Technical Report

UCAM-CL-TR-206
ISSN 1476-2986

Number 206

Formal verification of real-time protocols using higher order logic

Rachel Cardell-Oliver

August 1990

ISSN 1476-2986

Formal Verification of Real-Time Protocols using Higher Order Logic

Rachel Cardell-Oliver

Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG

August 1990

1 Introduction

A protocol is a distributed program which controls communication between machines in a computer network. Two or more programs are executed on different computers which communicate only via the medium connecting them. Usually, communication over the medium is unreliable.

Protocol implementations are difficult to understand and to write correctly because the interaction between programs and their non-deterministic, real time environment is complex. For this reason, protocols are often specified using an abstract model. Two cases in which abstractions are useful are

1. for the top down design of a protocol implementation
2. to verify the correctness of an implementation by showing that it is a member of a class of more general algorithms which have already been shown to be correct.

However, few abstract specification techniques model the problems which occur in real implementations. In particular, the correctness of many protocols depends on real time issues such as the correct setting of timers and fast responses to incoming messages.

This paper presents techniques for modelling real-time protocols at different levels of abstraction, from implementation behaviour to abstract requirements specifications. The language used for these models is higher order logic. The techniques are illustrated by the specification and verification of the class of sliding window protocols. The HOL system, a machine implementation of higher order logic [2], was used to both specify and verify this example and a full listing of the HOL theories for sliding window protocols is given in Appendix B.

2 Specifying Protocols in Higher Order Logic

One reason for describing protocols at different levels of abstraction is in order to verify their correctness. The idea is that a specification should be sufficiently abstract that it can be generally seen to describe the requirements the protocol should satisfy. An implementation, whose behaviour is more complex and cannot readily be seen to be correct, is then proved (formally or informally) to satisfy its specification.

In order to reason formally about the correctness of protocol implementations we require

1. a specification of the behaviour of a protocol implementation
2. a specification of the function or requirements of that protocol
3. a formalism for relating these descriptions : a relation, *satisfies*, is defined to be true if the given implementation model achieves its specification and false otherwise.

We shall call a specification which states *how* a protocol behaves a behavioural specification and one which states *what* the protocol must achieve a functional specification.

In this paper we argue that higher order logic is a suitable formal language for modelling protocols and show how techniques for specifying and verifying hardware using higher order logic can be adapted for communication protocols.

2.1 Signals

A real time computer system can be modelled by a set of *signals*. Signals represent the value of some aspect of the system's state at given sampling times. The quantity measured could represent input or output streams or internal state variables. For example, a protocol can be modelled by sampling the values input to the sender and output from the receiver. The model could also include signals representing the current internal state of the sender, receiver and channel. Possible sampling times for these quantities could be every machine cycle, every n nanoseconds, each event interrupt etc.

A signal is a function from time to values. For example, the sender's input could be modelled by a function from sampling times to the type of data which the protocol transmits. The model for time is discrete because we have assumed a system is sampled at intervals. The natural numbers are used to model time. However, each step of modelling time could have many interpretations: the sampling interval may be arbitrarily small or may represent some unspecified and variable interval of abstract time such as a state transition.

The idea of using signals to model real time systems has been widely applied in the specification and verification of hardware [11, 9, 3]. One difference in applying this method to protocols is that signals in and out of channels and to and from the protocol's outside world are likely to have more complex data types than the signals which have been used in the literature to model hardware.

The ouput stream from the receiver could be represented by a signal from time to an abitrary type, `*data`. An arbitrary type can be modelled in HOL by a type variable which can stand for any concrete type such as a record, a character, integer, n-bit word etc. Functions with domain `ty1` and range `ty2` are represented by the type `ty1 → ty2`. The receiver's output stream has type

```
time → *data
```

A channel's value at any time is either a packet or a null value representing nothing on the channel. The type constructor `+` is used to represent such a type. For example, `ty1 + ty2` represents the type whose members are either of type `ty1` or of type `ty2`. A packet is a pair containing a sequence number of type `seq` and some data of type `*data`.

This type is represented by the product type constructor ×. Values of type ty1 × ty2 have two fields: the first of type ty1 and the second of type ty2. A type non_packet, is represented by the type one which has only one element. Thus the type channel, a signal modelling values input or output from a physical channel, can be represented by [1]

```
channel = time → (seq × *data) + non_packet
```

Another difference between modelling hardware and software using higher order logic is that computer programs (software) manipulate variables stored in memory. These variables, too, can be modelled as signals: functions from time to their current values. For example, a program variable s whose type is the natural numbers can be modelled by the signal s : time → num. If the program variable, s, has value 5 at some time t' then the signal s reflects this by s t' = 5.

2.2 Functional Specifications

The relationships which must hold between signals are specified by logical predicates on those signals. These relationships may be stated functionally (as requirements) or behaviourally.

The function of a sliding window protocol could be specified by the relationship between the input signal, source:time → *data and the output signal, sink:time → *data. Assume that the data values of the source are unique and thus distinguishable in the sink.
1. every output value is a copy of an earlier input :

```
OutputWasInput(source:time → data,sink:time → data) =
    ∀ t:time.  ∃ t':time.  t'<t ∧ sink t = source t'
```

2. the order of outputs preserves the order of inputs. We assume inputs are unique and offered (possibly with duplicates) in their original order.

```
OrderPreserved(source:time → data,sink:time → data) =
    ∀ t1 t2:time.
       t1 < t2 ∧ (∃ t1'.  sink t1 = source t1') ∧
       (∃ t2'.  sink t2 = source t2')
    ⟹
       t1' < t2'
```

Higher order logic [2] is necessary for this modelling method because signals are first order functions and thus predicates with signals as parameters are second order or higher.

2.3 Behavioural Specifications

Behavioural specifications can also be represented using predicates on signals. These specifications describe how a process behaves rather than what it must achieve. For example, a counter signal, count:time → num, which is initially set to 0 and then updated whenever an input signal, tick:time → bool, is true can be specifed by:

[1]Alternatively, the type package in HOL could be used to define a channel as a structured type:

```
channel = PACKET seq *data | EMPTY void
```

[2]as opposed to first order or predicate logic

```
Counter( tick:time → bool, count:time → num ) =
   count 0 = 0 ∧
   ∀ t:time.
      count (t+1) = tick t ⇒ (count t)+1
                           | (count t)
```

The notation `a ⇒ b | c` means if `a` is true then return the value `b` otherwise `c`. It should be noted that when specifying behaviour using predicates there is an *implicit* notion of input and output signals: `tick` is an input parameter which is read by the `Counter` and `count` is an ouput parameter which is written by the `Counter`. The notion of input and output signals is implicit because it is part of an (informal) interpretation of the meaning of logical formulae rather than a property of the logic itself.

2.4 Combining Specifications and Hiding Internal Signals

As well as modelling the behaviour of single components in a system, we require a method for combining component specifications in order to specify complete systems.

Signals which occur as parameters to a number of predicates represent information which is shared. In behavioural specifications a signal may be used as output in one predicate and as input to another. For example, the `Counter` specified in the last section could be connected to a timer device which outputs an interrupt (the boolean truth value) every time the counter variable is a multiple of the constant MC.

```
Timer( count:time → num, clkint:time → bool, MC:num ) =
   ∀ t:time.  clkint t = ( ((count t) MOD MC) = 0 )
```

Predicates are combined using logical conjunction. Thus, the specification of the combined behaviour of these two devices, in which `count` is ouput by the `Counter` and input to the `Timer`, is given by `ClkInt`. The internal signals `count` and `MC` are *hidden* in `ClkInt` using existential quantification.

```
ClkInt(tick:time → bool,clkint:time → bool) =
   ∃ count:time → num.  ∃ MC:num.
     Counter(tick,count) ∧ Timer(count,clkint,MC)
```

Similarly, functional specifications are combined using logical conjunction. A functional specification for a sliding window protocol is :

```
SPEC(source:time → data,sink:time → data) =
    OutputWasInput(source,sink) ∧
    OrderPreserved(source,sink)
```

2.5 Verification: the *satisfies* relation

Behavioural and functional specifications for real time software systems such as protocols can be specified in higher order logic. How can we prove that an implementation satisfies its specification?

A relation, *satisfies*, between two predicates, `A` and `B`, which is true if the implementation model `A` meets its specification `B`, is given by [3]:

[3]Read the following formula as: A satisfies B is defined by A logically implies B

$$\texttt{A}\ \textit{satisfies}\ \texttt{B} \equiv \texttt{A} \Longrightarrow \texttt{B}$$

This relation captures the idea that an implementation is "more specified" than its specification. However, it has the undesirable property that an inconsistent implementation (one containing a statement and its negation) satisfies any specification since the logical constant false implies both true and false specifications.

Using logical equivalence instead of implication as the satisfaction relation would solve this problem, but then we could not express the idea of an implementation being more specified than its specification.

There are two types of partial solution. First, a verifier must prove more properties of her implementation model than just satisfaction. For example, a protocol should transmit messages at a rate greater than some minimum, and if a perfect channel is assumed then the protocol should terminate. We shall call such proofs the verification of "reasonableness" properties. The choice of a good set of reasonableness proofs cannot be automated and relies on the ingenuity of the protocol verifier.

A second solution is to prove properties of the *model* used for describing implementations. For example, if an implementation were described by a formal programming language semantics then the verifier could prove properties such as variables having one and only one value at any time (so that, say, `s t = 0` $\wedge$ `s t = 1` will not be physically possible). If an implementation specification does not contain inconsistencies, and that specification logically implies more abstract specifications, then the higher level specifications can also be deduced to be consistent.

2.6 HOL

HOL [2] is a theorem prover for higher order logic derived from LCF [4]. The version of higher order logic used in HOL is based on [1]. 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. HOL theorems are 'secure' in that every theorem must have a formal proof. The syntax of the subset of higher order logic used in this paper is given in Appendix A.

3 A Methodology for Modelling Protocols

3.1 A Four Level Model

In sections 1 and 2 we distinguished between a specification (which could generally be seen to be correct) and implementations (which were more complex). Between these extremes there exist a range of protocol descriptions at different levels of abstraction.

We also identified two specification styles which could be used to make formal descriptions more readable: functional (or requirements) and behavioural descriptions. In the former we describe *what* must be achieved by a protocol and in the latter we describe *how* a protocol behaves.

In this section we shall identify four levels of description with which we choose to characterize protocols: implementations (IMPL), algorithms (ALG), generic behavioural specifications (GEN), and a minimal functional specification (SPEC).

An implementation, modelled by IMPL, is a description in programming language code and hardware of an executable protocol. An algorithm, modelled by ALG, describes the behaviour of a protocol without specifying all its implementation details. A functional specification, SPEC, states what the protocol must achieve but not how. A protocol specification, GEN, describes the behaviour of the protocol but at a higher level of abstraction than in an algorithm description. Such abstract behavioural descriptions can be used to describe a *class* of algorithms. Each class specification will serve as the specification for many different algorithms. In turn, each algorithm specification will specify many different implementations.

To illustrate the difference between these levels of description consider a sliding window protocol. A minimal functional specification of a sliding window protocol is that it should transfer its input stream of data to an output stream. The output stream must preserve the order of the original input. The class of sliding window protocols achieves this specification using a number of standard techniques.

1. Physically, a sender and receiver communicate using an unreliable, bidirectional channel.

2. Data from the sender's input stream is labelled for transmission over the unreliable communication medium to enable the receiver to preserve the order of that stream.

3. A limit is placed on the number of data messages which may have been transmitted by the sender but are not known to have been output by the receiver: this is called a window.

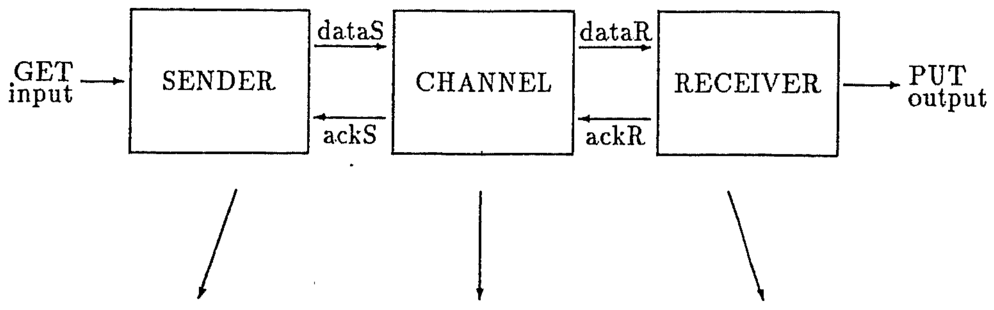
4. The mechanism for notifying the sender that data has been received is called positive acknowledgement: the receiver sends messages to the sender describing its current state.

A behavioural specification for the *class* of sliding window protocols describes how these techniques are combined to achieve the specification. Sliding window algorithms differ from one another in matters such as their choice of data for transmission, differing strategies for the receiver to show the sender its current state, and the class of communication media for which the protocol is designed. For example, TCP, HDLC and the alternating bit protocol are all sliding window algorithms. They vary in details such as their choice of window size, transmission strategies etc. An algorithm specification describes specific choices for each of these behavioural details. A protocol implementation can be described in terms of a particular programming language, computer hardware and network environment. The way in which messages are transmitted and received, and the implementation of timers and buffers should also be specified since these details may vary between implementations for the same physical environment.

The types, definitions and theorems which comprise the HOL theories representing SPEC, GEN and ALG for the class of sliding window protocol are given in Appendix B.

3.2 The use of multi-level specifications

Figure 1 describes a simple program code implementation of a protocol such as Stenning's data transfer protocol. We propose a verification strategy as follows. First, an implementation would be translated, using a formal programming language semantics, into a model in higher order logic: IMPL. We would then prove that the implementation model satisfied a particular algorithm. The algorithm would be described by a conjunction of properties chosen from a library of pre-defined protocol behaviours. The chosen algorithm would be then be proved to be a member of its protocol class. In practice, most of the required



```
s:=0; GET d;
WHILE true DO
  SEND (s,d);
  RECV ps;
  IF (OK ps) THEN
   IF (label ps)=(s+1)
     THEN (s:=s+1; GET d)
END.
```

```
Get input for each
channel and either
* deliver it after
  a random but
  bounded delay OR
* discard it

The channel may
also reorder or
duplicate its input.
```

```
r:=0;
WHILE true DO
  RECV pr;
  IF (OK pr) THEN
   IF (label pr)=r
     THEN (r:=r+1;
             PUT(mess pr));
  SEND (r,dummy)
END.
```

Figure 1: An Implementation of Stenning's Protocol

theorems would be part of the library. A theorem stating that the class algorithm satisfies its abstract specification would also be pre-proved.

The proposed verification would prove safety and liveness of the protocol under certain assumptions (see section 5.5). We do not claim that this verification covers all desirable properties of the protocol: further properties can be verified separately.

The remainder of this paper describes the class of sliding window protocols of which the implementation in Figure 1 is a member. We shall first use this example to identify important features of the protocol.

A SENDER reads input from its environment, using a `GET` command. This data is transmitted, with a label, `s`, over the channel `dataS`, `dataR` to a RECEIVER. The RECEIVER outputs data to its environment using the `PUT` command and also sends acknowledgement messages to the SENDER over the channel `ackR`, `ackS`. The RECEIVER's acknowledgement messages carry the label the RECEIVER is waiting to output. From this label the SENDER deduces whether it needs to retransmit its latest message (if the RECEIVER is still waiting for that message), or it can transmit a new message. The command, `SEND`, transmits a message containing a label and some data. The `RECV` command accepts an incoming message p, if there is one available. In this case, the function `OK p` is true and the packet's label is tested, otherwise `OK p` is false. The function `label` returns the first field of p when `OK p` and `mess` returns the data field. The labelled arrows in the diagram represent the direction of data flow. The behaviour of each box in the diagram is given below it, either in program code or in English.

The signals which characterize the protocol implementation of Figure 1 are

- the ports used by the SENDER and RECEIVER programs for communication with their calling environment : `input` and `ouput` of type `time` → `*data`

- the ports used by the SENDER and RECEIVER for communication with the network: `dataS, dataR, ackR, ackS` of type `time` $\rightarrow$ `(seq × data) + non_packet`
- the local variables of each program : `s, r` of type `time` $\rightarrow$ `num`, `d:time` $\rightarrow$ `*data`, `dummy:*data` and `ps, pr` of type `time` $\rightarrow$ `(seq × *data) + non_packet`

We can now describe a functional specification for the class of protocols of which this implementation is a member.

4 SPEC: a minimal functional specification

A sliding window protocol transfers a stream of data from one machine in a computer network to another. In section 2.2 such a protocol was specified in terms of input and output signals. Suppose instead that we abstract from the time messages are input and model the input stream as a (static) list of data and the output stream as a signal from time to the list of data output by that time. Suppose the input to a protocol was a stream of numbers `[0,1,2,3,4]`. Then the output list at time 0 would be `[]` and, as the protocol progressed, there would be times at which the output list was `[0]`, `[0,1]`, `[0,1,2]`, `[0,1,2,3]` and finally `[0,1,2,3,4]`. In the last case the protocol has achieved its purpose. A specification of this idea is

```
SPEC (source:*data list) (sink:time → *data list) =
    ∃ t:time.  sink t = source
```

Order is preserved since `source` and `sink` are lists and they will only be equivalent if `sink` has preserved the order of `source`. Using a list to specify input and output preserves information about the context of particular data elements so we no longer need to assume that each value of the source is unique. However, there are many incorrect protocols which satisfy this specification. We have said nothing about the physical structure of the protocol or that communication must be over an unreliable channel. Such properties are specified in the next level specification, GEN.

5 GEN: a generic class specification

5.1 PHYSICAL AND LOGICAL STRUCTURE

The structure of the protocol model, GEN, reflects the physical structure of the protocol: a sender on one computer, a receiver on another and a channel between them. Within this physical structure there is a logical structure. The SENDER and RECEIVER programs each have an initialization part, a message transmission part and a message reception part. The latter are executed for each traversal of each program's WHILE loop while the initialization part is executed once before the loop is entered. The channel between the SENDER and RECEIVER consists of two logical channels: one carrying data messages from the sender to the receiver and one carrying acknowledgements in the opposite direction. Figure 2 shows how this logical model fits into the physical model of Figure 1. As before, the arrows represent the direction of data flow.

In the HOL representation of this figure, the values shared between these entities are indicated by the shared parameters of the model.

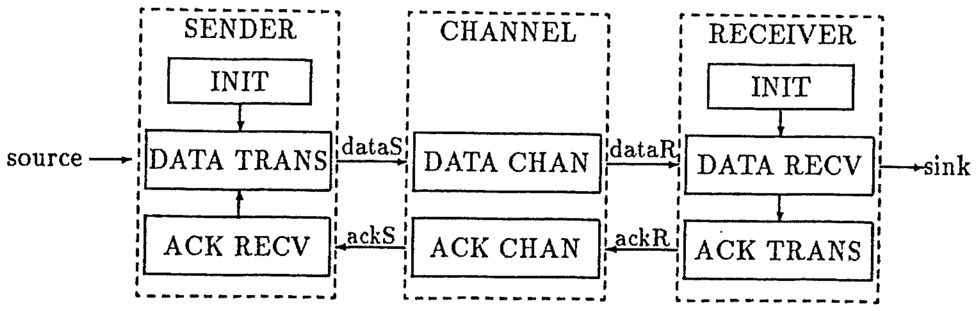


Figure 2: A Logical Structure for Sliding Window Protocol Models

```
∀ source sink.  SW_GEN source sink ≡
(∃ dataS dataR ackS ackR.
INIT sink ∧
DATA_TRANS source dataS ∧
CHANNEL dataS dataR ∧
DATA_RECV dataR sink ∧
ACK_TRANS ackR ∧
CHANNEL ackR ackS ∧
ACK_RECV ackS ∧
```

Note that the signals for channels are hidden inside the specification. For a full specification of GEN see Appendix B.

5.2 NON-DETERMINISM and CONTROL

The times at which messages are transmitted in an implementation are based on detailed timing of various events: for example whether a packet has been received and read from an input buffer, whether a new packet arrival has overwritten the first, whether previous operations have meant some events were ignored etc. In an abstract model these details should be left unspecifed. This is modelled here by making the choices to transmit or not to transmit and to accept a message or not to accept it a partially specified choice.

In situations where the designer has no control over the environment a choice may appear to be entirely non-deterministic. For example, the choice of whether a channel loses or delivers a packet is, for the protocol designer, completely non-deterministic and will be modelled by:

```
(Out t = set_non_packet) ∨ (Out t = In (t-(d t)))
```

On the other hand, in a program the designer can control which actions are taken, but may wish to only partially specify the choice to make her specification more general: that is, apply to a wider range of implementations. A particular implementation is said to be a refinement of such a specification.

The function PSC has been defined to represent non-determinism in specifications which will be refined. In the following definition, PSC performs either action d=t1 or d=t2 where t1 and t2 have different values. The conjuncts in the definition state that either d=t1 or d=t2 will occur, that t1 and t2 must be different,and that if action d=t1 occurs then the choice value a must be true. The type variables, * show that d,t1 and t2 can be of any type, but that they must have the same type.

```
PSC (a:bool) (d:*) (t1:*) (t2:*) =
    (d=t1 ∨ d=t2) ∧
    ¬ (t1 = t2) ∧
    (d=t1) ⟹ a
```

Partially specified choice is useful in two situations. First, in a programming language implementation all choices will be deterministic (e.g. an IF THEN ELSE command). This will be modelled by the fully specified choice a ⇒ d=t1|d=t2 ∧ ¬(t1=t2) which is a special case of PSC a d t1 t2. Second, different algorithms constrain the choice in different ways. For example, in one algorithm a new packet may be transmitted if a timeout occurs or whenever new input becomes available whilst in another algorithm packets are transmitted at any time. In both cases, any choice made with more information, PSC (a ∧ b) d t1 t2, is a special case of the original choice PSC a d t1 t2.

To see how partially specified choice is used in the generic specification, GEN, consider the transmission of data messages [4]. In the most general case, data messages may be transmitted at any time there is data available. A sliding window protocol limits the number of messages outstanding at any time by using a *window*. A window can be represented by a constant, say SW, and a rule that only the first SW data messages of those remaining to be transmitted at time t (represented by rem:time → data list) are available for transmission at that time. The predicate DATA_TRANS defines the data transmission behaviour of the class of sliding window protocols. The operator ⊕ represents addition modulo a protocol constant M. The function TLI n l returns the tail of the list l starting from its n-th element. Thus, ¬NULL(TLI (i t) (rem t)) checks that the element of rem t chosen for transmission is well defined.

```
DATA_TRANS ≡
   ∀ t:time.
      PSC (((i t)<SW ∧ ¬NULL(TLI (i t) (rem t)))
          (dataS t)
          (new_packet( s t ⊕ i t, HDI (i t) (rem t) )
          (set_non_packet)
```

The fragment above describes part of a *class* of algorithms because many behavioural details are left undefined. The constant SW and the transmission strategy function, i, are only partially specified. The choice between action and delaying action is non-deterministic so that the original specification still holds when the minimal transmission strategies defined in GEN are strengthened in ALG and IMPL specifications. For example, the condition for transmission in an algorithm could include the constraint that a timeout has occured or that new data for transmission has arrived. In an implementation, the transmission condition will include a condition that the time is one at which program execution has reached the command which causes a packet to be transmitted.

Another example of the use of PSC can be found in the specification of the receiver's data reception. The RECEIVER accepts packets transmitted by the SENDER from the

[4] In Appendix B, the operator NDC is used instead of PSC. NDC a c1 c2 ≡ (c1 ∨ c2) ∧ ¬ (c1 ∧ c2) ∧ (c1 ⟹ a). We could replace all occurrences of NDC with PSC since SPC a d t1 t2 ⟹ NDC a d=t1 d=t2

channel. If a message is received whose sequence number is equal to the next number to be output to the sink then the data part of the packet is output and the state signal r:time → sequence is updated for the next output.

```
DATA_RECV ≡
   ∀ t:time
      (PSC (good_packet(ackS t) ∧ (label(ackS t) = (r t)) )
         (r(t+1))
         ((r t) ⊕ 1)
         (r t) ) ∧
      (PSC (good_packet(ackS t) ∧ (label(ackS t) = (r t)) )
         (sink(t+1))
         (APPEND (sink t) (message(ackS t)))
         (sink t) )
```

5.3 REAL-TIME DELAY

Two approaches to the problem of modelling time and progress can be found in the protocol specification literature. The first method is to model protocols as state machines which use their inputs at any time to calculate their next state. These are behavioural state specifications. The behavioural specifications for DATA_TRANS and DATA_RECV given in the last section show how state specifications can be written in HOL.

A second method gives a functional specification, using real time intervals, of the time constraints which govern protocol behaviour. This method is used where the granularity of state model time is too coarse. That is, when a model which does not explicitly specify the time of its actions does not model the type of errors which are known to occur in practice. For example, real time intervals should be used to specify channel delay and timeouts.

Functional timing specifications state relationships between signals at different moments of real time. The output from a channel at time t, if a message is available, is a copy of the channel's input some time before t. Part of the channel specification, which was given in English in Figure 1, is Out t = In (t - (d t)). The specification is functional because it does not show *how* the channel stores and delivers messages but only the result of its doing so. The minimal constraints for the delay signal d are

```
∀ t.  0 < d t ∧ d t <= maxdelay
```

The full channel specification given below uses these constraints to describe a channel in which messages may be delivered or lost with variable but bounded delay and messages may also be duplicated or reordered by the channel. In section 5.4 it is shown how better behaved channels can be specified as special cases of the full channel.

```
CHANNEL In Out d maxdelay =
        (Out t = In t-(d t) ∨ Out t = set_non_packet) ∧
        (0 < d t) ∧ (d t ≤ maxdelay)
```

Figure 3 gives an example of the behaviour of the channel specified above. Messages are output by the channel at times 3, 4 and 6. The messages output at times 3 and 6 are duplicates because they were both input at time 2. The messages output at times 3 and 4 have been reordered since they were input at times 2 and 1 respectively.

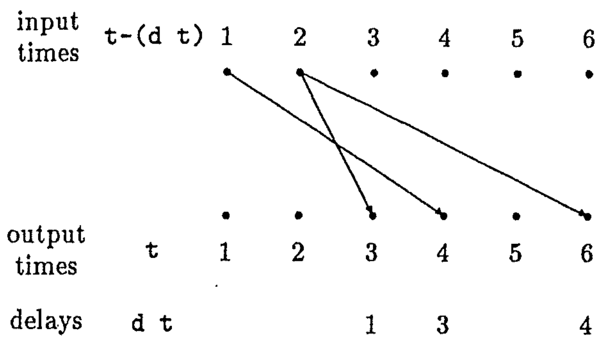


Figure 3: Sample behaviour of the CHANNEL specification

5.4 PARTIAL SPECIFICATION

The signal `i:time` → `sequence` in `DATA_TRANS` determines the next data message to be transmitted from the current window. In the specification, GEN, minimal constraints are specified for `i` : that it should be within the current window `((i t)<SW)` and should refer to data which still remains to be sent ¬`NULL(TLI (i t) (rem t))`. That is, `i` is only partially specified. In an algorithm a data transmission strategy is used to define when data packets are to be sent and which data message should be included in the packet each time. An algorithm's data transmission strategy will be given by a more detailed definition of the choice signal `i`.

Types can also be partially specified. For example, a packet's structure has some fields which will be used by all levels of specification such as a field for data and a field for labels. However, an implementation's packet structure may contain extra fields. This can be modelled by using arbitrary types (e.g. type variables in a polymorphic logic) for packet structures which are not instantiated until an implementation level description. The partial specification of structured types has not been used in the example described in this paper.

5.5 LIVENESS

In any computer system which allows non-deterministic choice between actions we have to decide how to model assumptions that the system makes progress. Traditionally, liveness is expressed in terms of

1. an assumption that, if a non-deterministic choice is offered infinitely often, then each branch of the conditional will eventually be chosen (fairness) and
2. a proof or assumption that events will be offered infinitely often.

However, in most implementations a program will not offer events or wait for them indefinitely, but only for some predetermined maximum time. If progress has not been made during this time interval then the program is aborted with a suitable signal to its caller. In the sliding window protocol specification, `GEN`, if the sender's state has not changed over `maxP` time units, then the sender assumes that the receiver or the channel has crashed and aborts the protocol. We define `abort:time` → `bool` to be true only when the protocol should abort and false otherwise.

```
ABORT ≡
∀ t:time.  abort t =
     ( (maxP ≤ t) ∧ (¬NULL(rem t)) ∧ ( rem t = rem (t-maxP) )
```

A protocol is said to be live if (∀ t. ¬ abort t). A protocol which is live in this sense will also satisfy traditional liveness criteria. The definition is general because the waiting interval maxP is not given a value. When details of an implementation environment have been specified then it can be proved that a given interpretation of the bound maxP is reasonable. That is, that the protocol does not abort in trivial cases.

5.6 VERIFICATION

Two lemmas about the behaviour defined by GEN are used to prove that the general class behavioural model, GEN, satisfies its specification, SPEC [5]. The first is a safety property relating the values of the sender's list of data remaining to be sent, rem, and the output list, sink. The operator ⊖ represents integer subtraction modulo a protocol constant M.

```
SAFETY ≡
    GEN ⟹
    ∀ t:time.
       APPEND (sink t) (TLI ((r t) ⊖ (s t)) (rem t)) = source
```

The second lemma states that as long as the protocol is not aborted then the list rem will be empty by time maxP*LENGTH(rem 0) at the latest [6]. The specification, GEN, contains the liveness assumption (∀ t:time. ¬ (aborted t)) where abort is specified by the predicate ABORT given above.

```
LIVENESS ≡
    GEN ⟹ (rem (maxP*LENGTH(rem 0)) = [])
```

The theorems SAFETY and LIVENESS are used to prove

```
GEN ⟹ SPEC
```

6 ALG: protocol algorithm specifications

6.1 TRANSMISSION STRATEGIES

A simple data transmission strategy can be used when window sizes are tailored to the delays of a channel: start transmitting data from the base of the window, transmit each message in turn until reaching the top of the window and then return to the bottom again. If no data or acknowledgement messages have been lost then the base of the window will have moved up by this time and new data will be transmitted. If acknowledgements have been lost then returning to the base of the window will initiate a string of retransmissions.

[5]The following statements of the SAFETY and LIVENESS assumptions slightly misuse our notation in that the parameters of GEN and SPEC are not listed. See Appendix B for a formal statement of these theorems

[6]The value of the list rem at time 0 is equal to the original input list, source.

```
BottomToTop ≡
   (∃ n.  ∀ t.
      (n 0 = 0) ∧
      (n(t + 1) =
        (good_packet(dataS t)
          ⇒ (n t ⊖ s t) < SW
              ⇒ (n t ⊕ 1)
              | (s t)
          | n t)) ∧
      (i t = n t ⊖ s t) )
```

A more common transmission strategy uses timeouts to signal when retransmissions should occur. A timeout occurs if the sender has made no progress for a certain period, TIMEOUT, after transmitting a packet because an acknowledgement for that packet has not arrived.

```
Timeout TIMEOUT dataS s rem t ≡
   ( (rem t) = rem(t-TIMEOUT)) ∧ (TIMEOUT ≤ t) ∧
   ( good_packet(dataS (t-TIMEOUT))) ∧
   ( s t = label(dataS (t-TIMEOUT)) ) ∧
```

In this transmission strategy all the data in a given window is transmitted once and then, if a timeout occurs for a packet which has not yet been acknowledged, that packet is retransmitted. If there is no new data to transmit and no timeout occurs, then (i t) is set to SW so that nothing will be transmitted.

```
TransWithTimeouts ≡
   ∀ t.  (Timeout t)
              ⇒ top(t+1)=top t ∧ i(t+1)=0
              | good_packet(dataS t) ∧ (top t ⊖ s t)<SW
                  ⇒ top(t+1)= ( (top t) ⊕ 1) ∧
                      i(t+1) = (top t) ⊖ (s t)
                  | top(t+1)=top t ∧ i t = SW
```

6.2 BUFFERS as CHANNELS: FUNCTIONAL MODEL

In some sliding window protocol algorithms, when the receiver receives a packet which it cannot immediately output to the sink, the receiver saves that packet in the hope that it can be output once some earlier packets have arrived. This strategy increases the efficiency of the protocol by reducing the number of packets which the sender may have to retransmit.

A buffer may be used by a sender to store data which is ready for transmission or which has been transmitted but not yet acknowledged.

In both these cases a buffer is simply a delay device which stores an input until such time as its output condition is satisfied. The abstract specification of such a buffer, used in ALG, is defined for all output times t which satisfy the output condition and for which at a previous time (ft' t) an input condition was satisfied. In order to bound the space required for buffers, input is only accepted if it is within the receiver's current window. In the specification below, the window size, RW:sequence, is a constant, but variable window sizes could be implemented by using rw:time → sequence to represent the window's size at different times. The variable window size (rw t) would always be bounded above by the constant RW. The function DataOut:time → data represents output values which have been input via the channel dataR:time → packet.

```
AbsBuffer dataR r RW DataOut ft' maxW t ≡
     (ft' t) ≤ t ∧ t-(ft' t) ≤ maxW ∧
     IN_WINDOW (dataR (ft' t)) (r (ft' t)) RW ∧
     r t = label(dataR(ft' t)) ∧
     DataOut t = message(dataR(ft' t))
```

6.3 DELAYS: BEHAVIOURAL MODELS

The definitions for channels, buffers, timeouts and liveness show how real time constraints in protocols can be specified using real time intervals. However, a protocol implementation does not have access to the values of its variables from any previous times. It must instead store and retrieve values at suitable times to achieve the same result. For example, a program can only compare the current value of some state variable and its value 10 seconds previously if, 10 seconds previously, the program had saved that variable's value and later retrieves it. The theory SW.ALG described in Appendix B contains an example which illustrates the use of a behavioural specification of a counter to implement the delay specified functionally in GEN by ABORT.

6.4 NEW CHANNELS

Some sliding window protocol algorithms are designed for a particular network environment. For example, many protocols only work correctly when their communication channel does not reorder packets.

A channel which does not reorder or duplicate its inputs is a special case of the general channel:

```
WELL_BEHAVED_CHANNEL In Out d maxd ≡
     CHANNEL In Out d maxd ∧
     (∀ t1 t2:time.
       good_packet(Out t1) ∧ good_packet(Out t2) ∧ t1 < t2
       ⟹ (t1 - (d t1)) < (t2 - (d t2)))
```

If two channels are connected in sequence, for example as adjoining links of a subnet, then the resulting process is also a channel. The new channel's maximum delay is the sum of the maximum delays of the original channels and the new delay function is a function of the originals.

```
COMPOSE_CHANNEL_THM ≡
     CHANNEL a b d1 maxd1 ∧ CHANNEL b c d2 maxd2 ⟹
     CHANNEL a c (λ t.  (d2 t) + (d1(t - (d2 t))))(maxd2 + maxd1)
```

Channels which may "crash" and refuse to deliver messages for a certain period can be modelled using auxilliary signals. For example, in the following specification the signal has_crashed is true at times when the channel is not available. The effect of a crash on the behaviour of the channel is given by:

```
CHANNEL_WITH_CRASHES In Out d maxd has_crashed ≡
     CHANNEL In Out d maxd ∧
     (∀ t.  has_crashed t ⟹ ¬ good_packet(Out t))
```

Many other behaviours can be modelled in this way. For example, a signal `garbled:time → bool` could be defined to distinguish between non_packet channel outputs which are the result of a packet damaged during transmission (say when `garbled t = T`) and those which represent no output at a given time (when `garbled t = F`).

7 RELATED WORK

The methodology for modelling protocols presented in this paper is based on methods designed for modelling hardware using higher order logic [6, 11, 9, 3]. In order to model protocols we extended the notion of values shared between components from wires to structures such as packets. Although abstraction techniques and the multi-level specification of hardware are covered in [11, 9] the treatment of partially specified choice and delay for buffers and channels in our work is new.

Gordon's methodology is used in [10] to combine specifications written in DRTL, an extension of Jahanian and Mok's RTL [7]. A requirements language and a design language are defined; both describe only the timing behaviour of a real time system. Communication is modelled implicitly by notification delays and not by a channel process as is done in our model.

Real time logic, RTL, [7, 8] is a language designed for the specification of real time systems. Systems are first specified using actions and events which can be combined in sequence or in parallel subject to real time constraints. Such specifications are then translated into RTL formulas in which actions are described by their start and completion events and the time of the i-th occurence of an event is expressed by an occurrence function. Two decision procedures have been proposed for verifying RTL specifications. The first is based on quantifier free Presberger arithmetic [7]. A more efficient decision procedure based on graph theory is proposed in [8].

In our methodology, actions and events are both represented by signals which are similar in expressiveness to RTL event occurence functions. Our verification proofs are performed at a higher level than the decision procedures for RTL. Thus, although our proofs are machine checked, and so rigorous, they do require considerable human input to direct the proof. The ability to describe real time constraints in a range of both functional and behavioural specifications is an advantage of our method which is not available in RTL.

Most protocol specification and verification methods in the literature describe specific algorithms and do not cover implementation models or more general abstract specifications. Few models can express real time constraints. A model which addresses both these problems is described in Shankar and Lam [15, 14] where an event-action model is used to described real time protocols.

In [14] a class of sliding window protocols is verified under two different assumptions about the behaviour of their communications channel. The model for protocol behaviour lies somewhere between those called GEN and ALG in this paper. The proof method involves deriving invariants for each action in the specification; projection can be used for modularity. Real time is modelled by an external clock which may be read by processes.

Rules are proposed to determine whether a real time constraint is implementable: for example, whether one process controls all the events necessary to realise the constraint. Our model is more concrete than this because we prove that a specification is implementable by showing that a particular implementation satisfies that specification.

Two logical models which can be used for real time systems are interval temporal logic [13, 5] and interval logic [12]. Whereas temporal logic formulae hold on an infinite interval, 'the future', interval temporal logics enable a finite interval to be specified on which a formula holds. Thus constraints of the type 'event x must occur within 3 seconds of event y' can be specified. The expressiveness of ITL is similar to that of the higher order logic model used in this paper. An advantage of interval temporal logic is that deterministic specifications are executable in the programming language Tempura [13]. Thus, a class of high level specifications in interval temporal logic could be checked by executing them.

8 CONCLUSIONS

This paper has shown that higher order logic can be used for the specification of protocols at a number of different levels of abstraction. We have successfully represented a range of real time behaviours such as unreliable channels with bounded but variable delay, timeouts and buffers.

Most specification languages are more suited to either requirements or behavioural specifications. The ease of using whichever style seems most appropriate is an advantage of our approach.

The use of a generic specification for the class of sliding window protocols has proved a useful way to structure the verification of real time algorithms and implementations since

1. GEN is a simpler model than that for any specific protocol algorithm and its verification proof, including real time properties, is not too hard, and

2. the extra predicates required for an ALG specification can be defined and verified independently of one another: that is, particular algorithm strategies such as timeouts for retransmission, negative acknowledgements, the use of buffers and different channel behaviours can be kept as a library of definitions with proofs of their properties.

The model GEN presented in this paper is not a canonical generalization of all sliding window protocols. For example, the transmission strategy for protocols such as TCP, where a list rather than a single element of data is transmitted, is not covered by GEN. Our model is a special case of one which would cover TCP and so could probably be rewritten to include this more general behaviour. Also, the bound we place on the range of sequence numbers is necessary and sufficient for channels which may duplicate and reorder packets, but is larger than required for channels which do not reorder or duplicate packets. However, the search for a completely generalized canonical model for sliding window protocols, if one exists, is beyond the scope of this research.

Further experience is needed in relating GEN and ALG specifications to implementation models. We have started to describe the semantics of a real time programming language in higher order logic using the model of GEN and ALG specifications. Implementation models could then be verified following the methodology suggested in section 3.2. Properties about the performance of protocols and the effect of particular transmission strategies, window sizes etc. in a given environment could also be verified from an IMPL model.

Acknowledgements

I would like to thank Mike Gordon for helpful comments on drafts of this paper and also Jeff Joyce and participants in the UBC HOL Course held in Vancouver in June 1990 for stimulating discussions. 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] Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[2] Mike Gordon. A proof generating system for higher-order logic. Technical Report 103, University of Cambridge Computer Laboratory, January 1987.

[3] M.J.C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Conference on VLSI*, pages 153–177. North Holland, 1986.

[4] 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.

[5] Roger Hale. Using temporal logic for prototyping: the design of a lift controller. In *Temporal Logic in Specification 1987*, pages 375–408, 1987. Lecture Notes in Computer Science 398.

[6] F.K. Hanna and N. Daeche. Specification and verification of digital systems using higher-order predicate logic. *IEE Proceedings E*, 133 Part E(5):242–254, September 1986.

[7] Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[8] Farnam Jahanian and Aloysius Ka-Lau Mok. A graph theoretic approach for timing analysis and its implementation. *ACM Transactions on Computers*, C-36(8):961–975, August 1987.

[9] Jeffrey J. Joyce. *Multi-Level Verification of Microprocessor-Based Systems*. PhD thesis, Computer Laboratory, University of Cambridge, December 1989.

[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] P.M. Melliar-Smith. Extending interval logic to real time systems. In *Temporal Logic in Specification 1987*, pages 224–242, 1987. Lecture Notes in Computer Science 398.

[13] Ben Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986.

[14] A. Udaya Shankar. Verified data transfer protocols with variable flow control. *ACM Transactions on Computer Systems*, 7(3):281–316, August 1989.

[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.

A Syntax for a fragment of the HOL logic

MACHINE VERSION	LOGICAL SYMBOL	MEANING
`T` and `F`	T and F	Truth and Falsity
`~p`	$\neg p$	not p
`p /\ q`	$p \wedge q$	p and q
`p \/ q`	$p \vee q$	p or q
`p ==> q`	$p \Longrightarrow q$	p implies q
`P x`	$P(x)$ or Px	property P of x
`!x. P x`	$\forall x.P\ x$	for all x, property P x is true
`?x. P x`	$\exists x.P\ x$	there exists at least one x such that P x is true
`x=y`	$x = y$	polymorphic equality : x and y have the same type and same value
`a=>b|c`	$a \Rightarrow b \mid c$	if a then b else c
`< <= + - *`	$< \leq + - *$	arithmetic infix operators on natural numbers
`:bool`	:bool	boolean type : T and F
`:num`	:num	natural number type : 0,1,2,3,...
`:one`	:one	the type with only one member: one
`:*ty`	:*ty	arbitrary type: a type variable
`:ty list`	:ty list	a list with elements of type ty
`hd, tl, NIL, []`	hd, tl, NIL, []	list operators: head, tail, and two notations for the empty list
`NULL` l	$NULL\ l$	true if l is the empty list and false otherwise
`APPEND l1 l2`	$APPEND\ l_1\ l_2$	a list which is the concatenation of two lists, l_1 and l_2, of the same type
`LENGTH l1`	$LENGTH\ l_1$	the number of elements in the list l_1
`:ty1 -> ty2`	$ty_1 \rightarrow ty_2$	type of any function with domain ty_1 and range ty_2
`:ty1 # ty2`	$ty_1 \times ty_2$	cartesian product type
	FST, SND	FST$(x,y) = x$ and SND$(x,y) = y$
`:ty1 + ty2`	: $ty_1 + ty_2$	disjoint union type. Use this type with the following functions :
	INL, INR, OUTL, OUTR	injections and projections of the sum
	ISL, ISR	a test for left or right summand
`|- thm`	$\vdash$ thm	thm is a theorem of higher order logic (that is a statement which has been formally proved)

B HOL theories for SPEC, GEN and ALG

B.1 Types: a summary

```
time = num
sequence = num
non_packet = one packet = (sequence # *data) + one
channel = time → packet
```

B.2 Signals and Constants: a summary

`source : *data list`	original input to the protocol
`sink : time → *data list`	output of the protocol over time
`rem : time → *data list`	data remaining to be transmitted by the sender, including data sent but not yet acknowledged
`s : time → sequence`	the sender's window marker
`r : time → sequence`	the receiver's window marker: s and r determine which incoming packets should be accepted
`i : time → sequence`	determines the next data message to be transmitted by the sender from its current window
`DataOut : time → *data`	data values returned to the receiver from its input buffer
`SW : sequence`	sender's window size
`RW : sequence`	receiver's window size
`M : sequence`	Maximum range of sequence numbers which can be used
`dataS dataR : channel`	input and ouput, respectively, of the data channel
`ackR ackS : channel`	input and output, respectively, of the acknowledgement channel
`maxdd maxda : time`	maximum delivery delays of the data and acknowledgement channels respectively
`Sdif, Rdif: sequence`	Maximum range of sequence numbers which could be transmitted during the time interval maxdd or maxda respectively.

B.3 HOL Theories

```
******************** SW.GEN THEORY ********************

The Theory SW.GEN
Parents --  HOL     arith     myarith     hdi_tli     SEQMOD

Type Abbreviations --
  time ":num"     sequence ":num"     non_packet ":one"
  seqtime ":num -> num"     delaytime ":num -> num"     time ":num"
  sequence ":num"     non_packet ":one"     seqtime ":num -> num"
  delaytime ":num -> num"

******************** DEFINITIONS ********************

  set_non_packet  |- set_non_packet = INR one
  good_packet  |- !p. good_packet p = ISL p
  new_packet  |- !ss dd. new_packet ss dd = INL(ss,dd)
  label  |- !p. label p = FST(OUTL p)
  message  |- !p. message p = SND(OUTL p)
  NDC
    |- !c a1 a2. NDC c a1 a2 = (a1 \/ a2) /\ ~(a1 /\ a2) /\ (a1 ==> c)
  INIT
    |- !source rem s sink r.
        INIT source rem s sink r =
        (rem 0 = source) /\ (s 0 = 0) /\ (sink 0 = []) /\ (r 0 = 0)
  CHANNEL
    |- !In Out d maxdelay.
        CHANNEL In Out d maxdelay =
        (!t.
          ((Out t = In(t - (d t))) \/ (Out t = set_non_packet)) /\
          (d t) > 0 /\
          (d t) <= maxdelay)
  DATA_TRANS
    |- !dataS rem s i M dtrA.
        DATA_TRANS dataS rem s i M dtrA =
        (!t.
          NDC
          (dtrA t)
          (dataS t = new_packet(plusm(s t,i t,M))(HDI(i t)(rem t)))
          (dataS t = set_non_packet))
  DATA_RECV
    |- !dataR sink DataOut r M drvA.
        DATA_RECV dataR sink DataOut r M drvA =
        (!t.
          NDC
          (drvA t)
          ((sink(t + 1) = APPEND(sink t)[DataOut t]) /\
          (r(t + 1) = plusm(r t,1,M)))
         ((sink(t + 1) = sink t) /\ (r(t + 1) = r t)))
ACK_TRANS
  |- !ackR r M ackty atrA.
       ACK_TRANS ackR r M ackty atrA =
       (!t.
         NDC
         (atrA t)
         (ackR t = new_packet(subm(r t,1,M))(ackty t))
         (ackR t = set_non_packet))
ACK_RECV
  |- !ackS rem s M arvA.
       ACK_RECV ackS rem s M arvA =
       (!t.
         NDC
         (arvA t)
         ((s(t + 1) = plusm(label(ackS t),1,M)) /\
          (rem(t + 1) = TLI(subm(s(t + 1),s t,M))(rem t)))
         ((s(t + 1) = s t) /\ (rem(t + 1) = rem t)))
ABORT
  |- !abort maxP rem.
       ABORT abort maxP rem =
       (!t.
         abort t = (rem t = rem(t - maxP)) /\ maxP <= t /\ ~NULL(rem t))
LIVE_ASSUMPTION  |- !abort. LIVE_ASSUMPTION abort = (!t. ~abort t)
dtrans_min
  |- !i SW rem t.
       dtrans_min i SW rem t = (i t) < SW /\ ~NULL(TLI(i t)(rem t))
IN_WINDOW
  |- !p b M ws.
       IN_WINDOW p b M ws = good_packet p /\ (subm(label p,b,M)) < ws
drecv_min
  |- !dataR r DataOut t.
       drecv_min dataR r DataOut t =
       good_packet(dataR t) /\
       (label(dataR t) = r t) /\
       (DataOut t = message(dataR t))
atrans_min  |- !t. atrans_min t = T
arecv_min
  |- !ackS s M SW t.
       arecv_min ackS s M SW t = IN_WINDOW(ackS t)(s t)M SW
Sdif_DEF
  |- !s M maxdd Sdif.
       Sdif_DEF s M maxdd Sdif =
       Sdif < M /\
       (!t n. n <= maxdd ==> (subm(s(t + n),s t,M)) <= Sdif)
Rdif_DEF
  |- !r M maxda Rdif.
```

```
        Rdif_DEF r M maxda Rdif =
        Rdif < M /\
        (!t n. n <= maxda ==> (subm(r(t + n),r t,M)) <= Rdif)
  M_ASSUM
    |- !M SW s r maxdd maxda Sdif Rdif.
        M_ASSUM M SW s r maxdd maxda Sdif Rdif =
        0 < M /\
        (SW + 1) < M /\
        Sdif_DEF s M maxdd Sdif /\
|       (Sdif + SW) < M /\
        Rdif_DEF r M maxda Rdif /\
        (Rdif + SW) < M /\
        (Rdif + 1) < M

******************** MAIN DEFINITIONS ********************

  SW_SPEC  |- !source sink. SW_SPEC source sink = (?t. sink t = source)
  SW_GEN
    |- !source sink.
        SW_GEN source sink =
        (?rem s i r SW M Sdif Rdif DataOut ackty maxP maxdd maxda abort
          dataS dataR ackS ackR dd da.
          INIT source rem s sink r /\
          DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
          CHANNEL dataS dataR dd maxdd /\
          DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) /\
          ACK_TRANS ackR r M ackty atrans_min /\
          CHANNEL ackR ackS da maxda /\
          ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
          ABORT abort maxP rem /\
          LIVE_ASSUMPTION abort /\
          M_ASSUM M SW s r maxdd maxda Sdif Rdif)


******************** THEOREMS ********************

  rem1_lemma
    |- M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
       ACK_RECV ackS rem s M(arecv_min ackS s M SW) ==>
       (!t.
         (rem(t + 1) = TLI(subm(s(t + 1),s t,M))(rem t)) /\
         (subm(s(t + 1),s t,M)) <= SW)
  remn_lemma
    |- M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
       ACK_RECV ackS rem s M(arecv_min ackS s M SW) ==>
       (!t n.
         n <= maxdd ==> (rem(t + n) = TLI(subm(s(t + n),s t,M))(rem t)))
  remdd_lemma
```

```
   |- M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
      ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
      CHANNEL dataS dataR dd maxdd ==>
      (!t. rem t = TLI(subm(s t,s(t - (dd t)),M))(rem(t - (dd t))))
 ChannelLemma
   |- !A B d maxd t.
       CHANNEL A B d maxd /\ good_packet(B t) ==>
       (B t = A(t - (d t))) /\ good_packet(A(t - (d t)))
 AckValueLemma
   |- !t.
       ACK_TRANS ackR r M ackty atrans_min /\
       CHANNEL ackR ackS da maxda /\
       arecv_min ackS s M SW t ==>
       (label(ackS t) = subm(r(t - (da t)),1,M))
 dataSvalues
   |- !t.
       good_packet(dataS t) /\
       DATA_TRANS dataS rem s i M(dtrans_min i SW rem) ==>
       (label(dataS t) = plusm(s t,i t,M)) /\
       (message(dataS t) = HDI(i t)(rem t)) /\
       (i t) < SW /\
       ~NULL(TLI(i t)(rem t))
 lemmaRmin
   |- DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
      CHANNEL dataS dataR dd maxdd /\
      DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) ==>
      (!t.
        (r(t + 1) = plusm(r t,1,M)) /\ drecv_min dataR r DataOut t \/
        (r(t + 1) = r t))
 R_CASES_THM
   |- DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
      CHANNEL dataS dataR dd maxdd /\
      DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) ==>
      (!t.
        (r(t + 1) = plusm(r t,1,M)) /\
        drecv_min dataR r DataOut t /\
        (sink(t + 1) = APPEND(sink t)[DataOut t]) \/
        (r(t + 1) = r t) /\ (sink(t + 1) = sink t))
 lemmaR
   |- DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
      CHANNEL dataS dataR dd maxdd /\
      DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) ==>
      (!t.
        (r(t + 1) = plusm(r t,1,M)) /\
        (r t = plusm(s(t - (dd t)),i(t - (dd t)),M)) /\
        (i(t - (dd t))) < SW \/
        (r(t + 1) = r t))
lemmaS
```

```
   |- ACK_TRANS ackR r M ackty atrans_min /\
      CHANNEL ackR ackS da maxda /\
      ACK_RECV ackS rem s M(arecv_min ackS s M SW) ==>
      (!t.
        (s(t + 1) = plusm(label(ackS t),1,M)) /\
        (label(ackS t) = subm(r(t - (da t)),1,M)) /\
        (subm(label(ackS t),s t,M)) < SW \/
        (s(t + 1) = s t))
sdifd_lemma
   |- Sdif_DEF s M maxdd Sdif /\ CHANNEL dataS dataR dd maxdd ==>
      (!t. (subm(s t,s(t - (dd t)),M)) <= Sdif)
rdifd_lemma
   |- Rdif_DEF r M maxda Rdif /\ CHANNEL ackR ackS da maxda ==>
      (!t. (subm(r t,r(t - (da t)),M)) <= Rdif)
rda_lemma
   |- 0 < M /\
      ACK_TRANS ackR r M ackty atrans_min /\
      CHANNEL ackR ackS da maxda /\
      ACK_RECV ackS rem s M(arecv_min ackS s M SW) ==>
      (subm(r(t - (da t)),s t,M) = subm(s(t + 1),s t,M)) /\
      (subm(r t,s(t + 1),M) = subm(r t,r(t - (da t)),M)) \/
      (s(t + 1) = s t)
sincr_lemma
   |- M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
      ACK_TRANS ackR r M ackty atrans_min /\
      CHANNEL ackR ackS da maxda /\
      ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
      DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
      CHANNEL dataS dataR dd maxdd /\
      DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) ==>
      (!t.
        (subm(r t,s(t + 1),M)) + (subm(s(t + 1),s t,M)) =
        subm(r t,s t,M))
RSdif_lemma
   |- INIT source rem s sink r /\
      M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
      ACK_TRANS ackR r M ackty atrans_min /\
      CHANNEL ackR ackS da maxda /\
      ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
      DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
      CHANNEL dataS dataR dd maxdd /\
      DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) ==>
      (!t. (subm(r t,s t,M)) <= SW)
rs1_lemma
   |- INIT source rem s sink r /\
      M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
      ACK_TRANS ackR r M ackty atrans_min /\
      CHANNEL ackR ackS da maxda /\
      ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
      DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
      CHANNEL dataS dataR dd maxdd /\
      DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) ==>
      (!t. subm(plusm(r t,1,M),s t,M) = (subm(r t,s t,M)) + 1)
 Rsincr_lemma
   |- M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
      INIT source rem s sink r /\
      ACK_TRANS ackR r M ackty atrans_min /\
      CHANNEL ackR ackS da maxda /\
      ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
      DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
      CHANNEL dataS dataR dd maxdd /\
      DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) ==>
      ((subm(r(t + 1),s(t + 1),M)) + (subm(s(t + 1),s t,M)) =
       subm(r(t + 1),s t,M))
 S_INC_THM
   |- M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
      INIT source rem s sink r /\
      ACK_TRANS ackR r M ackty atrans_min /\
      CHANNEL ackR ackS da maxda /\
      ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
      DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
      CHANNEL dataS dataR dd maxdd /\
      DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) ==>
      (!t.
        TLI(subm(r(t + 1),s(t + 1),M))(rem(t + 1)) =
        TLI(subm(r(t + 1),s t,M))(rem t))
 rsd_lemma
   |- INIT source rem s sink r /\
      M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
      ACK_TRANS ackR r M ackty atrans_min /\
      CHANNEL ackR ackS da maxda /\
      ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
      DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
      CHANNEL dataS dataR dd maxdd /\
      DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) ==>
      ((subm(r t,s t,M)) + (subm(s t,s(t - (dd t)),M)) =
       subm(r t,s(t - (dd t)),M))
 change_i
   |- (plusm(s(t - (dd t)),i(t - (dd t)),M) = r t) /\
      (i(t - (dd t))) < SW /\
      M_ASSUM M SW s r maxdd maxda Sdif Rdif ==>
      (subm(r t,s(t - (dd t)),M) = i(t - (dd t)))
 R_INC_THM
   |- !t.
       M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
       INIT source rem s sink r /\
        ACK_TRANS ackR r M ackty atrans_min /\
        CHANNEL ackR ackS da maxda /\
        ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
        DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
        CHANNEL dataS dataR dd maxdd /\
        DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) /\
        drecv_min dataR r DataOut t ==>
        (DataOut t = HDI(subm(r t,s t,M))(rem t)) /\
        ~NULL(TLI(subm(r t,s t,M))(rem t))

******************** SAFETY THEOREM ********************

  SAFETY_THM
    |- INIT source rem s sink r /\
       M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
       ACK_TRANS ackR r M ackty atrans_min /\
       CHANNEL ackR ackS da maxda /\
       ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
       DATA_TRANS dataS rem s i M(dtrans_min i SW rem) /\
       CHANNEL dataS dataR dd maxdd /\
       DATA_RECV dataR sink DataOut r M(drecv_min dataR r DataOut) ==>
       (!t. APPEND(sink t)(TLI(subm(r t,s t,M))(rem t)) = source)

*********************************************************

  NOT_ABORT
    |- !abort maxP rem.
        ABORT abort maxP rem /\ LIVE_ASSUMPTION abort ==>
        (!t. NULL(rem t) \/ t < maxP \/ ~(rem t = rem(t - maxP)))
  remd_LENGTH_lemma
    |- M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
       ACK_RECV ackS rem s M(arecv_min ackS s M SW) ==>
       (!n t. (LENGTH(rem(t + n))) <= (LENGTH(rem t)))
  remd_LENGTH_THM
    |- ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
       M_ASSUM M SW s r maxdd maxda Sdif Rdif ==>
       (!n t.
         ((~(rem(t + n) = rem t)) =>
          (LENGTH(rem(t + n))) < (LENGTH(rem t)) |
          (LENGTH(rem(t + n)) = LENGTH(rem t))))
  decreasing_rem_lemma
    |- !ackS s r rem maxP maxdd maxda SW M Sdif Rdif.
        ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
        M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
        ABORT abort maxP rem /\
        LIVE_ASSUMPTION abort ==>
        (!n. (LENGTH(rem(maxP * n))) <= ((LENGTH(rem 0)) - n))
```

```
******************** LIVENESS THEOREM ********************

  LIVENESS
    |- !ackS s r rem maxP maxdd maxda SW M Sdif Rdif.
        ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
        M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
        ABORT abort maxP rem /\
        LIVE_ASSUMPTION abort ==>
        (rem(maxP * (LENGTH(rem 0))) = [])

******************** CORRECTNESS THEOREM ********************

  TOTAL_CORRECTNESS_THM
    |- !source sink. SW_GEN source sink ==> SW_SPEC source sink

******************** END SW.GEN THEORY ********************
```

```
******************** SW.ALG THEORY *******************
The Theory SW.ALG
Parents --  HOL     SW.GEN

Definitions --
******************** TRANSMISSION STRATEGIES *************
  BottomToTop
    |- !i dataS s SW maxseq.
        BottomToTop i dataS s SW maxseq =
        (?n.
          !t.
           (n 0 = 0) /\
           (n(t + 1) =
            (good_packet(dataS t) =>
             ((subm(n t,s t,maxseq)) < SW => plusm(n t,1,maxseq) | s t) |
             n t)) /\
           (i t = subm(n t,s t,maxseq)))
  Timeout
    |- !TIMEOUT dataS s rem t.
        Timeout TIMEOUT dataS s rem t =
        (rem t = rem(t - TIMEOUT)) /\
        TIMEOUT <= t /\
        good_packet(dataS(t - TIMEOUT)) /\
        (label(dataS(t - TIMEOUT)) = s t)
  AckIfInput
    |- !ACK_TIMEOUT dataR ackR t.
        AckIfInput ACK_TIMEOUT dataR ackR t =
        good_packet(dataR(t - 1)) \/
        ACK_TIMEOUT <= t /\
        (!t'.
          (t - ACK_TIMEOUT) <= t' /\ t' < t ==> ~good_packet(ackR t'))

******************** RECEIVER BUFFER SPECS *******************

  CHANNEL_O
    |- !In Out d maxdelay.
        CHANNEL_O In Out d maxdelay =
        (!t.
          ((Out t = In(t - (d t))) \/ (Out t = set_non_packet)) /\
          0 <= (d t) /\
          (d t) <= maxdelay)
  BufCond
    |- !t t' maxW dataR r M RW.
        BufCond t t' maxW dataR r M RW =
        t' <= t /\
        (t - t') <= maxW /\
        IN_WINDOW(dataR t')(r t')M RW /\
        (!t''. t' < t'' /\ t'' < t ==> ~(r t'' = label(dataR t'))) /\
        (r t = label(dataR t'))
  AbsBuffer
    |- !dataR r M RW DataOut ft' maxW t.
        AbsBuffer dataR r M RW DataOut ft' maxW t =
        BufCond t(ft' t)maxW dataR r M RW /\
        (DataOut t = message(dataR(ft' t)))


******************** BUFFER IS CHANNEL THEOREMS  ******************

Theorems --
  COMPOSE_CHANNEL_O_THM
    |- !a b c d1 d2 maxd1 maxd2.
        CHANNEL a b d1 maxd1 /\ CHANNEL_O b c d2 maxd2 ==>
        CHANNEL a c(\t. (d2 t) + (d1(t - (d2 t))))(maxd2 + maxd1)
  BufCond_AS_CHANNEL
    |- !dataR r M RW maxW ft'.
        CHANNEL_O
        dataR
        (\t.
          (BufCond t(ft' t)maxW dataR r M RW =>
           dataR(ft' t) |
           set_non_packet))
        (\t. (BufCond t(ft' t)maxW dataR r M RW => t - (ft' t) | 0))
        maxW
  AbsBuf_AS_drecv_min
    |- !dataR r M RW DataOut ft' maxW t.
        AbsBuffer dataR r M RW DataOut ft' maxW t <=>
        drecv_min
        (\t.
          (BufCond t(ft' t)maxW dataR r M RW =>
           dataR(ft' t) |
           set_non_packet))
        r
        DataOut
        t
  AbsBuf_CHANNEL_THM
    |- !dataS dataR dA maxdA M RW DataOut ft' maxW.
        ?dataQ dd maxdd.
         CHANNEL dataS dataR dA maxdA /\
         DATA_RECV
         dataR
         sink
         DataOut
         r
         M
         (AbsBuffer dataR r M RW DataOut ft' maxW) ==>
         CHANNEL dataS dataQ dd maxdd /\
        DATA_RECV dataQ sink DataOut r M(drecv_min dataQ r DataOut)


******************** BUFFER BEHAVIOUR:  ********************
******************** A CLOCK EXAMPLE    ********************

  START_CLK  |- !clk wait. START_CLK clk wait = (clk 0 = wait)
  SET_CLK  |- !clk t wait. SET_CLK clk t wait = (clk(t + 1) = t + wait)
  TICK_CLK  |- !clk t. TICK_CLK clk t = (clk(t + 1) = clk t)
  RING_CLK  |- !clk t. RING_CLK clk t = (clk t) < t
  maxP_CLK_DEF
    |- !rem maxP clk.
        maxP_CLK_DEF rem maxP clk =
        START_CLK clk(maxP - 1) /\
        0 < maxP /\
        (!t.
          ((NULL(rem t) \/ ~(rem(t + 1) = rem t)) =>
           SET_CLK clk t maxP |
           TICK_CLK clk t))
  CLK_abort_def
    |- !abort clk.
        CLK_abort_def abort clk = (!t. abort t = RING_CLK clk t)


******************** CLOCK BEHAVIOUR IMPLIES  ********************
******************** ABORT SPECIFICATION      ********************

  rem_same_clk_lemma
    |- !rem maxP maxdd maxda clk ackS s r SW M Sdif Rdif.
        ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
        M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
        maxP_CLK_DEF rem maxP clk ==>
        (!n t.
          ~NULL(rem(t + n)) ==>
          ((rem(t + n) = rem t) =>
           (clk(t + n) = clk t) |
           (clk(t + n)) >= (t + maxP)))
  CLK_lemma3
    |- !rem maxP clk.
        maxP_CLK_DEF rem maxP clk ==> (!t. (clk t) < (t + maxP))
  CLK_lemma1
    |- !rem maxP maxdd maxda clk ackS s r SW M Sdif Rdif.
        ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
        M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
        maxP_CLK_DEF rem maxP clk ==>
        (!t. t < maxP /\ ~NULL(rem t) ==> ~(clk t) < t)
  CLK_lemma2
    |- !t rem maxP clk.
```

```
      maxP_CLK_DEF rem maxP clk /\ NULL(rem t) ==> ~(clk t) < t
  maxP_CLK_thm
    |- !rem maxP maxdd maxda abort clk ackS s r SW M Sdif Rdif.
      ACK_RECV ackS rem s M(arecv_min ackS s M SW) /\
      M_ASSUM M SW s r maxdd maxda Sdif Rdif /\
      maxP_CLK_DEF rem maxP clk /\
      CLK_abort_def abort clk ==>
      ABORT abort maxP rem


******************** NEW CHANNELS  ********************

  ORDERED_CH
    |- !In Out d maxd.
      ORDERED_CH In Out d maxd =
      CHANNEL In Out d maxd /\
      (!t1 t2.
        good_packet(Out t1) /\ good_packet(Out t2) /\ t1 < t2 ==>
        (t1 - (d t1)) <= (t2 - (d t2)))
  GOOD_CHANNEL
    |- !In Out d maxd.
      GOOD_CHANNEL In Out d maxd =
      CHANNEL In Out d maxd /\
      (!t1 t2.
        good_packet(Out t1) /\ good_packet(Out t2) /\ t1 < t2 ==>
        (t1 - (d t1)) < (t2 - (d t2)))
  CHANNEL_WITH_CRASHES
    |- !In Out d maxd has_crashed.
      CHANNEL_WITH_CRASHES In Out d maxd has_crashed =
      CHANNEL In Out d maxd /\
      (!t. has_crashed t ==> ~good_packet(Out t))

  COMPOSE_CHANNEL_THM
    |- !a b c d1 d2 maxd1 maxd2.
      CHANNEL a b d1 maxd1 /\ CHANNEL b c d2 maxd2 ==>
      CHANNEL a c(\t. (d2 t) + (d1(t - (d2 t))))(maxd2 + maxd1)

******************** END SW.ALG THEORY ********************
```

```
******************** SEQMOD THEORY ***********************
******* SUPPORT FOR MODULO SEQUENCE NUMS IN GEN  ********

The Theory SEQMOD
Parents --  HOL     int     arith     myarith
Constants --
  plusm ":num # (num # num) -> num"     N ":num -> (num -> num)"
  subm ":num # (num # num) -> num"
Definitions --
  plusm  |- !a b m. plusm(a,b,m) = ABS(((Int a) plus (Int b)) MODI m)
  N  |- !a m. N a m = ABS((neg(Int a)) MODI m)
  subm  |- !a b m. subm(a,b,m) = plusm(a,N b m,m)


Theorems --
  MOD_MOD
    |- !n. (Int 0) << (Int n) ==> (!a. (a MODI n) MODI n = a MODI n)
  MOD_0  |- !n. (Int 0) << (Int n) ==> ((Int 0) MODI n = Int 0)
  Add_MOD_MOD
    |- !m.
        (Int 0) << (Int m) ==>
        (!a b.
          ((a plus (b MODI m)) MODI m = (a plus b) MODI m) /\
          (((a MODI m) plus b) MODI m = (a plus b) MODI m))
  Int_abs  |- !i. (Int 0) Leq i ==> (Int(ABS i) = i)
  Int_abs_MOD
    |- !m. (Int 0) << (Int m) ==> (!i. Int(ABS(i MODI m)) = i MODI m)
  Leq_LESS_OR_EQ  |- !m. (Int 0) Leq (Int m) = 0 <= m
  MODI_IS_a  |- !a m. 0 < m /\ a < m ==> (ABS((Int a) MODI m) = a)
  plus_in_plusm
    |- !a b c m. 0 < m ==> (plusm(a + b,c,m) = plusm(plusm(a,b,m),c,m))
  subm_eq_sub
    |- !a b m.
        0 < (a - b) /\ (a - b) < m /\ 0 < m ==> (subm(a,b,m) = a - b)
  N_m  |- !m. 0 < m ==> (N m m = 0)
  N_less_m  |- !n m. 0 < m ==> (N n m) < m
  plusm_is_a  |- !a m. 0 < m /\ a < m ==> (plusm(a,0,m) = a)
  plusm_m  |- !a m. 0 < m ==> (plusm(a,m,m) = plusm(a,0,m))
  plusm_less_m  |- !a b m. 0 < m ==> (plusm(a,b,m)) < m
  PLUSM_COMMUTATIVE  |- !a b m. plusm(a,b,m) = plusm(b,a,m)
  PLUSM_ASSOC
    |- !a b c m.
        0 < m ==> (plusm(a,plusm(b,c,m),m) = plusm(plusm(a,b,m),c,m))
  PLUSM_ID  |- !a m. plusm(a,0,m) = ABS((Int a) MODI m)
  PLUSM_INV  |- !a m. 0 < m ==> (plusm(a,N a m,m) = 0)
  plusm_eq
    |- !a b c m.
        0 < m /\ a < m /\ c < m /\ (plusm(a,b,m) = plusm(c,b,m)) ==>
        (a = c)
```

```
n_plusm
  |- !a b m. 0 < m ==> (N(plusm(a,b,m))m = plusm(N a m,N b m,m))
plusm_plus_m  |- !a b m. 0 < m ==> (plusm(a + m,b,m) = plusm(a,b,m))
plus_in_subm  |- !a b m. 0 < m ==> (N(a + b)m = N(plusm(a,b,m))m)
subm_plusm_to_num
  |- !a b c m.
      0 < m /\ a < (b + c) /\ (b + c) < (a + m) ==>
      (subm(a,plusm(b,c,m),m) = (a + m) - (b + c))
subm_plusm_2_to_num
  |- !a b c m.
      0 < m /\ (a + b) < c /\ c < ((a + b) + m) ==>
      (subm(plusm(a,b,m),c,m) = ((a + b) + m) - c)
plus_eq_plusm
  |- !a b m. 0 < m /\ (a + b) < m ==> (a + b = plusm(a,b,m))
plusm_sub_sub
  |- !a b c m.
      0 < m ==> (plusm(subm(a,b,m),subm(b,c,m),m) = subm(a,c,m))
subm_plusm
  |- !m.
      0 < m ==>
      (!a b c. subm(plusm(a,b,m),c,m) = plusm(subm(a,c,m),b,m))
plusm_0  |- !a b m. 0 < m ==> (plusm(plusm(a,b,m),0,m) = plusm(a,b,m))
plusm_subm  |- !a b m. 0 < m /\ a < m ==> (plusm(subm(a,b,m),b,m) = a)
subm_self  |- !m. 0 < m ==> (!r. subm(r,r,m) = 0)
plusm_subm_self
  |- !m a b c.
      0 < m ==> (subm(plusm(subm(a,b,m),b,m),c,m) = subm(a,c,m))
plusm_subm_self2
  |- !m a b c.
      0 < m ==> (subm(c,plusm(subm(a,b,m),b,m),m) = subm(c,a,m))
change_sides_simpl
  |- !m a b c.
      (plusm(a,b,m) = c) /\ 0 < m /\ b < m ==> (b = subm(c,a,m))
change_sides
  |- !a b c d m.
      (subm(plusm(a,b,m),c,m) = d) /\ 0 < m /\ b < m ==>
      (b = plusm(subm(c,a,m),d,m))

******************** END SEQMOD THEORY *******************
```

```
******************** hdi_tli THEORY ********************

The Theory hdi_tli
Parents --  HOL     myarith     arith
Constants --
  TLo ":(*)list -> (*)list"     TLI ":num -> ((*)list -> (*)list)"
  HDI ":num -> ((*)list -> *)"
Definitions --
  TLo  |- !l. TLo l = ((~NULL l) => TL l | [])
  TLI  |- (!l. TLI 0 l = l) /\ (!n l. TLI(SUC n)l = TLo(TLI n l))
  HDI  |- !n l. HDI n l = HD(TLI n l)

Theorems --
  LENGTH_TL  |- !l. ~NULL l ==> (LENGTH(TL l) = (LENGTH l) - 1)
  NULL_LENGTH_0  |- !l. NULL l ==> (LENGTH l = 0)
  APPEND_NIL  |- !l. APPEND l[] = l
  TLI_NIL  |- !n. TLI n[] = []
  HDI_TLI_1
    |- !x l. ~NULL(TLI x l) ==> (APPEND[HDI x l](TLI(x + 1)l) = TLI x l)
  HDI_TLI_2  |- !x y l. TLI x(TLI y l) = TLI(x + y)l
  LIST_EQ_IMP_LENGTH_EQ
    |- !l1 l2. (l1 = l2) ==> (LENGTH l1 = LENGTH l2)
  LENGTH_LESS_IMP_NOT_EQ
    |- !l1 l2. (LENGTH l1) < (LENGTH l2) ==> ~(l1 = l2)
  LENGTH_NOT_NULL  |- !l1. ~NULL l1 ==> 0 < (LENGTH l1)
  HDI_TLI_3  |- !x l. LENGTH(TLI x l) = (LENGTH l) - x
  LENGTH_TLI
    |- !l1 l2 x.
         ~NULL l2 /\ (l1 = TLI x l2) /\ 0 < x ==>
         (LENGTH l1) < (LENGTH l2)

******************** END hdi_tli THEORY ********************
```