Using Higher Order Logic for Modelling Real-Time Protocols

Rachel Cardell-Oliver
Computer Laboratory, University of Cambridge
CAMBRIDGE, CB2 3QG, U.K.
(rco@cl.cam.ac.uk)

1 Introduction

A protocol is a set of rules which control communication between machines in a computer network; it is required in an environment where two or more computers communicate only via the medium connecting them. Usually, communication over the medium is unreliable. Communication protocols are difficult to understand and to implement correctly because the interaction between protocol programs and their uncooperative, real-time environment is complex.

We present a method for the formal specification and verification of real-time protocols using higher order logic. The main contributions of this paper are to show how higher order logic can be used to write *generic* specifications for a class of protocols and to illustrate the use of such specifications.

Our specification language is higher order logic. Processes are modelled by higher order predicates on signals (functions from time to values) which express relations between input, output and internal state. Protocols are specified at multiple levels of abstraction; a modular specification style is supported; both requirements (what is achieved) and behaviour (how it is achieved) can be modelled.

The class of sliding window protocols is used to illustrate our method. The HOL system, a machine implementation of higher order logic [Gor88], has been used to specify and verify this example. Further details of the definitions and theorems used may be found in [CO90] and an introduction to the class of sliding window protocols may be found in [BG87].

The specification method used here is not new: it has been widely used to model hardware. For example, [HD86, Mel89, Joy89, Gor86] describe abstraction techniques and generic specifications for hardware, and [MS88] describes the timing behaviour (only) of real time systems. However, to the best of our knowledge ours is the first use of such a model for generic specifications of software such as communication protocols.

The paper is organized as follows. Sections 2 and 3 describe our specification language and the framework in which we intend to verify protocols. Sections 4 to 6 give specifications for the class of sliding window protocols at three different logical levels. Verification proofs are also discussed. Sections 7 and 8 discuss the relationship of our model to existing techniques for the specification of real-time protocols and summarize our conclusions.

2 Specification Language

2.1 Higher Order Logic

The version of higher order logic used in this paper is based on Church's simple type theory [Chu40]. Higher order logic contains all the terms of first order logic and also higher order terms: predicates or functions with predicates and functions as arguments or results. Logical terms include truth and falsity, predicates (`P x` means `x` has property `P`), negation ($\neg$), disjunction ($\vee$), conjunction ($\wedge$), implication ($\Longrightarrow$), equivalence ($\equiv$), equality ($=$), quantification ($\forall$ and $\exists$) and conditionals (`a ⇒ c1 | c2`). The usual operators for natural numbers are available: $+, -, \times, <, \leq$. Functions may be denoted by lambda expressions: λ `t. t+3` is the function which returns a number three greater than its argument. Alternatively, the same function could be defined by `f t = t+3`.

Our version of higher order logic ensures that every logical term has a well defined type which is either atomic (e.g. `num`, `bool`, `string` etc.) or compound (e.g. *ty* `list`, $ty_1 \times ty_2$, $ty_1 + ty_2$, $ty_1 \rightarrow ty_2$). Compound types are explained in section 2.2. The notation `t:ty` means term `t` has type `ty`.

HOL [Gor88] is a theorem prover for higher order logic derived from LCF [GMW79]. HOL users may make definitions, prove theorems about them, and group constants, types, definitions and theorems together as theories which can be stored on disk. A theorem has a special type: a formula can only be a theorem if it has been formally deduced from the 5 axioms and 8 rules of inference which are the basis of this version of higher order logic. We write $\vdash$ `f` when `f` is a theorem. HOL theorems are "secure" in that each theorem is know to have a formal proof. Users can prove theorems by interacting with a goal directed proof handler and by programming proof strategies in the ML programming language.

2.2 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, a protocol 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 signals in and out of channels and to and from a protocol's outside world can be characterized by data types. For example, the output stream from a receiver could be represented by a signal from time to an arbitrary 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`. The type representing no output, `non_packet`, is 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

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

Computer programs (software) manipulate variables stored in memory. These variables 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`.

2.3 Specifications

In order to describe a computer system using a set of signals we must specify the relationships which hold between signals. Logical predicates on signals are used to describe such relationships. They may be stated as requirements (*what* must be achieved by the system) or as behaviours (*how* a requirement is implemented).

For example, the requirements for a sliding window protocol could be specified by the relationship which must hold 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, sink) ≡
    ∀ t:time.  ∃ t':time.  t'<t ∧ sink t = source t'
```

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

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

Higher order logic (as opposed to first order predicate logic) is necessary for this modelling method because signals are first order functions and thus predicates with signals as parameters are second order or higher.

In another example a counter signal, `count:time → num`, is initially set to 0 and then updated whenever an input signal, `tick:time → bool`, is true. `count` can be specified by:

```
Counter (tick, count) ≡
   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 output 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:num.

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

Predicates are combined using logical conjunction. Any number of requirements specifications may be combined:

```
SW_SPEC (source, sink) ≡
    OutputWasInput(source,sink) ∧
    OrderPreserved(source,sink)
```

When predicates represent process behaviour, then their conjunction can be thought of as the parallel execution of those processes. Thus, the specification of the combined behaviour of these two devices, in which count is output 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, clkint) ≡
    ∃ count:time → num.  ∃ MC:num.
      Counter(tick,count) ∧ Timer(count,clkint,MC)
```

2.5 Verification

Specifications for real time software systems such as protocols can be expressed in higher order logic. How can we prove that a specification at one level of abstraction satisfies another?

An implementation model A (a predicate) meets its specification B (another predicate) if A logically implies 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 that an implementation is more specified than its specification.

There are two types of partial solution. First, a verifier must prove more properties of his 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 properties "reasonableness" properties.

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 ∧ 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.

3 Verification Strategy

3.1 A Four Level Model

The formal specifications given in sections 4 to 6 assume the following model for protocol specifications.

We distinguish between a high level specification which is sufficiently simple to be easily seen to be correct and implementations in hardware and software which are more complex. Between these extremes there exist a range of protocol descriptions at different levels of abstraction. We have identified four levels of description which characterize protocols: implementations (IMPL), algorithms (ALG), generic specifications of protocol behaviour(GEN), and a minimal requirements specification (SPEC). In this paper examples of SPEC, GEN and ALG specifications are given. We do not discuss implementation models (where 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 requirements 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 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 requirements 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 (from the source) but are not known to have been output by the receiver (to the sink): 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.

3.2 A Sample Implementation

This section gives an informal description of an implementation from the class of protocols we have specified. It is included for readers not familiar with sliding window protocols and may be skipped.

The example of Figure 1 may be used to identify important features of sliding window protocols. 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

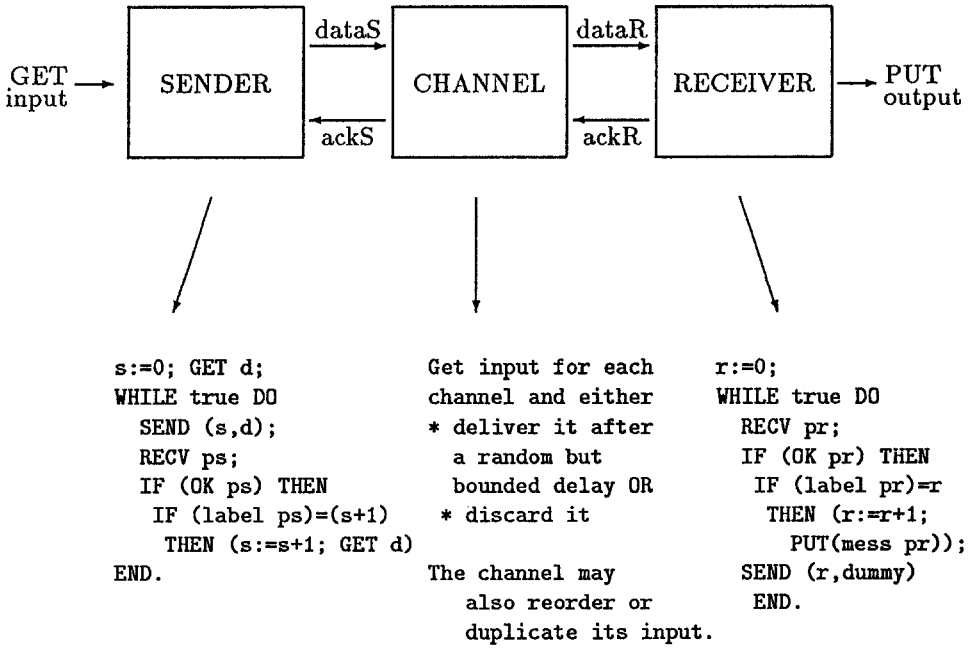

Figure 1: An Implementation of Stenning's Protocol

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

We can now discuss different specifications for the class of protocols of which this implementation is a member.

4 SPEC: a minimal requirements specification

A sliding window protocol transfers a stream of data from one machine in a computer network to another. In section 2.3 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.

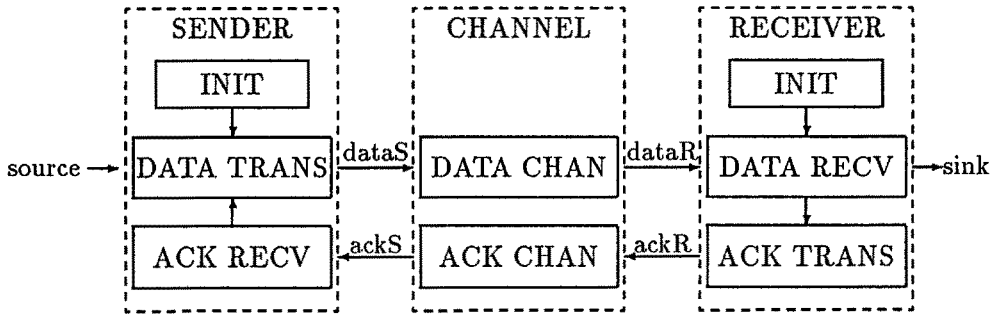

Figure 2: A Logical Structure for Sliding Window Protocol Models

In the HOL representation of this protocol, the values shared between these entities are indicated by the shared parameters. The signals for channels are hidden inside the specification.

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

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 resulted in some events being ignored etc. In an abstract model these details should be left unspecified. 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 (after some delay `d t`) is, for the protocol designer, completely non-deterministic and is 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 specify the choice only partially to make his 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. Its definition was chosen to meet the following criteria

- A partially specified choice made with extra information implies the original partially specified choice.
- A fully specified choice implies partially specified choice

The first requirement is needed because 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. The second requirement is used for programming language implementations. Here all choices will be deterministic (e.g. an IF THEN ELSE command) and will be modelled by the fully specified choice `((a ⇒ d=t1|d=t2) ∧ (¬(t1=t2)))`.

`PSC a d t1 t2` should be read as "if `a` then perhaps `d=t1` else `d=t2`". Formally, the requirements given above are,

- `PSC (a ∧ b) d t1 t2 ⟹ PSC a d t1 t2`
- `(a ⇒ d=t1 | d=t2) ∧ ¬(t1=t2) ⟹ PSC a d t1 t2`

The conjuncts in the definition of PSC state that `d` will be equal to either `t1` or `t2`, that `t1` and `t2` must be different (which is true in all our specifications), and that if `d=t1` is chosen then the choice value `a` must have been true. The type variable, `*`, shows 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
```

To see how partially specified choice is used in the generic specification, GEN, consider the transmission of data messages. 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 (represented by

rem:time → *data list) are available for transmission at any 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 and the function TLI n l returns the tail of the list l starting from its n-th element. The predicate NULL l is true if l is the empty list and false otherwise. 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 occurred 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 process. The RECEIVER accepts packets (dataR t) transmitted by the SENDER from the channel. If a message is received whose sequence number is equal to the next number to be output to the sink then two things happen: the data part of the packet is output to the sink list and the state signal r:time → sequence is updated. In our HOL specification we use a pair of type :(*data list) × seq and choose to set the values of sink(t+1) and r(t+1) either to updated values or to their values at time t guided by the predicate (good_packet(dataR t) ∧ (label(dataR t)=(r t))). The function APPEND joins two lists in the usual way.

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

A model for a receiver strategy which buffers incoming packets is described in Section 6.2.

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 extended state machines which use their inputs at any time to calculate their next state. The specifications for DATA_TRANS and DATA_RECV given in the last section show how state specifications can be written in our model.

Alternatively, requirements may be specified using real time intervals. 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. Real time intervals, for example, should be used to specify channel delay and timeouts, so that we can verify that a protocol responds correctly to all possible events and their orderings.

Interval 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)). This specification 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

$$\forall\ t.\quad 0 < d\ t \wedge d\ t \leq \mathtt{maxdelay}$$

The full channel specification 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 6.3 it is shown how better behaved channels can be specified as special cases of this channel.

$$\begin{aligned}&\mathtt{CHANNEL\ In\ Out\ d\ maxdelay} \equiv\\ &\qquad(\mathtt{Out\ t = In\ t\text{-}(d\ t)} \vee \mathtt{Out\ t = set\_non\_packet}) \wedge\\ &\qquad(\mathtt{0 < d\ t}) \wedge (\mathtt{d\ t} \leq \mathtt{maxdelay})\end{aligned}$$

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.

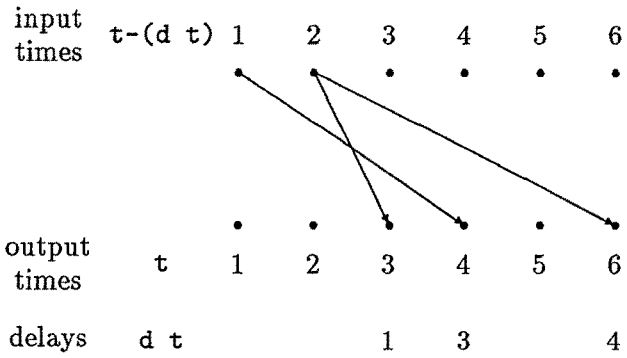

Figure 3: Sample behaviour of the CHANNEL specification

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. In Section 6.1 we show how an algorithm's data transmission strategy can be described 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 an interval of maxP:time 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, a parameter of GEN and also of ABORT, 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. 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
```

PROOF OUTLINE : The proof of this lemma is by induction. When t is 0 then the predicate INIT (one of the conjuncts of GEN) states that s 0 = 0 ∧ r 0 = 0 and that sink 0 = □ ∧ rem 0 = source. Thus, by elementary properties of APPEND, TLI and ⊖ the required equivalence holds.

The step case of the induction can be proved by considering both cases for the values of r, sink at t+1. First note that

$$\forall t.\ \mathtt{rem}(t+1) = \mathtt{TLI}\ (\mathtt{s}(t+1) \ominus (\mathtt{s}\ t))\ (\mathtt{rem}\ t)$$

since s and rem are always updated together. There are two cases for the value of r(t+1) and sink(t+1): either they are updated to (r t) $\oplus$ 1 and (APPEND (sink t) [message(dataR t)]) respectively, or the new values are the same as those at time t. In the second case our inequality is the same as the induction hypothesis and so the required equivalence holds. For the first case we show that

$$\texttt{message(dataR t)} = \texttt{HDI (r t} \ominus \texttt{s t) (rem t)}$$

since dataR t=dataS(t-(dd t)) and rem t can be related to rem(t-(dd t)) when certain conditions are specified on the range of sequence numbers used. Thus, the required equivalence holds for all possible cases of s,r,sink,rem.
END PROOF.

The second lemma states that as long as the protocol is not aborted then the list signal rem will be empty by time maxP*LENGTH(rem 0) at the latest. The function LENGTH l is equal to the length of list l. The specification, GEN, contains a liveness assumption: (∀ t:time. ¬(abort t)) where abort is specified by the predicate ABORT given in Section 5.5. Assuming the protocol is *not aborted*, the lemma

$$\forall\ \texttt{n.}\quad \texttt{LENGTH (rem(maxP*n))} \leq \texttt{LENGTH(rem 0) - n}$$

is used to prove

$$\texttt{LIVENESS} \equiv$$
$$\vdash \texttt{GEN} \Longrightarrow \texttt{(rem (maxP*LENGTH(rem 0)) = [])}$$

The theorems SAFETY and LIVENESS are used to prove

$$\vdash \texttt{GEN} \Longrightarrow \texttt{SPEC}$$

All these proofs have been mechanically checked using HOL. Further details are given in [CO90].

6 ALG: protocol algorithm specifications

This section illustrates how generic specifications given in section 5 can be used as the basis of a more detailed algorithm specification. A sliding window protocol algorithm is a "more defined" version of the class specification GEN. For example, in an algorithm choices about window sizes, when and what values to transmit, how to buffer messages and how to acknowledge them are made explicit. In our model this is done by adding information to a generic specification in one of the following ways:

1. give values for constant parameters (e.g. SW=7 ∧ M=16)

2. give definitions for partially defined parameters (e.g. see Section 6.1)

3. strengthen partially specified choices (e.g. see Sections 5.2 and 6.3)

4. add new behavioural predicates (e.g. see Sections 6.2 and 6.3)

5. replace requirements specifications by behavioural ones (e.g. see [CO90])

We define ALG $\equiv$(GEN $\wedge$ P) where P represents extra information as above and then prove the theorems $\vdash$ ALG $\Longrightarrow$ GEN and thus $\vdash$ ALG $\Longrightarrow$ SPEC. We can also prove that the extra information added to GEN does not contradict conditions already established. For example, a predicate M_ASSUM states necessary conditions for the choice of window size SW and sequence number range M. Choosing SW=8 $\wedge$ M=8 would make the predicate M_ASSUM equivalent to logical false, and so is not a "reasonable" extension of GEN.

6.1 Transmission Strategies

Transmission strategies may be specified by giving values for the function i (a parameter of DATA_TRANS) which specifies the next message to be transmitted and was only partially specified in GEN.

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.

The predicate is_sent, true whenever data has been transmitted, shows when next should be updated: either for the next data message in the current window or to return to the bottom of the window. There are two cases when transmission should return to the bottom of the window (to_bottom): when the top of the window has been reached ((next t)$\ominus$(s t) = SW) or when an update of s t moves the bottom of the window beyond the current value of next t (SW < (next t)$\ominus$(s t)). Since next is the sequence number of the next packet to be transmitted and we require (s t)$\oplus$(i t) = next t, we define i t as (next t)$\ominus$(s t).

```
is_sent N t = good_packet(dataS t) ∧ (label(dataS t)=N)

to_bottom t = ((next t)⊖(s t) = SW) ∨
                    (SW < (next t)⊖(s t))
```

```
next 0 = s 0 ∧
∀ t.  next(t+1) = is_sent (next t) t
                    ⇒ to_bottom t
                        ⇒ s t
                        | (next t)⊕ 1
                    | next t
i t = next 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:time`, after transmitting a packet because an acknowledgement for that packet has not arrived. That is,

$$\forall t.\ \texttt{timeout}\ t \ \equiv\ \texttt{rem}\ t \ =\ \texttt{rem}(t - \texttt{TIMEOUT})$$

(see [CO90] for details). In this transmission strategy all data from the current window is transmitted once as in the last example (see `next`). `next` is updated whenever data has been transmitted (see `is_sent`). If a timeout occurs, then the packet at the bottom of the window is transmitted again (see `current`). If there is no new data to transmit and no timeout occurs, then nothing will be transmitted (since `i t = SW`: see section 5.2).

```
is_sent N t = good_packet(dataS t) ∧ (label(dataS t)=N)

next 0 = s 0 ∧
∀ t.  next(t+1) = is_sent (next t) t
                    ⇒ next t ⊕ 1
                    | next t

current t = timeout t ⇒ s t | next t

i t = current t ⊖ s t
```

6.2 Buffers

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 later 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 sender may use a buffer 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: that is, a type of channel. An abstract specification for such a buffer may be defined for all output times `t` for which:

1. an input time g t exists which is at most maxW time steps before t,
2. some input condition IC is satisfied at g t,
3. an output condition OC is satisfied at t,
4. an output value OV is available at time t.

In order to restrict the space required for buffers, input is accepted only if it is within the receiver's current window. This is our input condition IC. 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 output condition OC for this buffer is that the label of the packet input at time g t is the same as the receiver's sequence number r t. The output value is the message part of the packet inputt at g t.

```
IC ≡ IN_WINDOW (dataR (g t)) (r (g t)) RW

OC ≡ r t = label(dataR(g t))

OV ≡ message(dataR(g t))

AbsBuffer dataR r RW DataOut g maxW ≡
  λ t.  (g t) ≤ t ∧ t ≤ (g t)+maxW ∧
        IC ∧ OC ∧ (DataOut t = OV)
```

We have proved that a receiver using this buffer to accept input is a "more defined" version of the receiver in GEN which does not buffer its input (see [CO90] for details). We showed that AbsBuffer is simply a type of channel and that since two channels connected in sequence are also a type of channel, the new specification using AbsBuffer implies the original.

6.3 New Channels

Some sliding window protocol algorithms are designed for a more restrictive network environment than that specified by the CHANNEL predicate. For example, a protocol may only work correctly if its 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.  t1 < t2 ∧
  good_packet(Out t1) ∧ good_packet(Out 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. A lambda expression is used to denote the new delay function of the composed channel.

```
newD d1 d2 = (λ t.  (d2 t) + (d1(t - (d2 t))))

COMPOSE_CHANNEL_THM ≡
 ⊢  CHANNEL a b d1 maxd1 ∧ CHANNEL b c d2 maxd2
   ⟹
    CHANNEL a c (newD d1 d2) (maxd2 + maxd1)
```

Channels which may "crash" and refuse to deliver messages for a certain period can be modelled using auxiliary 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` of type `: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 [HD86, Mel89, Joy89, Gor86]. Although abstraction techniques and the multilevel specification of hardware are covered in [Mel89, Joy89] the treatment of partially specified choice and delay for buffers and channels in our work is new.

Gordon's methodology is used in [MS88] to combine specifications written in DRTL, an extension of Jahanian and Mok's RTL [JM86]. A requirements language and a design language are defined. In [MS88] there is more emphasis on developing a specification language than in our work. For example, the language

determines how and when processes can be connected, what types of real-time events are possible etc. However, only the timing behaviour of a real-time system and not its function are modelled.

Few protocol specification methods in the literature can express real time constraints. A model which does address this problem is described in Shankar and Lam [SL87, Sha89] where an event-action model is used to describe real time protocols. Logical invariants on that model are used for specification and thus multilevel specifications cannot be expressed. In [Sha89] 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.

Interval temporal logic [Mos86, Hal87] and interval logic [MS87] are temporal logics which may be used to specify real time systems. 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: requirements and behavioural specifications such as SPEC, GEN and ALG can be expressed. The proof theory for verifying that one process defined in ITL satisfies another is not well developed. However, an advantage of ITL is that deterministic specifications are executable in the programming language Tempura [Mos86]. Thus, a class of high level specifications in interval temporal logic could be checked by executing them.

Real-time protocols can be modelled by extending process algebras such as CCS to model waiting times of processes. TCCS [MT90] extends CCS with time actions: one action which delays process actions for a fixed period and an action which delays for an indefinite interval. Verification that one process specifies another is by bisimulation. TCCS, like the model presented in this paper, is suitable for multilevel specifications because the language used for specifications and implementations is the same. However, a specification such as SPEC would have to be defined behaviourally in TCCS as, say, a unit buffer. A more important difference between the models is that since TCCS processes communicate using synchronized messages (like CCS processes) only lower bounds on real-time behaviour, and not upper bounds, can be expressed whilst our model can express the exact timing of events.

8 Conclusions

This paper has shown how higher order logic can be used for the specification of real-time protocols at three 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. Our examples have been formally specified and verified using the HOL theorem prover.

Many 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 since

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

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,

3. the development of a non-trivial specification is a difficult and time consuming task; we are able to *reuse* specifications such as SPEC, GEN and ALG and many of the proofs for their verification.

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 [CO90] 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. This problem can be solved by making our constraints on the range of sequence numbers used by a protocol more general, and making some changes to our proofs. 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 defined a semantics for a real time programming language in higher order logic using the model of GEN and ALG specifications. Implementation models can be verified to satisfy ALG specifications and thus also GEN and SPEC for the class of sliding window protocols. Properties about the

performance of protocols could also be verified from an IMPL model. For example, the model could be used to describe the efficiency of particular transmission strategies, window sizes etc. in a given environment.

Acknowledgements

I would like to thank Mike Gordon, Roger Hale, Neil Viljoen and an anonymous referee 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

[BG87] Dimitri Bertsekas and Robert Gallagher. *Data Networks.* Prentice-Hall International, Englewood Cliffs, NJ, 1987.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[CO90] Rachel Cardell-Oliver. Formal verification of real-time protocols using higher order logic. Technical Report 206, University of Cambridge Computer Laboratory, August 1990.

[GMW79] 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.

[Gor86] 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.

[Gor88] Mike Gordon. A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis, Proceedings of the Workshop on Hardware Verification, Calgary, 12-16 January, 1987, G.M. Birtwistle and P.A. Subrahmanyam, eds. Kluwer International Series in Engineering and Computer Science, SECS35*, pages 73–128. Kluwer Academic Publishers, 1988.

[Hal87] 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.

[HD86] 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.

[JM86] 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.

[Joy89] Jeffrey J. Joyce. *Multi-Level Verification of Microprocessor-Based Systems.* PhD thesis, Computer Laboratory, University of Cambridge, December 1989. also published as Computer Laboratory Technical Report 195.

[Mel89] Thomas F. Melham. *Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic.* PhD thesis, Computer Laboratory, University of Cambridge, August 1989. also published as Computer Laboratory Technical Report 201.

[Mos86] Ben Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[MS87] 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.

[MS88] 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.

[MT90] Faran Moller and Chris Tofts. *A Temporal Calculus of Communicating Systems*, pages 401–415. Springer Verlag, August 1990. Lecture Notes in Computer Science 458.

[Sha89] A. Udaya Shankar. Verified data transfer protocols with variable flow control. *ACM Transactions on Computer Systems*, 7(3):281–316, August 1989.

[SL87] 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.