proving resources of Coq can be focused directly on the concurrent system under investigation. Such a kind of 'focus' results in efficient reasoning as well.

Another novelty of our work is that the approach of 'specification by observation' is used, where system properties are expressed as properties of observation functions, which are list processing functions defined using the computational mechanism intrinsic to type theory. Such a design is deliberate to exploit the computational mechanism of type theory so that many cases can be proved automatically by computation.

Because of these advantages, highly efficient reasoning is achieved, so that even non-trivial communication protocols can be treated with reasonable cost. A simplified sliding window protocol is chosen as an example to illustrate the ideas. Since sliding window protocol usually serves as a benchmark example in protocol verification, the choice makes it easier to compare our approach with others.

The particular version of sliding window protocol, although simplified, is still an infinite state system, which lies beyond the reach of ordinary model checking. The formal verification of this protocol shows the potential of our approach. It is shown that both safeness and liveness can be treated using our approach.

The relevant Coq scripts for this paper can be found at: http://www.dur.ac.uk/xingyuan.zhang/sw/concCoq.zip.

The rest of this paper is organized as follows: Section II shows how a concurrent system can be modelled as an inductively defined set of event traces. Section III describes the architecture of sliding window protocol and the set of events needed to specify it. Section IV specifies the desired behavior of the protocol. In Section V, additional observation functions required to implement the sliding window protocol are defined. Section VI presents the sliding window protocol as a concurrent system SW. Section VII gives the proof of safeness. A detailed explanation of the proof of Lemma 7.4 shows how the computational mechanism is used in proof construction. Section VIII gives the proof of liveness. Section IX summarizes related works and gives some discussions. Section X is the conclusion. Auxiliary definitions used in this paper can be found in Appendix A. Some preliminary lemmas are given in Appendix B.

### A. Conventions

Because type theory is proposed to formalize constructive mathematics, we are able to present the work in standard mathematical notation. All these standard formlæ have precise meaning in type theory and this enables the results in this paper to be checked mechanically by Coq.

Type theory has a notion of computation, which is used as an definition mechanism, where the equation $a \overset{\mathsf{def}}{\Longrightarrow} b$ represents a 'computational rule' used to expand the definition of $a$ to $b$.

The equality symbol '$=$' used in this paper is the Leibniz equality in type theory. Let $a \overset{\mathsf{def}}{\Longrightarrow}^* b$ means '$a$ computes to $b$', we have: $(a \overset{\mathsf{def}}{\Longrightarrow}^* b) \Rightarrow (a = b)$ but not vise versa.

Free variables in formulae are assumed to be universally quantified, for example, $n_1 + n_2 = n_2 + n_1$ is an abbreviation of $\forall\, n_1, n_2.\ n_1 + n_2 = n_2 + n_1$.

## II. Concurrent system

A concurrent system on event set Evt is written as CS(Evt) and defined as a relation between event trace and event:

$$\mathsf{CS}(\mathsf{Evt}) \overset{\mathsf{def}}{\Longrightarrow} [\![\mathsf{Evt}]\!] \rightarrow \mathsf{Evt} \rightarrow \mathtt{Prop} \qquad (1)$$

where the event set Evt is represented as a type in type theory, event trace is represented as list of events: $[\![\mathsf{Evt}]\!]$ [1]. $\mathtt{Prop}$ is the type of propositions in type theory. For concurrent system $cs$ : CS(Evt), event trace $tr$ : $[\![\mathsf{Evt}]\!]$ and event $e$ : Evt, the expression

$$cs(tr, e)$$

means: in concurrent system $cs$, $e$ is a valid event to happen under the state defined by the event trace $tr$. The execution of

[1] $[\![\mathsf{Evt}]\!]$ is the type of lists consisting of elements from type Evt. List operations can be found in Appendix A.2.

$$\frac{}{\mathsf{VT}(cs, \langle\rangle)}\ \mathbf{vt\_nil} \qquad \frac{\mathsf{VT}(cs, tr) \quad cs(tr, e)}{\mathsf{VT}(cs, e \centerdot tr)}\ \mathbf{vt\_cons}$$

Fig. 1. Definition of VT

a concurrent system $cs$ starts with empty trace, at each step of execution, if $cs(tr, e)$ is true for some event $e$, then $e$ could be the next event to happen, in which case it is added to the head of the current trace $tr$. Instead of defining the state of a concurrent system as a value assignment for a certain set of variables, we represent the state of a concurrent system as a finite trace of events. Therefore, the state of the system changes with the occurrence of events. The system is concurrent in the sense that the choice between several possible valid events is nondeterministic.

Such a notion of execution is reflected in the definition of the relation $\mathsf{VT}(cs, tr)$ in Figure 1, where $\mathsf{VT}(cs, tr)$ means the trace $tr$ is a valid trace of the concurrent system $cs$. The rule **vt_nil** defines the empty trace (written as $\langle\rangle$) to be a valid trace; and the rule **vt_cons** says: if $tr$ is a valid trace, and $e$ is a valid event to happen under $tr$, then $e \centerdot tr$ (the list obtained by adding $e$ to the head of $tr$) is a valid trace.

In this way, the behavior of a concurrent system can be represented as an inductively defined set of event traces. Most of the lemmas verified in this paper have the form:

$$\forall\, tr\,.\, \mathsf{VT}(cs, tr) \Rightarrow P(tr) \qquad (2)$$

where $cs$ is the concurrent system under investigation, $P$ is the property we are interested in. Since Coq has comprehensive support for structural induction, by induction on the formation of $\mathsf{VT}(cs, tr)$, $P$ can be proved conveniently. Proofs of this kind often consist of a large number of cases. A particular appealing aspect of our approach is that many of them can be discharged automatically by the computational mechanism of type theory, so that people can concentrate more on interesting cases.

The composition of two concurrent systems $cs_1, cs_2$ : CS(Evt) is now simply defined as:

$$cs_1 \parallel cs_2 \overset{\mathsf{def}}{\Longrightarrow} \lambda\, tr, e.\, cs_1(tr, e) \ \vee\ cs_2(tr, e) \qquad (3)$$

## III. System architecture

Sliding window protocol is a communication protocol which provides reliable connection between two network nodes linked with an unreliable communication channel. It is required that all messages sent at one node will eventually be delivered at the other node in the same order as they were sent. The architecture of the sliding window protocol is given in Figure 2. The set of events determined by this architecture is given in Figure 3, where M is the type of messages, which is not defined explicitly in this paper.

In Figure 2, the the sliding window protocol is modelled by four event-generating entities, namely: Supplier, Sender, Channel and Receiver. Each class of events is represented as an arrow, from the entity which generates it, to the entity being affected. For example, the event $\rhd[m]$ (generated by Supplier)

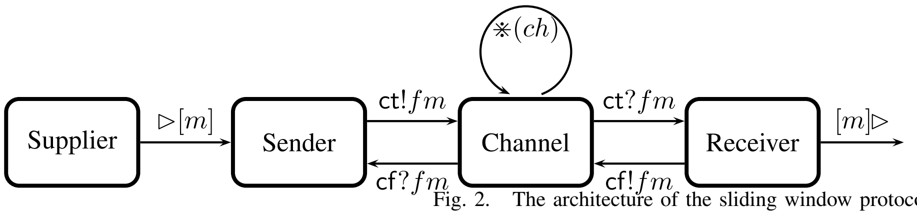

Fig. 2. The architecture of the sliding window protocol

$$\frac{}{\mathsf{Evt} : \mathsf{Set}}\ \textbf{Evt\_formation}$$

$$\frac{m : \mathsf{M}}{\rhd[m] : \mathsf{Evt}}\ \textbf{msg\_in} \qquad \frac{m : \mathsf{M}}{[m]\rhd : \mathsf{Evt}}\ \textbf{msg\_out} \qquad \frac{ch : \mathsf{Chan} \quad fm : \mathsf{Frm}}{ch!fm : \mathsf{Evt}}\ \textbf{send}$$

$$\frac{ch : \mathsf{Chan} \quad fm : \mathsf{Frm}}{ch?fm : \mathsf{Evt}}\ \textbf{receive} \qquad \frac{ch : \mathsf{Chan}}{\circledast(ch) : \mathsf{Evt}}\ \textbf{lost}$$

Fig. 3. Definition of Evt

represents the creation of a message $m$ to be transferred by the sliding window protocol. The event $[m]\rhd$ (generated by Receiver) represents the delivery of message $m$ to the outside world. There are two communication channels, ct is the one from Sender to Receiver and cf is the one from Receiver to Sender. The name of channels are defined in Figure 4 as the inductive type Chan.

For any channel $ch$, the sending of frame $fm$ over $ch$ is represented by the event $ch!fm$, the receiving of frame $fm$ from $ch$ is represented by the event $ch?fm$. The type of frames Frm is defined in Figure 5, where the frame $\boxed{\mathsf{data}(n, m)}$ is used by Sender to wrap the message $m$ before sending it over ct and $n$ is a sequence number managed by Sender. The frame $\boxed{\mathsf{ack}(n)}$ is used by Receiver to send acknowledging information back to Sender and $n$ is the sequence number of the message being acknowledged.

Interferences may happen to communication channels, the effect of which is to destroy frames under transmission. The occurrence of a single interference to channel $ch$ is represented by the event $\circledast(ch)$.

## IV. Specification by observation

Suppose the sliding window protocol is implemented as a concurrent system SW. Then the desired behavior of SW is expressed as properties of event traces generated by SW according to the inductive definition of VT given in Figure 1. Properties of event traces are described in terms of observation functions, which are simply list processing functions. For example, the list of all messages created by Supplier is defined by the observation function msgs_in:

$$\begin{cases} \mathrm{msgs\_in}(\rhd[m] \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \mathrm{msgs\_in}(tr) \dotplus m \\ \mathrm{msgs\_in}(\_ \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \mathrm{msgs\_in}(tr) \\ \mathrm{msgs\_in}(\langle\rangle) \overset{\mathsf{def}}{\Longrightarrow} \langle\rangle \end{cases} \tag{4}$$

where, the equation $\mathrm{msgs\_in}(\rhd[m] \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \mathrm{msgs\_in}(tr) \dotplus m$ says: when an event $\rhd[m]$ happens, the message it carries is appended to the end of the resulting list ($\dotplus$ is defined in (32)), and the equation $\mathrm{msgs\_in}(\_ \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \mathrm{msgs\_in}(tr)$ says: when any other kind of event happens, the resulting list remains unchanged, where 'all the other events' are written as _. Notice that the use of _ makes msgs_in resilient to the extension of Evt because there is no need to change the definition when new events are added to Evt. And finally, the equation $\mathrm{msgs\_in}(\langle\rangle) \overset{\mathsf{def}}{\Longrightarrow} \langle\rangle$ says: the resulting list is initially empty.

Similarly, the list of messages delivered by the the sliding window protocol is defined by the observation function msgs_out:

$$\begin{cases} \mathrm{msgs\_out}([m]\rhd \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \mathrm{msgs\_out}(tr) \dotplus m \\ \mathrm{msgs\_out}(\_ \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \mathrm{msgs\_out}(tr) \\ \mathrm{msgs\_out}(\langle\rangle) \overset{\mathsf{def}}{\Longrightarrow} \langle\rangle \end{cases} \tag{5}$$

With these two functions, the desired behavior of SW can be expressed by the following two lemmas:

*Lemma 4.1 (Safeness):*

$$\forall tr \,.\, \mathsf{VT}(\mathsf{SW}, tr) \Rightarrow \exists l.\ \mathrm{msgs\_in}(tr) = \mathrm{msgs\_out}(tr)^\frown l \tag{6}$$

which says: the list of messages delivered at the Receiver is a prefix of the messages injected into the system at the Sender.

*Lemma 4.2 (Liveness):*

$$\forall tr \,.\, \mathsf{VT}(\mathsf{SW}, tr) \Rightarrow \exists tr'.\ \mathsf{VT}(\mathsf{SW}, tr'^\frown tr)\ \wedge\ \mathrm{msgs\_in}(tr) = \mathrm{msgs\_out}(tr'^\frown tr) \tag{7}$$

which says: all the messages injected into the system at the Sender will eventually be delivered at the Receiver. The expression $\mathsf{VT}(\mathsf{SW}, tr'^\frown tr)$ means $tr'^\frown tr$ is a valid extension

$$\frac{}{\mathsf{Chan} : \mathsf{Set}}\ \textsf{Chan\_}\textbf{formation} \qquad \frac{}{\mathsf{ct} : \mathsf{Chan}}\ \textbf{chan\_to} \qquad \frac{}{\mathsf{cf} : \mathsf{Chan}}\ \textbf{chan\_from}$$

Fig. 4. Definition of Chan

$$\frac{}{\mathsf{Frm} : \mathsf{Set}}\ \textsf{Frm\_}\textbf{formation} \qquad \frac{n : \mathsf{Nat} \quad m : \mathsf{M}}{\boxed{\mathsf{data}(n,m)} : \mathsf{Frm}}\ \textbf{data} \qquad \frac{n : \mathsf{Nat}}{\boxed{\mathsf{ack}(n)} : \mathsf{Frm}}\ \textbf{ack}$$

Fig. 5. Definition of Frm

of $tr$. Lemma 4.2 guarantees that there is always a valid extension (or future) for any valid trace $tr$, when the messages delivered at the Receiver equals the messages injected at the Sender at the moment denoted by $tr$. If the scheduling mechanism is fair to all possible futures, the phenomena described by Lemma 4.2 is guaranteed to happen.

Notice that both Lemma 4.1 and Lemma 4.2 are in the general pattern described in (2), as most of the lemmas in this paper.

## V. More observation functions

For the construction of SW, additional observation functions are required. The function $\mathrm{msg\_no\_rcvd}$ computes the list of sequence numbers from data frames $\boxed{\mathsf{data}(n,m)}$ observable by the Receiver:

$$\begin{cases} \mathrm{msg\_no\_rcvd}(\mathsf{ct}?\boxed{\mathsf{data}(n,m)} \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \\ \qquad n \centerdot \mathrm{msg\_no\_rcvd}(tr) \\ \mathrm{msg\_no\_rcvd}(\_ \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \\ \qquad \mathrm{msg\_no\_rcvd}(tr) \\ \mathrm{msg\_no\_rcvd}(\langle\rangle) \overset{\mathsf{def}}{\Longrightarrow} \langle\rangle \end{cases} \tag{8}$$

The function $\mathrm{msgs\_rcvd}$ computes a finite mapping from sequence numbers to messages from data frames $\boxed{\mathsf{data}(n,m)}$ observable by the Receiver (the definition of 'finite mapping' is given in Appendix A.3):

$$\begin{cases} \mathrm{msgs\_rcvd}(\mathsf{ct}?\boxed{\mathsf{data}(n,m)} \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \\ \qquad \mathrm{msgs\_rcvd}(tr)\,[m^{\,n}] \\ \mathrm{msgs\_rcvd}(\_ \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \mathrm{msgs\_rcvd}(tr) \\ \mathrm{msgs\_rcvd}(\langle\rangle) \overset{\mathsf{def}}{\Longrightarrow} \epsilon \end{cases} \tag{9}$$

The definition of $\mathrm{msgs\_rcvd}(tr)\,[m^{\,n}]$ can be found in (38).

The function $\mathrm{msg\_no\_acked}$ computes the list of acknowledged sequence numbers from the acknowledging frames $\boxed{\mathsf{ack}(n)}$ observable by the Sender:

$$\begin{cases} \mathrm{msg\_no\_acked}(\mathsf{cf}?\boxed{\mathsf{ack}(n)} \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \\ \qquad n \centerdot \mathrm{msg\_no\_acked}(tr) \\ \mathrm{msg\_no\_acked}(\_ \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \mathrm{msg\_no\_acked}(tr) \\ \mathrm{msg\_no\_acked}(\langle\rangle) \overset{\mathsf{def}}{\Longrightarrow} \langle\rangle \end{cases} \tag{10}$$

$$\frac{}{\mathsf{Supplier}(tr, \triangleright[m])}\ \textbf{sp\_in}$$

Fig. 6. Definition of Supplier

The function $\mathrm{queue}(ch, tr)$ computes the list of frames observable by communication channel $ch$, waiting to be transmitted to their destination:

$$\begin{cases} \mathrm{queue}(ch, \overline{ch}!fm \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \\ \qquad \mathrm{queue}(ch, tr) \dotplus fm \quad \text{if } \overline{ch} = ch \\ \mathrm{queue}(ch, \overline{ch}?fm \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} fms \\ \qquad \text{if } \overline{ch} = ch \ \wedge\ \mathrm{queue}(ch, tr) = fm \centerdot fms \\ \mathrm{queue}(ch, \circledast(\overline{ch}) \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} fms \\ \qquad \text{if } \overline{ch} = ch \ \wedge\ \mathrm{queue}(ch, tr) = fm \centerdot fms \\ \mathrm{queue}(ch, \_ \centerdot tr) \overset{\mathsf{def}}{\Longrightarrow} \mathrm{queue}(ch, tr) \end{cases} \tag{11}$$

where, when a frame $fm$ is sent to a channel $\overline{ch}$ (represented by the event $\overline{ch}!fm$), if $\overline{ch} = ch$, the frame $fm$ is appended to the end of the resulting list. When a frame is delivered to destination (represented by the event $\overline{ch}?fm$), if $\overline{ch} = ch$ and the current value of the resulting list is $fm \centerdot fms$, the new value of the resulting list becomes $fms$. When an interference happens (represented by the event $\circledast(\overline{ch})$), if $\overline{ch} = ch$ and the current value of the resulting list is $fm \centerdot fms$, the new value of the resulting list becomes $fms$. For all other cases, the resulting list remains unchanged.

## VI. The construction of SW

The sliding window protocol SW can now be defined as the composition of four concurrent sub-systems, each corresponding to an entity in Figure 2:

$$\mathsf{SW} \overset{\mathsf{def}}{\Longrightarrow} \mathsf{Supplier} \parallel \mathsf{Sender} \parallel \mathsf{Channel} \parallel \mathsf{Receiver}$$

The definition of Supplier is given in Figure 6. There is only one rule for Supplier, which says: Supplier is allowed to inject message into the system at any time.

The definition of Sender is given in Figure 7, where the operation $\mathrm{maxn}(\ldots)$ is defined in (41) and the operation $\ldots[\ldots]$ is defined in (35). The rule **sd_send** says: any message injected into the system which has not yet been acknowledged may be sent over the channel ct. The premise $\mathrm{maxn}(\mathrm{msg\_no\_acked}(tr)) = acked$ means that there is a number ,$acked$, representing the maximum number acknowledged at the moment $tr$. For any sequence numbers

$$\frac{\text{maxn}(\text{msg\_no\_acked}(tr)) = acked \quad acked < n \quad \text{msgs\_in}(tr)[n] = m}{\mathsf{Sender}(tr, \mathsf{ct}!\boxed{\mathsf{data}(n,m)})}\ \textbf{sd\_send}$$

$$\frac{\text{mcat}(\text{msg\_no\_acked}(tr)) = \bot \quad \text{msgs\_in}(tr)[n] = m}{\mathsf{Sender}(tr, \mathsf{ct}!\boxed{\mathsf{data}(n,m)})}\ \textbf{sd\_send\_init}$$

Fig. 7. Definition of Sender

$n\,(acked < n)$, if the message for $n$ has been injected into the system (specified by the premise $\text{msgs\_in}(tr)[n] = m$), then, that massage $m$ is allowed to be sent over ct after being encapsuled in $\boxed{\mathsf{data}(n,m)}$. The rule **sd_send_init** says, if there is no acknowledgement received at all (specified by the premise $\text{mcat}(\text{msg\_no\_rcvd}(tr)) = \bot$), then, any message injected into the system is allowed to be sent over the the channel ct.

When defining an entity, the observation functions used must be observable by that entity. For example, the two observation functions msg_no_acked and msgs_in in Figure 7 are all observable by the Sender.

The definition of Receiver is given in Figure 8, where the operation $|\ldots|$ is defined in (33) and the operation $\text{mcat}(\ldots)$ is defined in (42). The rule **rv_out** says: messages stored in the finite map $\text{msgs\_rcvd}(tr)$ is allowed to be delivered sequentially. The observation function $|\text{msgs\_out}(tr)|$ records the number of messages delivered at the moment denoted by $tr$, therefore, the next message to deliver must have $|\text{msgs\_out}(tr)|$ as its sequence number. If this message is available in $\text{msgs\_rcvd}(tr)$ (specified by the premise $(\text{msgs\_rcvd}(tr))(|\text{msgs\_out}(tr)|) = m$), then the message can be delivered by $[m]\triangleright$. The rule **rv_ack** concerns the acknowledging frame $\boxed{\mathsf{ack}(rcvd)}$. As indicated by the premise

$$\text{mcat}(\text{msg\_no\_rcvd}(tr)) = rcvd$$

$rcvd$ is the largest continuous number of the list of sequence numbers which have arrived at the Receiver.

The definition of Channel is given in Figure 9, where the rule **ch_out** specifies that: if the list of frames waiting to be delivered in channel $ch$ is not empty, then the first frame in that list is delivered to its destination. The rule **ch_lost** specifies that: when the list of frames waiting to be delivered in channel $ch$ is not empty, interference may happen to $ch$.

## VII. Proof of safeness

The proof of the Safeness (Lemma 4.1) consists of a series of lemmas. The following lemma shows that: all frames contained in channel ct are of the form $\boxed{\mathsf{data}(n,m)}$.

*Lemma 7.1:*

$$\mathsf{VT}(\mathsf{SW}, tr) \Rightarrow$$
$$fm \Subset \text{queue}(\mathsf{ct}, tr) \Rightarrow \exists\, n, m \,.\, fm = \boxed{\mathsf{data}(n,m)} \quad (12)$$

The following lemma shows that all frames contained in channel cf are of the form $\boxed{\mathsf{ack}(n)}$.

*Lemma 7.2:*

$$\mathsf{VT}(\mathsf{SW}, tr) \Rightarrow$$
$$fm \Subset \text{queue}(\mathsf{cf}, tr) \Rightarrow \exists\, n \,.\, fm = \boxed{\mathsf{ack}(n)} \quad (13)$$

The following lemma shows that when the frame $\boxed{\mathsf{data}(n,m)}$ is in channel ct, the condition $\text{msgs\_in}(tr)[n] = m$ must hold.

*Lemma 7.3:*

$$\mathsf{VT}(\mathsf{SW}, tr) \Rightarrow$$
$$\boxed{\mathsf{data}(n,m)} \Subset \text{queue}(\mathsf{ct}, tr) \Rightarrow$$
$$\text{msgs\_in}(tr)[n] = m \quad (14)$$

The following lemma shows that for all pairs $(n,m)$ in $\text{msgs\_rcvd}(tr)$, the condition $\text{msgs\_in}(tr)[n] = m$ holds.

*Lemma 7.4:*

$$\mathsf{VT}(\mathsf{SW}, tr) \Rightarrow$$
$$\text{msgs\_rcvd}(tr)(n) = m \Rightarrow$$
$$\text{msgs\_in}(tr)[n] = m \quad (15)$$

A lot of proof cases in this paper are discharged automatically by the computational mechanism of type theory, but space does not allow us to explain each of them. To illustrate, we shall explain the use of computation through the proof of Lemma 7.4.

The proof is by induction on the structure of the $\mathsf{VT}(\mathsf{SW}, tr)$. After expanding the definition of SW, there is one case for each rule in Figures 6 – 9. Since Lemma 7.4 concerns only msgs_rcvd and msgs_in, the values of which are only affected by the occurrence of $\mathsf{ct}?\boxed{\mathsf{data}(n,m)}$ and $\triangleright[m]$, all those rules which do not generate these two events have no effect on the truth of Lemma 7.4. For these 'irrelevant rules', the goal can be proved automatically by the computational mechanism of Coq, because, after expanding the definition of msgs_rcvd and msgs_in, the goal is exactly the same as the induction hypothesis.

Only two nontrivial cases are left, the one for **sp_in** and the one for **ch_out**. For **sp_in**, since the occurrence of $\triangleright[m]$ has no effect on msgs_rcvd, only msgs_in gets extended at the end, the goal can be proved easily from the induction hypothesis. For **ch_out**, when $ch = \mathsf{cf}$, the occurrence of $ch?fm$ has no effect on either msgs_rcvd or msgs_in, the goal can be proved directly from the induction hypothesis. When $ch = \mathsf{ct}$, according to Lemma 7.1, we have $fm = \boxed{\mathsf{data}(n,m)}$. From this, Lemma 7.3 and the induction hypothesis, the goal can be proved. This ends the proof of Lemma 7.4.

$$\frac{(\mathrm{msgs\_rcvd}(tr))(|\mathrm{msgs\_out}(tr)|) = m}{\mathsf{Receiver}(tr, [m]\triangleright)}\ \textbf{rv\_out} \qquad \frac{\mathrm{mcat}(\mathrm{msg\_no\_rcvd}(tr)) = rcvd}{\mathsf{Receiver}(tr, \mathsf{cf}!\boxed{\mathsf{ack}(rcvd)})}\ \textbf{rv\_ack}$$

Fig. 8. Definition of Receiver

$$\frac{\mathrm{queue}(ch, tr) = fm \centerdot fms}{\mathsf{Channel}(tr, ch?fm)}\ \textbf{ch\_out} \qquad \frac{\mathrm{queue}(ch, tr) \neq \langle\rangle}{\mathsf{Channel}(tr, \divideontimes(ch))}\ \textbf{ch\_lost}$$

Fig. 9. Definition of Channel

The following lemma shows that: if $m$ is the $n$-th element of $\mathrm{msgs\_out}(tr)$, $(n, m)$ must be contained in $\mathrm{msgs\_rcvd}(tr)$.

*Lemma 7.5:*

$$\begin{aligned}&\mathsf{VT}(\mathsf{SW}, tr) \Rightarrow \\ &\qquad \mathrm{msgs\_out}(tr)[n] = m \Rightarrow \\ &\qquad\qquad \mathrm{msgs\_rcvd}(tr)(n) = m\end{aligned} \quad (16)$$

Combining Lemma 7.5 and Lemma 7.4, it can be proved that:

*Lemma 7.6:*

$$\begin{aligned}&\mathsf{VT}(\mathsf{SW}, tr) \Rightarrow \\ &\qquad \mathrm{msgs\_out}(tr)[n] = m \Rightarrow \\ &\qquad\qquad \mathrm{msgs\_in}(tr)[n] = m\end{aligned} \quad (17)$$

which says: the $n$-th elements of $\mathrm{msgs\_out}(tr)$ is also the $n$-th elements of $\mathrm{msgs\_in}(tr)$.

The following lemma shows that: if for any $n$, the $n$-th element of $l_1$ is also the $n$-th element of $l_2$, then $l_1$ is a prefix of $l_2$.

*Lemma 7.7:*

$$(\forall\, n, a.\, l_1[n] = a \Rightarrow l_2[n] = a) \Rightarrow \exists\, l\,.\, l_2 = l_1{}^\frown\ l \quad (18)$$

By applying Lemma 7.7 to Lemma 7.6, Lemma 4.1 can be proved.

## VIII. Proof of liveness

Instead of proving Liveness (Lemma 4.2) directly, we first prove Lemma 8.1, which says: if the messages delivered by the Receiver is less than the messages injected into the Sender at least by $m$ (expressed as $\mathrm{msgs\_in}(tr) = \mathrm{msgs\_out}(tr)^\frown\ (m \centerdot l)$), then there is a valid extension of $tr$ which delivers the message $m$ (expressed as $\mathrm{msgs\_out}(tr'^\frown\ tr) = \mathrm{msgs\_out}(tr) \dotplus m$). Lemma 4.2 can be proved easily by repeated application of Lemma 8.1.

*Lemma 8.1 (One step liveness):*

$$\begin{aligned}&\mathsf{VT}(\mathsf{SW}, tr) \Rightarrow \\ &\quad \mathrm{msgs\_in}(tr) = \mathrm{msgs\_out}(tr)^\frown\ (m \centerdot l) \Rightarrow \\ &\qquad \exists\, tr'\,.\,(\mathsf{VT}(\mathsf{SW}, tr'^\frown\ tr)\ \wedge \\ &\qquad\quad \mathrm{msgs\_out}(tr'^\frown\ tr) = \mathrm{msgs\_out}(tr) \dotplus m)\end{aligned}$$

Several preliminary lemmas are needed to prove this lemma. These lemmas are explained in Appendix B.

The proof of Lemma 8.1 is: It is important to notice that: since $|\mathrm{msgs\_out}(tr)|$ is the number of messages delivered at the moment $tr$, the next message to deliver must have the number $|\mathrm{msgs\_out}(tr)|$ (the definition of $|\ldots|$ is in (33)). The proof is to find where this message resides and to show that this message will eventually be delivered by the Receiver.

Depending on the value of $\mathrm{mcat}(\mathrm{msg\_no\_rcvd}(tr))$, there are two cases:

1) When $\mathrm{mcat}(\mathrm{msg\_no\_rcvd}(tr)) = rcvd$, from Lemma 1.4, we have:

$$|\mathrm{msgs\_out}(tr)| \leq rcvd + 1$$

Depending on the value of $|\mathrm{msgs\_out}(tr)|$, there are two sub-cases:

a) When $|\mathrm{msgs\_out}(tr)| = rcvd + 1$, from this and the premise

$$\mathrm{msgs\_in}(tr) = \mathrm{msgs\_out}(tr)^\frown\ (m \centerdot l) \quad (19)$$

it can be deduced that:

$$\mathrm{msgs\_in}(tr)[rcvd + 1] = m \quad (20)$$

The next message to be delivered must have sequence number $rcvd + 1$ and this message has not arrived at the Receiver yet. We first search this message in $\mathrm{queue}(\mathsf{ct}, tr)$. According to Lemma 1.1, there could be two cases:

i) When

$$\begin{aligned}&(\exists\, hd : [\![Frm]\!], \overline{m} : \mathsf{M}, tl : [\![Frm]\!]. \\ &\quad \mathrm{queue}(\mathsf{ct}, tr) = \\ &\qquad (hd \dotplus \boxed{\mathsf{data}(rcvd + 1, \overline{m})})^\frown\ tl\ \wedge \\ &\qquad (\forall fm.\, fm \Subset hd \Rightarrow \\ &\qquad\quad fm \neq \boxed{\mathsf{data}(rcvd + 1, \overline{m})}))\end{aligned}$$

which means the message is in $\mathrm{queue}(\mathsf{ct}, tr)$.

By applying Lemma 1.2 to this, we have:

$$\exists \overline{tr}, nl\,.$$
$$\mathsf{VT}(\mathsf{SW}, \overline{tr}^\frown\ tr) \ \wedge \qquad (21a)$$
$$\text{queue}(\mathsf{ct}, \overline{tr}^\frown\ tr) = \boxed{\mathsf{data}(rcvd+1, \overline{m})}\ \centerdot\ tl \ \wedge \qquad (21b)$$
$$\text{msg\_no\_rcvd}(\overline{tr}^\frown\ tr) = nl\ \centerdot\ \text{msg\_no\_rcvd}(tr) \ \wedge \qquad (21c)$$
$$\neg(rcvd+1 \Subset nl) \ \wedge \qquad (21d)$$
$$\text{msgs\_out}(\overline{tr}^\frown\ tr) = \text{msgs\_out}(tr) \ \wedge \qquad (21e)$$
$$\text{msgs\_in}(\overline{tr}^\frown\ tr) = \text{msgs\_in}(tr) \qquad (21f)$$

By assigning $[\overline{m}]\triangleright \centerdot \mathsf{ct}?\boxed{\mathsf{data}(rcvd+1, \overline{m})} \centerdot \overline{tr}$ to $tr'$, the goal can be proved as the following: The execution of $\overline{tr}$ moves $\boxed{\mathsf{data}(rcvd+1, \overline{m})}$ to the head of ct so that the event $\mathsf{ct}?\boxed{\mathsf{data}(rcvd+1, \overline{m})}$ can be generated by using the rule **ch_out**. Since **rv_out** is waiting for the message numbered $rcvd+1$ in order to deliver the next message, the event $[\overline{m}]\triangleright$ can now be generated by **rv_out**. From (21e), it can be deduced that:

$$\text{msgs\_out}(([\overline{m}] \triangleright \centerdot \mathsf{ct}?\boxed{\mathsf{data}(rcvd+1, \overline{m})} \centerdot \overline{tr})^\frown\ tr) = \text{msgs\_out}(tr) \dotplus \overline{m} \qquad (22)$$

From (19), (21f) and (22), it can be proved that $\overline{m} = m$. From this, the specialized goal:

$$\text{msgs\_out}(([\overline{m}] \triangleright \centerdot \mathsf{ct}?\boxed{\mathsf{data}(rcvd+1, \overline{m})} \centerdot \overline{tr})^\frown\ tr) = \text{msgs\_out}(tr) \dotplus m$$

can be proved from (22).

ii) When

$$(\forall fm.\, fm \Subset \text{queue}(\mathsf{ct}, tr) \Rightarrow (\forall m\,.\, fm \neq \boxed{\mathsf{data}(rcvd+1, m)})) \qquad (23)$$

which means the message is not in ct. We are going to show that the event $\mathsf{ct}!\boxed{\mathsf{data}(rcvd+1, m)}$ will eventually happen, which sends the message numbered $rcvd+1$ over channel ct. Therefore, by the same argument as Case 1(a)i, the goal can be proved. The rules capable of generating this event are **sd_send** and **sd_send_init**. Depending on the value of $\text{maxn}(\text{msg\_no\_acked}(tr))$, there are two sub-cases:

A) When $\text{maxn}(\text{msg\_no\_acked}(tr)) = ackd$, from Lemma 1.3, it can be derived that $ackd \leq rcvd$. Therefore, we have

$$ackd < rcvd + 1 \qquad (24)$$

Now, the event $\mathsf{ct}!\boxed{\mathsf{data}(rcvd+1, m)}$ can be generated by applying **sd_send** to (24) and (20).

B) When $\text{maxn}(\text{msg\_no\_acked}(tr)) = \bot$, by applying **sd_send_init** to this and (20), the event $\mathsf{ct}!\boxed{\mathsf{data}(rcvd+1, m)}$ is generated.

b) When $|\text{msgs\_out}(tr)| \leq rcvd$, since $rcvd$ is the largest continuous sequence number observed by the Receiver, all messages less or equal to $rcvd$ have already received by the Receiver. Therefore, we have:

$$\exists \overline{m}\,.\ \text{msgs\_rcvd}(tr)(|\text{msgs\_out}(tr)|) = \overline{m} \qquad (25)$$

which makes the rule **rv_out** applicable. The goal can be proved easily.

2) When $\text{mcat}(\text{msg\_no\_rcvd}(tr)) = \bot$, which means there is no largest continuous sequence number in all the frames observed by the Receiver. Therefore, it can be deduced that the message with sequence number 0 has not been observed by the Receiver. From Lemma 1.4, it can be deduced that $|\text{msgs\_out}(tr)| = 0$. Therefore, a frame $\boxed{\mathsf{data}(0, \overline{m})}$ must arrive for the Receiver to deliver the next message.

a) When $\boxed{\mathsf{data}(0, \overline{m})}$ is already in ct, a similar argument to Case 1(a)i can be used to prove the goal.

b) When $\boxed{\mathsf{data}(0, \overline{m})}$ is not in ct. By applying Lemma 1.3 to the fact that

$$\text{mcat}(\text{msg\_no\_rcvd}(tr)) = \bot$$

it can be deduced that:

$$\text{maxn}(\text{msg\_no\_acked}(tr)) = \bot \qquad (26)$$

By a similar argument to (20), we have:

$$\exists \overline{m}\,.\ \text{msgs\_in}(tr)[0] = \overline{m} \qquad (27)$$

By applying **sd_send_init** to (26) and (27), the event $\mathsf{ct}!\boxed{\mathsf{data}(0, m)}$ is generated. After the occurrence of this event, a similar argument to Case 1(a)i can be used to prove the goal.

This ends the proof of Lemma 8.1.

## IX. Related work and discussion

Similar approach has been used by Paulson to verify security protocols [17], [18]. Our work differs from Paulson's in the following sense:

- In section II, a formal notion of concurrent systems and their parallel composition is proposed, which is not present in Paulson's work. We believe, this makes the conception much more clearer.
- While Paulson's approach mainly deals security protocols, our approach is proposed as a general approach for protocol verification. To serve this purpose, a method to

specify and treat liveness is proposed in this paper, which is absent in Paulson's work as well.

As summarized in [19], a lot of experiments have been carried out in Focus, which, similar to our approach, is based on event traces and standard mathematics notation as well. However, the Focus allows infinite event traces, which is treated using denotational approach. The work in this paper shows that: the additional complexity of infinite traces is not strictly necessary.

In [20], Yodaiken proposed using basic mathematical languages directly. Our approach is much of the same spirit as theirs, but with more stress on mechanical support. Additionally, the notion of 'Specification by Observation' in this paper is absent in [20].

Gimenez [21] verified Alternate Bit Protocol using Coq co-inductive type, Bezem and Groote [22] verified the same protocol using an axiomatic approach. Both these two work are done using Coq, and both are using language embedding of process algebras. Therefore, despite the fact that we are using the same proof assistant as theirs, our approach is quite different from theirs.

## X. Conclusion

This paper proposes a approach of protocol verification in type theoretical proof assistant Coq. We believe that this approach is applicable to other type theory based proof assistants as well. A simplified version of sliding window protocol is used as an illustrating example. It shows that both safeness and liveness can be formalized and reasoned about by using *only* finite traces. Such a simplified notion of trace leads to efficient reasoning, so that the overall verification of a non-trivial protocol can be carried out within reasonable cost. The approach of 'specification by observation' is used so that many cases can be proved automatically by using the computational mechanism intrinsic to type theory.

## Appendix

### A. Auxiliary Definitions

*1) Exceptional Set:* For $A : \texttt{Set}$, exceptional set $\mathcal{M}(A)$ is defined as:

$$\begin{array}{c} \dfrac{A : \texttt{Set}}{\bot : \mathcal{M}(A)}\ \textbf{bottom\_value} \\ \dfrac{A : \texttt{Set} \quad a : A}{\mathsf{return}(a) : \mathcal{M}(A)}\ \textbf{normal\_value} \end{array} \tag{28}$$

Intuitively, $\mathcal{M}(A)$ represents the type obtained from $A$ by adding a special element $\bot$ to represent undefined value. A normal element $a$ of the original type $A$ is represented as $\mathsf{return}(a)$ in exceptional set $\mathcal{M}(A)$. However, the return is usually omitted in the sequel, unless there is possibility of confusion.

For $el : \mathcal{M}(A)$, $f : A \rightarrow \mathcal{M}(B)$, the operation $el \succcurlyeq f$ is used to compute a value in $\mathcal{M}(B)$, which is defined as:

$$\begin{cases} \mathsf{return}(a) \succcurlyeq f \overset{\mathsf{def}}{\Longrightarrow} f(a) \\ \bot \succcurlyeq f \overset{\mathsf{def}}{\Longrightarrow} \bot \end{cases} \tag{29}$$

*2) List:* In the type theory of Coq, data types used in concrete reasoning usually reside in the universe Set. So, the assertion '$A$ is a data types' is written as $A : \texttt{Set}$.

For any data types type $A : \texttt{Set}$, another data types $[\![A]\!]$ is defined for the lists made of elements from $A$:

$$\begin{array}{c} \dfrac{A : \texttt{Set}}{\langle\rangle : [\![A]\!]}\ \textbf{nil\_value} \\ \dfrac{A : \texttt{Set} \quad a : A \quad l : [\![A]\!]}{a \centerdot l : [\![A]\!]}\ \textbf{cons\_value} \end{array} \tag{30}$$

Intuitively, $\langle\rangle$ is the empty list, and the operator $\centerdot$ is used to put $a : A$ (an element of $A$) to the head of $l : [\![A]\!]$ (an element of $[\![A]\!]$).

The append operation $l_1 {}^\frown l_2$ is defined as:

$$\begin{cases} \langle\rangle {}^\frown l_2 \overset{\mathsf{def}}{\Longrightarrow} l_2 \\ (a \centerdot l_1) {}^\frown l_2 \overset{\mathsf{def}}{\Longrightarrow} a \centerdot (l_1 {}^\frown l_2) \end{cases} \tag{31}$$

The operation $l \dotplus a$ is used to append element $a$ to the end of $l$:

$$l \dotplus a \overset{\mathsf{def}}{\Longrightarrow} l {}^\frown (a \centerdot \langle\rangle) \tag{32}$$

The operation $|l|$ is used to compute the length of a list, which is defined as:

$$\begin{cases} |a \centerdot l| \overset{\mathsf{def}}{\Longrightarrow} |l| + 1 \\ |\langle\rangle| \overset{\mathsf{def}}{\Longrightarrow} 0 \end{cases} \tag{33}$$

The relation $i \mathrel{\underline{\in}} l$ means $i$ is contained in list $l$, which is define as:

$$\begin{cases} \bar{i} \mathrel{\underline{\in}} (i \centerdot l) \overset{\mathsf{def}}{\Longrightarrow} \bar{i} = i \ \vee\ (\bar{i} \mathrel{\underline{\in}} l) \\ \bar{i} \mathrel{\underline{\in}} \langle\rangle \overset{\mathsf{def}}{\Longrightarrow} \mathsf{False} \end{cases} \tag{34}$$

The operation $l[n]$ is used to select the $n$-th elements of $l$. It is defined as:

$$\begin{cases} a \centerdot l[0] \overset{\mathsf{def}}{\Longrightarrow} a \\ a \centerdot l[n+1] \overset{\mathsf{def}}{\Longrightarrow} l[n] \\ \langle\rangle[n] \overset{\mathsf{def}}{\Longrightarrow} \bot \end{cases} \tag{35}$$

*3) Finite Mapping:* Given any two type $A$ and $B$, $A \rightharpoonup B$ is the type of finite mappings from $A$ to $B$, which is defined as:

$$A \rightharpoonup B \overset{\mathsf{def}}{\Longrightarrow} A \rightarrow \mathcal{M}(B) \tag{36}$$

The empty mapping, which does not map anything, is written as $\epsilon$, and is defined as:

$$\epsilon \overset{\mathsf{def}}{\Longrightarrow} \lambda\, a : A.\, \bot \tag{37}$$

Intuitively, for finite map $\chi : A \rightharpoonup B$, $a : A$, $b : B$, the assertion $\chi(a) = b$ means: the mapping $\chi$ maps $a$ to $b$.

The operation $\chi\,[b^{\,a}]$ is used to add a new map '$a$ to $b$' to $\chi$, overriding the original maps for $a$ in $\chi$, it is formally defined as:

$$\begin{cases} \chi\,[b^{\,a}]\,(\overline{a}) \overset{\mathsf{def}}{\Longrightarrow} b & \text{if } \overline{a} = a \\ \chi\,[b^{\,a}]\,(\overline{a}) \overset{\mathsf{def}}{\Longrightarrow} \chi(\overline{a}) & \text{otherwise} \end{cases} \tag{38}$$

*4) Largest Continuous Number:* Suppose $nl$ is a list of natural numbers, if $n$ is the largest number in $nl$ such that all natural numbers less or equal to $n$ are also in $nl$, then $n$ is called the 'largest continuous number' in $nl$. This concept is used in our construction of the sliding window protocol.

To define largest continuous number constructively, we first need to define the concept of continuous number, if $n$ is in $nl$ and all the natural numbers less or equal to $n$ are also in $nl$, then $n$ is a continuous number in $nl$. This conception is captured by $(\!|nl|\!)_{0\ldots n}$, which is defined as:

$$\begin{cases} (\!|nl|\!)_{0\ldots 0} \overset{\mathsf{def}}{\Longrightarrow} 0 \mathrel{\underline{\in}} nl \\ (\!|nl|\!)_{0\ldots(n+1)} \overset{\mathsf{def}}{\Longrightarrow} (n+1) \mathrel{\underline{\in}} nl \ \wedge\ (\!|nl|\!)_{0\ldots n} \end{cases} \tag{39}$$

The operation $(\!|nl|\!)[\overline{nl}]$ is defined to compute the list of natural numbers in $\overline{nl}$ which are continuous in $nl$:

$$\begin{cases} (\!|nl|\!)[n \centerdot \overline{nl}] \overset{\mathsf{def}}{\Longrightarrow} n \centerdot (\!|nl|\!)[\overline{nl}] & \text{if } (\!|nl|\!)_{0\ldots n} \\ (\!|nl|\!)[n \centerdot \overline{nl}] \overset{\mathsf{def}}{\Longrightarrow} (\!|nl|\!)[\overline{nl}] & \text{if } \neg(\!|nl|\!)_{0\ldots n} \\ (\!|nl|\!)[\langle\rangle] \overset{\mathsf{def}}{\Longrightarrow} \langle\rangle \end{cases} \tag{40}$$

The operation $\mathrm{maxn}(nl)$ is defined to compute the largest natural number in $nl$:

$$\begin{cases} \mathrm{maxn}(n \centerdot nl) \overset{\mathsf{def}}{\Longrightarrow} \\ \qquad \mathrm{maxn}(nl) \succcurlyeq \lambda\,\overline{n}.\ \mathsf{if}\ n < \overline{n}\ \mathsf{then}\ \overline{n}\ \mathsf{else}\ n \\ \mathrm{maxn}(\langle\rangle) \overset{\mathsf{def}}{\Longrightarrow} \bot \end{cases} \tag{41}$$

With all these preliminary definitions, the largest continuous natural number in $nl$ can be computed with $\mathrm{mcat}(nl)$, which is defined as:

$$\mathrm{mcat}(nl) \overset{\mathsf{def}}{\Longrightarrow} \mathrm{maxn}((\!|nl|\!)[nl]) \tag{42}$$

## *B. Lemmas used in the proof of Lemma 8.1*

The following lemma is about searching for frame $\boxed{\mathsf{data}(n,m)}$ with sequence number $n$ in a list of frames $l$. It shows that: there could be only two cases: either such a frame is found, or there is not such a frame in $l$ at all.

*Lemma 1.1:*

$$\begin{aligned} &\forall\, l : [\![Frm]\!]\,, n\,. \\ &\quad (\exists\, hd : [\![Frm]\!],\ m : \mathsf{M},\ tl : [\![Frm]\!]. \\ &\qquad l = (hd \dotplus \boxed{\mathsf{data}(n,m)})^\frown\ tl\ \wedge \\ &\qquad (\forall fm.\, fm \mathrel{\underline{\in}} hd \Rightarrow fm \neq \boxed{\mathsf{data}(n,m)})) \ \vee \\ &\quad (\forall fm.\, fm \mathrel{\underline{\in}} l \Rightarrow (\forall\, m\,.\, fm \neq \boxed{\mathsf{data}(n,m)})) \end{aligned}$$

The list $hd$ is the list of frames ahead of $\boxed{\mathsf{data}(n,m)}$, which contains no frames with sequence number $n$, $tl$ is the list of frames behind $\boxed{\mathsf{data}(n,m)}$.

The following lemma shows that: if $\boxed{\mathsf{data}(n,m)}$ is in $\mathrm{queue}(\mathsf{ct}, tr)$, there is a valid execution $tr'$ extending $tr$, which delivers all frames ahead of $\boxed{\mathsf{data}(n,m)}$. Additionally, the extension does not change the values of $\mathrm{msgs\_out}$ and $\mathrm{msgs\_in}$.

*Lemma 1.2:*

$$\begin{aligned} &\mathsf{VT}(\mathsf{SW}, tr) \Rightarrow \\ &\mathrm{queue}(\mathsf{ct}, tr) = (hd \dotplus \boxed{\mathsf{data}(n,m)})^\frown\ tl) \Rightarrow \\ &(\forall fm.\, fm \mathrel{\underline{\in}} hd \Rightarrow fm \neq \boxed{\mathsf{data}(n,m)}) \Rightarrow \\ &\exists\, tr', nl\,. \\ &\quad \mathsf{VT}(\mathsf{SW}, tr'^\frown\ tr)\ \wedge \\ &\quad \mathrm{queue}(\mathsf{ct}, tr'^\frown\ tr) = \boxed{\mathsf{data}(n,m)} \centerdot tl\ \wedge \\ &\quad \mathrm{msg\_no\_rcvd}(tr'^\frown\ tr) = nl \centerdot \mathrm{msg\_no\_rcvd}(tr)\ \wedge \\ &\quad \neg(n \mathrel{\underline{\in}} nl)\ \wedge \\ &\quad \mathrm{msgs\_out}(tr'^\frown\ tr) = \mathrm{msgs\_out}(tr)\ \wedge \\ &\quad \mathrm{msgs\_in}(tr'^\frown\ tr) = \mathrm{msgs\_in}(tr) \end{aligned}$$

The following lemma shows: if there exists a maximum acknowledged sequence number $ackd$ at the Sender side, there must be a largest continuous received sequence number $rcvd$ at the Receiver side. Additionally, we have $ackd \leq rcvd$.

*Lemma 1.3:*

$$\begin{aligned} &\mathsf{VT}(\mathsf{SW}, tr) \Rightarrow \mathrm{maxn}(\mathrm{msg\_no\_acked}(tr)) = ackd \Rightarrow \\ &\quad \exists\, rcvd\,. \\ &\qquad \mathrm{mcat}(\mathrm{msg\_no\_rcvd}(tr)) = rcvd\ \wedge\ ackd \leq rcvd \end{aligned}$$

The following lemmas shows: if there is a largest continuous received sequence number $rcvd$ at the Receiver side, the number of messages delivered by Receiver is less than or equal to $rcvd + 1$; Otherwise, the number of messages delivered by Receiver is 0.

*Lemma 1.4:*

$$\begin{aligned} &\mathsf{VT}(\mathsf{SW}, tr) \Rightarrow \\ &\quad (\mathrm{mcat}(\mathrm{msg\_no\_rcvd}(tr)) = rcvd \Rightarrow \\ &\qquad |\mathrm{msgs\_out}(tr)| \leq rcvd + 1)\ \wedge \\ &\qquad (\mathrm{mcat}(\mathrm{msg\_no\_rcvd}(tr)) = \bot \Rightarrow |\mathrm{msgs\_out}(tr)| = 0) \end{aligned}$$

## REFERENCES

[1] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications," Austin, Tech. Rep., 1985.

[2] G. J. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: $10^{20}$ states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, June 1992.

[4] K. M. Chandy and J. Misra, *Parallel Program Design : a Foundation*. Reading, Mass.: Addison-Wesley, 1988.

[5] A. U. Shankar and S. S. Lam, "A stepwise refinement heuristic for protocol construction," *ACM Transactions on Programming Languages and Systems*, vol. 14, no. 3, pp. 417–461, July 1992.

[6] M. Abadi and L. Lamport, "The existence of refinement mappings," DEC Research Center, Palo Alto, CA, Tech. Rep. 29, 1988. [Online]. Available: ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-029.pdf

[7] ——, "Composing Specifications," in *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness*, ser. LNCS, J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., vol. 430. Berlin, Germany: Springer-Verlag, 1989, pp. 1–41. [Online]. Available: ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-066.pdf

[8] C. A. R. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

[9] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1989.

[10] N. Lynch and M. Tuttle, “An introduction to Input/Output automata,” *CWI-Quarterly*, vol. 2, no. 3, pp. 219–246, 1989.

[11] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. New York: Springer, 1992.

[12] L. Lamport, “The Temporal Logic of Actions,” Digital Equipment Corporation, Systems Research Centre, Tech. Rep. 79, Dec. 1991. [Online]. Available: ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-079.pdf

[13] D. Harel, “Statecharts: a visual approach to complex systems,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[14] G. Huet, G. Kahn, and C. Paulin-Mohring, “The Coq proof assistant, A tutorial, version 5.10,” Inria, Institut National de Recherche en Informatique et en Automatique, Technical Report RT-0178, 1995.

[15] E. Gimenez, “A tutorial on recursive types in Coq,” Inria, Institut National de Recherche en Informatique et en Automatique, Technical Report RT-0221, 1998.

[16] C. Cornes, J. Courant, J. Filliatre, G. Huet, P. Manoury, C. Munoz, C. Murthy, Christine, A. Saibi, and B. Werner, “The Coq proof assistant, reference manual, version 7.0,” INRIA (Institut National de Recherche en Informatique et en Automatique), Technical Report RT-0177, 2001.

[17] L. Paulson, “Proving properties of security protocols by induction,” in *10th IEEE Computer Security Foundations Workshop (CSFW ’97)*. Washington - Brussels - Tokyo: IEEE, June 1997, pp. 70–83.

[18] ——, “Proving security protocols correct,” in *14th Symposium on Logic in Computer Science (LICS’99)*. Washington - Brussels - Tokyo: IEEE, July 1999, pp. 370–383.

[19] M. Broy, M. Breitling, B. Schtz, and K. Spies, “Summary of case studies in Focus - Part II,” Technische Univerität München, Tech. Rep. TUM-I9740, 1997. [Online]. Available: http://www4.informatik.tu-muenchen.de/reports/TUM-I9740.html

[20] V. Yodaiken and K. Ramamritham, “Verification of a reliable net protocol,” in *Symp. on Formal Techniques in Real-Time and Fault Tolerant Systems*, 1992, pp. 194–215.

[21] M. A. Bezem and J. F. Groote, “A formal verification of the alternating bit protocol in the calculus of constructions,” Dept. of Philosophy, Utrecht University, Logic Group Preprint Series 88, Mar. 1993.

[22] E. Giménez, “Co-inductive types in Coq: An experiment with the alternating bit protocol,” Laboratoire de l’Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Research report RR 95-38, June 1995. [Online]. Available: ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR95/RR95-38.ps.Z