TIME-DEPENDENT DISTRIBUTED SYSTEMS: PROVING SAFETY, LIVENESS AND REAL-TIME PROPERTIES

A. Udaya Shankar* and Simon S. Lam**

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

TR-85-24 October 1985

---

*Department of Computer Science, University of Maryland, College Park, Maryland 20742. Work supported by National Science Foundation under Grant. No. ECS 85-02113.

**Work supported by National Science Foundation under Grant No. ECS 83-04734.

Table of Contents

Abstract

Most communication protocol systems utilize timers to implement real-time constraints between system event occurrences. Such 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 distributed systems. A distributed system is a network 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.

Each process in our distributed system model 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 to their values immediately after the event occurrence. The predicate embodies specifications of both the event's enabling condition and action. 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.

Measures of time are explicitly included in our model in the form of discrete-valued timer variables and time events that age the timer variables. Time constraints enforced within individual processes are modeled by including timer values in the enabling conditions of system events and time events of those processes. Time constraints between events in remote processes can then be specified and verified as safety assertions. Arbitrary constraints on the enabling conditions of time events can cause them to deadlock. We give sufficient conditions to avoid this possibility. These conditions correspond to time constraints that are implementable within individual processes and they are independent of the accuracy of the process timers.

We have applied our model and inference rules to construct and verify several large time-dependent communication protocols, including a transport-layer protocol of window size N and 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 verified.

1. INTRODUCTION

We consider distributed systems in which individual processes employ timers to enforce time constraints between event occurrences. These time constraints enforced within individual processes can give rise to global precedence relations, including time constraints, between events in remote processes. We refer to a distributed system as *time-dependent* when these global precedence relations are essential to the correct functioning of the system.

Our work has been motivated primarily by communication network protocols which are invariably time-dependent systems [1, 5, 6, 15]. Time-dependent behavior arises naturally in communication networks because errors and failures that occur in one process of the network are usually not communicated explicitly to other processes in the network that may be affected by these errors and failures. In such situations, only by the use of timeouts can a process 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 time-dependent behavior can be quite complex.

We present in this paper an event-driven process model for specifying and verifying distributed systems, both time-dependent and time-independent. We have applied this model to construct and verify several nontrivial communication protocols, including a transport-layer protocol of window size N [20] and a version of the High-level Data Link Control (HDLC) protocol [18]. To illustrate our model, we present below a 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.

1.1 Some features of our model

Each process in the distributed system has 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 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. (Any automated verification or implementation method that handles predicate logic expressions can be used on our system specifications.)

This compromise between implementation-dependent features (the state variables) and implementation-independent features (predicates for specifying events) has advantages. The state variables allow us to give compact specifications of large systems and their logical correctness properties. The predicate specifications of events allow us to directly substitute the events into proofs in predicate logic. By combining these features with the event-driven structure of the system model, we get simple inference rules for safety and liveness properties. Safety properties state that certain relations always

hold between the current values of system variables, irrespective of whether the values change or not. Liveness properties state that the values of system variables will indeed change in a certain manner within a finite time. We find it convenient to convert liveness properties concerning *unbounded-length* paths in a system's reachability graph into inductive liveness properties of *bounded-length* paths.

We use discrete-valued timer variables to measure the elapse of time, and define time events to age these timer variables. By imposing conditions, referred to as *accuracy axioms*, on the time events, we are able to model timers realistically: our timer variables are uncoupled and can tick at any rate within specified error bounds of a given rate. Time constraints enforced within a process of the form "event e *will occur only if* elapsed times satisfy certain bounds" are modeled by including timer variables in the enabling conditions of system events of the process. Time constraints enforced within a process of the form "event e *must occur within* certain elapsed times" are modeled by imposing conditions, referred to as *timer axioms*, on the time event of the process.

The time constraints enforced within individual processes give rise to more general time constraints which depend on the interaction between processes. With timer variables, such time constraints can be specified by safety assertions, and formally verified through the inference rules. We have found that such time constraints are very useful for describing progress in communication network protocols. 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 the assertion of a time constraint such as "progress is achieved within a time duration T provided that the channels have not lost more than n messages in that time duration."

Our distributed system specifications can be implemented by programmers who may not be familiar with system analysis. A correct implementation of any process is possible from the specifications of that process alone, without the implementor having to analyze that process's interactions with other processes. In the case of a time-dependent system, this means that the only time constraints allowed in the specifications are those which can be implemented within individual processes. We give a formal definition of implementable time constraints in Section 3.3. It is the task of the protocol designer or verifier, not the implementor, to establish that these implementable time constraints do indeed give rise to the desired global precedence relations.

Our approach to verification is rigorous: a system property is proved only when it is the result of a sequence of inference rules applications. Our proofs can therefore be checked by automated techniques.

1.2 Summary of this paper

In Section 2, we describe the notation needed in using predicates to specify events. A general transition system is modeled. It is then specialized to a time-independent distributed system model. In Section 3, we describe our modeling of timers and time constraints, and present our time-dependent distributed system model. In Section 4, we present inference rules for verifying safety properties and heuristics to generate the assertions needed to apply the inference rules. In Section 5, a time-dependent data transfer protocol example is constructed and its safety properties verified. In Section 6, we present inference rules for verifying liveness properties. We use them to verify the liveness of the data transfer protocol example. In Section 7, we modify the data transfer protocol example so that it now offers real-time service; i.e., data transfer within a specified response time. This real-time service is specified and then verified. In Section 8, we review our modeling and verification methods with respect to modularity, implementation-independence and high-level specifications, and discuss related works.

2. EVENTS AND THEIR SPECIFICATION

We use the term "predicate" to refer to a well-formed sentence of first-order predicate logic augmented by appropriate mathematics for the variables of the predicate. We use **and** and **or** to denote logical conjunction and disjunction respectively. We use *for all* and *for some* to denote universal and existential quantification respectively. Throughout, we assume that for every variable there is a specified domain of allowed values. We use *for all* x *in* X to denote a universally quantified variable x ranging over the domain X. The corresponding existential declaration is *for some* x *in* X. Where ambiguity may arise, the scope of the quantification is enclosed by square brackets. An example of a predicate is (for some x_1 in integers)[$x_1=x_2+1$ **or** $x_1=x_3$], where x_1, x_2 and x_3 are integer-valued variables; the free variables of this predicate are x_2 and x_3.

Let $\mathbf{x}=(x_1,x_2,...,x_n)$ be a set of variables that can take values from domain $\mathbf{X}$. Let p be a predicate expression with free variables from $\mathbf{x}$. For every value in $\mathbf{X}$, p evaluates to either True or False. p specifies a subset of $\mathbf{X}$, namely those values for which the predicate evaluates to True.

The notation $e(\mathbf{x})\equiv p$ declares that $e(\mathbf{x})$ refers to p; for example, $e(x_2,x_3) \equiv$ (for some x_1 in integers)[$x_1=x_2+1$ **and** $x_1=x_3$]. The expression to the right of "$\equiv$" is referred to as the *body* of $e(\mathbf{x})$. The expression to the left of the "$\equiv$" is referred to as the *header*. We will refer to the free variables in the header as *parameters* of the predicate $e(\mathbf{x})$. It is not neccessary that every parameter in the header must occur in the body (though that would normally be the case). We adopt the convention that any variable found in the body but not in the header is universally quantified, unless otherwise mentioned. For any given value of $\mathbf{x}$, we shall also use $e(\mathbf{x})$ to denote the value

that the predicate evaluates to. Thus, in the above example, e(1,2) is True while e(1,1) is False.

We use the terms "variable" and "parameter" in the mathematical sense, i.e., to denote some value from a domain of values. We use the term "state variable" to refer to a variable in the programming language sense, i.e., to denote both a location where a value may be stored, as well as the stored value. As in the case of variables, we assume that for every state variable there is a specified domain of allowed values. The Pascal-like syntax x:X is often used to indicate that state variable x takes values from domain X.

2.1 Input-output Predicates

An *input-output* predicate is a predicate whose free variables are classified into input parameters and output parameters. The notation e(**x**;**y**) denotes an input-output predicate named e where the input parameters $\mathbf{x}=(x_1,x_2,...,x_n)$ are listed before the semicolon, and the output parameters $\mathbf{y}=(y_1,y_2,...,y_m)$ are listed after the semicolon. Assume there are no parameters in common between **x** and **y**. If **x** has the domain **X** and **y** has the domain **Y**, then e(**x**;**y**) specifies a subset of $\mathbf{X}\times\mathbf{Y}$, namely those value pairs for which the predicate evaluates to True.

We use input-output predicates instead of algorithmic code. For example, given integers x and y, an algorithm that assigns to y the value $x+1$, can be modeled as e(x;y) $\equiv$ ($y=x+1$). The body of e(x;y) can also be (y-x=1). A nondeterministic algorithm that assigns to y either the value $x+1$ or the value x-2 provided that x is positive, can be modeled by e(x;y) $\equiv$ ($x>0$ **and** ($y=x+1$ **or** $y=x+2$)). Note that within the body of an input-output predicate there is no distinction between the input and output parameters because there are no assignment statements as found in algorithmic code. We insist that every output parameter in **y** must occur in the body of e(**x**;**y**). Every input parameter in **x** need not occur in the body of e(**x**;**y**) (although that will normally be the case).

An input-output predicate e(**x**;**y**) is said to be *enabled* for a given value of **x** if there is a value of **y** such that e(**x**;**y**) evaluates to True for that value pair. The *enabling condition* of e(**x**;**y**) is defined to be any predicate in **x** which is True for exactly those values of **x** where e(**x**;**y**) is enabled; e.g., the predicate (for some y in Y)e(**x**;**y**).

An input-output predicate e(**x**;**y**) can be composed from other predicates. For example, it is often very natural to have e(**x**;**y**) $\equiv$ (e_1(**x**) **and** e_2(**x**;**y**)), where e_2(**x**;**y**) is enabled for every value where e_1(**x**) is True. Then, e_1(**x**) is the enabling condition of e(**x**;**y**) and e_2(**x**;**y**) embodies the "action" of e(**x**;**y**). For another example, given input-output

predicates $e_1(\mathbf{x}_1;\mathbf{y}_1)$ and $e_2(\mathbf{x}_2;\mathbf{y}_2)$ where the domain of $\mathbf{y}_1$ is a subset of the domain of $\mathbf{x}_2$, we can construct $e(\mathbf{x}_1;\mathbf{y}_2) \equiv$ (for some $\mathbf{y}_1$) $[e_1(\mathbf{x}_1;\mathbf{y}_1)$ **and** $e_2(\mathbf{y}_1;\mathbf{y}_2)]$. A programming language analogy to this example is the sequential composition of two procedures to form a third procedure.

2.2 Specifying an Event-Driven System

We model a general event-driven system by a set of state variables whose values indicate the system state, a set of events that cause changes to the state variable values, and a set of initial conditions on the state variables.

Let $\mathbf{v}=(v_1,v_2,...,v_n)$ denote the set of state variables of the system. $\mathbf{v}$ is also referred to as the *state vector*. The domain of $\mathbf{v}$ is the *system state space*. In addition to state variables needed to model the system, $\mathbf{v}$ can contain auxiliary state variables needed for verification purposes only.

The initial conditions are specified by a predicate Initial($\mathbf{v}$). Any value of $\mathbf{v}$ that satisfies Initial($\mathbf{v}$) is an allowed *initial state* of the system.

Let $e_1,e_2,...,e_m$ be the set of events of the system. Each event e can occur only when the state vector $\mathbf{v}$ has certain values. Its occurrence causes the state vector $\mathbf{v}$ to assume a new value. Thus, the event e corresponds to a collection of input-output value pairs where the input and output values are the values of the state vector before and after the event occurrence. Applying our notation for input-output predicates, we specify a system event e by a predicate e($\mathbf{v};\mathbf{v}$"), where the input parameter $\mathbf{v}$ denotes the value of the state vector immediately before the event occurrence and the output parameter $\mathbf{v}$" denotes the value of the state vector immediately after the event occurrence. Such predicates are referred to as *event predicates*.

The notions of enabling condition, composition, action, etc., carry over from input-output predicates to event predicates. Unlike in the case of input-output predicates, we do not insist that every parameter in $\mathbf{v}$" need occur in the body of e($\mathbf{v};\mathbf{v}$"). If a parameter v" in $\mathbf{v}$" does not occur in the body of e($\mathbf{v};\mathbf{v}$"), then we take the convention that the value of the state variable v is not affected by the occurrence of e($\mathbf{v};\mathbf{v}$"); i.e., (v"=v) is an implicit conjunct in the body of e($\mathbf{v};\mathbf{v}$"). This convention helps in writing compact predicate specifications of events. We allow event e to be represented by the header e($\mathbf{v}_a;\mathbf{v}_b$"), where $\mathbf{v}_a$ and $\mathbf{v}_b$" can be any subsets of $\mathbf{v}$ and $\mathbf{v}$" respectively such that $\mathbf{v}_a \cup \mathbf{v}_b$" include all the free variables that occur in the body of e.

2.3 Distributed System Model

In this section, the event-driven model described above is specialized to model a distributed system of processes. Our distributed system model incorporates terminology from the networking area because our applications are from that area. Each process is either a protocol entity or a communication channel. The distributed system is a network of protocol entities $P_1,P_2,...,P_I$ interconnected by communication channels $C_1,C_2,...,C_K$. The interconnection topology can be arbitrary.

For each protocol entity P_i, let $\mathbf{v}_i$ be the set of state variables of P_i. For each channel C_i, let $\mathbf{z}_i$ be the sequence of messages in transit in the channel. The system state vector, also referred to as the *global state vector*, is defined by

$$\mathbf{v} = (\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_I, \mathbf{z}_1, \mathbf{z}_2, ..., \mathbf{z}_K)$$

As before, the system initial conditions are specified by a predicate Initial(**v**).

Each process in the distributed system has a set of events. The events of entity P_i can only involve the state vector $\mathbf{v}_i$ and the state vectors of channels accessible from P_i; i.e., only these state vectors can occur in its event headers. Entity events model message receptions, message sends, and internal activities such as timeout handling. The events of channel C_i can involve only the state vector $\mathbf{z}_i$. Channel events model channel errors such as loss, duplication, and reordering of messages in transit (see Appendix B for their predicate definitions).

Entity events can access channel state variables only via *send* and *receive primitives*. The send primitive for channel C_i is defined by $\text{Send}_i(\mathbf{z}_i,m;\mathbf{z}_i") \equiv (\mathbf{z}_i"=(\mathbf{z}_i,m))$; i.e., append the message value m to the tail of $\mathbf{z}_i$. (We will use a comma as the concatenation operator, and use parentheses to resolve ambiguities.) The receive primitive for channel C_i is defined by $\text{Rec}_i(\mathbf{z}_i;m,\mathbf{z}_i") \equiv ((m,\mathbf{z}_i")=\mathbf{z}_i)$; i.e., remove the message at the head of $\mathbf{z}_i$ and assign it to m, provided that $\mathbf{z}_i$ is not empty. When these primitives are used in the predicate bodies of entity events, the formal message parameter m is replaced by the actual message sent or received. (The definition of $\text{Send}_i(\mathbf{z}_i,m;\mathbf{z}_i")$ above assumes that C_i has unbounded message capacity; see Appendix B for the bounded-capacity case.)

3. DISTRIBUTED REAL-TIME SYSTEM MODEL

In Section 3.1, we define timers and time events. In Sections 3.2 and 3.3 respectively, we model time constraints that are enforced by individual processes, and time constraints that are enforced due to the cooperation of the processes. In these sections, we treat the distributed system as a general collection of processes; no distinction is made between channels and entities. In Section 3.4, we model bounded-delay communication channels.

3.1 Measures of Time

We use the term *local timers* to refer to the timers that are implemented within individual processes of a distributed system. Our local timers have several realistic properties. First, the interval between successive ticks of a local timer is not infinitesimally small. Second, local timers in different processes are not coupled: the ticks of one timer do not coincide in time with the ticks of another timer. Third, the rates of local timers in different processes are not constant but may vary within specified error bounds of a constant rate.

A local timer is a discrete-valued state variable that can take values from the domain {Off,0,1,2,...}. For this domain, define the successor function *next* as follows: $next(\text{Off})=\text{Off}$ and $next(i)=i+1$ for $i \neq \text{Off}$. When a timer is *aged*, if the original value is t then its new value is $next(t)$. (Other domains and aging definitions can be chosen if needed; e.g., domain of $\{0,1,...,n\text{-}1\}$ and aging corresponding to modulo n addition.)

For each process, there is a *local time event* (corresponding to a clock tick) whose occurrence ages all local timers within that process. Since no other timer is affected, local timers in this process are effectively decoupled from timers in other processes of the distributed system. We shall use the term *system events* to refer to the events of the process other than the time event; i.e., the communication and internal events of the process. (Note that a communication event is a system event of each process involved in the communication.) In addition to being aged by its time event, a local timer can be *reset* to some value by a system event of that process. Thus, a local timer can be used to measure the time elapsed (in number of occurrences of its local time event) following a system event occurrence.

At this point in the modeling, the time events of different processes are entirely decoupled and may occur at vastly different rates. To keep the rate of time event occurrences in different processes within specified bounds, we include in our model a hypothetical time event, referred to as the *ideal time event*, that is assumed to occur at an absolutely constant rate. Each local time event's occurrences will be allowed to drift within a specified bound of the ideal time event's occurrences. For process i, let η_i denote the number of occurrences of its local time event since system initialization, and let ϵ_i denote the maximum error in the tick rate. (Typically, $\epsilon_i \ll 1$; for a crystal oscillator, $\epsilon_i \approx 10^{-6}$.) Let η denote the number of occurrences of the ideal time event since initialization. The η's are auxiliary state variables that are not implemented, and can never be reset by any system event.

Neither the local time event for process i nor the ideal time event is allowed to occur if such an occurrence will violate the following *accuracy axiom* of the local time event of process i

AccuracyAxiom$_i(\eta_i,\eta)$: For any earlier instant a,

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

where $\eta(a)$ refers to the value of η at instant a, and η refers to the current value of the state variable η. The above accuracy axiom states that over any time span since initialization, the number of occurrences of process i's local time event differs from the number of ideal time event occurrences by at most ϵ_i times the number of ideal time event occurrences. (The minimum upper bound of 1 is necessary since the η's are integer-valued.) This accuracy axiom is a discrete version of the following drift condition for continuous clocks

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

usually found in the literature [8].

For analyzing the relationships between time constraints enforced within processes and the resulting system-wide time constraints, it is convenient to have timers that are driven by the ideal time event. These timers are referred to as *ideal timers*. Ideal timers are *not* available to the implementation. Rather they are auxiliary variables used to record the actual time elapsed between occurrences of system events (not neccessarily of the same process). The analysis can then be divided into two phases: a global analysis phase involving only ideal timers, followed by a local analysis phase during which implementable time constraints (defined below) expressed with ideal timers are realized using local timers (see example in Section 5.1). In this latter phase, processes need to approximate the values of ideal timers by using local timers. Given an ideal timer u and a local timer v, we say: (u,v) *started at* (a,b), to mean that at some instant in the past u and v were simultaneously reset to a and b respectively, and after that instant there has been no reset to either u or v. If a=b then we say: (u,v) *started at* a.

If ϵ_i is the error rate of the local time event that drives timer v, then the following property clearly holds.

Started-at Property. (u,v) started at (a,b) $\Rightarrow$ |(u-a) - (v-b)| $\leq$ max(1, ϵ_i(u-a))

For convenience in specifying timers with limited counting capacity, we also allow a timer v to have the domain {Off,0,1,...,M} where M is some positive integer. The time event for aging v will reset v to Off if v=M before the time event occurrence. Note that this action is referred to as a reset (see started-at definition above), and not as aging. This reset action is incorporated by defining *next*(M) to be Off. Also, we shall extend the function *next* as follows: for any structured value **u**, *next*(**u**) returns **u** with every time value in it aged.

3.2 Implementable Time Constraints

Implementable time constraints are time constraints that are realizable by individual processes without any cooperation from the rest of the distributed system. They are guaranteed by the implementations of individual processes, and are not properties that have to be verified by analyzing the interaction of processes.

Let $\mathbf{v}_i$ denote the state vector of process i ($\mathbf{v}_i$ is a component of the global state vector $\mathbf{v}$). Let e_1 and e_2 be system events of process i, and let v be a timer in $\mathbf{v}_i$ that is reset to 0 by e_1, and reset to Off by e_2 where e_2 is different from e_1. Then, the time constraint (E1) "e_2 will not occur within T time units of e_1's occurrence" is modeled by including $v \geq T$ in the enabling condition of e_2. The time constraint (E2) "if e_2 occurs, then it occurs within T time units of e_1's occurrence" is modeled by including $v \leq T$ in the enabling condition of e_2. The time constraint (E3) "e_2 must occur within T time units of e_1's occurrence" is modeled by including $v < T$ in the enabling condition of the time event that ages v. Note that E3 *cannot* be modeled by including $v \leq T$ in the enabling condition of e_2, because in our model an enabled event is not forced to occur.

E1 and E2 are examples of time constraints of the form "system event e *will occur only if* the elapsed times satisfy a condition $TC(\mathbf{v}_i)$." They are modeled by including $TC(\mathbf{v}_i)$ in the enabling condition of event e.

E3 is an example of time constraints of the form "system event e *must occur while* the elapsed times satisfy a condition $TC(\mathbf{v}_i)$" They are modeled by including $TC(next(\mathbf{v}_i))$ in the enabling conditions of time events.

The TC conditions introduced above on the time events will be referred to as *timer axioms*. Let $\text{TimerAxiom}_i(\mathbf{v}_i)$ denote the conjunct of all the timer axioms of process i.

The local time event for process i is formally defined by the event predicate

(for all local timer v in $\mathbf{v}_i$)[v"=$next(v)$] **and** $\text{TimerAxiom}_i(\mathbf{v}_i")$
and $\eta_i" = \eta_i+1$ **and** $\text{AccuracyAxiom}_i(\eta_i",\eta)$

The ideal time event is defined by

(for all ideal timer v in $\mathbf{v}$)[v"=$next(v)$] **and** $\eta" = \eta+1$
and (for all process i)[$\text{TimerAxiom}_i(\mathbf{v}_i")$ **and** $\text{AccuracyAxiom}_i(\eta_i,\eta")$]

Observe that the time events are completely defined by the ideal and local timers, the timer axioms, and the error rates of the local time events. Unlike system events, time events are not implemented.

Because of the timer axioms, time events may deadlock. Consider the example E3 where e_1 and e_2 are both events of process i but e_2 involves the reception of a message (we shall refer to this example as E4). If there is no message in the channel to receive, then the time events will deadlock. We now present conditions to ensure that this will not happen. These conditions correspond to a formal definition of implementable time constraints.

A system event e of process i is said to be *controlled* by process i if the enabling condition of e depends only on the value of $\mathbf{v}_i$; i.e., e is an internal event or a message send into a nonblocking channel.

Definition. $\text{TimerAxiom}_i(\mathbf{v}_i)$ is *implementable* if the following conditions hold:

(IC1) $\text{TimerAxiom}_i(\mathbf{v}_i)$ holds initially.

(IC2) No system event of process i sets $\mathbf{v}_i$ to a value such that $\text{TimerAxiom}_i(\mathbf{v}_i)$=False **or** $\text{TimerAxiom}_i(\mathit{next}(\mathbf{v}_i))$=False.

(IC3) For every value of $\mathbf{v}_i$ such that $\text{TimerAxiom}_i(\mathit{next}(\mathbf{v}_i))$=False, there is a sequence of system events $e_1,e_2,...,e_n$ controlled by process i whose occurrence will set $\mathbf{v}_i$ to a value such that $\text{TimerAxiom}_i(\mathit{next}(\mathbf{v}_i))$=True.

Note that IC3 is the key condition. In example E4, IC3 is violated because the receive event e_2 is not controlled by process i. (The events referred to in IC3 can generally be found in the English statements of the time constraints, e.g., the event e_2 in example E3.) Also, note that the above definition does not depend on whether the timers involved are ideal or local. Therefore, it has the desirable feature of being independent of the error rate of the local time event of process i.

The distributed system is said to have implementable time constraints if the TimerAxiom of each process is implementable. The following result justifies our definition of implementable time constraints:

Theorem 1. If each process in a distributed system has an implementable TimerAxiom, then all timer and accuracy axioms hold at all times, and the time event counts will increase without bound. (Proof in Appendix A.)

3.3 Derived Time Constraints

Derived time constraints are time constraints that hold for the distributed system as a result of individual processes enforcing implementable time constraints. They are not guaranteed by the implementation but must be verified for the distributed system.

Derived time constraints can be global time constraints on the elapsed times between events in different processes. Derived time constraints can also be time constraints on events of the same process. An instance of that is example E4 where e_2 is a receive event.

A derived time constraint of the form "system event e *will occur only if* the elapsed times satisfy a condition TC(**v**)" is logically equivalent to the statement that TC(**v**) holds whenever e is enabled. It is established by proving that $e(\mathbf{v};\mathbf{v}") \Rightarrow TC(\mathbf{v})$ is invariant (proving invariance is covered in Section 4).

A derived time constraint of the form "system event e *must occur while* the elapsed times satisfy a condition TC(**v**)" is logically equivalent to the statement that TC(**v**) holds immediately after the occurrence of any time event. It is established by proving that (for every time event e_t)$[e_t(\mathbf{v};\mathbf{v}") \Rightarrow TC(\mathbf{v}")]$ is invariant.

3.4 Modeling Real-Time Channels

In this section, we model a channel C_i that displays a maximum message lifetime $MaxDelay_i$; i.e., any message attempting to stay in channel C_i for longer than $MaxDelay_i$ time units is lost or removed by some intermediate network node [21]. Such behavior is not only common in communication networks, but is crucial for the correct operation of communication protocols.

With each message in transit we associate a timer *age* that indicates the age of the message (time spent in the channel). For notational convenience, we assume that age is an ideal timer. The state variable $\mathbf{z}_i$ of C_i now denotes the sequence of <message,age> value pairs in C_i. Initially, any age timer in $\mathbf{z}_i$ has the value 0. The maximum message lifetime $MaxDelay_i$ constraint is modeled by the following timer axiom

$$TimerAxiom_{C_i}(\mathbf{z}_i) \equiv (\text{for every } <\text{message,age}> \text{ in } \mathbf{z}_i)[0 \leq \text{age} \leq MaxDelay_i]$$

The send and receive primitives are modified to the following: $Send_i(\mathbf{z}_i,m;\mathbf{z}_i") \equiv (\mathbf{z}_i=(\mathbf{z}_i,<m,0>))$, i.e., append message m with an age of 0 to the tail of $\mathbf{z}_i$. $Rec_i(\mathbf{z}_i;m,\mathbf{z}_i") \equiv$ (for some t)$[(<m,t>,\mathbf{z}_i")=\mathbf{z}_i]$, i.e., receive the message m from the head of $\mathbf{z}_i$ irrespective of its age.

In practice, to check that $TimerAxiom_i(\mathbf{z}_i)$ is implementable amounts to ensuring that at least one of the following conditions holds:

(a) The channel can delete any message of age $MaxDelay_i$. (Typically, a channel will have a loss event that can delete any <message,age> pair including those of age $MaxDelay_i$.)

(b) The entity that receives messages from C_i is always enabled to receive the first message (and hence by successive applications, any message) in C_i.

This guarantees IC3. IC1 is ensured since all *age* timers are initially zero. IC2 is guaranteed because neither the channel events nor the channel receive primitive reset *age* timers, and the channel send primitive resets *age* timers to 0.

4. PROVING SAFETY PROPERTIES

In this section, we present the inference rule for verifying safety properties, including those which involve started-at statements. We also describe how the method of preconditions can be used to generate the assertions needed to apply the safety inference rule. Our inference rules do not distinguish between distributed and non-distributed systems. Hence, in this section, we will consider the system model of Section 2.2: i.e, specified by a state vector **v**, an initial condition predicate Initial(**v**), and event predicates $e_1(\mathbf{v};\mathbf{v}")$, ..., $e_n(\mathbf{v};\mathbf{v}")$. Our reasons for adopting this approach are explained in the conclusion.

To justify the inference rules, we will appeal to the following state transition representation of the system model. The set of all possible value assignments to the system state variables defines the state space of the system. Those system states that satisfy Initial(**v**) are referred to as *initial system states*. Each event specifies a set of transitions between system states; each transition is from a system state where the event is enabled to a system state that can result from changes to the state variable values. A system state that can be reached from an initial state via a sequence of event transitions is referred to as a *reachable system 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 system behavior is represented by some path in the reachability graph starting from an initial state.

A safety property of the 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 **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 $A_0(\mathbf{v})$ 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 for proving invariance (following our convention, variables in the predicates are universally quantified over their domains unless otherwise indicated).

Inference Rule for Safety. If I(**v**) is invariant and A(**v**) satisfies

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

(ii) (for every event e)[(I(**v**) **and** A(**v**) **and** e(**v**;**v**")) $\Rightarrow$ A(**v**")]

(iii) $A(\mathbf{v}) \Rightarrow A_0(\mathbf{v})$

then we can infer that $A_0(\mathbf{v})$ is invariant.

Note that the inference rule is quite simple because of our use of predicates to define events. $A_0(\mathbf{v})$ represents a desired safety property and $I(\mathbf{v})$ can be any safety property whose invariance has already been verified. In particular, any timer or accuracy axiom can be a conjunct of $I(\mathbf{v})$ (from Theorem 1). Generating $A(\mathbf{v})$ from $A_0(\mathbf{v})$ and $I(\mathbf{v})$ is a nontrivial task analogous to generating loop invariants in program verification (see below).

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

Note that since $I(\mathbf{v})$ is given to be invariant, part (ii) of the above inference rule can be modified to the following:

(for every event e)[($I(\mathbf{v})$ **and** $I(\mathbf{v}")$ **and** $A(\mathbf{v})$ **and** $e(\mathbf{v};\mathbf{v}")$) $\Rightarrow A(\mathbf{v}")$]

Because this change strengthens the left hand side of the implication, it usually helps in deriving the right hand side.

Safety assertions with started-at statements

We will allow safety assertions to contain started-at statements; e.g., $x>y \Rightarrow$ ((u,v) started at 0). We next present rules for applying the safety inference rule to such assertions (below, u is an ideal timer and v is a local timer).

Started-at Rule 1. In part(i) of the safety inference rule, if

Initial($\mathbf{v}$) $\Rightarrow$ (u=a **and** v=b)

then we can infer

Initial($\mathbf{v}$) $\Rightarrow$ ((u,v) started at (a,b))

Started-at Rule 2. In part(ii) of the safety inference rule,

(i) If system event e satisfies

($I(\mathbf{v})$ **and** $A(\mathbf{v})$ **and** $e(\mathbf{v};\mathbf{v}")$) $\Rightarrow$ (u"=a **and** v"=b)

then we can infer

($I(\mathbf{v})$ **and** $A(\mathbf{v})$ **and** $e(\mathbf{v};\mathbf{v}")$) $\Rightarrow$ (u",v") started at (a,b)

(ii) If system event e satisfies

$$(I(\mathbf{v}) \textbf{ and } A(\mathbf{v}) \textbf{ and } e(\mathbf{v};\mathbf{v}")) \Rightarrow (u"=u \textbf{ and } v"=v \textbf{ and } ((u,v) \text{ started at } (a,b)))$$

then we can infer

$$(I(\mathbf{v}) \textbf{ and } A(\mathbf{v}) \textbf{ and } e(\mathbf{v};\mathbf{v}")) \Rightarrow ((u",v") \text{ started at } (a,b))$$

(iii) If time event e does not reset u or v (recall that time events can reset timers with bounded domains), then we can infer

$$(e(\mathbf{v};\mathbf{v}") \textbf{ and } ((u,v) \text{ started at } (a,b))) \Rightarrow ((u",v") \text{ started at } (a,b))$$

Method of preconditions

The inputs for the safety inference rule consist of $A_0(\mathbf{v})$, $I(\mathbf{v})$ and the system specifications. The method of preconditions [2] can be used to generate from these inputs either an assertion $A(\mathbf{v})$ that satisfies the requirements of the rule, or a sequence of events that takes the system from an initial state to a state that violates $A_0(\mathbf{v})$.

For an event $e(\mathbf{v};\mathbf{v}")$ and an assertion $C(\mathbf{v})$, the *weakest precondition* of C with respect to e can be defined by the following predicate in $\mathbf{v}$: (for all g)[$e(\mathbf{v};g) \Rightarrow C(g)$]. It specifies the set of all states from where the occurrence of e, if it is enabled, takes the system to a state that satisfies $C(\mathbf{v})$. The following iterative procedure generates a sequence of events together with a sequence of assertions $A_0(\mathbf{v})$, $A_1(\mathbf{v})$, $A_2(\mathbf{v})$, ..., where $A_{i+1}(\mathbf{v}) \Rightarrow A_i(\mathbf{v})$ for all i. The procedure will terminate provided that $A_0(\mathbf{v})$ is a decidable property.

1. Initially i=1, $A_1(\mathbf{v}) \equiv I(v)$ **and** $A_0(\mathbf{v})$, and the event sequence is empty.
2. If there exists a state g such that Initial(g) = True and A(g) = False, then terminate procedure: $A_0(\mathbf{v})$ is not invariant and the event sequence will take the system from initial state g to a state that violates $A_0(\mathbf{v})$.
3. If for every event e in the event set, the weakest precondition of $A_i(\mathbf{v})$ with respect to e is implied by $A_i(\mathbf{v})$, then terminate procedure: $A_0(\mathbf{v})$ is invariant and $A_i(\mathbf{v})$ is the weakest desired assertion.
4. Choose an event e'_i from the event set. Let A_{i+1} be the conjunct of A_i and the weakest precondition of A_i with respect to e'_i. Append e'_i to the event sequence. Increment i by 1 and go to step 2.

From our experience, a blind application of the above iterative procedure often leads to an expression for $A_i(\mathbf{v})$ that grows unmanageably with increasing i. The choice of the event in step 4 of each iteration is usually crucial in avoiding this. Unfortunately, the correct choice becomes clear only with increasing intuitive understanding of the system's behavior.

In practice, it is fruitful to strengthen $A_i(\mathbf{v})$ by including "relevant" properties that are either known to be invariant or are suspected of being invariant. In this case, the precondition obtained upon termination may not be the weakest precondition; this method is illustrated in Section 5.1 with an example. If we included a property C which turns out to be not invariant, then the above procedure may terminate improperly by declaring $A_0(\mathbf{v})$ to be not invariant while it is actually invariant. This improper termination may be detected by trying out the event sequence generated by the procedure on the initial state g. It will take the system to a state that violates C rather than $A_0(\mathbf{v})$. It is then necessary to resume the procedure from the point just prior to where C was introduced and not to utilize C.

5. A 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 each channel C_i has a bounded-delay of $MaxDelay_i$, and can lose, duplicate and reorder messages in transit. There is a source at P_1 which produces new data blocks to be transferred to a destination at P_2 which consumes them.

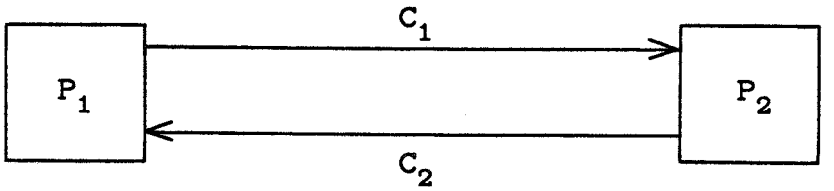

Figure 1. Network configuration of protocol example.

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 is the name of the message type, data can be any data block from DataSet, and ns is a sequence number that identifies the data block. P_2 sends messages of type (ACK,nr) where nr is a sequence number identifying the next expected data block. In this example, ns and nr are restricted to the values of 0 and 1. (The reader is referred to [9] for the general case of ns and nr taking values from $\{0,1,...,N\text{-}1\}$ for any $N \geq 2$.)

Let Source[n] for $n \geq 0$ denote the sequence of data blocks that have been accepted by P_1 from its source. P_1 sends Source[n] accompanied by sequence number equal to n mod 2. Let Sink[n] for $n \geq 0$ denote the sequence of data blocks that have been passed by P_2 to the destination. 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) where nr equals the next expected value of ns. When P_1 receives an (ACK,nr), if there is an outstanding data block Source[n] such that nr=(n+1)mod 2 then the (ACK,nr) is considered to acknowledge Source[n] (as well

as any data blocks accepted earlier). An *outstanding* data block is one that has been accepted but not yet acknowledged. (This is compatible with the usual notion of outstanding because we allow P_1 to send an outstanding block at any time.)

In order to ensure that received sequence numbers are correctly interpreted, P_1 accepts a new data block Source[n] only when the following three conditions hold: First, all data blocks accepted earlier have been acknowledged. Second, at least $MaxDelay_1$ time has elapsed since the last send of data block Source[n-1]. Third, at least $MaxDelay_2$ time has elapsed since P_1 first received the acknowledgement to Source[n-1].

If the first condition is violated, then both Source[n] and Source[n-1] would become outstanding, and a received (ACK, n+1 mod 2) could signify either an acknowledgement for Source[n] or an old acknowledgement for Source[n-2]. The second and third conditions are implementable time constraints. In Section 5.1, we will derive these time constraints as sufficient conditions for the correct interpretation of received sequence numbers.

Once Source[n] is accepted by P_1, it is repeatedly sent until it is acknowledged. Neither of the above time constraints apply to the retransmissions of Source[n]. 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. (further discussions in Section 7). 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. Auxiliary state variables needed for stating and verifying desired logical correctness properties are also included. Below, $\oplus$ and $\ominus$ denote modulo 2 addition and subtraction respectively. Also, $MDelay_i = 1+(1+\epsilon_i) \times MaxDelay_i$ for i=1 and 2 (recall that P_1's timers have an accuracy of ϵ_1). Finally, for brevity in stating an event predicate, we use the following guarded command [2] notation $e_1(v;v") \rightarrow e_2(v;v")$ to mean that the action in e_2 is done only if e_1 is enabled. Formally, $e_1(v;v") \rightarrow e_2(v;v")$ expands to $(e_1(v;v") \Rightarrow e_2(v;v")$ **and** (not $e_1(v;v") \Rightarrow$(for every v" in the body of e_2)[v"=v]).

Variables of P_1

Source : array[0..s-1] of DataSet; {Source is an auxiliary history variable that records the sequence of data blocks accepted by P_1. s indicates the length of Source. Initially, s=0 and Source is the null sequence.}

a : 0..∞; {Auxiliary variable indicating the number of data blocks that have been acknowledged; i.e., Source[a],...,Source[s-1] are outstanding. a is initialized to 0 and is always less than or equal to s.}

vs : 0..1; {Initialized to 0 and always equals s mod 2}

va : 0..1; {Initialized to 0 and always equals a mod 2}

DTimer : (Off,0,1,2,...,$MDelay_1$); {Local timer which measures elapsed time since last D message sent. Initialized to Off}

DTimerG : (Off,0,1,...); {Ideal timer associated with DTimer. Auxiliary variable initialized to Off}

ATimer : (Off,0,1,2,...,$MDelay_1$); {Local timer that measures time elapsed since reception of ACK message that first acknowledged Source[s-1]. Initialized to Off}

ATimerG : (Off,0,1,...); {Ideal timer associated with ATimer. Auxiliary variable initialized to Off}

Let $\mathbf{v}_1$ denote a list of the above variables along with the local time event count η_1. The initial condition of P_1 is given by

$Initial_1(\mathbf{v}_1) \equiv$ (a=s=vs=va=0 **and** DTimer=DTimerG=ATimer=ATimerG=Off).

Events of P_1

1. AcceptData($\mathbf{v}_1$;$\mathbf{v}_1$")
   $\equiv$ vs=va **and** DTimer=Off **and** ATimer=Off {If new data can be accepted}
   and Source[s]" ϵ DataSet **and** s"=s+1 **and** vs"=vs $\oplus$ 1 {then do so}

2. Send_D($\mathbf{v}_1$,$\mathbf{z}_1$;$\mathbf{v}_1$",$\mathbf{z}_1$")
   $\equiv$ vs $\neq$ va **and** $Send_1$((D,Source[s-1],vs-1), $\mathbf{z}_1$;$\mathbf{z}_1$") {send outstanding data}
   and DTimer"=0 **and** DTimerG"=0 {start DTimer}

3. Rec_ACK($\mathbf{v}_1$,$\mathbf{z}_2$;$\mathbf{v}_1$",$\mathbf{z}_2$")
   $\equiv$ (for some nr in {0,1})[Rec_2($\mathbf{z}_2$;(ACK,nr),$\mathbf{z}_2$") {Receive nr}
   and ((vs$\neq$va **and** nr=va $\oplus$ 1) $\rightarrow$ {if outstanding data acknowledged}
   (a"=a+1 **and** va"=va $\oplus$ 1 {then update state}
   and ATimer"=0 **and** ATimerG"=0))] {and start ATimer}

Variables of P_2

Sink : array[0..r-1] of DataSet; {Sink is an auxiliary history variable that records the sequence of data blocks passed on to the destination. r indicates its length. Initially r=0 and Sink is the null sequence}

vr : 0..1; {Sequence number of the next expected data block. vr is initialized to 0 and always equals (r mod 2)}

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.

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

Events of P_2

1. Send_ACK $(\mathbf{v}_2,\mathbf{z}_2;\mathbf{v}_2",\mathbf{z}_2")$
   $\equiv$ SendACK = True **and** Send_2 ((ACK,vr), $\mathbf{z}_2;\mathbf{z}_2"$)
   and SendACK"=False

2. Rec_D $(\mathbf{v}_2,\mathbf{z}_1;\mathbf{v}_2",\mathbf{z}_1")$
   $\equiv$ (for some data in DataSet)(for some ns in {0,1})[Rec_1 $(\mathbf{z}_1;$(D,data,ns),$\mathbf{z}_1")$
   and (ns=vr $\rightarrow$ {if next expected sequence number}
   (Sink[r]"=data **and** r"=r+1 **and** vr"=vr $\oplus$ 1)) {then accept data}
   and SendACK" = True]

Other events

$\mathbf{z}_i$ is the sequence of (message,age) pairs in C_i. The channel events of C_i are specified by a predicate ChannelError($\mathbf{z}_i,\mathbf{z}_i"$) that allows all possible losses, duplications and reorderings of (message,age) pairs in the channel. The time axiom for C_i is $\text{TimerAxiom}_i(\mathbf{z}_i) \equiv$ (for all <m,t> in $\mathbf{z}_i$)$[0 \leq t \leq \text{MaxDelay}_i]$.

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

$\eta_1" = \eta_1 + 1$ {age η_1 and its associated local time variables}
and DTimer" = next(DTimer) **and** ATimer" = next(ATimer)
and AccuracyAxiom($\eta_1",\eta$) {if aging does not violate any accuracy or time axioms}
and DTimer" $\leq$ MDelay_1 **and** ATimer" $\leq$ MDelay_2

The *ideal time event* is specified by

AccuracyAxiom_1 $(\eta_1,\eta")$ **and** $\text{TimeAxiom}_1(\mathbf{z}_1")$ **and** $\text{TimeAxiom}_2(\mathbf{z}_2")$
and $\eta"=\eta+1$ **and** $\mathbf{z}_1"=\text{next}(\mathbf{z}_1)$ **and** $\mathbf{z}_2"=\text{next}(\mathbf{z}_2)$
and DTimerG" = next(DTimerG) **and** ATimerG" = next(ATimerG)

System initial condition

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

Initial(**v**) ≡ Initial$_1$($\mathbf{v}_1$) **and** Initial$_2$($\mathbf{v}_2$) **and** $\mathbf{z}_1$ is empty **and** $\mathbf{z}_2$ is empty
and η=η_1=0.

5.1 Safety Verification and Timing Analysis

For the above protocol, we would like to verify that following safety property is invariant:

A_0 (a) Sink[i] = Source[i] for $0 \leq i < r$
(b) $a \leq r \leq s \leq a + 1$

A_0 states that the sequence of data blocks passed to the destination at P_2 is a prefix of the sequence of data blocks accepted from the source at P_1, and that a data block is acknowledged at P_1 only after it has indeed been passed on to the destination at P_2. We have included in part (b) of A_0 the requirement that P_1 can have at most one outstanding data block. In order to show that A_0 is invariant, we need to find an assertion A that implies A_0 and satisfies the inference rule for safety (Section 4). We will obtain such an assertion using the heuristic approach outlined in Section 4.

For brevity, we shall refer to a predicate A(**v**) as simply A, and use A" to refer to A(**v**"). Also, we shall refer to an event e(**v**;**v**") as simply e. When we prove that (A **and** e)⇒A" holds, the proof will be presented as a sequence of steps, each consisting of a statement *L* (at the left) and a list of statements R_1,R_2, ... (at the right). *L* derives (in predicate calculus) from the list R_1,R_2, ... Each R_i is (a) a statement that has been derived in an earlier step; or (b) a statement that is implied by the event under consideration; or (c) an assumption when the implication R_i⇒*L*, rather than *L*, is being proved. Finally, we say that event e does *not affect* predicate A if event e implies that v"=v for every variable v in A; clearly, if e does not affect A then (e **and** A)⇒A" holds.

The following property is obviously invariant:

A1 vs = s mod 2 **and** va = a mod 2 **and** vr = r mod 2

Proof of A1's invariance
Each individual conjunct in A1 satisfies the safety inference rule. We will give the details for (vs=s mod 2) only.

Initial(**v**)⇒(vs=s=0)⇒(vs=s mod 2).
For every event e other than AcceptData, e does not effect A1.
AcceptData:

vs"=s" mod 2 (from vs=s mod 2 (from A1), s"=s+1, vs"=vs $\oplus$ 1)

End of proof

In order that P_2 correctly interprets received D messages, it is neccessary that at any time all the D messages in C_1 must have the same sequence number. Suppose this is not the case, and C_1 contains the messages (D,$data_1$,1) and (D,$data_2$,0). Because C_1 can duplicate and reorder messages, P_2 can receive the messages in the following alternating sequence: (D,$data_i$,i), (D,$data_{i \oplus 1}$,i $\oplus$ 1), (D,$data_i$,i), ..., where i=vr. Then, P_2 will pass the sequence $data_i$,$data_{i \oplus 1}$,$data_i$,..., to the destination, thus violating A_0.

Recall that once P_1 accepts a data block, it can be sent immediately. Therefore, to ensure that all the D messages in C_1 have the same sequence number, P_1 must ensure before accepting a new data block Source[s] that Source[s-1] is no longer in C_1. This desired precedence relation can be enforced by exploiting the maximum message lifetime property of C_1. Specifically, P_1 waits until the time elapsed since the last send of Source[s-1] exceeds $MaxDelay_1$. This explains the implementable timer constraint DTimer=Off in the AcceptData event.

From the Send_D event and part (b) of A0, observe that whenever a (D,data,ns) is sent into C_1, we have data = Source[s-1], ns = vs $\ominus$ 1, and DTimerG reset to 0 so that DTimerG lower bounds the age of every D message in C_1. Further, s and vs are changed only when when DTimer = Off, i.e., when there are no D messages in $\mathbf{z}_1$. Thus, the following assertion is invariant:

A2 <(D,data,ns),age> in $\mathbf{z}_1$ $\Rightarrow$
- (a) (data = Source[s-1] **and** ns = vs $\ominus$ 1
- (b) **and** (DTimerG, DTimer) started at 0
- (c) **and** age $\geq$ DTimerG)

Proof of A2's invariance

Initial($\mathbf{v}$)$\Rightarrow$$\mathbf{z}_1$ is empty$\Rightarrow$A2.
Events Rec_ACK and Send_ACK do not affect A2.
Consider the channel events and Rec_D:
(a) A2" (from A2, $\mathbf{v}_1$"=$\mathbf{v}_1$, <m,t> in $\mathbf{z}_1$" $\Rightarrow$ <m,t> in $\mathbf{z}_1$)
Consider AcceptData:
(a) not ((DTimerG,DTimer) started at 0) (from DTimer=Off)
(b) A2" (from a, A2)
Consider Send_D:
(a) DTimerG"=DTimer"=0, s"=s, vs"=vs, (Send_D)
$\mathbf{z}_1$"=($\mathbf{z}_1$,<(D,Source[s-1],vs $\ominus$ 1),0>)
(b) A2" (from a, A2, Timer axiom for C_1)

Consider the ideal time event:

(a) A2" (from A2, age"=age+1, DTimerG=next(DTimerG), $\mathbf{v}_i$"=$\mathbf{v}_i$)

Consider the local time event of P_1:

(a) DTimer<$MDelay_1$ (from (DTimerG,DTimer) started at 0)
⇒(DTimerG",DTimer") started at 0

(b) DTimer=$MDelay_1$ (from A2, Started-at property 1, Timer axiom for C_1)
⇒DTimerG>$MaxDelay_1$.
⇒no D messages in $\mathbf{z}_1$

(c) A2" (from a, b)

End of proof

In order that P_1 correctly interprets a received (ACK,nr) message, it is necessary that the following is invariant:

A3 (((ACK,va $\oplus$ 1) in $\mathbf{z}_2$) **and** vs $\neq$ va) ⇒ r = s

Otherwise, the reception of the ACK message will cause P_1 to violate A_0.

To have A3 invariant, P_1 must ensure before sending a new data block Source[n] that C_2 does not contain old (ACK,n-1 mod 2) messages which were used to acknowledge the reception of Source[n-2]. Thus, before sending Source[n], P_1 must ensure that the elapsed time since the last send of (ACK,n-1 mod 2) exceeds $MaxDelay_2$. Unlike in the above case of the D messages, P_1 does not have access to this elapsed time. However, note that P_2 does not send (ACK,n-1 mod 2) once r equals n. Also, from A_0, a=n is true only after r=n is true. Thus, the time elapsed since a=n became true is a lower bound on the time elapsed since P_2's last send of (ACK,n-1 mod 2). Furthermore, P_1 does have access to this elapsed time. In our specifications, ATimerG indicates this elapsed time. Thus, P_1 can ensure that A3 is invariant by ensuring that ATimerG exceeds $MaxDelay_2$ before sending Source[n]. This is the derivation of the time constraint ATimer=Off in the AcceptData event.

Immediately after a and ATimerG are set equal to n and 0 respectively, we have s=a=n and ATimerG lower bounding the age of every (ACK,n-1 mod 2) message in C_2. vs and va remain constant until new data is accepted, at which point C_2 has no (ACK,n-1 mod 2) messages and vs≠va. Thus, we expect the following property to be invariant (note that va $\ominus$ 1 = va $\oplus$ 1).

A4 ((<(ACK,va $\oplus$ 1),age> in $\mathbf{z}_2$) **and** vs = va) ⇒ (age $\geq$ ATimerG
and ((ATimerG, ATimer) started at 0)

Proof of invariance of A0 and A3 and A4

The following proof relies upon the invariance of A1 and A2 (proved above).
Initial(**v**)⇒(s=r=0 **and** $\mathbf{z}_2$ is empty)⇒ (A0 **and** A3 **and** A4)

Consider channel events. They do not affect A0.

(a) A3" **and** A4" (A3, A4, <m,t> in $\mathbf{z}_2$"⇒<m,t> in $\mathbf{z}_2$)

Consider AcceptData:

(a) A0" (A0, s"=s+1, r"=r, Sink"=Sink)
(b) not ((ACK,va $\oplus$ 1) in $\mathbf{z}_2$) (A4, vs=va, ATimer=Off)
(c) A3", A4" (b, vs"≠va"=va, $\mathbf{z}_2$"=$\mathbf{z}_2$)

Send_D does not affect A0, A3, and A4.

Consider Rec_D. If received (D,data,ns) has ns≠vr, then A0,A3,A4 unaffected.
We now consider the other case:

(a) Rec($\mathbf{z}_1$;(D,data,ns),$\mathbf{z}_1$") **and** ns=vr (assumption)
(b) data=Source[s-1], ns=vr=vs $\ominus$ 1=(s-1) mod 2 (a, A1, A2)
(c) vs $\ominus$ 1=vr, r=s-1, vs≠va (b, A0)
(d) vs=vr", r"=s, A0" (c, r"=r+1, vr"=vr $\oplus$ 1, Sink[r]"=data)
(e) A3", A4" (r=s, vs≠va)

Consider Send_ACK. A0 is not affected.

(a) A3",A4" (va=vr (assumption), A3, A4, $\mathbf{z}_2$"=($\mathbf{z}_2$,<(ACK,vr),0>))
(b) r=s=a+1, vs≠va (vr=va $\oplus$ 1 (assumption), A0, A1)
(c) A3", A4" (b, A3, A4, $\mathbf{z}_2$"=($\mathbf{z}_2$,<(ACK,vr),0>))

Consider Rec_ACK:

(a) (for some age)[$\mathbf{z}_2$=(<(ACK,nr),age>,$\mathbf{z}_2$")] (Rec_ACK)
(b) A0", A3", A4" ((not (nr=vs≠va)) (assumption), a, s"=s, a"=a)
(c) nr=vs≠va (assumption)
(d) r=s=a+1 (c, A0, A1, A3)
(e) A0", A3" (d, A0, a"=a+1)
(f) A4" (ATimerG"=ATimer"=0, Timer axiom for C_2)

Consider the ideal time event. A0 and A3 are unaffected.

(a) A4" (A4, age"=age+1, ATimerG"=ATimerG+1, ATimer not reset)

Consider the local time event of P_1. A0 and A3 not affected.

(a) A4" (ATimer<$MDelay_2$ (assumption), ATimer not reset)
(b) ATimerG>$MaxDelay_2$ (ATimer=$MDelay_2$ (assmpt.), A4, Started-at ppty.)
(c) not ((ACK,va $\oplus$ 1) in $\mathbf{z}_2$) (b, Timer axiom of C_2, A4)
(d) A4" (a, c)

End of proof

6. PROVING LIVENESS PROPERTIES

In this section, we describe how liveness properties are specified, and give the inference rules for verifying them. As an application, we verify the liveness of the protocol example. A liveness property of a distributed 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 system 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 verifying liveness properties is based on specifying and verifying inductive properties of *bounded-length* paths in the reachability graph. 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.

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.

For stating most liveness properties, it necessary to relate values of variables when A(**v**) holds to values of variables some time later when B(**v**) holds. For example, we will need to make statements such as: (for all integers m_1)[$x \geq m_1$ leads-next-to $x \geq m_1+1$]. For this purpose, we need to consider assertions whose free variables now include variables different from **v**. Let **m** denote the set of these new variables. Then, A(**v**,**m**) leads-next-to B(**v**,**m**) means that A(**v**,**m**) leads-next-to B(**v**,**m**) for each possible value of **m**. For notational convenience, we will assume each variable m_i in **m** takes nonnegative integer values.

We now present the inference rule used to establish the leads-next-to property.

Inference Rule for leads-next-to. If I(**v**) is invariant and there is a set of events E such that A(**v**,**m**) and B(**v**,**m**) satisfy:

(i) (for every event e in the system)[(I(**v**) **and** A(**v**,**m**) **and** e(**v**;**v**")) ⇒ (A(**v**",**m**) **or** B(**v**",**m**))]

(ii) (I(**v**) **and** A(**v**,**m**)) ⇒ (for some event e in E) (for some **v**") [e(**v**;**v**")) **and** B(**v**",**m**)]

then we can infer that A(**v**,**m**) leads-next-to B(**v**,**m**) via E.

This inference rule is very similar to the definition of *leads-next-to*, except that instead of referring to reachable states it allows us to utilize any safety property I(**v**) that is known. It also explicitly indicates the set E of events that allow the system to achieve the leads-next-to property. Therefore, a leads-next-to statement can be verified by examining each event individually. As in the case of the inference rule for safety, we can replace I(**v**) by (I(**v**) **and** I(**v'**)) in parts (i) and (ii) of the above inference rule. This will make it easier to establish the right hand sides of the implications.

We now extend the leads-next-to definition. Given assertions A(**v**,**m**) and B(**v**,**m**), we say that A(**v**,**m**) *leads-to* B(**v**,**m**) if for every value of **m** and every reachable system state g_0 where A(g_0,**m**)=True, the following holds: for every unbounded-length path $g_0,g_1,g_2,\ldots$ in the reachability graph, either there is a state g_n where B(g_n,**m**)=True, or there is an event e which is enabled at an infinite number of states in the path but never occurs.

The leads-to construct is similar to the "eventually" operator of temporal logic [14]: (A(**v**,**m**) leads-to B(**v**,**m**)) is equivalent to (A(**v**,**m**) $\Rightarrow \diamond$ B(**v**,**m**)). It is quite adequate for stating desired liveness properties; e.g., the liveness property example mentioned above can be stated by ($x \geq 0$ **and** $y \geq 0$) leads-to ($x \geq m_1$ **or** $y \geq m_2$)). We will now state two inference rules for the leads-to relationship.

Inference Rule for leads-to. Given assertions A(**v**,**m**), B(**v**,**m**), and C_1(**v**,**m**), C_2(**v**,**m**), ..., C_n(**v**,**m**) that satisfy

(i) A(**v**,**m**) leads-next-to (B(**v**,**m**) **or** C_1(**v**,**m**))

(ii) For $1 \leq i < n$: C_i(**v**,**m**) leads-next-to (B(**v**,**m**) **or** C_{i+1}(**v**,**m**))

(iii) C_n(**v**,**m**) leads-next-to B(**v**,**m**)

then we can infer that A(**v**,**m**) leads-to B(**v**,**m**).

This inference rule can be used to derive properties such as (($x \geq m_1$ **and** $y \geq m_2$) leads-to ($x \geq m_1+1$ **or** ($x \geq m_1$ **and** $y \geq m_2+1$))). The next inference rule allows us to take the above leads-to statement and infer that (($x \geq 0$ **and** $y \geq 0$) leads-to ($x \geq m_1$ **or** $y \geq m_2$).

Induction Rule for leads-to. Given assertions A(**v**), B(**v**), D_1(**v**,m_1), D_2(**v**,m_2), ..., D_l(**v**,m_l) that satisfy

$(A(\mathbf{v})$ **and** $D_1(\mathbf{v},m_1)$ **and ...** **and** $D_l(\mathbf{v},m_l))$ leads-to
$(B(\mathbf{v})$ **or**
$(A(\mathbf{v})$ **and**
$(D_1(\mathbf{v},m_1+1)$ **or**
$(D_1(\mathbf{v},m_1)$ **and**
$(D_2(\mathbf{v},m_2+1)$ **or**
$(D_2(\mathbf{v},m_2)$ **and**
$(D_3(\mathbf{v},m_3+1)$ **or**
(...**or**
$(D_{l-1}(\mathbf{v},m_{l-1})$ **and**
$D_l(\mathbf{v},m_l+1))...)$

we can infer

$(A(\mathbf{v})$ **and** $D_1(\mathbf{v},0)$ **and ...** **and** $D_l(\mathbf{v},0))$ leads-to
$(B(\mathbf{v})$ **or** $D_1(\mathbf{v},m_1)$ **or ...** **or** $D_l(\mathbf{v},m_l))$.

We refer to the first leads-to statement in the induction rule as an *inductive leads-to* statement. The induction rule merely applies mathematical induction to inductive leads-to statements. The validity of the above induction rule may require some thought. The case when l=1 is obvious: Given

$(A(\mathbf{v})$ **and** $D_1(\mathbf{v},m_1))$ leads-to $(B(\mathbf{v})$ **or** $(A(\mathbf{v})$ **and** $D_1(\mathbf{v},m_1+1)))$

we can infer

$(A(\mathbf{v})$ **and** $D_1(\mathbf{v},0))$ leads-to $(B(\mathbf{v})$ **or** $D_1(\mathbf{v},m_1))$

The case when l=2 is fairly obvious: Given

$(A(\mathbf{v})$ **and** $D_1(\mathbf{v},m_1)$ **and** $D_2(\mathbf{v},m_2))$ leads-to
$(B(\mathbf{v})$ **or** $(A(\mathbf{v})$ **and** $(D_1(\mathbf{v},m_1+1)$ **or** $(D_1(\mathbf{v},m_1)$ **and** $D_2(\mathbf{v},m_2+1)))))$

we can infer

$(A(\mathbf{v})$ **and** $D_1(\mathbf{v},0)$ **and** $D_2(\mathbf{v},0))$ leads-to $(B(\mathbf{v})$ **or** $D_1(\mathbf{v},m_1)$ **or** $D_2(\mathbf{v},m_2))$

We now list a few rather obvious properties of the leads-to relationship.

Leads-to Property 1. If $(A(\mathbf{v},\mathbf{m})$ leads-to $B(\mathbf{v},\mathbf{m}))$ and $(C(\mathbf{v},\mathbf{m})$ leads-to $D(\mathbf{v},\mathbf{m}))$, then $((A(\mathbf{v},\mathbf{m})$ **or** $C(\mathbf{v},\mathbf{m}))$ leads-to $(B(\mathbf{v},\mathbf{m})$ **or** $D(\mathbf{v},\mathbf{m})))$.

Leads-to Property 2. If $(A(\mathbf{v},\mathbf{m})$ leads-to $B(\mathbf{v},\mathbf{m}))$ and $(B(\mathbf{v},\mathbf{m})$ leads-to $C(\mathbf{v},\mathbf{m}))$, then $(A(\mathbf{v},\mathbf{m})$ leads-to $C(\mathbf{v},\mathbf{m}))$.

Leads-to Property 3. If $(A(\mathbf{v},\mathbf{m}) \Rightarrow B(\mathbf{v},\mathbf{m}))$, then $(A(\mathbf{v},\mathbf{m})$ leads-to $B(\mathbf{v},\mathbf{m}))$.

Leads-to Property 4. If $(A(\mathbf{v},\mathbf{m})$ leads-to $B(\mathbf{v},\mathbf{m}))$, $I(\mathbf{v})$ is invariant, and $C(\mathbf{v},\mathbf{m})$ is any assertion, then $((A(\mathbf{v},\mathbf{m})$ **and** $C(\mathbf{v},\mathbf{m}))$ leads-to $(B(\mathbf{v},\mathbf{m})$ **and** $I(\mathbf{v}))$.

6.1 Liveness Verification of Data Transfer Protocol

For the protocol example, we would like to prove the following liveness property: If P_1 accepts data block Source[n-1] then it is eventually acknowledged, provided C_1 does not continuously lose (D,Source[n-1],n-1 mod 2) messages and C_2 does not continuously lose (ACK,n mod 2) messages. Note that once Source[n-1] is acknowledged, P_1 will be eventually ready to accept Source[n] (in fact, P_1 will be ready within $\max(\text{MaxDelay}_1, \text{MaxDelay}_2)$ time units).

To formally state this property, we define the following auxiliary variable arrays:

LCount1, LCount2: array[0..s-1] of integers; {initialized to 0}

For the duration that Source[n-1] is outstanding, LCount1[n-1] indicates the number of times that C_1 has lost a (D,Source[n-1],n-1 mod 2) message since the previous reception of such a message at P_2. LCount1[n-1] is reset to 0 at each reception of (D,Source[n-1],n-1 mod 2), and incremented by 1 whenever a loss event of C_1 deletes a (D,Source[n-1],n-1 mod 2) message. For the duration that Source[n-1] is outstanding, LCount2[n-1] indicates the number of times that C_2 has lost a (ACK,n mod 2) message. (Note that Source[n-1] is not outstanding as soon as an (ACK,n mod 2) message is received at P_1.)

The desired liveness property can be stated by

L_0 s=a+1=n leads-to (s=a=n **or** LCount1[n-1]$\geq m_1$ **or** LCount2[n-1]$\geq m_2$)

We first prove that Source[n-1] will be received at P_2 unless C_1 continuously loses (D,Source[n-1],n-1 mod 2) messages. This property is formally stated as follows:

L1 s=a+1=r+1=n leads-to (s=r=a+1=n **or** LCount1[n-1]$\geq m_1$)

Proof of L1

In the proof, LCount1 and <n-1> denote LCount1[n-1] and (D,Source[n-1],n-1 mod 2) respectively. It can be easily checked that the following leads-next-to statements satisfy the inference rule for leads-next-to.

(i) (s=a+1=r+1=n **and** LCount1 $\geq m_1$) leads-next-to
(s=a+1=r=n **or** (s=a+1=r+1=n **and** (<n-1> in $\mathbf{z}_1$) **and** LCount1 $\geq m_1$))
via {Rec_D, Send_D}

(ii) (s=a+1=r+1=n **and** (<n-1> in $\mathbf{z}_1$) **and** LCount1 $\geq m_1$) leads-next-to
(s=a+1=r=n **or** (s=a+1=r+1=n **and** LCount1 $\geq m_1+1$)
via {Rec_D, Loss event for C_1}

From the inference rule for leads-to, (i) and (ii) imply the following inductive leads-to property:

(iii) (s=a+1=r+1=n **and** LCount1 $\geq m_1$)
leads-to (s=r=a+1=n **or** (s=a+1=r+1=n) **and** LCount1 $\geq m_1$+1))

From the induction rule for leads-to for the case l=1, (iii) implies L1 (since LCount1 $\geq$ 0 is always true). **End of proof**

We next prove that once P_2 has received Source[n-1], then P_1 eventually receives the required acknowledgement (ACK,n mod 2), unless C_2 continuously loses (ACK,n mod 2) messages or C_1 continuously loses (D,Source[n-1],n-1 mod 2) (which means that P_2 may not repeatedly send ACK messages). This property is formally stated as follows:

L2 s=r=a+1=n leads-to
(s=r=a=n **or** LCount1[n-1] $\geq m_1$ **or** LCount2[n-1] $\geq m_2$)

Proof of L2

In the proof, LCount1 and LCount2 denote LCount1[n-1] and LCount2[n-1] respectively; <n-1> in $\mathbf{z}_1$ denotes (D,Source[n-1],n-1 mod 2); <n> in $\mathbf{z}_2$ denotes (ACK,n mod 2). It can be easily checked that the following leads-next-to statements satisfy the inference rule for leads-next-to.

(i) (s=r=a+1=n **and** LCount$_1 \geq m_1$ **and** LCount$_2 \geq m_2$) leads-next-to (s=r=a=n **or**
(M1) (s=r=a+1=n **and** LCount$_1 \geq m_1$ **and** LCount$_2 \geq m_2$ **and** <n-1> in $\mathbf{z}_1$))
via {Rec_ACK, Send_D}

(b) M1 leads-next-to (s=r=a=n **or**
(M2) (s=r=a+1=n **and** LCount$_1 \geq m_1$+1 **and** LCount$_2 \geq m_2$)
or
(M3) (s=r=a+1=n **and** LCount$_2 \geq m_2$ **and** SendACK = True))
via {Rec_ACK, Loss event of C_1, Rec_D}

(iii) M3 leads-next-to (s=r=a=n **or**
(M4) (s=r=a+1=n **and** LCount$_2 \geq m_2$ **and** <n> in $\mathbf{z}_2$)
via {Rec_ACK, Send_ACK}

(iv) M4 leads-next-to (s=r=a=n
or (s=r=a+1=n **and** LCount$_2 \geq m_2$+1)
via {Rec_ACK, Loss event of C_2}

From the inference rule for leads-to, (i), (ii), (iii) and (iv) imply the following inductive leads-to property:

(v) (s=r=a+1=n **and** LCount$_1 \geq m_1$ **and** LCount$_2 \geq m_2$) leads-to
(s=r=a=n **or** (s=r=a+1=n **and** LCount$_1 \geq m_1$+1 **and** LCount$_2 \geq m_2$)

or (s=r=a+1=n **and** $LCount_2 \geq m_2+1$))

From the induction rule for the case l=2, (v) implies L2 (let $LCount_2 \geq m_2$ be D_1 and let $LCount_1 \geq m_1$ be D_2). **End of proof**

The two liveness properties L1 and L2 together imply L_0.

Proof of L_0

Applying leads-to property 2 to L1, L2, we have

(s=a+1 **and** r=a) leads-to (s=r=a **or** LCount1 $\geq m_1$ **or** LCount2 $\geq m_2$)

Applying leads-to property 1 to the above and L2, we have

(s=a+1 **and** $a \leq r \leq a+1$) leads-to (s=a **or** LCount1 $\geq m_1$ **or** LCount2 $\geq m_2$)

Since ($a \leq r \leq a+1$) is invariant, it can be deleted from the above, leading to L_0. **End of proof**

7. REAL-TIME DATA TRANSFER PROTOCOL

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, let $Delay_i (\leq MaxDelay_i)$ be the delay that a message is *expected* to encounter in channel C_i ($Delay_i \ll MaxDelay_i$ for a realistic channel). Third, entity P_1 transmits data block Source[n-1] as soon as it is accepted, and retransmits it once every RoundTripDelay time units until it is acknowledged, where RoundTripDelay = $Delay_1$ + $Delay_2$ + MaxResponseTime. P_1 retransmits Source[n-1] at most MaxRetryCount number of times, after which P_1 aborts the connection (enters a state called LinkDown).

We say that a message m in C_i is *overdelayed* if it is not received within $Delay_i$ time of its send. Note that if $Delay_i = MaxDelay_i$ then overdelaying message m corresponds to losing m and any of its duplicates. For this more realistic model, we will prove the following: within a time T (= RoundTripDelay × MaxRetryCount) of P_1 accepting data block Source[n-1], either that data block is acknowledged or [the number of times that C_1 has overdelayed the message (D,Source[n-1],n-1 mod 2)] + [the number of times that C_2 has overdelayed the message (ACK, (n+1) mod 2)] exceeds MaxRetryCount.

7.1 Modified Protocol

To implement the above real-time behavior, the previous data transfer protocol is modified as follows. At P_2, let SendACKTimer be a local timer which is reset to Off in the Send_ACK event and reset to 0, if it was Off, in the Rec_D event. SendACKTimer is initially Off. SendACKTimer is constrained by the timer axiom

SendACKTimer $\leq$ MResponseTime

where MResponseTime = $((1-\epsilon_2) \times$ MaxResponseTime - 1) and ϵ_2 is the maximum error rate in the local time event of P_2. Let SendACKTimerG be an auxiliary ideal timer that is reset along with SendACKTimer.

We have the following modifications at P_1. First, let LinkDown be a boolean variable which is set to True when the current outstanding data block Source[s-1] has been sent MaxRetryCount number of times and no acknowledgement has been received within RoundTripDelay of the last send. LinkDown is initially False. Once LinkDown = True, P_1 does not send or receive any more messages. Second, let DTimer be constrained by the timer axiom

(LinkDown=False **and** vs$\neq$va) $\Rightarrow$ DTimer $\leq$ RTripDelay

where RTripDelay = $(1+(1+\epsilon_1) \times$ RoundTripDelay). (D,Source[s-1], vs $\ominus$ 1) will be retransmitted when vs $\neq$ va and DTimer = RTripDelay. Third, let the AcceptData event also transmit the accepted data block. Fourth, for the current value of s, let the variable RetryCount indicate the number of times that (D,Source[s-1], vs $\ominus$ 1) has been sent. RetryCount is initially 0.

In addition to the above variables, we need to define auxiliary variables in order to formally state and verify the desired real-time property.

Trynumber: 0..MaxRetryCount; {An auxiliary field in each D message. Set to the (updated) value of RetryCount when the D message is sent}

DRecd: {sequence of the try numbers of currently outstanding D messages received at P_2. DRecd is set to empty when a equals s. When P_2 receives a D message, the try number of the received D message is appended in DRecd if a<s holds at that time. DRecd is initially empty}

AckSent: {Sequence of ideal timers indicating the times elapsed since transmissions of acknowledgements to the currently outstanding data block Source[s-1]. AckSent is set to empty when a equals s. AckSent is updated by global time value 0 when P_2 sends (ACK,vr) and vr = vs $\neq$ va. AckSent is initially empty}

$SCount_1$: integers; {number of times that C_1 overdelays message (D,Source[s-1], vs $\ominus$ 1) while Source[s-1] is outstanding. Set to 0 when a equals s. $SCount_1$ is incremented by 1 whenever a global tick occurs and DTimerG = $Delay_1$ but RetryCount is not in DRecd for any t. $SCount_1$ is intially 0}

$SCount_2$: integers; {number of times that C_2 overdelays message (ACK,vs) while Source[s-1] is outstanding. Set to 0 when a equals s. $SCount_2$ is incremented by 1 whenever a global tick occurs and there exists a time value equal to $Delay_2$ in ACKSent but a<s holds. $SCount_2$ is initially 0}

OutTimer: (Off,0,1,...); {Local timer of P_1 which when active indicates the local time elapsed since Source[s-1] became outstanding. OutTimer is initially Off}

The events of the protocol system are now specified. The AcceptData event, which now also sends the accepted data block, has been renamed as SendNewData.

Events of P_1

1. SendNewData($\mathbf{v}_1,\mathbf{z}_1;\mathbf{v}_1",\mathbf{z}_1"$)
   ≡ LinkDown=False
   and vs=va **and** DTimer=Off **and** ATimer=Off {If new data can be accepted}
   and Source[s]" ∈ DataSet **and** s"=s+1 **and** vs"=vs ⊕ 1 {then do so}
   and $Send_1$((D,Source[s],vs,1), $\mathbf{z}_1$; $\mathbf{z}_1$") {and send it with try number 1}
   and DTimer"=DTimerG"=0 **and** RetryCount"=1 **and** OutTimer"=0

2. Send_D($\mathbf{v}_1,\mathbf{z}_1;\mathbf{v}_1",\mathbf{z}_1"$)
   ≡ LinkDown=False
   and vs ≠ va **and** DTimer=RTripDelay **and** RetryCount<MaxRetryCount
   and $Send_1$((D,Source[s-1],vs-1,RetryCount), $\mathbf{z}_1;\mathbf{z}_1"$) {send outstanding data}
   and DTimer"=0 **and** DTimerG"=0 {start DTimer}
   and RetryCount"=RetryCount+1

3. Abort($\mathbf{v}_1;\mathbf{v}_1"$)
   ≡ LinkDown=False
   and vs ≠ va **and** DTimer=RTripDelay **and** RetryCount=MaxRetryCount
   and LinkDown"=True

4. Rec_ACK($\mathbf{v}_1,\mathbf{z}_2;\mathbf{v}_1",\mathbf{z}_2"$)
   ≡ LinkDown=False **and** Rec_2($\mathbf{z}_2$;(ACK,nr),$\mathbf{z}_2$") {Receive nr}
   and ((vs≠va **and** nr=va ⊕ 1) → {if outstanding data acknowledged}
   (a"=a+1 **and** va"=va ⊕ 1 {then update state}
   and ATimer"=ATimerG"=0 **and** RetryCount"=0
   and OutTimer"=0ff **and** DRecd"=empty {and auxiliary variables}
   and ACKSent"=empty **and** $SCount_1$=$SCount_2$=0))

Events of P_2

1. Send_ACK (v_2,z_2;v_2",z_2")
   ≡ SendACK = True **and** Send$_2$ ((ACK,vr), z_2;z_2")
   and SendACK"=False **and** SendACKTimer"=SendACKTimerG"=Off
   and (r=s=a+1 → ACKSent"=(ACKSent,0)) {update auxiliary variables}

2. Rec_D (v_2,z_1;v_2",z_1")
   ≡ Rec$_1$ (z_1;(D,data,ns,try number),z_1")
   and (ns=vr → {if next expected sequence number}
   (Sink[r]"=data **and** r"=r+1 **and** vr"=vr ⊕ 1)) {then accept data}
   and SendACK" = True **and**
   and (SendACKTimer=Off → SendACKTimer"=SendACKTimerG"=0)
   and (s=a+1 → DRecd"=(DRecd, try number)

Time Events

The ideal time event is defined by the conjunct of the previous ideal time event predicate (in Section 5) and the following:

AccuracyAxiom$_2$(η_2,η")
and ACKSent" = next(ACKSent)
and SendACKTimerG" = next(SendACKTimerG)
and (DTimerG = Delay$_1$ **and** RetryCount not in DRecd
→ SCount$_1$" = SCount$_1$ + 1)
and (Delay$_2$ **in** ACKSent → SCount$_2$" = SCount$_2$ +1)

The local time event for P_1 is the conjunct of the previous local time event predicate (in Section 5) with the timer axiom for DTimer (stated above).

The local time event for P_2 is
η_2" = η_2 + 1 **and** AccuracyAxiom$_2$(η_2,η)
and SendACKTimer" = next(SendACKTimer)
and SendACKTimer" ≤ MResponseTime

7.2 Verification of Real-Time Property

The desired real-time property can be stated as follows:

D0 (a) LinkDown = False ⇒ OutTimer ≤ MaxRetryCount × RTripDelay

(b) LinkDown = True
⇒ (OutTimer ≥ MaxRetryCount × RTripDelay **and** SCount$_1$ + SCount$_2$ ≥ MaxRetryCount)

Notice that this real-time property is a safety property and not a liveness property requiring the leads-to operator.

Observe that the modified protocol is a *refinement* of the previous data transfer protocol in the following sense: For each event e_{mod} in the modified protocol, either there is a corresponding event e in the previous protocol such that $e_{mod}(\mathbf{v};\mathbf{v}") \Rightarrow e(\mathbf{v};\mathbf{v}")$, or e_{mod} does not affect any variables of the previous protocol (the Abort event falls in this category). Thus, the safety property A that was shown to be invariant for the previous protocol continues to be invariant for the modified protocol.

The safety assertions D1-D7 below satisfy the safety inference rule, given that A is invariant (details of proof in Appendix C). D1 and D2 imply D0, which is therefore verified.

D1. LinkDown = False ⇒

(a) ((OutTimer=Off **and** s=a **and** RetryCount=0
and $SCount_1$=$SCount_2$=0 **and** DRecd=ACKSent=empty)

(b) **or** (s=a+1 **and** ((DTimerG,DTimer) started at 0
and RetryCount≥1
and OutTimer = (RetryCount - 1)×RTripDelay + DTimer

D2. LinkDown = True ⇒ (OutTimer ≥ MaxRetryCount × RTripDelay
and $SCount_1$ + $SCount_2$ ≥ MaxRetryCount)

D3. (a) trynumber in DRecd ⇒ 1 ≤ trynumber ≤ RetryCount
(b) ((D,Source[s-1], vs ⊖ 1, trynumber) in $\mathbf{z}_1$ **and** s=a+1) ⇒
1 ≤ trynumber ≤ RetryCount

D4. 0 ≤ DTimerG ≤ $Delay_1$
⇒ ($SCount_1$ + $SCount_2$ ≥ RetryCount -1
and (RetryCount in DRecd ⇒ (r=s=a+1
and (SendACKTimerG ≥ 0 **or** (for some t in ACKSent)[t ≥ 0])))

D5. $Delay_1$ < DTimerG ≤ $Delay_1$ + MaxResponseTime
⇒ ($SCount_1$ + $SCount_2$ ≥ RetryCount
or ($SCount_1$ + $SCount_2$ ≥ RetryCount - 1 **and** r=s=a+1
and (SendACKTimerG ≥ DTimerG - $Delay_1$
or (for some t in ACKSent)[t ≥ 0])))

D6. $Delay_1$ + MaxResponseTime < DTimerG ≤ RoundTripDelay
⇒ ($SCount_1$ + $SCount_2$ ≥ RetryCount
or ($SCount_1$ + $SCount_2$ ≥ RetryCount - 1 **and** r=s=a+1
and (for some t in ACKSent)

$$[t \geq DTimerG - (Delay_1 + MaxResponseTime)]))$$

D7. $DTimerG > RoundTripDelay \Rightarrow SCount_1 + SCount_2 \geq RetryCount$

8. CONCLUSION

We have developed a model for specifying time-dependent distributed systems, and presented inference rules for verifying safety and liveness properties. Our proofs can be checked by automated techniques. A data transfer protocol example was used to illustrate the model and application of inference rules.

Our prime objective in developing this model was to be able to specify, verify and ultimately construct realistic communication network protocols. In [20], we have constructed, starting from desired safety requirements, a time-dependent data transfer protocol that generalizes the protocol in this paper to one with an arbitrary window size of N. In [18], we used a preliminary version of this model to specify and verify a version of the HDLC protocol, complete with all its principal functions and mechanisms, including cyclic sequence numbers, polling, checkpointing, connection management, and flow control.

Our model has several distinctive features. First, measures of real time are explicit in our model. Real-time constraints enforced within individual processes are incorporated into the time and system events of the process. Unlike other models [9, 11, 16] which explicitly incorporate measures of time, our model does not require that an enabled event *must* occur, and thereby avoids an explicit scheduling of event occurrences. Derived real-time properties which hold for the distributed system can be stated as safety assertions, and formally verified using inference rules. Our system specifications can then be implemented by programmers who may not be familiar with system analysis. It is our experience that measures of time are central to the working and understanding of communication protocols. Utilizing them is not only realistic, but, perhaps contrary to popular notion, it also greatly simplifies the specification, verification and construction of communication protocols [20].

Second, our specification of processes involves state information within the processes. In particular, we do not specify a process strictly in terms of the *trace* of message communications that the process has already performed with its environment [4, 12, 17]. We did not adopt trace specifications because we find that the lack of internal state structure typically results in large and cumbersome specifications of processes, especially when considering real-life aspects of communication protocols such as bounded capacity channels, flow control, real-time constraints, connection management, etc. This effect has been pointed out in [17] for the case of the alternating-bit protocol. A secondary reason why we did not adopt traces is that the implementor often prefers guidance (and the resulting constraints) to the total independence and lack of guidance resulting from trace specifications of processes.

The third major difference is in our inference rules. To verify a system property A_0 (either safety or liveness), it is neccessary to obtain an assertion A which implies A_0 and satisfies the requirements of the inference rules. The difficulty of verification lies in obtaining A. Invariably, one obtains A as the end product in a sequence of approximations A_1, A_2,... Each successive A_i is determined by examining how the earlier approximations failed to satisfy the inference rules. One approach that is popular in the distributed systems area is to insist that each A_i be expressed as a collection of properties $A_{i,j}$, where j ranges over the processes, such that each $A_{i,j}$ can be verified by examining the behavior of process j alone [4, 12, 13]. This approach, which will be refered to as the process decomposition approach, is often used in conjunction with trace specifications of processes.

Our experience in verifying distributed systems, and communication protocols in particular, has been that it is easier to obtain A (as well as to present a formal proof of it) if we do not have to decompose each A_i into such $A_{i,j}$'s. For this reason, our inference rules do not follow the process decomposition approach. Instead, the inputs to our inference rules are assertions that can involve state variables in different processes, and events that specify changes to the values of variable. *Events, and not processes, are the units of composition in our system model.* Note that each event affects only a small part of the distributed system. Therefore, when applying an inference rule to an assertion A_i, most of A_i is not affected and need not be examined. Thus, the presence of distribution does in fact help our verification, even though it is not formally captured in the syntax of our inference rules. The comments above apply also to the construction of communication protocols, and not only to their verification [20].

The combination of an event-driven system model and the use of predicates to specify events results in some rather simple inference rules for safety and liveness properties. It reduces the gap between specification and verification, and has proved to be useful in protocol construction [20]. The use of predicates to specify events follows the recent trend towards logic programming languages [3, 7], and should not present difficulty in implementation. This is especially so in communication network systems, where each event corresponds to the limited amount of processing associated with the handling of a message transfer or timeout condition.

Protocol projections

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 reduce the analysis of a multifunction protocol system into analyses of smaller single-function protocol systems, called image protocol systems [10]. The theory of projections was originally developed in [10] using a set-theoretic notation. In a companion paper [19], we specialize this theory to the time-dependent system model herein. The use of state variables and predicates (to specify events) greatly facilitates the construction of image protocol systems.

REFERENCES

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

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

[3] Ferguson, R., "PROLOG: A Step Toward the Ultimate Computer Language," *Byte,* Vol. 6, No. 11, Nov. 1981, pp. 384-399.

[4] Hoare, C. A. R., "A Calculus of Total Correctness for Communicating Processes," *Science of Computer Programming,* 1, 1981, pp. 49-72.

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

[6] 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.

[7] Kowalski, R., *Logic for Problem Solving,* Elsevier North-Holland, Amsterdam, 1979.

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

[9] Lamport, L., "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," *ACM Trans. Prog. Lang. Syst.,* Vol. 6, 2, April 1984, 254-280.

[10] 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.

[11] Merlin, P. and D. Farber, "A Methodology for the Design and Implementation of Communications Protocols," *IEEE Trans. Commun.,* Vol. COM-24, 6, June 1976.

[12] Misra, J. and K. M. Chandy, "Proofs of Networks of Processes," *IEEE Trans. Soft. Eng.,* Vol. SE-7, No. 4, July 1981.

[13] Owicki, S. and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Commun. ACM,* Vol. 19, No. 5, May 1976.

[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] Razouk, R. R., "The Derivation of Performance Expressions for Communication Protocols from Timed Petri Net Models," *Proc. IFIP 4th Int. Workshop on Protocol Specification, Verification and Testing*, Columbia University, June 11-14, 1984, North-Holland, 1985.

[17] Schwartz, R. L. and P. M. Melliar-Smith, "From state machines to temporal logic: Specification methods for protocol standards," *IEEE Trans. Commun.*, Vol. COM-30, pp. 2486-2496, Dec. 1982.

[18] 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.

[19] Shankar, A. U. and S. S. Lam, "Verification of Communication Networks via Projections," Tech. Rep. in preparation, Dept. of Computer Sciences, Univ. of Texas at Austin, 1985.

[20] Shankar, A. U. and S. S. Lam, "An Exercise in Protocol Construction," Tech. Rep. in preparation, Dept. of Computer Sciences, Univ. of Texas at Austin, 1985.

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

Appendix A

Proof of Theorem 1

The accuracy and timer axioms hold initially (from IC1). No system event can reset the time event counts or timers to values that can violate the axioms (from IC2 and definition of time events). Thus, the axioms are invariant and part (a) is established.

We now prove part(b), i.e., that time events do not deadlock. Let **s** be any reachable system state where all the time events are blocked. Because **s** is reachable we know that all the accuracy and timer axioms hold at state **s**. From condition IC3, we know

that there exists a sequence of enabled events that will take the system from state **s** to state **r** where TimerAxiom(next(**v**)) = True for every timer axiom in the system. Thus, from the definitions of the ideal and local time events, we see that the only way that all time events can be blocked at state **r** is if

(for all η_i)[AccuracyAxiom$_i$(η_i+1,η) = False]

and (for some η_i)[AccuracyAxiom$_i$(η_i,η+1) = False].

Thus, we prove the theorem by establishing the following: AccuracyAxiom$_{\eta_i}$(η_i+1,η) = False $\Rightarrow$ AccuracyAxiom$_{\eta_i}$(η_i,η+1) = True.

Let the current values of η_i and η be denoted by η_i(c) and η(c). Now, AccuracyAxiom$_{\eta_i}$ (η_i,η) For all instants a and b where b is later than a, $|(\eta_i(b) - \eta_i(a)) - (\eta(b) - \eta(a))| \leq \max(1,\epsilon_i(\eta(b) - \eta(a)))$. Because the AccuracyAxiom has held at every instant so far, the only way that AccuracyAxiom$_{\eta_i}$(η_i+1,η) = False, is if there is some instant a in the past (i.e. somewhere in the current execution path) such that

$$|(\eta_i(c) + 1 - \eta_i(a)) - (\eta(c) - \eta(a))| > \max(1,\epsilon_i(\eta(c) - \eta(a))). \quad \text{(A1)}$$

We know that $|(\eta_i(c) - \eta_i(a)) - (\eta(c) - \eta(a))| \leq \max(1,\epsilon_i(\eta(c) - \eta(a)))$ (A2)

Similarly, AccuracyAxiom$_{\eta_i}$(η_i,η+1) = False implies that there is some instant b in the past such that

$$|(\eta_i(c) - \eta_i(b)) - (\eta(c) + 1 - \eta(b))| > \max(1,\epsilon_i(\eta(c) - \eta(b))). \quad \text{(A3)}$$

and

$$|(\eta_i(c) - \eta_i(b)) - (\eta(c) - \eta(b))| \leq \max(1,\epsilon_i(\eta(c) - \eta(b))). \quad \text{(A4)}$$

For brevity, let η(a), η(b), η(c), η_i(a), η_i(b), η_i(c), be denoted respectively by t_a, t_b, t_c, u_a, u_b, u_c.

From A1, either

$(u_c + 1 - u_a) - (t_c - t_a) > \max(1,\epsilon_i(t_c - t_a))$

or $(u_c + 1 - u_a) - (t_c - t_a) < -\max(1,\epsilon_i(t_c - t_a))$

But the second alternative means that

$(u_c - u_a) - (t_c - t_a) < -\max(1,\epsilon_i(t_c - t_a))$,

which violates A2. Hence only the first alternative can hold.

A3 implies either

$(u_c - u_b) - (t_c + 1 - t_b) > \max(1,\epsilon_i(t_c + 1 - t_b))$,

or $(u_c - u_b) - (t_c + 1 - t_b) < -\max(1,\epsilon_i(t_a + 1 - t_b))$

The first alternative implies that

$(u_c - u_b) - (t_c - t_b) > \max(1,\epsilon_i(t_c + 1 - t_b)) \geq \max(1,\epsilon_i(t_c + 1 - t_b))$,

which violates A4. Hence, only the second alternative can hold.

Thus, we now have to derive a contradiction from

$$(u_c + 1 - u_a) - (t_c - t_a) > \max(1,\epsilon_i(t_c-t_a)) \quad \text{(A5)}$$

and $$(u_c - u_b) - (t_c + 1 - t_b) > -\max(1,\epsilon_i(t_c+1-t_b)) \quad \text{(A6)}$$

Before deriving the contradiction, we will next show that $| \epsilon_i(t_a - t_b) | > 1$.

From A5, we have

$(u_c - u_a) - (t_c - t_a) > \max(1,\epsilon_i(t_c-t_b)) \geq 0.$

This implies

$$(u_c - u_a) - (t_c - t_a) \geq 1 \quad \text{(A7)}$$

From A6, we have

$(u_c - u_b) - (t_c - t_b) < -\max(1,\epsilon_i(t_c+1-t_b)) + 1 \leq 0.$

This implies

$$(u_c - u_b) - (t_c - t_b) \leq -1 \Rightarrow (t_c - t_b) - (u_c - u_b) \geq 1 \quad \text{(A8)}$$

Adding A7 and A8, we have

$$(t_a - t_b) - (u_a - u_b) \geq 2 \quad \text{(A9)}$$

Since AccuracyAxiom holds at instants a and b, we know that

$$\epsilon_i(t_a - t_b) > 1 \quad \text{(A10)}$$

(This simplifies the proof in that we do not have to consider $\epsilon_i(t_a - t_b) \leq 1$.)

Case 1: Assume a is later than b

Case 1(a): $\epsilon_i(t_c - t_a) < 1$. Since $\epsilon_i(t_a - t_b) > 1$ (from A10), we have $\epsilon_i(t_c - t_b) > 1$.

A5 reduces to

$$(u_c + 1 - u_a) - (t_c - t_a) > 1$$

$$\Rightarrow (u_c - u_a) - (t_c - t_a) \geq 1$$

$$\Rightarrow (t_c - t_a) - (u_c - u_a) \leq -1$$

A6 reduces to

$$(u_c - u_b) - (t_c - t_b) - 1 < -\epsilon_i(t_c-t_b+1)$$

Adding the two, we get

$$(u_a - u_b) - (t_a - t_b) < -\epsilon_i(t_c-t_b+1)$$

Now $-\epsilon_i(t_c-t_b+1) = -\epsilon_i(t_c-t_a+t_a-t_b) - \epsilon_i$

$= -\epsilon_i(t_c-t_a) - \epsilon_i(_a-t_b) - \epsilon_i$

$< -\epsilon_i(t_a-t_b)$

This contradicts the AccuracyAxiom at instants a and b.

Case 1(b): $\epsilon_i(t_c-t_a) \geq 1$

A5 reduces to

$(u_c - u_a) - (t_c - t_a) + 1 > \epsilon_i(t_c - t_a)$

$\Rightarrow (t_c - t_a) - (u_c - u_a) - 1 < -\epsilon_i(t_c - t_a)$

A6 reduces to

$(u_c - u_b) - (t_c - t_b) - 1 < \epsilon_i(t_c - t_b + 1)$

Adding the two, we get

$(u_a - u_b) - (t_a - t_b) - 2 < -\epsilon_i(2t_c - t_a - t_b + 1) = -\epsilon_i(2(t_c - t_a) + (t_a - t_b) + 1)$

From conditions of Case 1(b) we have $\epsilon_i(t_c - t_a) \geq 1$. Hence, RHS $\leq -2 - \epsilon_i(t_a - t_b + 1)$. Thus, we have $(u_a - u_b) - (t_a - t_b) < -\epsilon_i(t_a - t_b + 1)$. Again, we have a contradiction with the AccuracyAxiom at instants a and b.

Case 2: Assume b is later than a

Case 2(a): $\epsilon_i(t_c - t_b) < 1$. Since $\epsilon_i(t_b - t_a) > 1$ (from A10), we have $t_i(t_c - t_a) > 1$. A5 reduces to (as in Case 1(b)) $(t_c - t_a) - (u_c - u_a) - 1 < -\epsilon_i(t_c - t_a)$

A6 reduces to

$(u_c - u_b) - (t_c - t_b) - 1 < -1$

$\Rightarrow (u_c - u_b) - (t_c - t_b) \leq -1$

Adding the two, we get

$(t_b - t_a) - (u_b - u_a) < -\epsilon(t_c - t_a) < -\epsilon(t_b - t_a)$

which contradicts the AccuracyAxiom at instants a and b.

Case 2(b): $\epsilon_i(t_c - t_b) \geq 1$.

This is like Case 1(b), and we have

$(u_a - u_b) - (t_a - t_b) - 2 < -\epsilon_i(2t_c - t_a - t_b + 1)$

$\Rightarrow (u_b - u_a) - (t_b - t_a) > \epsilon_i(2t_c - t_b + t_b\ t_a) + 1) - 2$

Since $\epsilon(t_c - t_b) \geq 1$, we have $(u_b - u_a) - (t_b - t_a) > \epsilon_i(t_b - t_a)$, which contradicts the AccuracyAxiom at instants a and b.

End of proof of Theorem 1

Appendix B

Modeling a variety of channels

Recall that a channel C_i with state variable $\mathbf{z}_i$ has three predicates specifying it: $Send_i(m,\mathbf{z}_i;\mathbf{z}_i")$, $Rec_i(\mathbf{z}_i;m,\mathbf{z}_i")$, and $ChannelError(\mathbf{z}_i;\mathbf{z}_i")$.

Infinite-buffer, finite-buffer blocking, and finite-buffer loss channels can be modeled by appropriate Send primitives.

1. *Infinite-buffer channel.* This is the one modeled in Section 4.
   $\text{Send}_i(m,\mathbf{z}_i;\mathbf{z}_i") \equiv (\mathbf{z}_i" = (\mathbf{z}_i,m))$

2. *Finite-buffer blocking channel.* Send is blocked if the channel is full.
   $\text{Send}_i(m,\mathbf{z}_i;\mathbf{z}_i") \equiv (|\mathbf{z}_i| < J \textbf{ and } \mathbf{z}_i" = (\mathbf{z}_i,m))$,
   where $|\mathbf{z}_i|$ denotes the length of $\mathbf{z}_i$ and J denotes the channel capacity.

3. *Finite-buffer loss channels.* Sending of a message into a full channel causes a message (either the new one or one already in the channel) to be lost.
   $\text{Send}_i(m,\mathbf{z}_i;\mathbf{z}_i") \equiv$ (for some message sequences **a,b,c**)(for some message n)
   $[(\mathbf{a}=(\mathbf{z}_i,m))$ **and** $(|\mathbf{a}| \leq J \Rightarrow \mathbf{z}_i" = \mathbf{a})$
   and $(|\mathbf{a}| = J+1 \Rightarrow (\mathbf{a}=(\mathbf{b},n,\mathbf{c})$ **and** $\mathbf{z}_i" = (\mathbf{b},\mathbf{c})))]$

In the above, we have assumed $\mathbf{z}_i$ is a sequence of messages. If $\mathbf{z}_i$ is a sequence of <message,age> pairs, then m is replaced by <m,0> in the body of the send primitive.

Minimum delay channels can be modeled by an appropriate Rec primitive. For channel C_i, let state variable $\mathbf{z}_i$ be the sequence of <message,age> pairs. Then, a minimum delay of D can be modeled by

$\text{Rec}_i(\mathbf{z}_i;m,\mathbf{z}_i") \equiv$ (for some t)$[(\mathbf{z}_i=(<m,t>,\mathbf{z}_i"))$ **and** $t \geq D]$

Recall that the internal behavior of channel C_i is specified by $\text{ChannelError}(\mathbf{z}_i;\mathbf{z}_i)$. We formally specify different types of channel errors below (**a**, **b**, **c** are existentially quantified over sequences of messages, while m is existentially quantified over messages).

$\text{Loss}(\mathbf{z}_i;\mathbf{z}_i") \equiv (\mathbf{z}_i=(\mathbf{a},m,\mathbf{b})$ **and** $\mathbf{z}_i"=(\mathbf{a},\mathbf{b}))$
$\text{Duplicate}(\mathbf{z}_i;\mathbf{z}_i") \equiv (\mathbf{z}_i=(\mathbf{a},m,\mathbf{b})$ **and** $\mathbf{z}_i" = (\mathbf{a},m,m,\mathbf{b}))$
$\text{Reorder}(\mathbf{z}_i;\mathbf{z}_i") \equiv ((\mathbf{z}_i = (\mathbf{a},m,\mathbf{b},\mathbf{c})$ **and** $\mathbf{z}_i" = (\mathbf{a},\mathbf{b},m,\mathbf{c}))$
or $(\mathbf{z}_i = (\mathbf{a},\mathbf{b},m,\mathbf{c})$ **and** $\mathbf{z}_i" = (\mathbf{a},m,\mathbf{b},\mathbf{c}))$

We can have combinations of the above; e.g.,

$\text{ChannelError}(\mathbf{z}_i;\mathbf{z}_i") \equiv (\text{Loss}(\mathbf{z}_i,\mathbf{z}_i")$ **or** $\text{Duplicate}(\mathbf{z}_i;\mathbf{z}_i")$ **or** $\text{Reorder}(\mathbf{z}_i;\mathbf{z}_i"))$

Appendix C

Proof that D satisfies safety inference rule

Throughout this proof, safety property A is assumed to be invariant, as also are the timer and accuracy axioms.

Initial Conditions

OutTimer = DTimer = Off, s=a=0, $SCount_1$ = $SCount_2$ = 0, RetryCount = 0, DRecd = ACKSent = empty, $\mathbf{z}_1$ = $\mathbf{z}_2$ = empty. Thus, D1 holds non vacuously, while D2-D7 hold vacuously.

SendNewData

(i)	LinkDown = LinkDown" = False	(SendNewData)
(ii)	OutTimer" = DTimer" = DTimerG" = 0	(SendNewData)
(iii)	RetryCount" = 1	(SendNewData)
(iv)	s" = s + 1 = a + 1	(SendNewData)
(v)	$\mathbf{z}_1$"=$\mathbf{z}_1$, ((D,Source[s"-1], vs" $\ominus$ 1,1),age)	(SendNewData)
(a)	D1"	(from (i), (ii), (iii), (iv))
(b)	D2" holds vacuously	(from (i))
(c)	D3"(a)	(from D3(a))
(d)	D3"(b)	(from D3(b) and (v))
(e)	D4"	(from (iii), DRecd = empty (from D1(a)))
(f)	D5", D6", D7" hold vacuously	(from DTimerG" = 0 (from (ii)))

Send_D

(i)	LinkDown = LinkDown" = False	(Send_D)
(ii)	DTimer = RTripDelay	(Send_D)
(iii)	DTimer" = DTimerG" = 0	(Send_D)
(iv)	RetryCount" = RetryCount + 1	(Send_D)
(v)	s" = s = a + 1 = a" + 1	(Send_D)
(vi)	$\mathbf{z}_1$" = $\mathbf{z}_1$, ((D,Source[s"-1], vs" $\ominus$ 1, RetryCount")	(Send_D)
(a)	D1" holds	(from D1(b), (ii), (iii), (iv), (v))
(b)	D2" holds vacuously	(from (i))
(c)	D3" holds	(from D3, (vi))
(d)	DTimerG > RoundTripDelay	(from (ii), ((DTimerG,DTimer) started at 0) (from D1(b)), Started-at property)
(e)	$SCount_1$ + $SCount_2$ $\geq$ RetryCount	(from d, D7)
(f)	D4"	(from e, (iv), D4)
(g)	D5" - D7" hold vacuously	(from (iii))

Abort

(i)	LinkDown = False	(Abort)
(ii)	LinkDown" = True	(Abort)
(iii)	DTimer"=DTimer = RTripDelay, s=s"=a"+1=a+1	(Abort)
(iv)	RetryCount = MaxRetryCount	(Abort)
(a)	D1" holds vacuously	(from (ii))
(b)	DTimerG > RoundTripDelay	(from (ii), ((DTimerG,DTimer) started

at 0) (from D1(b)), Started-at property)
(c) D2" holds (from (i), (iii), (iv), D1(b), b, D7)
(d) D3" holds (from D3)
(e) D4" - D6" hold vacuously (from (iii))
(f) D7" holds (from b, D7)

Rec_ACK

Case 1. vs = va **or** nr = va.
D" holds from D

Case 2
(i) vs $\neq$ va **and** nr = va $\oplus$ 1 (Rec_ACK)
(ii) s = s" = a" = a + 1 (Rec_ACK)
(iii) DTimer" = DTimerG" = OutTimer = Off (Rec_ACK)
and ACKSent" = DRecd" = empty
and $SCount_1 = SCount_2" = 0$
(iv) LinkDown = LinkDown" = False (Rec_ACK)

(a) D1" holds (from (ii), (iii), (iv))
(b) D2" - D7" hold vacuously (from (iii), (iv))

Send_ACK
(i) r = s = a + 1 $\rightarrow$ ACKSent" = (ACKSent, 0) (Send_ACK)
(ii) SendACK = True (Send_ACK)

(a) D1", D2", D3" hold (D1 and (i); D2 and D3 not affected)
(b) RetryCount in DRecd $\Rightarrow$ r = s = a + 1 (from D4)
$\Rightarrow$ (for some t in ACKSent")[t = 0] (from (i))
(c) D4" holds (from b, D4)
(d) D5" holds (from D5, b)
(e) D6", D7" hold (D6, D7 not affected)

Rec_D
(i) $\mathbf{z}_1 = (<$(D,data,ns,trynumber),age$>, \mathbf{z}_1")$ (Rec_D)
(ii) SendACKTimerG" $\geq$ 0, SendACKTimer" $\geq$ 0 (Rec_D)
(iii) s = a + 1 $\rightarrow$ DRecd" = DRecd, trynumber (Rec_D)
(iv) s = r" (from A2)

(a) D1" holds (from D1, (iii))
(b) D2" holds (from D2)
(c) D3"(a) holds (from D3(b), D3(a))
(d) D3"(b) holds (D3(b) not affected)
(e) D4" ((RetryCount in DRecd or s=a or trynumber$\neq$RetryCount), D4)

(f) D4" (RetryCount not in DRecd, s=a+1, trynumber=RetryCount, (iv), (ii), (iii))

(g) D5", D6", D7" hold (D5, D6, D7 not affected)

Local Time event for η_1

(a) D1(a)" holds (from D1(a), local time event)

(b) D1(b)" holds (from D1(b), (OutTimer$\neq$Off $\Rightarrow$ OutTimer"=OutTimer+1), (LinkDown=False and s$\neq$a $\Rightarrow$ DTimer"$\leq$RTripDelay $\Rightarrow$ DTimer"=DTimer+1))

(c) D2" - D7" hold (from D2 - D7 not affected)

Local time event for η_2

Does not effect D

Global time event

D1, D2, D3 not affected.

Assume DTimerG $<$ $Delay_1$. Then DTimerG" $\leq$ $Delay_1$

D4" holds (from D4)

D5" - D7" holds vacuously

Assume DTimerG $=$ $Delay_1$. Then DTimerG" $=$ $Delay_1 + 1$

Assume RetryCount in DRecd. Then D5" holds with

$SCount_1 + SCount_2 \geq$ RetryCount -1 (from D4)

Assume RetryCount not in DRecd. Then D5" holds with

$SCount_1 + SCount_2 \geq$ RetryCount (from D4, updating of $SCount_1$ in global time event)

Thus D5" holds

D6", D7" hold vacuously

Assume $Delay_1 <$ DTimerG $<$ $Delay_1$ + MaxResponseTime

Then, D5" holds (from D5, and global time event increments SendACKTimerG, DTimerG, t if they are active)

D4", D6", D7" hold vacuously

Assume DTimerG $=$ $Delay_1$ + MaxResponseTime

Assume $SCount_1 + SCount_2 \geq$ RetryCount. Then D6" holds

Assume $SCount_1 + SCount_2 =$ RetryCount -1. Then, since

SendACKTimerG" $\leq$ MaxResponseTime (from Timer Axiom),

we have (for some t in ACKSent)[t $\geq$ 0] holding.

Thus, (for some t in ACKSent")[t $\geq$ 1] holds.

Thus D6" holds

D4", D5", D7" hold vacuously

Assume $Delay_1$ + MaxResponseTime < DTimerG < RoundTripDelay
Then D6" holds (from D6, and t (if it exists) and DTimerG are both incremented by 1).

Assume DTimerG = RoundTripDelay
Then, from D7, either $SCount_1 + SCount_2 \geq$ RetryCount
or (for some t in ACKSent)[$t \geq Delay_2$].
Then, $SCount_2" = SCount_2 + 1$
(in update of $SCount_2$ in global time event).
D7" holds
D4" - D6" hold vacuously.

Assume DTimerG > RoundTripDelay.
D7" holds (from D7)
D4" - D6" hold vacuously.
End of proof