Protocol Specification, Testing, and Verification, IV
Y. Yemini, R. Strom, and S. Yemini (Editors)
Elsevier Science Publishers B.V. (North-Holland)

SPECIFICATION AND VERIFICATION OF TIME-DEPENDENT COMMUNICATION PROTOCOLS

A. Udaya Shankar
Department of Computer Science
University of Maryland
College Park, Maryland 20742

Simon S. Lam
Department of Computer Sciences
University of Texas
Austin, Texas 78712

ABSTRACT

Real-life communication protocol systems are invariably large time-dependent distributed systems whose correct functioning depends upon time relationships between system event occurrences. We present a model for specifying and verifying such time-dependent systems. Each component (protocol entity or channel) of the protocol system is specified by a set of state variables and a set of events. Each 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. 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. 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. We have specified and verified the protocol's safety, liveness and real-time properties.

1. INTRODUCTION

Real-life protocol systems are typically large time-dependent systems. By *time-dependent*, we mean that they display real-time behavior (in the form of time constraints between system event occurrences) that is crucial to their logical correctness as well as their performance [16, 17]. For example, an entity must send an acknowledgement to a message within a specified response time of receiving that message; the time duration that a message resides in a channel is less than a specified maximum channel delay and/or larger than a specified minimum delay, the time between an entity crashing and its re-entry into the protocol system must satisfy certain bounds, etc. Examples of such protocols include the High-level Data Link Control (HDLC) protocol, the Transmission Control Protocol (TCP), Carrier Sense Multiple Access (CSMA), etc. [8, 15, 7, 3]. (In each of these cases, we are

referring to the protocol as defined in its reference manuals.)

We describe an event-driven process model that is suitable for *specifying and verifying* such distributed systems, both time-dependent and time-independent. In this model, clocks are not coupled and they can tick at any rate within some specified error bounds. This real-time modeling is an extension of results initially presented in [16]. We have applied this model to the analysis of several nontrivial protocol examples: a version of the High-level Data Link Control (HDLC) protocol [17], the physical clock synchronization protocol of Lamport [13], and a transport-level protocol for reliable data transfer over bounded-delay channels that can lose, reorder and duplicate messages in transit (using cyclic sequence numbers, timers and timeouts). Only the last example appears in this paper.

Summary of the protocol model features

Our protocol system model differs significantly from other models in regards to both specification and verification aspects. Each component (protocol entity or channel) is specified by a set of *state variables* and a set of *events*. The events of a component manipulate the values of state variables local to the component and transfer messages to adjacent components. Each event is specified 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. There is *no* algorithmic code in our model.

What we have is a compromise between implementation-dependent features (the state variables) and implementation-independent features (the use of predicates to specify events). It provides a very convenient and uniform specification of the safety, liveness, and even performance properties of the system. It allows for very simple inference rules for safety and liveness properties, system specifications that can be directly substituted into assertions used within proofs, and it simplifies our modeling of time measures and the application of projections. For example, we do not use any special notation (such as temporal logic [14]) to express liveness properties of *unbounded-length* paths in a system's reachability graph. Instead, we describe such liveness properties in terms of inductive properties of bounded-length paths. Real-time system properties can usually be expressed as safety assertions.

Summary of the real-time modeling

We model the real-time behavior of systems by using discrete-valued time variables to measure the elapse of physical (real) time, and time events to age the time variables. By imposing certain conditions, referred to as *accuracy axioms*, on the occurrence of 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 all types of time constraints between system events. With accuracy and time axioms we can specify and verify the time-dependent behavior of distributed systems. We also formalize the notion of *feasible* time constraints; i.e., time constraints that can be realistically guaranteed by a component without any cooperation from the rest of the system.

One consequence of the time-dependent behavior of real-life communication network protocols is that 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 such as "eventually (i.e., within a finite but unbounded time duration) a data block will be transferred" is not realistic. More appropriate is a real-time specification 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." With our model of real-time behavior, such real-time specifications can usually be stated as safety assertions. We model both types of progress specifications in our data transfer protocol example. In each case, there is a compact verification of the specification [18].

Summary of the rest of this paper

In Section 2, we present more details of our real-time modeling. In Section 3, we specify our protocol system model. In Section 4, we characterize safety and liveness properties for the protocol model, and present the inference rules used to establish these properties. In Section 5, a time-dependent data transfer protocol example is specified. In Section 6, we present safety, liveness and real-time properties for the example protocol.

2. MODELING MEASURES OF TIME

The elapse of physical time in any distributed system component is indicated by devices, such as crystal oscillators, that issue "ticks" at (almost) regular time intervals. We refer to such devices as *local tickers*. Systems typically employ counters (i.e., clocks and timers) to accumulate the number of elapsed ticks generated by a local ticker since the occurrence of some system event. We refer to such counters as *time variables*.

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. In a distributed system, we would insist that all the time variables driven by a local ticker must lie within a single component of the distributed system. In addition to being aged, a time variable can be *reset* to some value by an event of its component, thereby indicating the time elapsed since that event occurrence.

In order to keep local tickers within specified drifts, we include in our model a hypothetical ticker, referred to as the *global ticker*, that is assumed to tick at an absolutely constant rate. 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 implemented clocks or timers, and can never be reset by any system event. The local and global time events are constrained so that each local ticker i satisfies the following *accuracy axiom* (below, the notation $\eta(a)$ refers to the value of η at instant a):

AccuracyAxiom$_i(\eta_i,\eta)$: For all instants a and b where b is later than a,
$|\ (\eta_i(b) - \eta_i(a)) - (\eta(b) - \eta(a))| \leq \max(1, \epsilon_i(\eta(b)-\eta(a)))$.

(This is a discrete analog of $|1 - \frac{d\eta_i}{d\eta}| \leq \epsilon_i$ in [13].)

Recall that time variables can 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.

Time events will be allowed to occur only if their occurrence will not violate any accuracy or time axioms. This modeling of real-time behavior is valid provided that the time events do not get deadlocked. We have shown that the tickers will continue to accumulate ticks if the accuracy and time axioms correspond to *feasible constraints* [18].

3. DISTRIBUTED SYSTEM MODEL

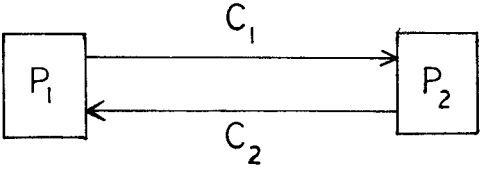

Fig. 1. Network configuration of protocol example in Section 5.

For the sake of brevity, the special configuration and channel behavior of the protocol example in Section 5 are considered. (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 [15, 19]). (For arbitrary network configurations with various types of channels, see [18].)

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 $n \geq 0$ and f_i is a parameter ranging over a specified set of values. (Henceforth, each variable and parameter used in our model is assumed to take values from a specified domain of values.) Let Q_i be the set of message types sent by P_i and received by P_j $(j \neq i)$.

Let $\mathbf{v}_i$ be the set of state variables of P_i. $\mathbf{v}_i$ can have auxiliary variables used for verification only and time variables used for modeling time constraints. 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. A time variable v in $\mathbf{v}_i$ may be constrained by a time axiom.

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 describe our use of predicates to specify relations between sets of input and output parameters. Let $\mathbf{x}$ and $\mathbf{y}$ be sets of parameters with no parameters in common. The notation e($\mathbf{x}$;$\mathbf{y}$) denotes a predicate named e which involves parameters in $\mathbf{x}$ and $\mathbf{y}$. The parameters before the semicolon (i.e., $\mathbf{x}$) are referred to as *input parameters*; the parameters after the semicolon (i.e., $\mathbf{y}$) are *output parameters*. e($\mathbf{x}$;$\mathbf{y}$) specifies the set of input-output relations {($\mathbf{s}$,$\mathbf{r}$):e($\mathbf{s}$;$\mathbf{r}$)=True}. The predicate e($\mathbf{x}$;$\mathbf{y}$) is said to *enabled* for any value of $\mathbf{x}$ such that there is a value of $\mathbf{y}$ for which e($\mathbf{x}$;$\mathbf{y}$)=True; Note that within the body of the predicate e($\mathbf{x}$;$\mathbf{y}$) there is no

distinction between the input and output parameters; there are no assignment statements either. There is no requirement that each parameter in **x** and **y** be used in the body of e(**x**;**y**). If a parameter y in **y** is not explicitly equated to a value in the body of e(**x**;**y**), then we take the convention that y can assume any value in its domain.

To form an analogy with programming language concepts, we can consider e(**x**;**y**) to model a (perhaps non-deterministic) procedure e with formal input parameters **x** and formal output parameters **y** as follows: procedure e can be executed only when the value of **x** is such that e(**x**;**y**) is enabled; then, the execution of procedure e causes the output parameters **y** to be assigned a value such that e(**x**;**y**)=True. For example, given integers x and y, a procedure e that assigns to y the value $x+1$, can be modeled by $e(x;y) \equiv (y=x+1)$, or by $e(x;y) \equiv (y-x=1)$. A non-deterministic procedure e that assigns to y either the value $x+1$ or the value $x+2$ provided that y is positive, can be modeled by $e(x;y) \equiv (y>0 \ \& \ (y=x+1 \vee y=x+2))$. (We use the symbols & and $\vee$ to denote logical conjunction and disjunction respectively.)

We now describe how events are specified. A system event e corresponds to a set of transitions in the global state space, and is specified 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. Typically, an event e involves only a few components of the global state vector; when specifying the event e, we will include only those components in its parameter list.

For each channel C_i, the channel behavior (loss, reordering etc.) can be specified by a predicate to be called ChannelError ($\mathbf{z}_i$; $\mathbf{z}_i$"). In addition, there are two service primitives $\text{Send}_i(m)$ and $\text{Rec}_i(m)$, where m is a message parameter. Formally, $\text{Send}_i(m)$ is ($\mathbf{z}_i$"=($(m,0),\mathbf{z}_i$)) i.e., append m with an age of 0 to the tail of $\mathbf{z}_i$. Similarly, $\text{Rec}_i(m)$ is ($(\mathbf{z}_i$",(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.

Entity P_i has the following events:

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

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

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

The time events are completely specified by the accuracy and time axioms in a straightforward manner. (See the protocol example in Section 5 for an illustration.)

4. PROOF RULES FOR SAFETY AND LIVENESS

The state variables and events of the system model define the reachability graph of the model. The reachability graph captures all the system properties. Specifically, safety assertions are predicates on the set of global states, and liveness assertions are predicates on sequences of global states. We now present the inference rules used to establish safety and liveness properties for a system specified in our model. The inference rules are quite simple because of our use of predicates to define the events.

An assertion A(**v**) is *invariant* if it holds at every reachable global state.

Inference rule for safety. If $B(\mathbf{v})$ is invariant and $A(\mathbf{v})$ satisfies
$((\text{Initial}(\mathbf{v}) \Rightarrow A(\mathbf{v}))$
$\&\ (\forall\ e(\mathbf{v};\mathbf{v}''): B(\mathbf{v})\ \&\ A(\mathbf{v})\ \&\ e(\mathbf{v};\mathbf{v}'') \Rightarrow A(\mathbf{v}''))$
$\&\ (A(\mathbf{v}) \Rightarrow A_0(\mathbf{v}))),$
then $A_0(\mathbf{v}))$ is invariant.

If in the above rule, $B(\mathbf{v}) = \text{True}$, then $A(\mathbf{v})$ is said to be *inductively complete*. Generating $A(\mathbf{v})$ given $A_0(\mathbf{v})$ and $B(\mathbf{v})$ can be done using the method of weakest preconditions [4] or symbolic execution. The time events have been defined so that every time and accuracy axiom is invariant.

We state liveness properties by specifying inductive properties of *bounded-length* paths in the system's reachability graph (space), or more generally, in a graph obtained by aggregating nodes of the reachability graph. Given assertions $A(\mathbf{v})$ and $B(\mathbf{v})$, we say that $A(\mathbf{v})$ *leads-next-to* $B(\mathbf{v})$ if for every reachable global state $\mathbf{v}$ where $A(\mathbf{v}) = \text{True}$, the following holds: for every enabled event its occurrence takes the system to a state $\mathbf{v}''$ where either $A(\mathbf{v}'') = \text{True}$ or $B(\mathbf{v}'') = \text{True}$, and at least one event $e(\mathbf{v};\mathbf{v}'')$ is enabled whose occurrence can take the system to a state $\mathbf{v}''$ where $B(\mathbf{v}'') = \text{True}$.

This definition guarantees that in any implementation that is fair, on any outgoing path from a reachable state $\mathbf{v}$ where $A(\mathbf{v}) = \text{True}$, we will eventually reach a state $\mathbf{v}''$ where $B(\mathbf{v}'') = \text{True}$. A *fair* implementation is one in which events are scheduled so that any event that is enabled infinitely often is not indefinitely delayed.

Inference rule for liveness. If $I(\mathbf{v})$ is invariant, and $A(\mathbf{v})$ and $B(\mathbf{v})$ satisfy
$(\forall\ e(\mathbf{v};\mathbf{v}''): (I(\mathbf{v})\ \&\ A(\mathbf{v})\ \&\ e(\mathbf{v};\mathbf{v}'')) \Rightarrow B(\mathbf{v}'') \vee A(\mathbf{v}''))$
$\&\ ((A(\mathbf{v})\ \&\ I(\mathbf{v})) \Rightarrow \exists$ enabled $e(\mathbf{v};\mathbf{v}''): (I(\mathbf{v})\ \&\ A(\mathbf{v})\ \&\ e(\mathbf{v};\mathbf{v}'')) \Rightarrow B(\mathbf{v}'')),$
then $A(\mathbf{v})$ leads-next-to $B(\mathbf{v})$.

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

As an illustration of how the *leads-to* relation can be used to specify liveness properties of unbounded-length paths, observe that the temporal logic specification $(\sim(\forall n: \Diamond\ x>n) \Rightarrow (\forall m: \Diamond\ y>m))$ can be specified by $(\forall\ n,m: (x=n\ \&\ y=m)$ leads-to $(x>n \vee y>m))$ where all variables are non-negative integers. For a more interesting example, see the liveness property in Section 6.

5. A TIME-DEPENDENT DATA TRANSFER PROTOCOL

We present herein 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 that is 0 or 1. Successive data blocks are sent with cyclically increasing sequence numbers. P_2 sends messages of type (ACK,nr) where nr is a receive sequence number. 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 behavior, 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 ACK that

acknowledged previously unacknowledged data. Neither of these time constraints apply for any retransmission of a previously sent but unacknowledged data block. The time to wait before retransmitting a previously sent but unacknowledged data block must 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 examples, the separation is not always so clear.

We now list the state variables and events of the entities. (Below, $\text{MDelay}_i = (1+\epsilon_1) \times \text{MaxDelay}_i$ for i=1 and 2.)

Variables of P_1

Source: array[0..∞] of DataSet; {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}

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: (0,1,2,..); {time elapsed since last D message sent}

ACKTimer: (0,1,2,..); {time elapsed since reception of last ACK message that caused progress}

$\text{Initial}_{P_1} \equiv$(s=vs=0 & ws=1 & DTimer > MDelay1 & ACKTimer > MDelay2)

Events of P_1

1. Send_D ($\mathbf{v}_1$,$\mathbf{z}_1$; $\mathbf{v}_1$",$\mathbf{z}_1$")
   = ((ws=vs) {Retransmit old data}
   ∨ (ws≠vs & Dtimer > MDelay1 & ACKTimer > MDelay2)) {Transmit new data}
   & Send_1((D,Source[s],vs)) {Send message}
   & s"=s & vs"=vs & ws"=vs {update state vector}
   & DTimer"=0 & ACKTimer"=ACKTimer & Source"=Source

2. Rec_ACK($\mathbf{v}_1$,$\mathbf{z}_2$; $\mathbf{v}_1$",$\mathbf{z}_2$")
   = Rec_2 ((ACK,nr)) {Receive nr}
   & ((nr=vs⊕ 1 & vs=ws {Outstanding data acknowledged}
   & s"=s+1 & vs"=nr & ACKTimer"=0)
   ∨ (nr=vs & s"=s & vs"=vs & ACKTimer"=ACKTimer)) {Old acknowledgement}
   & ws"=ws & DTimer"=DTimer & Source"=Source

In the above, ⊕ denotes addition modulo 2.

Variables of P_2

Sink: array [0..∞] of DataSet; {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]}

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

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

$\text{Initial}_{P_2} \equiv$ (r=vr=0 & SendACK=False)

Events of P_2

1. Send_ACK ($\mathbf{v}_2,\mathbf{z}_2$; $\mathbf{v}_2$", $\mathbf{z}_2$")
   = (SendACK = True) & Send_2 ((ACK,vr))
   & Sink"=Sink & r"=r & vr"=vr & SendACK"=False

2. Rec_D($\mathbf{v}_2,\mathbf{z}_1$; $\mathbf{v}_2$", $\mathbf{z}_1$")
   = Rec_1((D,data,ns))
   & ((ns=vr & Sink"[r]=data & r"=r+1 & vr"=vr⊕1) {in-sequence data}
   ∨ (ns≠vr & Sink"=Sink & r"=r & vr"=vr)) {out-of-sequence data}
   & 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.

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

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

The *global time event* is specified by

$\text{AccuracyAxiom}_1(\eta_1,\eta+1)$ & $\text{TimeAxiom}_1(\mathit{next}(\mathbf{z}_1))$ & $\text{TimeAxiom}_2(\mathit{next}(\mathbf{z}_2))$
& η"=η+1 & $\mathbf{z}_1$"=$\mathit{next}(\mathbf{z}_1)$ & $\mathbf{z}_2$"=$\mathit{next}(\mathbf{z}_2)$

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

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

Safety specification and verification

For this example protocol, 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 is invariant because A1 & A2 & A3 & A4 & A5 can easily be seen to be inductively complete, where

A2. (vs = s mod 2) & (vr = r mod 2)

A3. (∀ (m,t) in $\mathbf{z}_1$: m = (D,Source[s],vs) & vs=ws & (r=s ∨ r=s+1) & t ≥ DTimer)
∨ (∀ (m,t) in $\mathbf{z}_1$: m = (D,Source[s-1],vs⊖1) & vs=ws⊖1 & r=s & t ≥ DTimer)

A4. SendACK = True => r=s $\vee$ (vs=ws & r=s+1)

A5. $\forall$ (m,t) in z_2: (m=(ACK,vr) & (r=s $\vee$ (vs=ws & r=s+1)))
$\vee$ (m=(ACK,vr$\ominus$1) & ((r=s & t $\geq$ ACKTimer & vs $\neq$ ws)
$\vee$ (r=s+1 & vs=ws)))

where $\ominus$ denotes subtraction modulo 2. A proof that the above is inductively complete may be found in [18].

Liveness specification and verification

For this protocol, we would like to prove the following: if the channels do not continuously 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 the last (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 the last (D,Source[n],n mod 2) message in C_1, LCount1 is incremented by 1; whenever a (D,Source[n],n mod 2) 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 & LCount1=m1) leads-to ((s=n & r=n+1)
$\vee$ (s=r=n & LCount1 > m1))

L2. (s=n & r=n+1 & LCount1=m1 & LCount2=m2) leads-to
((s=r=n+1) $\vee$ ((s=n & r=n+1) & ((LCount2 > m2)
$\vee$ (LCount2 $\geq$ m2 & 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 5. The verification is very short and may be found in [18].

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 MaxRoundTripDelay = MaxDelay1 + MaxDelay2 + MaxResponseTime (1+ η_1+η_2). If after sending Source[n] for MaxRetryCount times, P_1 does not receive an (ACK, (n+1) mod 2) within MaxRoundTripDelay 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 (=MaxRoundTripDelay $\times$ 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.

LCount1: 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. LCount1 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.) LCount1 is set to 0 whenever P_1 gets an ACK that causes progress.

LCount2: Same as LCount1, 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 => (ResetTime - ReferenceTime) $\leq$ T
& LCount1 + LCount2 $\geq$ MaxRetryCount.

Notice that this real-time specification is a safety assertion and not a liveness assertion requiring the leads-to operator. Its verification may be found in [18].

7. CONCLUSION

We have presented an event-driven process model suitable for specifying and verifying large time-dependent protocol systems. 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. Real-time properties are expressed as safety assertions in our model.

The model specification consists entirely of state variables and predicates. There is no algorithmic code in our model. This compromise between implementation-dependent and implementation-independent features provides us with a uniform characterization of safety, liveness and performance properties, and with simple inference rules for safety and liveness properties.

We have applied this model to the analysis of several nontrivial protocol examples: a version of the High-level Data Link Control (HDLC) protocol, the physical clock synchronization protocol of Lamport, and a transport-level protocol for reliable data transfer over bounded-delay channels that can lose, reorder and duplicate messages in transit (using cyclic sequence numbers, timers and timeouts).

REFERENCES

[1] Bochmann, G. V. and Chung, R. J., "A Formalized Specification of HDLC Classes of Procedures," *Conf. Rec. Nat. Telecommun. Conf.*, Los Angeles, December 1977.

[2] Bochmann, G. V., "Finite State Description of Communication Protocols," *Computer Networks*, Vol. 2, 1978.

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

[4] Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[5] DiVito, B. L., "Mechanical Verification of a Data Transport Protocol," *Proc. ACM SIGCOMM '83,* Austin, Texas, March 1983.

[6] Hailpern, B. T. and Owicki, S. S., "Verifying Network Protocols using Temporal Logic," Tech. Rep. 192, Computer Systems Laboratories, Stanford University, June 1980.

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

[8] "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.

[9] Kurose, J., "The Specification and Verification of a Connection Establishment Protocol using Temporal Logic," *Proc. 2nd Int. Workshop on Protocol Specification, Testing and Verification,* Idyllwild, California, May 1982.

[10] Lam, S. S. and Shankar, A. U., "An Illustration of Protocol Projections," *Proc. 2nd Int. Workshop on Protocol Specification, Testing and Verification,* Idyllwild, California, May 1982.

[11] Lam, S. S. and Shankar, A. U., "Verification of Communication Protocols via Protocol Projections," *Proc. INFOCOM'82,* Las Vegas, April 1982.

[12] Lam, S. S. and Shankar, A. U., "Protocol Verification via Projections," Tech. Rep. 207, Dept. of Computer Sciences, Univ. of Texas at Austin, August 1982 (revised September 1983). To appear in *IEEE Trans. on Software Eng.,* July 1984.

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

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

[15] 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.

[16] Shankar, A. U. and Lam, S. S., "On Time-Dependent Communication Protocols and their Projections," *Proc. 2nd Int. Workshop on Protocol Specification, Testing and Verification*, Idyllwild, California, May 1982.

[17] 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.

[18] Shankar, A. U. and Lam, S. S., "Specification and Verification of Communication Networks; Part 1: safety, liveness and real-time properties; Part 2: method of projections," Tech. Rep. 214, Dept. of Computer Sciences, Univ. of Texas at Austin, 1984 (In preparation).

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

[20] Stenning, N. V., "A Data Transfer Protocol," *Computer Networks*, Vol. 1, September 1976.