Technical Report

UCAM-CL-TR-183
ISSN 1476-2986

Number 183

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

The specification and verification of sliding window protocols in higher order logic

Rachel Cardell-Oliver

October 1989

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

The Specification and Verification of Sliding Window Protocols in Higher Order Logic

Rachel Cardell-Oliver

Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG

October 1989

1 Introduction

This report describes the formal specification and verification of a class of sliding window protocols using higher order logic. It is proved that a model for implementations of the protocol logically implies safety and liveness invariants and that these invariants in turn imply an abstract specification of the protocol. The specification states that a sequence of data is transferred from one computer in a network to another after some elapsed time. The implementation model is an interpretation of the behaviour, over time, of the variables of a Pascal program for a sliding window protocol. Real time is modelled explicitly. This means that timeouts, channel delays, response times and retransmission limits can be expressed directly and real protocol implementations can be modelled. All levels of the protocol model and proof are expressed in higher order logic, and proofs are checked by the HOL proof assistant. My aim is to demonstrate, by use of a non-trivial example, the feasibility of protocol verification using formal methods and computer checked deductive proofs.

The specification and implementation models used in the proof are based on techniques developed for hardware verification in HOL at Cambridge [4,8,11]. The physical structure of a protocol and its environment is reflected in the structure of the logical predicates which model it. Variables of the protocol's programs are modelled as functions of time. I have proved the total correctness - that is safety and liveness - of a general model for a class of protocols. This model and proof will eventually be used as the basis for a more general sliding window protocol model and for correctness proofs for real implementations of specific sliding window protocols.

1.1 Program Correctness

Verifying that a computer system is *correct* means showing that an implementation of the system *satisfies* its specification. This process is often done informally when a computer program is written. A specification for the program is given in English, and from that a programmer produces executable computer code. Informal methods are prone to errors and misunderstandings. Formal verification, however, uses formal languages for specification and implementation models, and a mathematical framework for correctness. A

formal programming language semantics can be used as the model for implementations. The programmer aims to prove that an implementation satisfies its specification.

Two techniques for protocol verification are model checking and deductive proof. Model checking involves showing that a specification will be satisfied by all possible models of a protocol implementation, where the model represents all execution states which may be reached by the protocol. Verification by model checking is a form of proof by exhaustion. It has the advantage that the model checking can be performed automatically by a computer, but there is the disadvantage that the state space for realistic protocols will be very large, perhaps making exhaustive search infeasible. Deductive proofs for various logics, which do not involve exhaustive model checking, may avoid this problem. Some of the advantages of automated model checking proofs can be retained if a computer is used to check, assist with, and manage the proofs written by a human verifier. The use of the HOL theorem prover for higher order logic in such a role is described in this report.

Formal methods can be used to verify that an implementation satisfies its specification, but there remain a number of other ways in which computer systems may not be correct. For example, a formal specification may contain errors; it may not specify what the designer intended or may not correctly model the environment with which the program must interact. Tools are needed to help designers check that their formal specifications actually model their intentions. A model for an implementation is a mathematical model of a real physical entity (a computer running a program) and, like all models, may incorrectly model some aspects of that entity. Experience with formal models for real systems and comparisons between models and their real counterparts should gradually improve our ability to model computer systems. Only the first problem, verifying that an implementation model satisfies its specification, will be considered here.

1.2 A Framework for Protocol Verification

I shall distinguish between algorithms, implementations, and specifications. A specification for a protocol, for example, defines *what* the protocol must do without defining *how* to achieve it. An algorithm defines how the specification could be achieved, but in general terms, and an implementation is a realization of an algorithm. An implementation is expressed in terms of particular computer programming languages, compilers and computer hardware. A description of an algorithm, by contrast, should have sufficient information from which to derive an implementation, but be sufficiently general to lead to many different implementations for different computers, programming languages and environments.

A sliding window protocol may be described at many different levels of abstraction; from computer hardware, at the lowest level, to a logical description of its data transfer function at the highest level. A long term aim of the work reported here is to show that a programming language description of a protocol satisfies a high level specification of the protocol's function and that the hardware and compiler which realize the programming language code correctly implement the semantics of that programming language.

I contend that it is important to develop methods for the verification of protocol implementations rather than algorithms. This is because, at least for the class of data transfer protocols, although the algorithms are well known, experience shows that writing correct implementations of these algorithms is a difficult task. I have chosen to work with the most general high level specification possible. At the highest level, an abstract specification should be satisfied by any protocol from the class of sliding window protocols. This

report describes the verification of a protocol model which is somewhere between a general algorithm and an implementation. Work is in progress to develop good implementation models.

There are many different styles of protocol specification for different requirements at various levels of abstraction. Specifications can be classified into those which provide an outside view of the protocol and those giving an inside view. From an outside view, a protocol's programs and communication channel are treated as a black box which consumes input and produces output. Such a specification would be used to implement different protocols which perform the same task. An inside view of a protocol defines what services the protocol provides. This type of specification would be used to implement a protocol which needs to communicate with other protocols written using the same specification. Either style of specification may be written as logical assertions using the notation of mathematics or as procedural descriptions in, say, pseudo program code.

I have used an outside view style of specification to express the relationship between the input and output data streams of a sliding window protocol. Time is the only other variable used in the specification. This specification defines the function of any sliding window protocol without details of its implementation. Both safety and liveness requirements for an implementation are encapsulated in the specification. Other protocol properties such as performance characteristics and inside view specifications can be defined separately.

An implementation is a physical entity and so in order to prove formal properties of that implementation a mathematical *model* must be developed for the implementation. An important problem is deciding how much detail should be modelled, and in what manner. Formal programming language semantics define an interpretation for the parts of an implementation written in programming language code. Alternatively, programming language compilers and computer hardware can be modelled. The latter choice provides a concrete interpretation of program code and the network environment. The performance properties of operating systems, computer load, network activity and various timing models for hardware could also be taken into account in an implementation model.

My implementation model consists of a set of assertions in higher order logic which define the behaviour of the protocol's programs and the channel through which they communicate. The program model defines the changing values of the variables of Pascal style programs. The communication channel is modelled in a similar manner. The relationship between the program code and the model is an informal one at present. However, formal models such as an operational semantics for the programming language will be used eventually.

This section has introduced a method for defining an implementation and specification for a protocol as assertions in higher order logic which describe the same physical system at different levels of abstraction. An implementation model satisfies its specification if the former logically implies the latter.

1.3 Related Work

The work presented in this report combines ideas from the fields of protocol specification and verification and theorem proving using higher order logic. Specifically, this report investigates :

- the verification of total correctness (that is, safety and liveness properties) of protocols [6,10,12,15]

- models for real-time properties of protocols [8,10,12,15]
- modular specifications and proofs [4,6,8,10,11,12,14,18,19]
- computer aided theorem proving [1,4,8,12,17,18,19]
- the verification of a non-trivial example: a sliding window protocol with cyclic sequence numbers [1,3,13] (see also [6,9,16,19])
- the use of higher order logic for specificaton and verification [4,8,10,11]
- proofs that a protocol implementation model satisfies a high level functional specification rather than proving that protocol algorithms satisfy their specifications [1,6]

Hailpern and Owicki [6] used Floyd-Hoare verification techniques to prove safety and liveness properties of Stenning's data transfer protocol (a sliding window protocol without cyclic sequence numbers). Temporal logic was used to describe liveness properties, auxilliary history variables were used to simplify verification proofs, and pre-, post- and liveness assertions about small parts of a protocol were combined to verify larger parts. Hailpern's method has many of the same goals as mine. However, the use of temporal logic to express liveness rather than modelling real time precludes the proofs of correctness of specific implementations since these rely on the correct setting of timeout, delay, response time and retransmission intervals. Hailpern's implementation model is written in programming language code, but his liveness commitments rely on assumptions that events enabled infinitely often will eventually occur, whereas I have chosen to model protocol implementations with explicit real-time bounds.

Shankar and Lam have developed methods for verifying safety, liveness and real time properties of protocols [14,13,15]. Shankar has modelled a protocol with a set of assertions representing guarded events which may be performed by the protocol. Events are guarded (or enabled) by predicates on state variables. An event may occur at any time at which the state variables satisfy its predicate guard. An event may cause messages to be transmitted, state variables to be updated and timers to be updated. Timing properties such as timeouts are modelled by counter events which increment a counter subject to assertions controlling the time differences between local clocks. Modular proofs can be written using image protocols [14] in which only a subset of the total state variable space is considered at any time. Proofs were not checked automatically, but hand proofs of various protocol properties have been reported. For example, safety properties of a sliding window protocol algorithm with cyclic sequence numbers, given real-time constraints, were verified in [13] and safety properties of the HDLC sliding window protocol in [14]. Safety, liveness and real-time properties of alternating bit protocols (a simple case of the general sliding window protocol) were proved in [15]. Shankar and Lam's event model does not attempt to model programming language code, but rather an algorithm for the protocol.

Sunshine [17] has reported on a project to investigate the use of the AFFIRM theorem prover for protocol verification in which he has modelled protocols as extended finite state machines. These in turn were modelled in AFFIRM as abstract data structures. For example, an AFFIRM data structure for a two state machine contains a state type of possible states (say A and B) and defines the operations allowed on variables of that type (say transitions from A to B and B to A). The protocol machines are 'extended' in that state variables may be used to store values such as the current state of a communication channel, or a counter variable. Verification work cited in [17] included proofs of safety,

liveness and performance properties for a wide range of protocols. As with Shankar's work, the implementation model of [17] does not model programming language code, and inside view specifications were used instead of outside view or black box specifications.

DiVito [18,19] verified the safety of a sliding window protocol (*without* cyclic sequence numbers) using the Boyer Moore theorem prover. His process model uses buffers, message histories and guarded events on state variables. Proofs of liveness or real-time properties were not attempted. Brand [1] proved safety properties of a sliding window protocol with cyclic sequence numbers using automatic symbolic execution but, again, the model reported cannot be used to prove liveness properties. Brand proved the correctness of a sliding window protocol implementation while DiVito proved the correctness of a protocol algorithm.

Crocker's state delta formalism has been applied to alternating bit protocols by Overman and Crocker [12]. Their proof model uses symbolic execution (see also [1]) for the automatic proof of safety, liveness and real time properties and also supports modular specifications. To my knowledge, these methods have not been applied to more general implementations such as the generalised sliding window protocol described in this report.

Finally, work on the use of higher order logic and the HOL theorem prover for higher order logic for verifying hardware [4,11,8] and real time distributed systems [10] has influenced the protocol verification style proposed here. A number of abstraction mechanisms have been developed for hardware verification with higher order logic [11]. Joyce [8] used HOL to verify a hardware version of a handshaking protocol. MacEwen [10] combined the structural abstraction methods of [4] with event occurrence functions to develop a theory for the verification of hard real time algorithms. I do not know of any examples of the use of MacEwen's theory for practical verification examples.

2 Sliding Window Protocols

2.1 A Definition

A protocol is a program for communication between machines in a computer network. Sliding window protocols are a class of protocols which transfer an ordered stream of data from one computer to another. The term *sliding window protocol* is used in the literature to describe a wide range of specific protocols as well as classes of protocols. I shall use the following class definition.

A sliding window protocol transfers an ordered sequence of data messages between two computers via an unreliable channel. In the literature, assumptions about the nature of the unreliable channel vary. I shall assume that the communication channel carries messages in both directions but may lose, garble, reorder or duplicate them. Message delivery time is bounded but variable within that bound. *How* data is transferred in this environment is defined in Section 2.3 which outlines an implementation model for this protocol.

2.2 A Sliding Window Protocol Specification

Let the input sequence for a sliding window protocol be called the source. The source is available on one computer and the output sequence, which is called the sink, is produced on another computer. Since the sink data list is dynamic it will not be available until some time after the protocol has begun and it is modelled as a function from time to finite

time $(0 < t_1)$	0	t_1
source	[1,2,3]	[1,2,3]
sink	[]	[1,2,3]

Figure 1: An Unrealistic System Satisfying the Specification

time $(0 < t_1 < t_2)$	0	t_1	t_2
source	[1,2,3]	[1,2,3]	[1,2,3]
message sent	1	2	3
message received	1	2	3
cumulative sink	[1]	[1,2]	[1,2,3]

Figure 2: Messages transmitted one by one, but with 0 delay

output data lists. Time starts at time 0 and extends infinitely into the future in discrete time steps. Thus $sink(t)$ refers to the value of the ouput sequence, *sink*, at time t where $t \geq 0$. The source data list is a static variable, its value is determined outside the protocol model and never changed by the model, so *source* is just modelled as a finite list of data messages and does not have a time parameter. The specification for a sliding window protocol can be described mathematically by Equation 1.

$$\exists t : time.\ sink(t) = source \tag{1}$$

Figure 1 shows an example of the behaviour of a system which satisfies Equation 1.

In practice, the list of data to be transferred will usually be too large to be sent in one operation. Instead, items are transmitted one at a time. Figure 2 illustrates how, with the unrealistic assumption that the delivery time for messages is 0, Figure 1 would be realised.

Another unrealistic assumption of Figure 1 is that the channel reliably delivers messages. In fact, the channel which transfers data between the computers is unreliable and may lose or garble transmitted messages. Thus, messages may need to be transmitted several times before they are correctly received. This is illustrated for message 2 in Figure 3, where the symbol '-' represents a lost or garbled message.

Finally, the unrealistic assumption of 0 delay is dropped. There will be a variable but bounded delay, introduced by the channel, between sending a message and either receiving

time $(0 < \cdots < t_4)$	0	t_1	t_2	t_3	t_4
source	[1,2,3]	[1,2,3]	[1,2,3]	[1,2,3]	[1,2,3]
message sent	1	2	2	2	3
channel	1	2	2	2	3
message received	1	-	-	2	3
cumulative sink	[1]	[1]	[1]	[1,2]	[1,2,3]

Figure 3: System where messages may be lost, but still 0 delay

time ($0 < \ldots < t_7$)	0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
source (S=[1,2,3])	S	S	S	S	S	S	S	S
message sent	1	-	2	-	2	-	3	-
channel	1	-	2	-	2	-	3	-
message received	-	1	-	-	-	2	-	3
cumulative sink	[]	[1]	[1]	[1]	[1]	[1,2]	[1,2]	[1,2,3]

Figure 4: Sample Execution with Lost and Delayed Messages

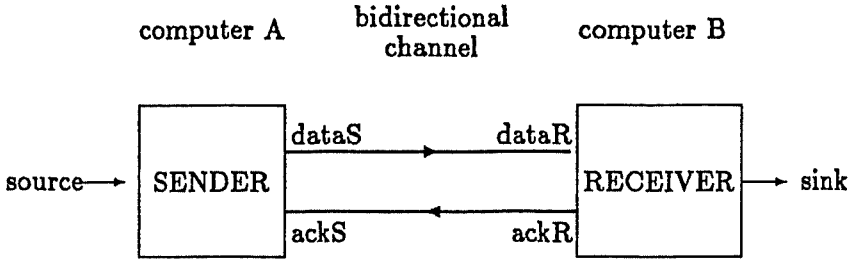

Figure 5: Physical environment for a sliding window protocol

it, or the message being garbled or lost. The symbol '-' is used in Figure 4 to represent no message as well as a garbled or lost message.

The specification of Equation 1 expresses an outside view of the protocol at a high level of abstraction. At this level the programs implementing the protocol and the channel connecting them are treated as a black box in which the source is consumed and the sink produced. I shall now describe how an implementation achieves this specification.

2.3 A Sliding Window Protocol Implementation

Figure 5 shows the type of environment in which the sliding window protocol would be executed. Two computers are connected by a bidirectional communication channel. One program, the sender, is executed on the computer which stores the source data and another program, the receiver, accepts and outputs the data on another computer. The two programs run concurrently. The network which contains the computers, including the wire between them, provides a hard real time environment with which the programs must interact. That is, the programs are unable to control some aspects of their environment and actions must be taken in real time. For example, the channel may lose packets (which is beyond the control of the programs) and the programs should respond to an incoming message before another arrives.

— ACTIONS ORDERED IN TIME —

SENDER	CHANNEL	RECEIVER
send message 1		
	message lost	
resend message 1		
	message delivered	
		message 1 received and output
		send ack 1
	ack lost	
resend message 1		
	message delivered	
		duplicate message 1 not ouput
		send ack 1
	message delivered	
ack 1 received		

Figure 6: Lost data and acknowledgement messages in a typical transaction

2.3.1 Positive Acknowledgement

The technique used by a sliding window protocol to achieve reliable data transmission over an unreliable channel is called *positive acknowledgement.* Every data message transmitted by a sender carries a label which distinguishes it from other transmitted messages. The label provides a receiver with sufficient information to output messages in their source order and to discard duplicated messages. A label, message pair is called a *packet.* For each data packet received the receiver returns an *acknowledgement* packet to the sender. An acknowledgement packet contains a copy of the message packet's label signifying a particular data message has been received.

Since packets may be lost or garbled the sender and receiver may need to retransmit packets (see Figure 4). The sender retransmits a packet for which it does not receive an acknowledgement within a certain time. The receiver reads the label of each incoming packet and, if the label is in order, outputs the packet's message to the sink. Otherwise, the packet is discarded. The receiver does, however, send an acknowledgement for the latest correctly received packet on reception of any incoming packet lest earlier acknowledgements have been lost. An example of a typical transaction which uses this retransmission strategy is illustrated in Figure 6.

2.3.2 Sliding Windows

The *windows* of a sliding window protocol determine how much of the input or output data sequences are visible to a sender or receiver. Windows may be implemented as message buffers. The windows slide because once the data in the window has been acknowledged (for the sender) or output (for the receiver), the window slides up to reveal new input messages or space for output messages. Window size is a parameter of an implementation. For example, the alternating bit protocol has both sender and receiver window size 1, while the HDLC protocol has sender window size 7 or, in extended mode, 127.

At one extreme, when the sender's window size is unlimited, the sender uses all of the input data at once, transmitting any message at any time. At the other extreme, sender window size one, messages are transmitted one at a time and the sender always waits for

an acknowledgement before transmitting the next message. The sender window always contains data messages which have been (or may be) transmitted and have not yet been acknowledged.

The receiver window size determines which messages may be accepted and possibly output to the sink. The choice is made on the basis of incoming message labels. If the receiver window has size one then only messages which can be output immediately are accepted. If the window has a size greater than one then some messages can be buffered to be output in the future.

The size of sender and receiver windows affects the choice of labels. Informally, a label can only be reused once any old message carrying that label is guaranteed to be out of the system. Let the sender's window size be SW, the receiver's window size RW and the range of labels be $maxseq$. Message labels are natural numbers in the range $0, \ldots, maxseq - 1$ called sequence numbers and the sender selects labels in the cycle $0, 1, 2, \ldots, maxseq - 1, 0, 1, \ldots$.

In this report, I consider a sliding window protocol with receiver window size one. For this case, the range of distinguishable labels must be at least one greater than the sender's window size :

$$(maxseq \geq SW + 1) \ \wedge \ (RW = 1)$$

The reason for this restriction is best illustrated with an example. Suppose that the above constraint on SW is not met because $SW = maxseq$. Then the sender may transmit messages $0 \ldots maxseq - 1$ before acknowledgements for any of those messages have been received. Suppose the receiver does, in fact, receive all messages and sends the appropriate acknowledgement message but this acknowledgement is lost. Then when the sender resends messages $0 \ldots maxseq - 1$, the receiver will not know whether its acknowledgement packet has been correctly received and these messages are new ones, reusing the old sequence numbers, or whether the messages are a retransmission of the first $maxseq$ messages. Thus the receiver would be unable to correctly deliver messages to its sink. However, for $SW < maxseq$ it can be shown that the receiver will always be able to distinguish between new and duplicate messages on the basis of their sequence numbers.

In the more general case, where the receiver's window is greater than one, the range of sequence numbers required is

$$maxseq \geq 2 \times RW \ \wedge \ SW = RW$$

Ideas for modelling this more general protocol are discussed in Section 6.4.

Usually, window sizes are chosen to maximize the efficient use of a protocol's communication channel. For example, SW should be large enough to ensure the sender is continuously transmitting data while waiting for acknowledgements. However, if the channel delay between sending and receiving a message is much larger than the time for a program to put that message onto the channel, then the previous strategy could result in unreasonable demands for a window's buffer space and a smaller window size might be chosen (see Section 6.3).

2.3.3 A Pascal Implementation

The Pascal code of Figure 7 is an example of the program code which would be executed by the sender and receiver programs of Figure 5. The programs show how the mechanisms of positive acknowledgement and sliding windows are combined in a sliding window protocol. The protocol illustrated uses the go-back-n retransmission strategy. The sender's window

size in this example is 7, the receiver's window size 1, and *maxseq* is 8, so the sender may transmit up to 7 packets before it receives an acknowledgement for the first packet. Since the receiver's window size is 1, it can only accept packets in order and has no buffer space for out of order packets.

The protocol's communication channel is represented by the four variables `dataS`, `dataR`, `ackR`, `ackS` corresponding to the same variable names in Figure 5. The predefined procedures `SEND` and `RECV` transmit and receive the packets assigned to `dataS`, `ackR` and `dataR`, `ackS` respectively. The procedure `RECV` returns a dummy packet if no incoming packet is available on the channel. The behaviour of the network and the channel will be stated directly in higher order logic in the full proof.

The program also uses the predefined proceure `IN_WINDOW` which checks whether a valid packet, i.e. not a dummy packet, has been received and if so whether its label is within the sender or receiver's window. Executing `(IN_WINDOW P bw ws maxseq)` checks that packet `P` is not a dummy packet and then that its label is between the bottom of the window, `bw`, and its top, `((bw+ws) mod maxseq)` where `ws` is the window size, and `maxseq` the range of sequence numbers used as labels.

Since acknowledgements may be lost, the sender may miss some low acknowledgements (acks) but receive higher ones. For example, messages 0 to 6 are transmitted but acks 0,1,2,3 are lost and only acks 4,5 and 6 are received. From this the sender can deduce that messages 0,1,2,3 have been received too because the receiver only updates its acknowledgement number when it accepts packets in order. The loop `(while s<>stop)` in the `SENDER` program manages this situation. The functions `choosei` and `headi` find the next message within the sender's window to be transmitted.

3 Verification in Higher Order Logic

3.1 An Introduction to Higher Order Logic

The HOL system is a theorem prover for higher order logic derived from LCF [5]. The version of higher order logic used here is based on [2] and an introduction to the HOL system can be found in [4]. Higher order logic contains all the terms of first order logic and also contains higher order terms : predicates or functions with predicates and functions as parameters. HOL is a typed logic so each term has a well defined type. Figure 8 summarises the subset of HOL which will be used in this report.

3.2 Modelling Structure with HOL

The technique of modelling hardware devices as HOL predicates which describe the behaviour of those devices is well established [4,7,8,11]. Parameters of predicates are used to model physical connections or other shared information and local, existentially quantified variables are used for local or hidden information. The conjunction of predicates is used to model concurrent behaviour. These techniques are also suitable for protocol verification. For example, in a simplified model of the sliding window protocol consisting of transmission and reception devices (computer programs) and a wire device (the channel between them) the protocol's structure can be modelled by :

$$\begin{aligned}&\mathsf{IMPL}\ \mathit{inseq}\ \mathit{outseq} \equiv \exists C : \mathit{channel}.\\ &(\mathsf{TRANS}\ \mathit{inseq}\ C) \wedge (\mathsf{WIRE}\ C) \wedge (\mathsf{RECP}\ \mathit{outseq}\ C)\end{aligned}$$

The internal structure of each of these predicates can be defined in a similar manner.

```
program SENDER;

{external variables}
  source : input list of data
  ackS   : input from channel via ackR
  dataS  : output to channel

const maxseq = 8; {for example}
      SW = 7;     {SENDER window}

type data = num;    {or char or integer or record etc.}
     sequence = [0..maxseq-1]  {message labels}

var rem : list of data;
    s, stop, i: sequence;

begin
  rem:=source;
  s:=0;
  while not(NULL rem) do begin

{receive ack packet}
    RECV( ackS ) ;
    if IN_WINDOW(ackS,s,SW,maxseq)
    then begin
      stop:=(label(ackS)+1) MOD maxseq;
      while s<>stop do begin
        rem:=tail(rem);
        s:= (s+1) MOD maxseq;
      end; {while}
    end {if};

{transmit data}
    i:=choosei(SW);
    dataS:=((s+i) MOD maxseq,
            headi(i,rem));
    SEND( dataS )
  end {while};
end {program}.
```

```
program RECEIVER;

{external variables}
  sink  : output list of data
  ackR  : output to channel
  dataR : input from channel via dataS

 {constants and types as SENDER}
   RW = 1;     {RECEIVER window}

var r : sequence;
    dummy : data;

begin
  sink:=[];
  r:=0;
  while TRUE do begin

{receive data packet}
   RECV( dataR ) ;
   if IN_WINDOW(dataR,r,RW,maxseq)
   then begin

     sink:=append(sink, dataR);
     r:= (r+1) MOD maxseq;

   end {if};

{transmit ack}

  ackR:=((r-1) MOD maxseq, dummy);

  SEND( ackR )
  end {while};
 end {program}.
```

Figure 7: Pascal Programs for a Sliding Window Protocol

SYMBOL	MEANING
T	Truth
F	Falsity
$\neg p$	not p
$p \wedge q$	p and q
$p \vee q$	p or q
$p \Longrightarrow q$	p implies q
$P(x)$ or Px	Property P of x
$\forall x.P\ x$	for all x, property P x is true
$\exists x.P\ x$	there exists at least one x such that P x is true
$x = y$	polymorphic equality : x and y have the same type and same value
$a \Rightarrow b \mid c$	if a then b else c
$x < y$, $x \leq y$	x is less than y, x is less than or equal to y
bool	boolean type : T and F
num	natural number type : 0,1,2,3,...
one	the type with only one element: *one*
$*$	polymorphic type: $*$ is a type variable
list	list type : [1,2,3] is a list
hd, tl, NIL, []	list operators: head, tail, and two notations for the empty list
NULL l	NULL l is true if l is the empty list and false otherwise
APPEND l_1 l_2	APPEND returns a list which is the concatenation of two lists of the same type, l_1 and l_2
$ty_1 \rightarrow ty_2$	type of function with domain ty_1 and range ty_2
$ty_1 \times ty_2$	cartesian product type
FST, SND	FST(x,y) = x, SND(x,y) = y
$ty_1 + ty_2$	disjoint union type. Use this type with the following functions :
INL, INR	Injections for inserting elements into a variable of the sum type.
OUTL, OUTR	Projections out of the sum
ISL, ISR	Test whether an element of a sum is the left or right summand

Figure 8: HOL syntax

3.3 Modelling Behaviour and Time with HOL

A protocol can be described by the changing values of all the data it manipulates. This may be modelled in higher order logic as functions representing the program variables of an implementation from time to their values.

Time is modelled by the natural numbers. Time steps are discrete, and time is defined from a starting time, 0, into an infinite future. The same time scale is shared by all parts of the protocol model, and any assertion can refer to time explicitly using variables of the *time* type.

As an example of the use of this time model consider a counter, $C : time \rightarrow num$, which is updated whenever some condition $P : time \rightarrow bool$ is true or a timeout occurs. A timeout occurs if C has not been updated in the last max time steps. The following assertion models C where C_t means $C(t)$.

$$C_0 = 0 \ \wedge \ \forall t.(P_t \vee (C_{t-max} = C_t \wedge t \geq max)) \Rightarrow C_{t+1} = C_t + 1 \mid C_{t+1} = C_t$$

This assertion means that the counter is initially set to 0 and at each time step C takes on a new value which may be its old value, or its old value plus one. The counter is incremented if either P is true at time t, or if C has not been incremented in the last max time steps. The latter case is recognised when the current value of C is the same as its value max time steps earlier.

The time model described above is very simple. Possible extensions include the addition of local time scales and interrupt events. Local clocks, which may run at slightly different rates, could be modelled in a mannar similar to that of [15]. Interrupts could be modelled by event functions from *time* to *bool* which are true whenever the interrupt they represent occurs and false otherwise.

4 Specification of the Sliding Window Protocol in HOL

This section describes a model for the Pascal program of Figure 7 using higher order logic. The correspondence between the logic and the program code is that each variable of the Pascal program is modelled by a function from time to its value type. Eventually the logical model will be formally derived from program code. It should be noted that the model presented here is more general than the Pascal code of Figure 7 because it generalizes a class of sliding window protocols, of which the given Pascal program is one example.

4.1 Structure of a Sliding Window Protocol

The structure of my implementation model is illustrated in Figure 9. Each box in the diagram represents a predicate of higher order logic which defines how and when variables are updated. The full implementation model is the conjunction of all the predicates in this diagram.

The model of Figure 9 contains predicates representing a SENDER, a RECEIVER and the CHANNEL between them. The SENDER and RECEIVER transmit data packets and acknowledgements over an unreliable channel. Each is described by transmission and reception predicates for their behaviour over time: AckRecv and DataTrans describe the SENDER and DataRecv and AckTrans the RECEIVER. In addition the predicate INIT describes the initial state of SENDER and RECEIVER variables and ABORT monitors the

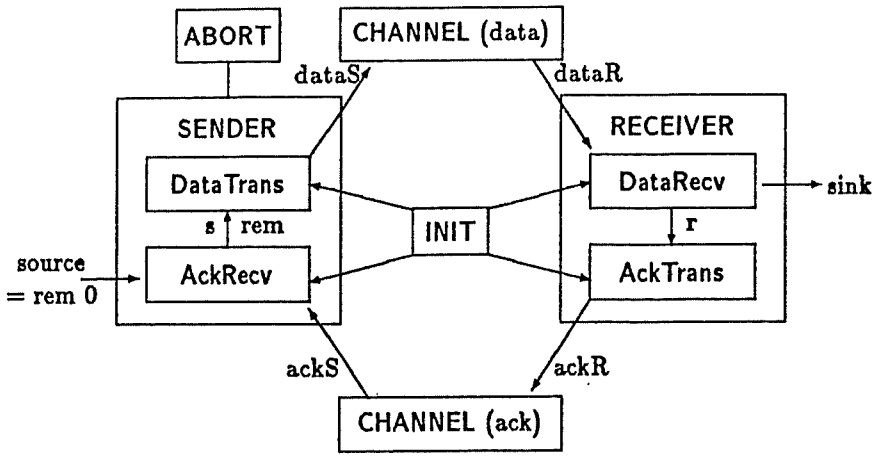

Figure 9: HOL Sliding Window Protocol Model

real time progress of the SENDER. The CHANNEL predicate defines the behaviour of an unreliable channel. The two logical channels used in this sliding window protocol are defined by (CHANNEL *dataS dataR*) and (CHANNEL *ackR ackS*). In practice, the physical realisation of these logical channels is a single channel such as a coaxial cable, telephone circuit, optical fibre, or satellite channel.

4.2 Behaviour of a Sliding Window Protocol

This section describes a model for the behaviour of a sliding window protocol as defined using the HOL system. The values of functions of time, say C, at time t will be denoted by C_t to improve readability.

4.2.1 Preliminary Definitions

Data messages are modelled by a polymorphic type *data* which might be instantiated to characters, integers, bits or records by a particular implementation. Sequence numbers are modelled by a *sequence* type which is equivalent to the natural numbers. Data and acknowledgement packets may be of two types:

- a sequence number, data message pair, or
- a lost or garbled packet.

The HOL model for this is the disjoint union, or *sum*, type. The following definition means that a packet is either a sequence number, data message pair, or an element of type *one*. Type *one* is the type with only one element *one* which is used to represent a lost or garbled packet in this context. The main types used in the proof are :

time = *num*
data = $*$
sequence = *num*
NonPacket = *one*
packet = (*sequence* $\times$ *data*) + *NonPacket*

The four variables which represent the bidirectional channel between the SENDER and RECEIVER programs are modelled by the *channel* type. A channel is a function from *time* to *packet* which returns the value of the packet being transmitted or received on the channel at any instant of time.

$$channel = time \rightarrow packet$$

The next two definitions are simply to improve the readability of HOL specifications for packet operations. The first constant is for assigning the value 'lost or garbled' to a packet variable. The second function checks that the packet, p, which is passed as a parameter, is *not* a lost or garbled packet.

$$\begin{aligned} &\mathsf{SetNonPacket} = \mathsf{INR}(one) \\ &\mathsf{GoodPacket}(p : packet) = (\mathsf{ISL}\ p) \end{aligned}$$

A new packet is created from a label and a message using NewPacket.

$$\mathsf{NewPacket}(ss : sequence)(dd : data) = \mathsf{INL}(ss, dd)$$

Those fields can be extracted from an incoming packet using label and message.

$$\begin{aligned} \mathsf{label(p:}\ packet) &= \mathsf{FST(OUTL}\ p) \\ \mathsf{message(p:}\ packet) &= \mathsf{SND(OUTL}\ p) \end{aligned}$$

The symbols $\oplus$ and $\ominus$ are used to represent functions which perform addition and subtraction modulo *maxseq*, the range of sequence numbers used by a specific implementation. The functions HDI and TLI return the head or tail of a list starting from a given position within the list.

4.2.2 Top Level Definitions

The complete sliding window protocol is represented by the following HOL predicate:

IMPL
source : *data list* *maxseq* : *sequence* *rem* : *time* → *data list*
s : *time* → *sequence* *SW* : *sequence*
p : *time* → *sequence* → *bool* *i* : *time* → *sequence*
aborted : *time* → *bool* *c* : *time* → *num* *maxT* : *num*
sink : *time* → *data list*
r : *time* → *sequence* *RW* : *sequence* *q* : *time* → *bool*
dataS : *channel* *dataR* : *channel*
ackS : *channel* *ackR* : *channel* ≡

INIT *source maxseq rem s SW sink r RW* ∧
SENDER *maxseq SW rem s p i dataS ackS* ∧
ABORT *c aborted maxT maxseq SW s rem ackS* ∧
CHANNEL *dataS dataR* ∧
RECEIVER *maxseq RW sink r q ackR dataR* ∧
CHANNEL *ackR ackS*

where

SENDER $maxseq\ SW\ rem\ s\ p\ i\ dataS\ ackS$ $\equiv$
DataTrans $rem\ s\ SW\ maxseq\ p\ i\ dataS$ $\wedge$
AckRecv $ackS\ SW\ maxseq\ rem\ s$

and

RECEIVER $maxseq\ RW\ sink\ r\ q\ ackR\ dataR$ $\equiv$
AckTrans $r\ maxseq\ q\ ackR$ $\wedge$
DataRecv $dataR\ RW\ maxseq\ sink\ r$

From now on the types of predicate parameters will not be given: they are the same as those given above for IMPL.

4.2.3 Middle Level Definitions

The starting state of the protocol is given by the INIT predicate. In a network environment an agreement protocol would be used to establish these initial values but a sliding window protocol may assume that the values have already been agreed upon.

INIT $source\ maxseq\ rem\ s\ sink\ r$ $\equiv$
$1 < maxseq \wedge rem_0 = source \wedge s_0 = 0 \wedge sink_0 = \mathsf{NIL} \wedge r_0 = 0$

DataTrans chooses any packet from the SENDER window to transmit. The SENDER's window consists of the first SW elements of rem_t. The message chosen is the i-th element of rem, $HDI\ i_t\ rem_t$, for $i_t < SW$. Its sequence number is $s_t \oplus i_t$ (that is $s_t + i_t$ mod $maxseq$) where s_t is the true sequence number of the first element of rem_t. The choice function p guarantees that i_t is within the window and that there are at least i_t data messages available in rem_t. If p_t is false then a non-packet is transmitted instead. With the intention of making this implementation model as general as possible, only the minimum information needed about p, its type, is given at this stage. Later, different implementations can be defined by more fully defining such parameters. For example, in Section 6 it is shown how this general definition of p and i can be strengthened to define an efficient retransmission strategy. The conjunct $\neg \mathsf{NULL} rem_t$ is included in DataTrans so that, after all data has been transmitted, non-packets are transmitted forever. That is, the program never terminates although the protocol will reach a stable state once transmission of source data is completed.

DataTrans $rem\ s\ SW\ maxseq\ p\ i\ dataS$ $\equiv$
$SW = maxseq - 1 \wedge$
$\forall\ t : time.$
$(p_t i_t \Longrightarrow \neg\ \mathsf{NULL}\ (\mathsf{TLI}\ i_t\ rem_t) \wedge i_t < SW) \wedge$
$(p_t i_t \wedge \neg\ \mathsf{NULL}\ rem_t)$
$\Rightarrow dataS_t = \mathsf{NewPacket}\ (s_t \oplus i_t)\ (\mathsf{HDI}\ i_t\ rem_t)$
$\mid dataS_t = \mathsf{SetNonPacket}$

The packet, $dataS_t$, which is produced by DataTrans is then transferred by the channel (CHANNEL $dataS\ dataR$) to $dataR_t$.

A zero delay channel either copies its input to its output or loses or garbles it. The latter is modelled by outputing a non-packet. The copy or loss operation occurs on each tick of the global clock.

$$\text{CHANNEL}\ \ A : \mathit{channel}\ \ B : \mathit{channel}\ \ \equiv$$
$$\forall t : \mathit{time}.\quad (B_t = A_t) \vee (B_t = \text{SetNonPacket})$$

The resulting packet in $dataR$ is read by DataRecv. If $dataR$ is acceptable then the state variables for the RECEIVER window, r, and output list, $sink$, are updated, otherwise their old values are maintained. The DataRecv predicate shown is designed for a RECEIVER window of size 1.

For any packet, p, (InWindow $p\ b\ ws\ maxseq$) tests whether p is inside the window by testing whether label p is less than window-size, ws, sequence numbers from the bottom edge of the window, b.

$$\text{InWindow}\ \ p\ b\ ws\ maxseq\ \ \equiv$$
$$\text{GoodPacket}\ p\ \ \wedge\ \ ((\text{label}\ p) \ominus b) < ws$$

$$\text{DataRecv}\ \ dataR\ RW\ maxseq\ sink\ r\ \ \equiv$$
$$RW = 1\ \ \wedge$$
$$\forall\, t : \mathit{time}.$$
$$\quad \text{InWindow}\ \ dataR_t\ r_t\ RW\ maxseq$$
$$\quad\quad \Rightarrow (r_{t+1} = r_t \oplus 1)\ \ \wedge$$
$$\quad\quad\quad (sink_{t+1} = \text{APPEND}\ sink_t\ [\text{message}\ dataR_t])$$
$$\quad\quad \mid\ \ r_{t+1} = r_t\ \ \wedge\ \ sink_{t+1} = sink_t$$

The variables updated in DataRecv are used in the next acknowledgement packet. Since r is the value of the next sequence number expected, the acknowledgement sequence is one less than r to represent the sequence number of the last message correctly received. The boolean function q is like p in DataTrans. An acknowledgement packet is only transmitted at times when q_t is true. The data field of an acknowledgement packet is never used, and can be filled with any dummy data message.

$$\text{AckTrans}\ \ r\ maxseq\ q\ ackR\ \ \equiv$$
$$\forall\, t : \mathit{time}.\ \ \forall\, dummy : \mathit{data}.$$
$$\quad q_t \Rightarrow ackR_t = \text{NewPacket}\ (r_t \ominus 1)\quad (dummy)$$
$$\quad\quad \mid\ ackR_t = \text{SetNonPacket}$$

Finally, $ackR$ is transmitted by the return channel, (CHANNEL $ackR\ ackS$), to $ackS$ and is checked by AckRecv

$$\text{AckRecv}\ \ ackS\ SW\ maxseq\ rem\ s\ \ \equiv$$
$$\forall\, t : \mathit{time}.$$
$$\quad \text{InWindow}\ \ ackS_t\ s_t\ SW\ maxseq$$
$$\quad\quad \Rightarrow s_{t+1} = (\text{label}\ ackS_t) \oplus 1\ \ \wedge$$
$$\quad\quad\quad rem_{t+1} = \text{TLI}\ (s_{t+1} \ominus s_t)\ rem_t$$
$$\quad\quad \mid\ \ s_{t+1} = s_t\ \ \wedge\ \ rem_{t+1} = rem_t$$

4.2.4 Monitoring Progress

The liveness, or progress, of the sliding window protocol is monitored by the ABORT predicate which would be part of the sender's program code in an implementation. The

ABORT predicate counts the lengths of times during which no valid acknowledgement arrives. If any such waiting period exceeds *maxT* then *aborted* is set to true and remains so. It is up to the program which is using the protocol to decide what action to take. The *aborted* variable is needed for the verification of liveness because it can be proved that the protocol will eventually deliver all its data as long as *aborted* remains false. Note that *aborted* will be always false once all *source* data has been delivered (NULL rem_t).

$$\begin{array}{l}\mathsf{ABORT}\ \ c\ aborted\ maxT\ maxseq\ SW\ s\ rem\ ackS\ \ \equiv \\ \quad c_0 = 0\ \ \wedge\ aborted_0 = \mathsf{F}\ \ \wedge \\ \quad\quad \forall\, t : time. \\ \quad\quad\quad c_{t+1} = ((\mathsf{InWindow}\ \ ackS_t\ \ s_t\ \ SW\ \ maxseq) \Rightarrow 0 \mid c_t + 1)\ \ \wedge \\ \quad\quad\quad aborted_{t+1} = (\ (c_t \geq maxT \vee aborted_t)\ \ \wedge \neg\ \mathsf{NULL}\ rem_t\ )\end{array}$$

The above is a 'real time' interpretation of liveness and models the solution used in most implementations. It can be contrasted with traditional liveness statements which say that if an event is continuously enabled then it will eventually occur. Real time liveness is a safety property: one that holds at all times. In fact real time liveness implies traditional liveness, so it is a stronger property. The following example illustrates the difference between them.

Let $p : time \rightarrow num$ be a non-decreasing function. Progress is said to have been made if $p_{t+1} > p_t$, otherwise $p_{t+1} = p_t$ and no progress has been made. An example of a real time liveness statement is: 'always, in MAX time steps from now, progress has been made or the attempt abandoned'.

$$\forall t.(p_{t+MAX} > p_t) \vee (aborted_{t+MAX})$$

The last statement may be contrasted with traditional liveness statements of the type: 'if the choice between progress and non-progress is presented infinitely often, then there exists some time in the future at which the progress choice will be chosen'.

$$(\forall t.\ (p_{t+1} = p_t \vee p_{t+1} > p_t)) \implies (\forall t.\ \exists t'.\ (t' > t)\ \wedge\ (p_{t'} > p_t))$$

5 Verification of the Sliding Window Protocol in HOL

I have proved that the sliding window protocol implementation as described above implies its specification (see Equation 1). The proof uses several intermediate theorems: a safety theorem about the relationship between the input and output lists, *source* and *sink*, and the windows of the protocol and a liveness theorem about the progress made by an implementation.

The following abbreviations will be used to describe the proof. Variables are universally quantified unless otherwise indicated.

$$\begin{array}{rcl}\mathbf{SPEC} & \equiv & \exists t : time.\ sink(t) = source \\ \\ \mathbf{IMPL} & \equiv & \mathsf{IMPL}\ source\ maxseq\ rem\ s\ SW\ p\ i\ c\ aborted\ maxT \\ & & sink\ r\ RW\ q\ dataS\ dataR\ ackS\ ackR \\ \\ \mathbf{SAFETY} & \equiv & \forall t : time. \\ & & \mathsf{APPEND}\ sink_t\ (\mathsf{TLI}\ r_t \ominus s_t\ rem_t)\ \ =\ \ source\end{array}$$

The main safety theorem is

IMPL $\Longrightarrow$ SAFETY.

Its proof uses induction over time, case analysis and a number of lemmas. The most important lemmas relate the contents of received packets to those of transmitted packets.

LEMMA1 $\equiv$
$\forall\, t : time.$
AckTrans $r\ maxseq\ q\ ackR\ \wedge$
CHANNEL $ackR\ ackS\ \wedge$
InWindow $ackS_t\ s_t\ SW\ maxseq$
$\Longrightarrow$
label $ackS_t\ =\ (r_t \ominus 1)$

LEMMA2 $\equiv$
$\forall\, t : time.$
DataTrans $rem\ s\ SW\ maxseq\ p\ i\ dataS\ \wedge$
CHANNEL $dataS\ dataR\ \wedge$
$RW = 1\ \wedge$
InWindow $dataR_t\ r_t\ RW\ maxseq\ \wedge$
$0 < maxseq$
$\Longrightarrow$
HDI $(r_t \ominus s_t)\ rem_t\ =$ message $dataR_t\ \wedge$
$\neg$ NULL(TLI $(r_t \ominus s_t)\ rem_t)\ \wedge$
$(r_t \ominus s_t) + 1 < maxseq$

This completes the safety part of the proof. Liveness is defined in terms of the time the sender has been waiting for a good acknowledgement; one which will allow it to progress. This definition of liveness avoids counting how many times a particular packet has been transmitted, or specifying that repeatedly transmitted packets will eventually be delivered as a result of fairness properties of the channel. All these types of liveness reasonably well capture the property of real channels that they are unlikely to continuously *not* deliver packets. The parameters $maxT$, p and i can easily be chosen so that the definition of liveness given below implies traditional liveness specifications based on fairness over a range of transmissions.

The liveness proof requires only SENDER and ABORT definitions. The following predicate describes the states of progress which the SENDER can reach. The disjuncts in the conclusion of this theorem represent

- completion : the list of data still to send is now empty,
- progress : the list of data still to send has shortened,
- no progress : the protocol has been aborted.

LIVE_CHOICES $\equiv$
INIT $source\ maxseq\ rem\ s\ SW\ sink\ r\ RW\ \wedge$
SENDER $maxseq\ SW\ rem\ s\ p\ i\ dataS\ ackS\ \wedge$
ABORT $c\ aborted\ maxT\ maxseq\ SW\ s\ rem\ ackS$
$\Longrightarrow$
$\forall\, t : time.$

$$\mathsf{NULL}\ rem_{t+maxT} \ \vee$$
$$(\exists\ x : num.\ (rem_{t+maxT} = \mathsf{TLI}\ x\ rem_t) \ \wedge\ (0 < x)\ ) \ \vee$$
$$aborted_{t+maxT+1}$$

Liveness is not a property which can be enforced, because a hard real time environment, such as a computer network, imposes its own behaviour on protocol programs. Instead, having proved that the protocol will either make progress or be aborted, *assume* that the protocol is not aborted and work from there. Let **LIVE-ASSUM** be the assumption that the protocol is *not* aborted.

$$\textbf{LIVE-ASSUM} \equiv \forall\ t{:}time.\quad (\neg\ aborted_{t+maxT+1})$$

It can now be proved that the protocol will reach a state where all the source data has been correctly delivered. In fact, an upper bound can be given:

$$\textbf{LIVENESS} \equiv \quad rem_{maxT \times LENGTH(rem_0)} = \mathsf{NIL}$$

and it is proven that the **LIVE-CHOICES** with **LIVE-ASSUM** implies **LIVENESS**. Finally:

$$\textbf{IMPL} \wedge \textbf{LIVE-ASSUM} \Longrightarrow \textbf{SAFETY} \wedge \textbf{LIVENESS}$$

and

$$\textbf{SAFETY} \wedge \textbf{LIVENESS} \Longrightarrow \textbf{SPEC}$$

5.1 Constructing the Proof

The proof described in the last section was written over several months. That time included my learning HOL and much time spent remodelling the protocol and rewriting the proofs. However, I would still expect a similar proof for a new class of protocols (such as handshaking protocols) to take at least a month for modelling, and a month for the proof. The main part of the definitions and theorems for the total correctness proof is a file of approximately 1200 lines. This is supported by another 2700 lines of general definitions and proofs (for example a theory of the integers and modulo arithmetic).

6 Discussion

This section describes ways in which the sliding window protocol model defined in this report might be made more general. Some directions for longer term work are also suggested.

6.1 Modelling Delay

The most unrealistic part of my protocol model is that channels deliver messages with no delay. Some possible communication channel characteristics which occur in physical media are :

1. Bounded but variable transmission delay with packets received, if at all, in sending order.

2. Bounded but variable transmission delay with packets possibly reordered by the transmission medium.

3. Bounded but variable transmission delay with packets possibly spontaneously duplicated by the transmission medium.

4. Packets may be damaged during transmission but not lost i.e. something is received for every packet transmitted.

5. Transmission delay is unbounded.

All of the above characteristics, except the last, are covered by the general delay channel model proposed below. Case 5 will not be modelled because no protocol can guarantee eventual delivery of a stream of data in this case.

The new channel predicate, DCHANNEL, uses a delay function which maps a time to the period for which a packet transmitted at that time would be delayed in the network channel. The time at which a packet is delivered is always later than its transmission time, but not more than *maxd* time units later. Within these boundaries, delivery time is variable.

$$\mathsf{DCHANNEL}\ \mathit{In} : \mathit{channel}\ \mathit{Out} : \mathit{channel}\ d : \mathit{time} \rightarrow \mathit{time}\ \mathit{maxd} : \mathit{time} \equiv$$

$$(\mathit{Out}_t = \mathit{In}_{t-d_t} \vee \mathit{Out}_t = \mathsf{SetNonPacket}) \wedge (d_t > 0 \wedge d_t \leq \mathit{maxd})$$

Originally, I intended to extend the zero delay channel model to one with delay by showing that a delay channel was equivalent to a particular transmission strategy. That is, delaying a packet in the channel is equivalent to delaying its transmission time. The latter can be expressed by refining the definitions of p and i in DataTrans and q in AckTrans. However, in a general sliding window protocol a delay channel introduces sequences of events which could not occur with zero delay. For example, an acknowledgement for a packet may arrive after that packet was retransmitted. Thus, in general, the timing behaviour of a channel is too complex to be expressed simply as a type of transmission strategy. This experience suggests that the time behaviour of a sliding window protocol is fundamental to its correctness and that

6.2 An Efficient Transmission Strategy

In the generalised sliding window protocol neither of the time functions i or p are fully defined. Any data packet may be transmitted so long as it is within the sender's window. A typical transmission strategy for real implementations is to send all the data in the window, from the bottom to the top, and then return to the bottom of the window to retransmit any packets for which an acknowledgement has not arrived. This may be defined as :

$$\begin{aligned}&\mathsf{BottomToTop}(p) \equiv\\ &\forall t.\ p_t \wedge i_0 = 0 \wedge\\ &i_{t+1} = ((i_t \ominus s_t) \geq (s_{t+1} \ominus s_t) \Rightarrow i_t \oplus (s_{t+1} \ominus s_t) + 1 \mid 0)\end{aligned}$$

6.3 Timeouts

If source data is not always available to send or if buffer space is not sufficient to allow data to be continuously transmitted then timeouts should be used. A timeout is a retransmission strategy. For each packet transmitted, the transmission time and the time by which an acknowledgement is expected are noted. If no acknowledgement arrives within this time interval then the original packet is retransmitted.

Timeouts could be added to the model by defining p and i in DataTrans as follows : if $s_t = s_{t-timeout}$ then $p_t i_t = \mathsf{T} \wedge i_t = 0$. The test $s_t = s_{t-timeout}$ establishes whether any progress has been made by the sender.

If timeouts are being used then p, i and q should be defined so that packets are only transmitted when an incoming packet is received or if no packet has been received for some time:

$$p_t i_t = (\mathsf{GoodPacket}(ackS_t) \vee s_t = s_{t-timeout})$$

$$q_t = (\mathsf{GoodPacket}(dataR_t) \vee (\forall t'.\ (t - timeout \leq t' \wedge t' < t) \Longrightarrow \neg q_{t'}))$$

6.4 Extending the Receiver Window

When the receiver window has size 1, the loss of any packet in a transmitted sequence of packets results in the retransmission of every packet in the sequence following the damaged one. All packets will be retransmitted even if they were received intact. A more efficient strategy is for the receiver to buffer packets which arrive out of order but may be needed in the future. The receiver window defines which packets 'may be needed in the future'. Messages in the receiver's window are passed on to the sink as they become available.

A predicate defining a receiver with window size greater than one can not easily be added to the current model. This is because the extension would be more general than the existing predicate. Ideally, the most general predicate should have been my starting point, and a single window receiver predicate introduced as a particular case of this model. However, since this protocol model was intended to test the feasibility of verifying protocols in HOL, it was decided to avoid the extra complexity of a variable receiver window size until it was known how difficult such verification proofs might be.

6.5 Negative Acknowledgement Packets

A new type of acknowledgement packet may be introduced to signal when an out of order packet has arrived at the receiver. The sender can then react more quickly when packets may have been lost than when using only a timeout strategy.

A NakTrans predicate similar to AckTrans could be defined which transmits acknowledgements with a special data field (to distinguish them from normal acknowledgements) whenever

$$\mathsf{GoodPacket}(dataR_t) \wedge \neg(\mathsf{InWindow}\ dataR_t\ r_t\ RW\ maxseq)$$

6.6 Dynamic Source Data and Two-way data flow

Instead of assuming that the source data is a static list of data, as in the proof above, source data may be modelled dynamically. This situation may arise if the receiver, or the network, is imposing flow control on the sender, or if source data is being produced from another computer or process while the protocol is running. A boolean function of time, *available*, is defined to be true whenever data is available, and false otherwise. A new predicate could be defined to receive data and buffer it in *rem* as it becomes available. A protocol with dynamic source data may be non-terminating, when new data may arrive at any time, or may use a special message or signal to tell the sender that there is no more data to come. In the non-terminating case, the specification should be modified to state that for every time, t, there exists a time, t', such that $t' > t$ and all the source data made available by time t has been delivered to the sink by time t'.

Two way data flow can be modelled by adding all the RECEIVER predicates to the SENDER and vice versa and adding new logical channels for the reverse data flow and its acknowledgements. The correctness proofs given above will hold for data transmitted in either direction. Although the communication medium for the protocol is now modelled by four logical channels, in an implementation data and acknowledgements will actually be transmitted in the same packet (called piggybacking) and all messages will be transmitted over one physical channel.

6.7 Future Work

My aim has been to demonstrate the feasibility of protocol verification using the HOL proof assistant. Having established a feeling for the size of the task, I intend to extend the protocol model reported here for a more general range of sliding window protocols and to extend verification work to particular implementations of the protocol. In order to generalize the protocol model I shall investigate a wider range of abstraction techniqes and further study sliding window protocols to establish properties common to all members of the class. Verification issues which may be investigated include performance and consistency. The techniques developed for sliding window protocols might then be used for the verification of other protocol families such as three way handshake protocols and other agreement protocols.

Acknowledgements

I would like to thank Mike Gordon and Roger Hale for many helpful discussions and their practical help. Both read and commented on drafts of this report. Mike Gordon showed me how his hardware specification style could be applied to protocols and Roger Hale suggested a specification style derived from executable specifications written in the Tempura language. Roger Hale also wrote the specification and verification of an alternating bit protocol in HOL which has formed the starting point for the sliding window protocol proofs described in this report.

Mike Gordon and Avra Cohn dedicated time to help me learn HOL relatively painlessly. Tom Melham kindly shared his HOL theory of the integers which I have used for my $\oplus$ and $\ominus$ functions. I would also like to acknowledge helpful discussions with Juanito Camilleri, Steve Crocker, Jeff Joyce, Tom Melham and Roger Needham.

I am grateful for financial support from the Australian Defence Science and Technology Organisation, the Australian Committee of the Cambridge Commonwealth Trust and an Overseas Research Studentship.

References

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

[2] Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[3] R. Duke, I. Hayes, P. King, and G. Rose. Protocol specification and verification using Z. In *Protocol Specification, Testing and Verification, VIII*, pages 33–46, 1988.

[4] Mike Gordon. *A Proof Generating System for Higher-Order Logic.* Technical Report 103, University of Cambridge Computer Laboratory, January 1987.

[5] M.J.C. Gordon, A.J.R.G. Milner, and C.P. Wadsworth. *Edinburgh LCF: a mechanized logic of Computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer Verlag, 1979.

[6] Brent T. Hailpern and Susan S. Owicki. Modular verification of computer communication protocols. *IEEE Transactions on Communications*, COM-31(1):56–68, January 1983.

[7] F.K. Hanna and N. Daeche. *Specification and Verification Using Higher Order Logic.* North Holland, 1986.

[8] Jeffrey J. Joyce. *Formal Specification and Verification of Asynchronous Processes in Higher-Order Logic.* Technical Report 136, University of Cambridge Computer Laboratory, June 1988.

[9] G.M. Lundy and Raymond E. Miller. A variable window protocol specification and analysis. In *Protocol Specification, Testing and Verification, VIII*, pages 361–372, 1988.

[10] Glenn H. MacEwen and David B. Skillicorn. *Using Higher Order Logic for Modular Specification of Real-Time Distributed Systems*, pages 37–66. Springer Verlag, September 1988. Lecture Notes in Computer Science 331.

[11] Thomas F. Melham. *Abstraction Mechanisms for Hardware Verification.* Technical Report 106, University of Cambridge Computer Laboratory, May 1987.

[12] William T. Overman and Stephen D. Crocker. Verification of concurrent systems : function and timing. In *Protocol Specification, Testing and Verification, II*, pages 401–409, 1982.

[13] A. Udaya Shankar. A verified sliding window protocol with variable flow control. In *Communications Architectures and Protocols*, pages 84–91, ACM SIGCOMM, August 1986.

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

[15] A. Udaya Shankar and Simon S. Lam. Time-dependent distributed systems: proving safety, liveness and real-time properties. *Distributed Computing*, 2(2):61–79, August 1987.

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

[17] Carl A. Sunshine, David H. Thompson, Roddy W. Erickson, Susan L. Gerhart, and Daniel Schwabe. Specification and verification of communication protocols in AFFIRM using state transition models. *IEEE Transactions on Software Engineering*, SE-8(5):460–489, September 1982.

[18] Benedetto L. Di Vito. Integrated methods for protocol specification and verification. In *Protocol Specification, Testing and Verification, II*, pages 411–433, 1982.

[19] Benedetto L. Di Vito. Mechanical verification of a data transport protocol. In *Communications Architectures and Protocols*, pages 30–37, ACM SIGCOMM, March 1983.