TIME-DEPENDENT COMMUNICATION PROTOCOLS[1]

A. Udaya Shankar[2] and Simon S. Lam

TR-84-26 September 1984

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

---

[1]To appear in *Principles of Communication and Networking Protocols*, S. S. Lam (ed.), IEEE Computer Society Press, Silver Spring, Maryland, 1984.

[2]Department of Computer Science, University of Maryland, College Park, Maryland 20742.

Table of Contents

Abstract

Most real-life communication protocol systems utilize timers and clocks to implement real-time constraints between system event occurrences. Such protocol systems are said to be *time-dependent* if these real-time constraints are crucial to the correct functioning of the systems. We present a model for specifying and verifying time-dependent systems. Specifically, we consider networks of processes that communicate with one another by message-passing. In the context of communication network protocols, each process is either a communication channel or a protocol entity.

A process in our model is specified by a set of state variables and a set of events. An event is described by a predicate that relates the values of the system state variables immediately before to their values immediately after the event occurrence. The predicate embodies specifications of both the event's enabling condition and action. Measures of time are explicitly included in our model. Furthermore, clocks are not coupled and they can tick at any rate within some specified error bounds. Inference rules for both safety and liveness properties are presented. Liveness properties are expressed in the form of inductive properties of bounded-length paths in a system's reachability space. Real-time properties are expressed as safety properties.

We have applied our methodology to the verification of several large communication protocols including a version of the High-level Data Link Control (HDLC) protocol. For the sake of brevity, a relatively small data transfer protocol is modeled herein for illustration. This protocol can reliably transfer data over bounded-delay channels that can lose, reorder and duplicate messages in transit. The protocol's safety, liveness and real-time properties are presented.

1. INTRODUCTION

We consider real-time systems that employ clocks and timers to enforce time constraints between system event occurrences. We refer to a system as *time-dependent* when it contains time constraints that are essential to the correct functioning of the system. Our work has been motivated primarily by real-life communication network protocols which are invariably time-dependent systems [1-4].

Time-dependent behavior arises naturally in communication networks because errors and failures that occur in a network component are usually not communicated explicitly to other network components that may be affected by these errors and failures. In such situations, only by the use of timeouts can a component infer that a failure (or an error) has occurred and initiate recovery action. Because such recovery mechanisms are themselves subject to the same kinds of failures or errors, the real-time behavior can be very complex.

We present in Sections 2, 3 and 5 an event-driven process model for specifying and verifying distributed systems, both time-dependent and time-independent. We have applied this model to the analysis of several nontrivial protocol examples, including a version of the High-level Data Link Control (HDLC) protocol [5]. To illustrate our model, we present below a transport-level protocol for reliable data transfer over bounded-delay channels that can lose, reorder and duplicate messages in transit. The protocol employs cyclic sequence numbers, timers and timeouts.

Review of protocol verification

The aim of protocol verification is to establish that certain desired logical correctness properties of a protocol system are guaranteed by its specifications. For distributed systems in general, such logical correctness properties are generaly categorized into safety properties and liveness properties. Informally, a *safety property* states that certain relations always hold between a set of system variables irrespective of whether the system progresses or not (e.g., if data blocks are delivered to a remote user, then they are delivered in the same order as the order in which they were submitted by the local user). A *liveness property* states that the system will indeed progress in a certain manner (e.g., a data block that is submitted by the local user will *eventually* be delivered to the remote user).

However, in the case of communication network protocols, it is often desirable to describe progress in terms of real-time properties. Typically, if a protocol does not achieve progress (transfer of data, establishment of a connection, etc.) within a bounded time duration T, then the protocol resets or aborts. Hence, a liveness assertion stating progress within a finite but *unbounded* time duration is not realistic. More appropriate is a real-time assertion such as "if within a time duration T the data block is not transferred, then at least n retransmissions of the data block have occurred, all of which failed." In our model, such real-time assertions can usually be stated as safety assertions.

Protocol verification consists of proving, in some deductive inference system, that the correctness properties of a protocol system follow from the system specifications. An inference system consists of a syntax for expressing statements, as well as a collection of axioms and inference rules. An axiom is a statement whose truth is taken for granted. An inference rule specifies how a new statement can be derived from other statements. A proof of a statement A_0 in such a system is a sequence of statements (with A_0 at the end) where each statement is either an axiom or is derived from previous statements by an application of an inference rule. First-order predicate logic is an example of a deductive inference system. An inference system for protocol verification can be obtained by extending first-order predicate logic with additional inference rules for proving safety and liveness properties. These additional inference rules define the semantics of constructs that are used in protocol specification.

Some features of our model

We model distributed systems as networks of processes that communicate with one another by message-passing. For communication network protocols in particular, each process in a network is either a communication channel or a protocol entity. Each process has a set of *state variables* and a set of *events*. An event is described by a predicate that relates the values of the system state variables immediately before the event occurrence to their values immediately after the event occurrence. The predicate embodies specifications of both the event's enabling condition and action. There is no algorithmic code in our model.

What we have is a compromise that incorporates both implementation-dependent features (the state variables) and implementation-independent features (the use of predicates to specify events). Such a combination has several advantages. First, it allows for very simple inference rules for safety and liveness properties. In particular, we do not need any special notation (such as temporal logic [6]) to express liveness properties of *unbounded-length* paths in a system's reachability graph. Instead, we describe such liveness properties in the form of inductive properties of bounded-length paths. Second, because the events are specified by predicates, their specifications can be directly substituted into proofs in predicate logic. Third, the use of state variables simplifies the modeling of time measures.

We use discrete-valued time variables to measure the elapse of time, and define time events to age these time variables. By imposing certain conditions, referred to as *accuracy axioms*, on the time events, we are able to model clocks realistically: our clocks are uncoupled and can tick at any rate within specified error bounds of a given rate. Additional conditions on the time events, referred to as *time axioms*, allow us to model many types of time constraints between system events.

Related work

In addition to time-dependent behavior, another characteristic of real-life communication protocols is that each protocol typically performs multiple distinct functions, such as connection management, one-way data transfers, etc. The method of projections provides an approach to transform the analysis of a multi-function protocol into analyses of smaller single-function protocols, called image protocols [7].

The theory of projections was originally developed in [7] using a set-theoretic notation. In Part 2 of [8], we specialize the theory to the time-dependent protocol system model herein. The use of state variables and predicates (to specify events) greatly facilitates the construction of image protocols.

2. MODELING MEASURES OF TIME

Protocol system components have devices, such as crystal oscillators, that issue "ticks" at (almost) regular time intervals (e.g., once every microsecond). We refer to such devices as *local tickers*. In order to measure intervals of time larger than that between consecutive ticks, the system components typically employ counters to accumulate the number of elapsed ticks generated by a local ticker since the occurrence of some system event. These counters are the clocks and timers used in the protocol system. We refer to such counters as *time variables*.

The tickers in protocol systems have several properties that should not be ignored in any realistic modeling. First, the interval between successive ticks is not infinitesimally small. Second, tickers in different components are not coupled: the ticks of one ticker do not coincide in time with the ticks of another ticker. Third, the rate at which a ticker ticks is not constant but may vary within certain error bounds of a constant rate. We account for each one of these features in our model.

The time variables in our model are discrete-valued variables. Without loss of generality, we consider them to take values from the set of non-negative integers. For each local ticker i, there is a *local time event* (corresponding to a tick) whose occurrence ages (increments) all time variables driven by the ticker. Since no other ticker is affected, this ticker is effectively decoupled from other tickers. All the time variables driven by a local ticker must necessarily lie within a single component of the protocol system. In addition to being aged, a time variable can be *reset* to some value by an event of its component. Thus, a time variable can be used to measure the time elapsed (in number of ticks of its local ticker) since an event occurrence.

At this point in the modeling, the tickers are entirely decoupled. There is nothing to prevent different local tickers from ticking at vastly different rates. To complete the modeling of real-time measures, we must keep local tickers within specified drifts. To do this we include in our model a hypothetical ticker, referred to as the *global ticker*, that is assumed to tick at an absolutely constant rate. Each local ticker will be allowed to drift

within a specified bound of the global ticker. For each local ticker i, let η_i denote the number of ticks issued since system initialization, and let ϵ_i denote the maximum error in the tick rate. Let η denote the number of global ticks since initialization. The η's are (auxiliary) time variables that do not correspond to actually implemented clocks or timers, and can never be reset by any system event. Neither the local time event for ticker i nor the global time event is allowed to occur if such an occurrence will violate the following *accuracy axiom* of local ticker i (below, the notation $\eta(a)$ refers to the value of η at instant a, while η refers to the current value of the time variable η):

AccuracyAxiom$_i$(η_i,η): For any earlier instant a,

$| (\eta_i - \eta_i(a)) - (\eta - \eta(a)) | \leq \max(1,\epsilon_i(\eta-\eta(a)))$.

The above accuracy axiom states that over any time span since initialization, the number of ticks of local ticker i differs from the number of global ticks by at most ϵ_i times the number of global ticks. (The minimum upper bound of 1 is necessary since the tickers are integer-valued.) This accuracy axiom is a discrete version of the following drift condition for continuous clocks

$$\left| 1 - \frac{d\eta_i}{d\eta} \right| \leq \epsilon_i$$

usually found in the literature [9].

Recall that time variables are used to measure the time elapsed since a system event occurrence. Thus, by including time variables in the enabling condition of a system event e, we can model time constraints of the form "event e *will not occur unless* certain time intervals have elapsed." To model time constraints of the form "event e *will occur within* certain elapsed time intervals," we impose conditions, referred to as *time axioms* on the allowed values of time variables. As a result, in addition to the constraints imposed by accuracy axioms, time events will not be allowed to occur if their occurrence will violate any time axioms. This modeling of real-time behavior is valid provided that the time events do not get deadlocked because of the accuracy and time axioms. We have shown that the tickers will continue to accumulate ticks if the time axioms correspond to *feasible constraints*; i.e., constraints that can be realistically met by the protocol entities [8]. A precise notation for specifying the time events is presented with the protocol example in Section 4.

We shall refer to time variables driven by local tickers as *local time variables*. We also allow time variables to be driven by the global ticker. These time variables are referred to as *global time variables*; they do not correspond to clocks and timers that are implemented in the system. Rather they are used to record the exact (i.e., global) time elapsed between system events. Such measurements may be needed in stating assertions of time-dependent behavior.

In particular, for a local time variable v, we often find it convenient to specify a global time variable v* such that any system event that resets v also resets v* to the same value. v* is referred to as the global time variable associated with v. Clearly, if v is driven by η_i, then AccuracyAxiom$_i$(v,v*) holds between all instants after the last reset.

3. DISTRIBUTED SYSTEM MODEL

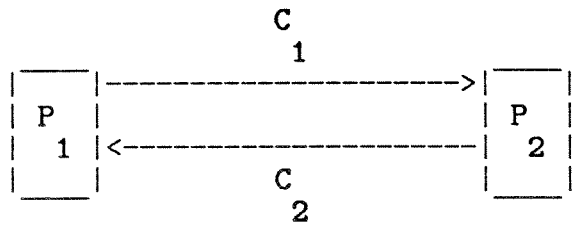

Fig. 1. Network configuration of protocol example in Section 4.

In this section, we specify the messages, state variables and events of the protocol system model. Instead of describing the model for communication networks in general, we present the model for the special configuration and channel behavior of the protocol example in Section 4. (See Fig. 1.) P_1 and P_2 are two protocol entities connected by bounded-delay channels C_1 and C_2. For i=1 and 2, any message attempting to stay in channel C_i for longer than a specified time MaxDelay_i is lost (e.g., removed by some intermediate network node [3, 10]). (For a general network configuration with various types of channels, the reader is referred to [8].)

Messages and state variables

For i=1 and 2, the messages sent by P_i are categorized into *message types*. Each message type q specifies multi-field messages of the form $(q, f_1,...,f_n)$ where $n \geq 0$. The first component indicates the name of the message type, while each of the other components f_i is a parameter ranging over a specified set of values. The vector notation $(q,\mathbf{f})$ is used to refer to the message type q. We note that multi-field messages are characteristic of most real-life communication protocols.

Let $\mathbf{v}_i$ be the set of state variables of P_i. Every variable in $\mathbf{v}_i$ takes values from a specified domain of values. In order to model timing constraints of P_i, $\mathbf{v}_i$ can include time variables (which may be constrained by time axioms). Assume that all time variables in $\mathbf{v}_i$ (if any) are driven by a local ticker with count η_i and maximum error ϵ_i. $\mathbf{v}_i$ can also have auxiliary variables used for verification purposes. An auxiliary variable is used only to record the behavior of the entity over time (e.g., sequence of data blocks delivered to a user); its value never affects the behavior of the entity.

For each message in channel C_i, we associate with the message a time variable *age* that indicates the age of the message (time spent in the channel). Let $\mathbf{z}_i$ be the sequence of (message, age) pairs in C_i. We let the *age* time variable be driven by the global ticker. The channel's bounded-delay behavior is modeled by the *time axiom*

$\text{TimeAxiom}_i(\mathbf{z}_i)$: For every (m,t) in $\mathbf{z}_i$, $t \leq \text{MaxDelay}_i$.

The global state vector is defined as $\mathbf{v} = (\mathbf{v}_1,\mathbf{v}_2,\mathbf{z}_1,\mathbf{z}_2)$. The initial conditions of the protocol system are given by a predicate named Initial($\mathbf{v}$).

Events

Before we describe how events are specified, we first introduce our use of predicates to specify relations between sets of input and output parameter values. Let **x** and **y** be sets of parameters that can take values from domains **X** and **Y** respectively. Assume that **x** and **y** have no parameters in common.

Consider a predicate with parameters from **x** and **y**. For example, the expression $(x_1=y+1)$ **or** $(x_2=y)$ is a predicate. For every value pair in $\mathbf{X}\times\mathbf{Y}$, a predicate evaluates to either True or False. The predicate specifies a set of value pairs in $\mathbf{X}\times\mathbf{Y}$, namely those where the predicate evaluates to True. The predicate is said to be *enabled* for any value of **x** if there is a value of **y** such that the predicate is True for that value pair.

Instead of using algorithmic code, we use predicates to specify input-output relations. For example, given integers x and y, an algorithm that assigns to y the value $x+1$, can be modeled by the predicate $(y=x+1)$. It can also be modeled by the predicate $(y-x=1)$. A nondeterministic algorithm that assigns to y either the value $x+1$ or the value $x-2$ provided that y is positive, can be modeled by the predicate ($y>0$ **and** ($y=x+1$ **or** $y=x+2$)). (We use **and** and **or** to denote logical conjunction and disjunction respectively.) Note that within a predicate there is no distinction between the input and output parameters because there are no assignment statements as found in algorithmic code.

We now introduce some notation. We use e(**x**;**y**) to denote a predicate named e with parameters from **x** and **y**. Note that the input parameters are listed before the semicolon. The output parameters are listed after the semicolon. For any given values of **x** and **y**, we shall also use e(**x**;**y**) to denote the value of the predicate.

We now describe how to use predicates to specify events. Since a system event may cause certain changes to the values of the system state variables, it corresponds to a collection of input-output value pairs where the input and output parameters are the values of the system variables before and after the event occurrence. Applying the above notation, we specify a system event e by a predicate e(**v**; **v'**), where the parameter **v** denotes the value of the global state vector immediately before the event occurrence and the parameter **v'** denotes the value of the global state vector immediately after the event occurrence.

The events of a distributed system usually have structure that we wish to explicitly indicate. First, an event e typically affects only a few components of the global state vector. When specifying such an event e, we will include only these components in its parameter list, and adopt the convention that the missing components are not affected by the event. Second, though an entity can affect the value of a channel state variable, it can do so only in certain ways: namely, sending a message (appending it to the tail of the channel state variable) and receiving a message (removing it from the head of the channel state variable). To incorporate these constraints in the model, we define send

and receive *service primitives* for the channels, and allow entity events to access the channel state variables only through these primitives.

For channel C_i, the send service primitive is $\text{Send}_i(m,\mathbf{z}_i;\mathbf{z}_i")$ which denotes the predicate $(\mathbf{z}_i" = ((m,0),\mathbf{z}_i))$ i.e., append m with an age of 0 to the tail of $\mathbf{z}_i$. Similarly, the receive service primitive is $\text{Rec}_i(\mathbf{z}_i;m,\mathbf{z}_i")$ which denotes the predicate $((\mathbf{z}_i",(m,t))=\mathbf{z}_i)$ i.e., if $\mathbf{z}_i$ has m at its head, then irrespective of m's age, remove it and pass it out. When these service primitives are included in the predicates of the entity events, the formal message parameter m is replaced by the actual message sent or received.

Thus the events of entity P_i have the structures indicated below:

(1) for each message type $(q,\mathbf{f})$ sent by P_i,
$\text{Send}\_q\ (\mathbf{v}_i,\mathbf{z}_i;\mathbf{v}_i",\mathbf{z}_i") \equiv e_{sq}(\mathbf{v}_i;\mathbf{v}_i",\mathbf{f})$ **and** $\text{Send}_i((q,\mathbf{f}),\mathbf{z}_i;\mathbf{z}_i")$;

(2) for each message type $(q,\mathbf{f})$ received by P_i from Channel C_j,
$\text{Rec}\_q(\mathbf{v}_i,\mathbf{z}_j;\mathbf{v}_i",\mathbf{z}_j") \equiv e_{rq}(\mathbf{v}_i,\mathbf{f};\mathbf{v}_i")$ **and** $\text{Rec}_j(\mathbf{z}_j;(q,\mathbf{f}),\mathbf{z}_j")$;

(3) internal events of the form $e_{int}(\mathbf{v}_i;\mathbf{v}_i")$ used to model timeouts, etc.

where the predicates e_{sq}, e_{rq} and e_{int} are to be specified for a particular protocol system.

For each channel C_i, the channel behavior (loss, reordering, etc.) is specified by a predicate called $\text{ChannelError}(\mathbf{z}_i;\mathbf{z}_i")$.

The time events are completely specified by the accuracy and time axioms in a straightforward manner. (See Part 1 of [8] and see the protocol example in Section 4 for an illustration.)

4. A TIME-DEPENDENT DATA TRANSFER PROTOCOL

We now present a data transfer protocol that reliably transfers data blocks from entity P_1 to P_2 using channels C_1 and C_2 (see Fig. 1), where we allow the bounded-delay channels to lose, duplicate and reorder messages in transit.

Let DataSet be the set of data blocks that can be sent in this protocol. P_1 sends messages of type (D,data,ns) where D identifies the name of the message type, data is a data block from DataSet, and ns is a send sequence number. P_2 sends messages of type (ACK,nr) where nr is a receive sequence number. In this protocol, ns and nr are restricted to the values of 0 and 1. When P_2 receives a (D,data,ns) message, if ns equals the next expected sequence number then the data block is passed on to the destination, else it is ignored. In either case, P_2 sends an (ACK,nr).

In order that the data transfer be reliable in spite of the channel errors, P_1 must ensure before sending a *new* data block that MaxDelay_1 time has elapsed since the last

data block was sent, and $MaxDelay_2$ time has elapsed since receiving the last acknowledgement to a previously unacknowledged data block. P_1 will repeatedly transmit this data block until it is acknowledged. However, neither of the time constraints apply to the retransmission of a previously sent but unacknowledged data block. The time to wait before a retransmission should be chosen on the basis of performance goals and the probability distributions of channel delays, channel loss, etc. Here we see a system with two different types of time constraints: one necessary for logical correctness and one concerned only with performance. In other protocols, the separation is not always so clear.

We now list the state variables and events of the entities. (Below, $MDelay_i = (1 + \epsilon_1) \times MaxDelay_i$ for i=1 and 2.)

Variables of P_1

Source: array[0..∞] of DataSet; {auxiliary history variable initialized to the sequence of data blocks to be sent to P_2}

s: 0..∞; {Source[s] is the data block in the next D message to be sent; an auxiliary variable}

vs: 0..1; {sequence number to be used in the next D message to be sent}

ws: 0..1; {sequence number used in the last D message sent}

DTimer,DTimerG: (0,1,2,..); {time elapsed since the last D message sent; DTimer is a local time variable and DTimerG is the global time variable associated with DTimer}

ACKTimer,ACKTimerG: (0,1,2,..); {time elapsed since reception of the last ACK message that caused progress; ACKTimerG is the global time variable associated with ACKTimer}

Let $\mathbf{v}_1$ denote a list of the above variables. The initial condition of P_1 is given by the following predicate.

$Initial_1(\mathbf{v}_1) \equiv$ (s=vs=0 **and** ws=1 **and** DTimer > MDelay1 **and** ACKTimer > MDelay2 **and** DTimerG = DTimer **and** ACKTimerG = ACKTimer)

Events of P_1

P_1 has two events, one for sending messages of type D and the other for receiving messages of type ACK. These events are specified below.

1. Send_D ($\mathbf{v}_1$,$\mathbf{z}_1$;$\mathbf{v}_1$",$\mathbf{z}_1$")
   ≡ ((ws=vs) {Retransmit old data}
   or (ws≠vs **and** Dtimer > MDelay1 **and** ACKTimer > MDelay2)) {Transmit new data}
   and $Send_1$((D,Source[s],vs),$\mathbf{z}_1$;$\mathbf{z}_1$") {Send message}

and s"=s **and** vs"=vs **and** ws"=vs {Update state vector}
and DTimer"=0 **and** ACKTimer"=ACKTimer **and** Source"=Source
and DTimerG"=0 **and** ACKTimerG"=ACKTimerG

2. Rec_ACK($\mathbf{v}_1$,$\mathbf{z}_2$;$\mathbf{v}_1$",$\mathbf{z}_2$")
≡ Rec$_2$ ($\mathbf{z}_2$; (ACK,nr), $\mathbf{z}_2$") {Receive nr}
and ((nr=vs $\oplus$ 1 **and** vs=ws {Outstanding data acknowledged}
and s"=s+1 **and** vs"=nr **and** ACKTimer"=0 **and** ACKTimerG"=0)
or (nr=vs **and** s"=s **and** vs"=vs {Old acknowledgement}
and ACKTimer"=ACKTimer
and ACKTimerG" = ACKTimerG))
and ws"=ws **and** DTimer"=DTimer **and** Source"=Source
and DTimerG" = DTimerG

In the above, $\oplus$ denotes addition modulo 2.

Variables of $\mathbf{P}_2$

Sink: array [0..∞] of DataSet; {auxiliary history variable that records the sequence of data blocks passed on to the destination}

r: 0..∞; {the next data block received in sequence will be saved in Sink[r]; an auxiliary variable}

vr: 0..1; {sequence number of next expected data block}

SendACK: Boolean; {True iff a received D message has not been acknowledged}

Let $\mathbf{v}_2$ denote a list of the above variables. The initial condition of P_2 is given by the following predicate.

Initial$_2$($\mathbf{v}_2$) ≡ (r=vr=0 **and** SendACK = False)

Events of $\mathbf{P}_2$

1. Send_ACK ($\mathbf{v}_2$,$\mathbf{z}_2$;$\mathbf{v}_2$",$\mathbf{z}_2$")
≡ (SendACK = True) **and** Send$_2$ ((ACK,vr), $\mathbf{z}_2$;$\mathbf{z}_2$")
and Sink"=Sink **and** r"=r **and** vr"=vr **and** SendACK"=False
2. Rec_D ($\mathbf{v}_2$,$\mathbf{z}_1$;$\mathbf{v}_2$",$\mathbf{z}_1$")
≡ Rec$_1$ ($\mathbf{z}_1$;(D,data,ns), $\mathbf{z}_1$")
and ((ns=vr **and** Sink"[r]=data **and** r"=r+1 **and** vr"=vr $\oplus$ 1) {in-sequence data}
or (ns ≠ vr **and** Sink"=Sink **and** r"=r **and** vr"=vr)) {out-of-sequence data}
and SendACK" = True

Other events

The channel events of C_i are specified by the predicate ChannelError($\mathbf{z}_i$,$\mathbf{z}_i$") that allows all possible losses, duplications and reorderings of messages in the channel.

The *local time event* for the local ticker at P_1 is specified by

AccuracyAxiom$_1(\eta_1+1,\eta)$ {if local tick will not violate accuracy axiom}
and η_1"$=\eta_1+1$ **and** DTimer"=DTimer+1 **and** ACKTimer"=ACKTimer+1
{then age η_1 and all time variables driven by local ticker}

The *global time event* is specified by

AccuracyAxiom$_1(\eta_1,\eta+1)$ **and** TimeAxiom$_1$(next($\mathbf{z}_1$)) **and** TimeAxiom$_2$(next($\mathbf{z}_2$))
and η"$=\eta+1$ **and** $\mathbf{z}_1$"=next($\mathbf{z}_1$) **and** $\mathbf{z}_2$"=next($\mathbf{z}_2$)
and DTimerG" = DTimerG + 1 **and** ACKTimerG" = ACKTimerG + 1

where next($\mathbf{z}_i$) is $\mathbf{z}_i$ with all ages in it incremented by 1.

System initial condition

The initial condition of the system is given by the following predicate

Initial($\mathbf{v}$) $\equiv$ Initial$_1(\mathbf{v}_1)$ **and** Initial$_2(\mathbf{v}_2)$ **and** $\mathbf{z}_1$ is empty **and** $\mathbf{z}_2$ is empty.

5. PROVING SAFETY AND LIVENESS PROPERTIES

The set of all possible value assignments to the system state variables define the global state space of the protocol system. Those global states that satisfy Initial($\mathbf{v}$) are referred to as *initial global states.* Each event specifies a set of transitions between global states; each transition is from a global state where the event is enabled to a global state that can result from changes to the state variable values. A global state that can be reached from an initial state via a sequence of event transitions is referred to as a *reachable global state.* The graph whose nodes are the reachable global states and whose arcs are the event transitions is referred to as the *reachability graph* of the system. A realization of the protocol system behavior is represented by some path in the reachability graph starting from an initial state.

Safety properties

A safety property of the protocol system states relationships between values of the system state variables. It can be represented by a predicate in the variables of the global state vector $\mathbf{v}$. An example of a safety property involving two integer state variables x and y is $(x \leq y \leq x + 1)$. A safety property A($\mathbf{v}$) holds for the system if it holds at every reachable state. Such a property is said to be *invariant.* We now present the inference rule to prove the invariance of a predicate A($\mathbf{v}$).

Inference rule for safety. If B(**v**) is invariant and A(**v**) satisfies

(i) ((Initial(**v**) $\Rightarrow$ A(**v**)) **and**

(ii) ($\forall$ e(**v**;**v**") : B(**v**) **and** A(**v**) **and** e(**v**;**v**") $\Rightarrow$ A(**v**")) **and**

(iii) (A(**v**) $\Rightarrow$ A_0(**v**))),

then A_0(**v**) is invariant.

The validity of the rule is obvious from the following. From (ii) we know that at every global state g where A(g) = B(g) = True, every enabled event e takes the protocol system to a state h where A(h) = True. Because B(**v**) is invariant, B(h) = True. Hence, once the protocol is in a state where A(**v**) and B(**v**) hold, all future states also satisfy the two predicates. From (i), we know that any initial state satisfies A(**v**) (and B(**v**) because of its invariance). Hence all reachable states satisfy A(**v**). Because A_0(**v**) is implied by A(**v**) (from (iii)), we know that all reachable states also satisfy A_0(**v**).

Note that the inference rule is quite simple because of our use of predicates to define events. Typically, we are given A_0(**v**) as a service requirement, and are asked to verify that it is invariant. The above rule does not explain how to obtain A(**v**) once we are given A_0(**v**) and B(**v**). It just states sufficient conditions that an A(**v**) must satisfy in order for us to conclude that A_0(**v**) is invariant. Generating A(**v**) given A_0(**v**) and B(**v**) can be done using the method of weakest preconditions [11] or symbolic execution. (In general, this is a nontrivial task analogous to generating loop invariants in program verification.)

If in the above rule, B(**v**)= True, then A(**v**) is said to be *inductively complete.* The time events have been defined so that each of the time and accuracy axioms is inductively complete (and hence invariant) [8].

Liveness properties

A liveness property of the protocol system states relationships that values of the system variables eventually satisfy. An example of a liveness property involving integer state variables x and y is as follows: during the course of the protocol operation, if x does not increase without bounds then y will increase without bounds. Note that a liveness property is not a property of each reachable state (and cannot be stated as a predicate in the variables of **v**). Rather it is a property of the paths in the reachability graph.

Our method of stating liveness properties is based on specifying inductive properties of *bounded-length* paths in the reachability graph. First, we assume that any implementation of the protocol system is *fair*, by which we mean the following: any event that is enabled infinitely often will eventually occur. Next, we have the following definition.

Given predicates A(**v**) and B(**v**), we say that A(**v**) *leads-next-to* B(**v**) if for every reachable global state g where A(g) = True, the following holds: for every event enabled in state g, its occurrence takes the system to a state h where either A(h) = True or B(h) = True, and there is at least one event enabled in state g whose occurrence can take the system to a state h where B(h) = True.

In any system implementation that is fair, if A(**v**) leads-next-to B(**v**) then on any outgoing path from a reachable state g where A(g) = True, the system will eventually reach a state h where B(h) = True. We now present the inference rule used to establish the leads-next-to property.

Inference rule for liveness. If I(**v**) is invariant, and A(**v**) and B(**v**) satisfy

(i) ($\forall$ e(**v**;**v"**): (I(**v**) **and** A(**v**) **and** e(**v**;**v"**)) $\Rightarrow$ B(**v"**) **or** A(**v"**)) **and**

(ii) ((A(**v**) **and** I(**v**)) $\Rightarrow$ $\exists$ e(**v**;**v"**): (e(**v**;**v"**) = True
and (I(**v**) **and** A(**v**) **and** e(**v**;**v"**)) $\Rightarrow$ B(**v"**))),

then A(**v**) leads-next-to B(**v**).

This inference rule is very similar to the definition of *leads-next-to*, except that it allows us to utilize any safety property I(**v**) that is known. Next, we extend the leads-next-to definition.

Given assertions A(**v**) and B(**v**), we say that A(**v**) *leads-to* B(**v**) if for some specified integer $n \geq 1$ the following holds: (A(**v**) leads-next-to (B(**v**) **or** C_1(**v**))) **and** (C_1(**v**) leads-next-to (B(**v**) **or** C_2(**v**))) **and** ... **and** (C_{n-1}(**v**) leads-next-to (B(**v**) **or** C_n(**v**))) **and** (C_n(**v**) leads-next-to B(**v**)).

Using the leads-to construct, the liveness property example above can be specified by the inductive statement ($\forall$ n,m: ($x = n$ **and** $y = m$) leads-to ($x > n$ **or** $y > m$)). There are other ways to specify liveness properties. For example, one could state liveness properties as predicates on the set of paths in the reachability graph, but that would be too cumbersome. Temporal logic offers another way to specify liveness properties. The liveness property example would be specified in temporal logic by (**not**($\forall n$: $\Diamond$ $x > n$) $\Rightarrow$ ($\forall m$: $\Diamond$ $y > m$)), where the operator $\Diamond$ is to be read as "eventually." It has been our experience that to prove temporal logic statements, it is often necessary to rephrase them into inductive statements of the leads-to type. Hence, by stating our liveness properties directly using the leads-to construct, we avoid some unneccessary overhead.

6. SAFETY, LIVENESS AND REAL-TIME PROPERTIES OF PROTOCOL EXAMPLE

Safety specification and verification

For the protocol in Section 4, we would like to prove that the following safety property is invariant:

A1. (a) Source[i] = Sink[i] for $0 \leq i < r$;
(b) $0 \leq s \leq r \leq s + 1$.

A1(a) states that the sequence of data blocks placed in Sink is a prefix of the data blocks in Source. A1(b) states that at most one data block is outstanding (i.e., sent but not acknowledged).

A1 is invariant because A1 **and** A2 **and** A3 **and** A4 **and** A5 is inductively complete (this can be checked easily by applying the inference rule for safety).

A2. (vs = s mod 2) **and** (vr = r mod 2)

A3. ($\forall$(m,t) in $\mathbf{z}_1$: m = (D,Source[s],vs) **and** vs=ws **and** (r=s **or** r=s+1) **and** t $\geq$ DTimerG)
or ($\forall$(m,t) in $\mathbf{z}_1$: m = (D,Source[s-1],vs $\oslash$ 1) **and** vs=ws $\oslash$ 1 **and** r=s **and** t $\geq$ DTimerG)

A4. SendAck = True $\Rightarrow$ r=s **or** (vs=ws **and** r=s+1)

A5. $\forall$(m,t) in $\mathbf{z}_2$: (m=(ACK,vr) **and** (r=s **or** (vs=ws **and** r=s+1)))
or (m=(ACK,vr $\oslash$ 1) **and** ((r=s **and** t $\geq$ ACKTimerG **and** vs $\neq$ ws)
or (r=s+1 **and** vs=ws)))

where $\oslash$ denotes subtraction modulo 2. The above can be shown to be inductively complete by a straightforward application of the inference rule for safety. (See Part 1 of [8].)

Liveness specification and verification

For this protocol, we would like to prove the following: if the channels do not continually lose messages, then s and r will grow without bound.

To specify and verify this formally in our model, we define the auxiliary variables LCount1 and LCount2. LCount1 counts the number of times that a (D,Source[n],n mod 2) message in C_1 has been lost since the previous reception of such a message at P_2. Formally, LCount1=0 initially; whenever a loss event of C_1 deletes a (D,Source[n],n mod 2) message in C_1, LCount1 is incremented by 1; whenever a (D,Source[n],n mod 2) message is received at P_2, LCount1 is reset to 0. LCount2 is similarly specified, except that (D,Source[n],n mod 2), C_1 and P_2 are replaced by (ACK, (n+1) mod 2), C_2 and P_1 respectively.

The desired liveness property is then stated as follows: For any non-negative integer n

L1. (s=r=n **and** LCount1=m1) leads-to ((s=n **and** r=n+1)
or (s=r=n **and** LCount1 > m1))

L2. (s=n **and** r=n+1 **and** LCount1=m1 **and** LCount2=m2) leads-to
((s=r=n+1) **or** ((s=n **and** r=n+1) **and**
((LCount2 > m2) **or** (LCount2 ≥ m2 **and** LCount1 > m1)))).

L1 assures us that from any state where s=r=n, we will get to a state where s=n and r=n+1, provided that LCount1 does not grow without bound. L2 assures us that the system will then get to a state where s=r=n+1, provided that neither LCount1 nor LCount2 grows without bound. Thus, assuming the desired channel behavior, L1 and L2 allow us to say that s and r will grow without bound.

The above liveness (leads-to) property has been verified for the data transfer protocol in Section 4. The verification is very short and may be found in Part 1 of [8].

Real-time specification and verification

To make our data transfer protocol more realistic, we include the following real-time behavior into its model.

First, entity P_2 will send an ACK message within a specified time interval (MaxResponseTime) of receiving a D message. Second, entity P_1 will retransmit a given data block Source[n] at most MaxRetryCount times. Let MRoundTripDelay = MDelay1 + MDelay2 + MaxResponseTime $\times$ $(1+\epsilon_1+\epsilon_2)$. If after sending Source[n] for MaxRetryCount times, P_1 does not receive an (ACK, (n+1) mod 2) within MRoundTripDelay of the last send, it assumes that the channels C_1 and C_2 are bad and aborts the connection (enters a state called RESET).

For this more realistic model, we would like to prove that if P_1 has reset, then indeed over a time period T (=MRoundTripDelay $\times$ MaxRetryCount), more than MaxRetryCount messages sent by P_1 and P_2 have been lost by C_1 and C_2 collectively.

To formally state this real-time specification, define the following auxiliary variables:

MessagesSent1: sequence of (m,t) pairs where m is a message sent by P_1 and t denotes the time at which m was sent; updated whenever P_1 does a send.

MessagesSent2: as above but for P_2.

SCount1: Number of times (D,Source[n], n mod 2) was sent into C_1 but did not get

received at P_2 within MaxDelay1 of sending. SCount1 is incremented by 1 whenever a global tick occurs and ((D,Source[n], n mod 2), t) is in MessagesSent1 and r=n and η = t + MaxDelay1. (Recall that η is the global ticker's count.) SCount1 is set to 0 whenever P_1 gets an ACK that causes progress.

SCount2: Same as SCount1, except that (D,Source[n], n mod 2), C_1 and P_2 are replaced by (ACK, n+1 mod 2), C_2 and P_1.

ReferenceTime: Value of η when P_1 last got an ACK that caused progress.

ResetTime: Value of η when P_1 last reset.

With all these auxiliary variables, the real-time specification can be stated as

P_1 at RESET $\Rightarrow$ (ResetTime - ReferenceTime) $\leq$ T
and SCount1 + SCount2 $\geq$ MaxRetryCount.

Notice that this real-time specification is a safety assertion and not a liveness assertion requiring the leads-to operator. Its verification can be found in Part 1 of [8].

Acknowledgement

The work reported in this article was supported by a grant from the National Science Foundation (Grant No. ECS 83-04734).

REFERENCES

[1] International Standards Organization, "Data Communication--High-level Data Link Control Procedures--Frame Structure," Ref. No. ISO 3309, Second Edition, 1979. "Data Communications--HDLC Procedures--Elements of Procedures," Ref. No. ISO 4335, First Edition, 1979. International Standards Organization, Geneva, Switzerland.

[2] IEEE Project 802 Local Area Network Standards, "CSMA/CD Access Method and Physical Layer Specifications," Draft IEEE Standard 802.3, Revision D, December 1982.

[3] Postel, J. (ed.), "DOD Standard Transmission Control Protocol," Defense Advanced Research Projects Agency, Information Processing Techniques Office, RFC 761, IEN 129, January 1980; in *ACM Computer Communication Review*, Vol. 10, No. 4, October 1980, pp. 52-132.

[4] Clark, D. D., "Protocol Implementation: Practical Considerations," ACM SIGCOMM'83 Tutorial, University of Texas at Austin, March 7, 1983.

[5] Shankar, A. U. and S. S. Lam, "An HDLC Protocol Specification and its Verification Using Image Protocols," *ACM Trans. on Computer Systems*, Vol. 1, No. 4, November 1983, pp. 331-368.

[6] Owicki, S. and L. Lamport, "Proving Liveness Properties of Concurrent Programs," *ACM TOPLAS*, Vol. 4, No. 3, July 1982, pp. 455-495.

[7] Lam, S. S. and A. U. Shankar, "Protocol Verification via Projections," *IEEE Trans. on Software Eng.*, Vol. SE-10, No. 4, July 1984, pp. 325-342.

[8] Shankar, A. U. and S. S. Lam, "Specification and Verification of Communication Networks, Part 1: Safety, Liveness and Real-time Properties; Part 2: Method of Projections," Tech. Rep. in preparation, Dept. of Computer Sciences, Univ. of Texas at Austin, 1984.

[9] Lamport, L, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, Vol. 21, No. 7, July 1978, pp. 558-565.

[10] Sloan, L., "Mechanisms that Enforce Bounds on Packet Lifetimes," *ACM Trans. Comput. Syst.*, Vol. 1, No. 4, Nov. 1983, pp. 311-330.

[11] Dijkstra, E. W., *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, N.J., 1976.