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
Revised, October 1986

*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.

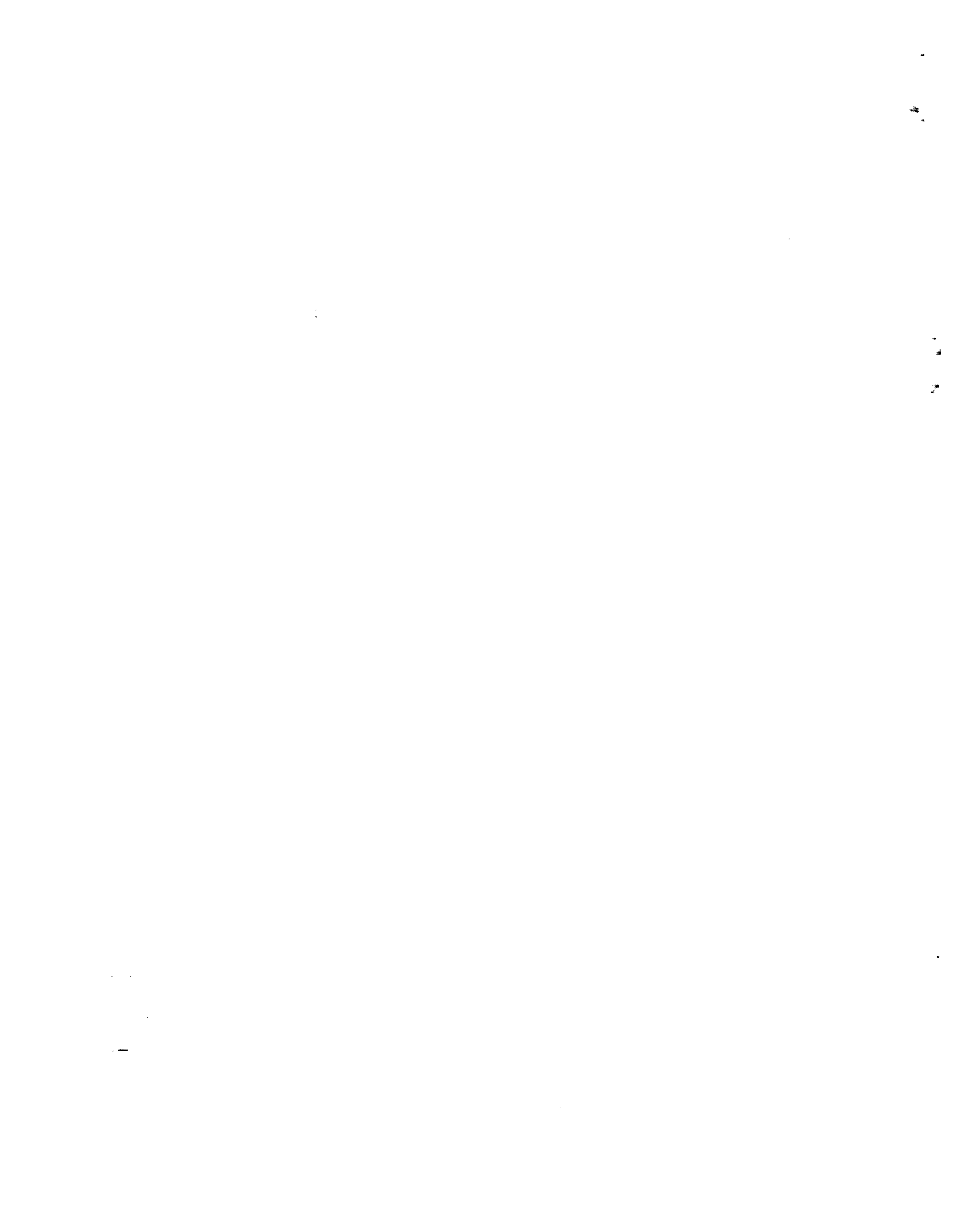

Table of Contents

1. INTRODUCTION **1**
2. EVENT SPECIFICATIONS AND INFERENCE RULES **3**
2.1 Specifying an event-driven system 3
2.2 Proving safety properties 4
2.3 Proving liveness properties 6
2.4 Distributed system model 7
3. A TIME-INDEPENDENT PROTOCOL **8**
3.1 Safety verification 10
3.2 Liveness verification 14
4. REAL-TIME SYSTEM MODEL **15**
4.1 Measures of Time 15
4.2 Implementable Time Constraints 17
4.3 Modeling Real-Time Channels 18
4.4 Derived Time Constraints 19
5. A TIME-DEPENDENT PROTOCOL **19**
5.1 Safety verification 22
5.2 Liveness verification 25
6. A PROTOCOL WITH REAL-TIME PROGRESS **25**
6.1 Protocol specifications 26
6.2 Safety and liveness verification 29
6.3 Real-time progress verification 29
REFERENCES **30**
Appendix A **32**
Appendix B **33**

Abstract

Most communication protocol systems utilize timers to implement real-time constraints between event occurrences. Such systems are said to be *time-dependent* if the real-time constraints are crucial to their correct operation. We present a model for specifying and verifying time-dependent distributed systems. We consider networks of processes that communicate with one another by message-passing. Each process has a set of state variables and a set of events. An event is described by a predicate that relates the values of the network's 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. Real-time progress properties can be verified as safety properties.

We illustrate with three sliding window data transfer protocols that use modulo-2 sequence numbers. The first protocol operates over channels that only lose messages. It is a time-independent protocol. The second and third protocols operate over channels that lose, reorder, and duplicate messages. For their correct operation, it is necessary that messages in the channels have bounded lifetimes. They are time-dependent protocols.

1. INTRODUCTION

Our work has been motivated primarily by communication network protocols which are invariably time-dependent systems [3, 7, 8, 20, 22, 26]. 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 that may be affected by these errors and failures; only by the use of timeouts can a process infer that certain failures or errors have occurred and initiate recovery action.

We present in this paper an event-driven process model for specifying and verifying distributed systems, both time-dependent and time-independent. In this model, events are specified by predicates; there is no algorithmic code. Event specifications can be directly substituted into proofs of safety and liveness properties. Combining this feature with the event-driven structure of the system model, we get simple inference rules for safety and liveness properties, including a lexicographic induction rule for the temporal operator "leads-to" [2, 14, 22]. Using predicates to specify events has been advocated by Lamport [14, 16] as well as by us [22].

To verify a desired safety property A_0, we present a heuristic method to generate a sequence of assertions A_0, A_1, ..., A_n which jointly satisfy the inference rules and imply A_0. Assertion A_i in the sequence is obtained by taking the precondition [4] of an assertion A_j, $0 \leq j < i$, with respect to an event. Unlike the methods in [17, 18, 6], we do not require A_i to be syntactically composed of assertions each of which depends on a single process. We find it more convenient to have *events, rather than processes, as the units of composition*. In this respect, our approach is similar to the recent work of Lamport [13, 16] and Chandy and Misra [2].

Real time modeling

Processes use a special type of state variables, referred to as *timers*, to measure the passage of time since event occurrences in discrete ticks. The timers of a process can be started and stopped by events of that process. Timers of different processes are uncoupled. By imposing conditions, referred to as *accuracy axioms*, we ensure that the timers of a process tick at rates that are within specified error bounds of a constant rate.

We say that a time constraint is *implementable* by a process if it can be enforced by that process alone, without cooperation from the rest of the distributed system. An implementable time constraint of a process of the form "event *e will occur only if* some elapsed times satisfy certain bounds" is modeled by including timers in the enabling conditions of *e*. An implementable time constraint of the form "event *e must occur within* certain elapsed times" is modeled by requiring that the desired time constraints are not violated by the ticking of timers. Note that in our model, it is not required that an enabled event *must* occur. The idea of inhibiting timers from ticking was proposed and investigated by us in [22] and was also suggested by Lamport recently [16].

The implementable time constraints enforced within processes give rise to network-wide time constraints which depend on interactions between processes. With timers, such time constraints can be specified and verified as safety assertions. We have found that such time constraints are very useful for describing progress in communication network protocols. Typically,

if a communication 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 [9]. Hence, a liveness assertion stating progress within a finite but *unbounded* time duration is often inappropriate. More useful 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."

The only time constraints allowed in the specification of a process are implementable ones. Otherwise, a correct implementation of a process would not be possible from the specification of that process alone. 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.

A system with an explicit auxiliary variable indicating the current time was presented by Francez and Pnueli [5]. However, their model was used for establishing liveness properties rather than real-time properties. Subsequent work on liveness properties eschewed explicit modeling of real time [2, 6, 14, 15, 19].

Sliding window protocol examples

The three sliding window protocols presented in this paper use modulo-2 sequence numbers to achieve reliable data transfer between a source and a sink connected by unreliable channels. The first protocol is time-independent and assumes that the channels can only lose messages. This protocol is like the alternating bit protocol considered in [14]; it is slightly different from the original alternating bit protocol [1, 6], which assumes that the channels can corrupt (but not lose) messages. This example illustrates the compactness of a verification in our model, and can be compared with other verifications of the alternating bit protocol [6, 14, 17]. We note that this protocol is a special case of data link protocols that use modulo-N ($N \geq 2$) sequence numbers and assume that channels can only lose messages [8, 11, 21] (Reference [11] also covers the case of channels that reorder messages to a limited extent).

The second and third protocols assume that the channels can lose and duplicate messages as well as reorder messages arbitrarily. For correct operation, it is necessary that message lifetimes have an upper bound. The source must then enforce certain implementable time constraints to achieve correct operation. The resulting protocols are novel. Our second protocol is best compared with the simplified Stenning's protocol [6, 17], which also allows channels to lose, reorder, and duplicate messages. Both protocols maintain at most one outstanding data block at the source, but the simplified Stenning's protocol is a time-independent protocol that is forced to use unbounded sequence numbers.

The original Stenning's protocol [27] allowed arbitrary (but fixed) send and receive window sizes. Based on a formal verification using unbounded sequence numbers, Stenning argued informally that correct operation would result with modulo-N sequence numbers if N exceeded a certain bound in terms of channel message lifetimes, transmission rate, and window sizes. In [24, 25], we have extended the second protocol example by constructing several protocols that use modulo-N sequence numbers for arbitrary $N \geq 2$. In addition to the use of modulo-N sequence numbers, these protocols extend the original Stenning's protocol in several other respects (e.g.,

variable windows for flow control, selective acks, etc.). It has been our experience that the use of timers actually simplifies the verification and construction of communication protocols.

Organization of this report

In Section 2, we describe the predicate specification of events, the safety and liveness inference rules, and the time-independent distributed system model. In Section 3, we present the first protocol example and verify its safety and liveness properties. In Section 4, we describe our modeling of timers and time constraints, and present our time-dependent distributed system model. In Section 5, we present the second protocol example and verify its safety and liveness properties. In Section 6, we refine the second protocol to offer real-time service; i.e., data transfer within a specified time. This real-time service is verified. The refinements are introduced in such a way that the safety and liveness properties of the second protocol continue to hold.

2. EVENT SPECIFICATIONS AND INFERENCE RULES

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**, **or** and $\Rightarrow$ to denote logical conjunction, disjunction and implication respectively. We use *for all* and *for some* to denote universal and existential quantification respectively. Where ambiguity may arise, the scope of the quantification is enclosed by square brackets. Throughout, we assume that for every variable there is a specified domain of allowed values. We use $\equiv$ to denote predicate definition. An example of a predicate definition is p $\equiv$ (for some x_1)[$x_1 = x_2 + 1$ **and** $x_1 = x_3$], where x_1, x_2 and x_3 are integer-valued variables; the free variables of predicate p are x_2 and x_3.

Given a predicate p and a set of variables $\mathbf{x}=(x_1,x_2,...,x_n)$, we say that p is a predicate in $\mathbf{x}$ to indicate that the free variables of p are from $\mathbf{x}$. We can also indicate this by the notation $p(\mathbf{x})$. This notation facilitates substituting expressions for free variables in p. In particular, for any given value of $\mathbf{x}$, we shall also use $p(\mathbf{x})$ to denote the value that the predicate evaluates to. Thus, in the above example, $p(1,2)$ is True while $p(1,1)$ is False.

2.1 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 their 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*. 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. We assume that each event occurrence is atomic; i.e., for the properties of interest, the simultaneous occurrence of multiple events is equivalent to the occurrence of those events in any order [2, 14, 16]. Instead of using algorithmic code, we specify the event e by a predicate in $\mathbf{v}$ and $\mathbf{v}'$, where $\mathbf{v}$ denotes the value of the state vector immediately before the event occurrence, and $\mathbf{v}'$ denotes the value of the state vector immediately after the event occurrence. Such predicates are referred to as *event predicates*. For implementation purposes, the variables $\mathbf{v}$ and $\mathbf{v}'$ can be treated respectively as input and output parameters of an algorithmic procedure. Notice that we use the term "variable" 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.

For example, consider the state vector $\mathbf{v}=(v_1,v_2)$, where v_1 and v_2 are integer-valued state variables. The event $e_1 \equiv (v_1<5$ **and** $v_1'=v_1+v_2$ **and** $v_2'=v_2)$ can be implemented by the algorithmic code "if $v_1<5$ then $v_1:=v_1+v_2$." The event $e_2 \equiv ((v_1'=1$ **or** $v_1'=2)$ **and** $v_2'=v_2)$ can be implemented by the algorithmic code "$v_1:=1 \ \square \ v_1:=2$," where $\square$ represents nondeterministic choice [4]. For compactness in specifying events, we adopt the convention that any variable v' in $\mathbf{v}'$ that does not occur in an event predicate is not affected by the event occurrence; i.e., the conjunct $v'=v$ is implicit in the event predicate. Thus, the above two examples can be written as $e_1 \equiv (v_1<5$ **and** $v_1'=v_1+v_2)$, and $e_2 \equiv (v_1'=1$ **or** $v_1'=2)$.

An event predicate e is said to be *enabled* for a given value of $\mathbf{v}$ if there is a value of $\mathbf{v}'$ such that e evaluates to True for that value pair. The *enabling condition* of e, denoted by *enabled*(e), is defined to be any predicate in $\mathbf{v}$ which is logically equivalent to the predicate (for some $\mathbf{v}'$)[e]. It is very natural to have $e \equiv (e_1$ **and** $e_2)$, where e_1 is a predicate in $\mathbf{v}$, e_2 is a predicate in $\mathbf{v} \cup \mathbf{v}'$, and e_2 is enabled for every value where e_1 is True. In this case, e_1 is the enabling condition of event e and e_2 specifies the "action" of event e.

In addition to state variables needed to model the system, $\mathbf{v}$ can contain auxiliary state variables needed for verification purposes only [18]. Auxiliary variables record the occurrence of events; they do not inhibit event occurrences nor do they affect the resulting values of non-auxiliary variables. Formally, if $\mathbf{v} = \mathbf{u} \cup \mathbf{w}$ where $\mathbf{u}$ are auxiliary variables and $\mathbf{w}$ are not, then the following holds for every event e: for any given value of $\mathbf{w}$ and $\mathbf{w}'$, the value of (for some $\mathbf{u}'$)[e] is independent of the value of $\mathbf{u}$.

2.2 Proving safety properties

A safety property of the event-driven system states relationships between the values of the state variables. It can be represented by a predicate in the variables of the system state vector $\mathbf{v}$. An example of a safety property involving two integer state variables v_1 and v_2 is $(v_1 \leq v_2 \leq v_1+1)$. A safety property A_0 holds for the system if it holds at every system state that is possibly reachable from an initial state. Such a property is said to be *invariant*. We now present the inference rule for proving invariance.

With the execption of event predicates, every predicate that we specify is either entirely in $\mathbf{v}$ or entirely in $\mathbf{v}'$. Thus, we shall refer to a predicate A in $\mathbf{v}$ as simply A, and use A' to denote $A(\mathbf{v}')$. Following convention, $A \Rightarrow B$ holds iff it holds for all values of the free variables in A and B.

Inference rule for safety. If I is invariant and A satisfies

(i) $Initial \Rightarrow A$

(ii) for every event e: (I **and** A **and** e) $\Rightarrow A'$

(iii) $A \Rightarrow A_0$

then we infer that A_0 is invariant.

A_0 represents a desired safety property and I can be any safety property whose invariance has already been verified; $I \equiv$ True if no invariant property is known. A has to be generated from A_0 and I; obviously this is a nontrivial task analogous to generating loop invariants in program verification (see below). Note that the inference rule is quite simple because of our use of predicates to define events.

The validity of the rule is obvious. Part (i) ensures that A holds initially. Because I is invariant, every reachable state satisfies I. Thus, part (ii) ensures that A is preserved by any event occurrence in a reachable system state. Thus A is invariant. Therefore, by part (iii), A_0 is invariant. Because I is given to be invariant, we can replace I by (I **and** I') in part (ii) of the above inference rule; this strengthening of the left hand side sometimes helps in deriving A'.

A heuristic

We now describe a heuristic, based on preconditions [4], which can be used to generate A from A_0, I, and the system specifications. The heuristic builds up iteratively a sequence of assertions A_0, A_1, ..., A_n. At any time, let $A \equiv$ (A_0 **and** ... **and** A_n). If the heuristic terminates successfully, A will satisfy the safety inference rule.

Any predicate P that is logically equivalent to (for all $\mathbf{v}'$)[(A **and** e) $\Rightarrow A_i'$] is referred to as a *weakest precondition* of A_i with respect to e. P is a predicate in $\mathbf{v}$ that is False over only those states in A where e is enabled and whose occurrence may violate A_i. (P corresponds to the weakest liberal precondition of Dijkstra [4].) An assertion that implies a weakest precondition is referred to as a *precondition*.

In the heuristic, an event-assertion pair (e, A_i), $0 \leq i \leq n$, is *marked* when we ensure that ($Initial \Rightarrow A_i$) and ((A **and** e) $\Rightarrow A_i'$) hold; otherwise, it is *unmarked*.

initially:

If we have an invariant assertion I at the beginning, then A consists of the two assertions A_0 and $A_1 \equiv I$, and $n=1$. For every event e, (e, A_0) is unmarked and (e, A_1) is marked.

If we do not have a (nontrivial) invariant I at the beginning, then A consists of the single assertion A_0, and n=0.

while there is an unmarked (e,A_i) **do**
generate a weakest precondition P of A_i with respect to e;
if $Initial \Rightarrow P$ does not hold **then** terminate the heuristic (A_0 not invariant)
else begin
mark (e, A_i);
if P is not identically True then let $A_{n+1} \equiv P$ and increment n by 1
end
enddo

result: If the heuristic terminates with all the (e,A_i) pairs marked, then A satisfies the safety inference rule and implies A_0.

To avoid the expressions for A_i, $0 \leq i \leq n$, from growing unmanageably, it is crucial to simplify the expression for P as much as possible in each iteration. Insight is very important here. First, the choice of the next unmarked (e,A_i) pair is crucial. Second, rather than P being a weakest precondition, it can be strengthened to a precondition. This technique can significantly simplify the expression for P (see Section 5.1 for an example). However, care should be taken so that P is not strengthened to the point where it is no longer invariant; if that happens, then the heuristic may terminate by incorrectly declaring A_0 to be not invariant. (As shown in [23], it is possible to recover from such an error.)

2.3 Proving liveness properties

A liveness property of the system states relationships that values of the system variables eventually satisfy; e.g., the value of state variable v_1 eventually exceeds n for some integer n. In this paper, we shall consider liveness properties that are expressed using the leads-to operator [2, 14, 16, 22]. Throughout, we assume that system implementations meet the following *fairness* condition: any event that is enabled continuously will eventually occur.

Given predicates A and B in **v**, A *leads-to* B means that if the system is in a state that satisfies A, then within a finite number of event occurrences it will be in a state that satisfies B. The following rule is used for deriving leads-to statements from system specifications.

Inference rule for leads-to. If I is invariant and, for some event e_0, assertions A and B satisfy the following:

(i) (I **and** A **and** not B) $\Rightarrow$ ($enabled(e_0)$ **and** (for all $\mathbf{v}'$)[$e_0 \Rightarrow B'$])

(ii) for every event e other than e_0: (I **and** A **and** not B **and** e) $\Rightarrow$ (A' **or** B')

then we infer that A leads-to B via e_0.

Part (i) of the rule ensures that at every reachable state where (A **and** not B) holds, event e_0 is enabled and its occurrence takes the system to a state where B holds. Part (ii) ensures that at every reachable state where (A **and** not B) holds, the occurrence of any other event will take the system to a state where either A or B holds. Thus, in any fair implementation, the system will eventually reach a state where B holds. A leads-to B via e_0 is a special case of A leads-to B, and is similar to "A until B" [14]. As in the case of the inference rule for safety, we can replace I by (I **and** I') in parts (i) and (ii) to facilitate the derivation of the right hand sides.

In addition to the above rule, we have the following (rather obvious) rules:
Leads-to rule 1. If ($A \Rightarrow B$) then (A leads-to B)
Leads-to rule 2. If (A leads-to (B **or** C)) and (C leads-to D), then (A leads-to (B **or** D))
Leads-to rule 3. If (A leads-to B) and (C leads-to D), then ((A **or** C) leads-to (B **or** D))

Finally, we present a rule that applies lexicographic induction [10] to leads-to statements. Let $\mathbf{u}=(u_1, u_2, ..., u_n)$ be an ordered set of integer-valued state variables from $\mathbf{v}$; $\mathbf{u}$ is a subset of $\mathbf{v}$. Below, $\mathbf{i}$ ranges over all n-tuples of natural numbers, and the $\leq$ relation is derived from the lexicographic ordering of integer n-tuples.

Induction rule for leads-to. If assertions A and B satisfy

(for all $\mathbf{i}$)[(A **and** $\mathbf{u} \geq \mathbf{i}$) leads-to ($B$ **or** (A **and** $\mathbf{u} > \mathbf{i}$))]

then we can infer (for all $\mathbf{i}$)[(A **and** $\mathbf{u} \geq \mathbf{0}$) leads-to ($B$ **or** (A **and** $\mathbf{u} > \mathbf{i}$))]

The first leads-to statement in the induction rule is referred to as an *inductive leads-to* statement. The induction rule merely applies mathematical induction to inductive leads-to statements.

2.4 Distributed system model

In this section, we specialize the event-driven model described above to that of a network of processes. We shall use terminology from the network protocols area because our examples 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 one-way communication channels $C_1, C_2, ..., C_K$.

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*($\mathbf{v}$).

Each process has a set of events. The events of entity P_i involve only the state vector $\mathbf{v}_i$ and the state vectors of channels accessible from P_i. There is one exception to this rule: Auxiliary variables can be accessed by more than one entity. Entity events model message

receptions, message sends, and internal activities such as timeout handling. The events of channel C_i 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 A for their predicate definitions).

Entity events access channel state variables only via *send* and *receive primitives*. The send primitive for channel C_i is defined by $\text{Send}_i(m) \equiv (\mathbf{z}_i'=(\mathbf{z}_i,m))$; i.e., append the message value m to the tail of $\mathbf{z}_i$. We use a comma as the concatenation operator, and parentheses to resolve ambiguities. The receive primitive for channel C_i is defined by $\text{Rec}_i(m) \equiv ((m,\mathbf{z}_i')=\mathbf{z}_i)$; 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. Note that $\text{Rec}_i(m)$=False if $\mathbf{z}_i$ is empty. When these primitives are used in entity events, the formal message parameter m is replaced by the actual message sent or received. The definition of $\text{Send}_i(m)$ above assumes that C_i has unbounded message capacity; see Appendix A for the bounded-capacity case.

To verify liveness properties of a distributed system, it is necessary to make an assumption regarding the progress of messages within a channel [6, 14]. Consider a channel from P_i to P_j, and a message m that can be sent by P_i into the channel. Let P_j be always ready to accept message m from the channel. The typical progress assumption is that if message m is sent repeatedly into the channel by P_i then it will be eventually delivered by the channel to P_j [6, 14]. To specify this assumption formally, define the *send count* of message m to be the number of times that m has been sent into the channel by P_i since the last time that m was delivered by the channel to P_j, or since system initialization. The send count of m is an auxiliary state variable that is incremented by 1 whenever m is sent, set to 0 whenever m is delivered, and is initially 0. We will assume that these updates are included in the send and receive primitives. The following axiom specifies that the send count does not grow unboundedly:

Channel liveness axiom: For any send count α:
if (for all i)[A leads-to ($\alpha > i$ **or** B)] holds, then we can infer (A leads-to B)

3. A TIME-INDEPENDENT PROTOCOL

We 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 can lose 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.

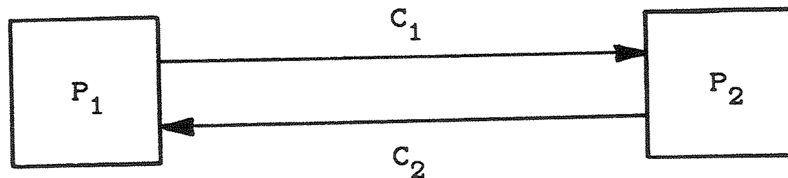

Figure 1. Network configuration of protocol example.

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 identifies the type of message, *data* is a data block from DataSet, and *ns* is a sequence number. P_2 sends messages of type (ACK,*nr*) where *nr* is a sequence number. Here, *ns* and *nr* are restricted to the values 0 and 1.

Throughout this report, for any integer value n, we shall use $\underline{n}$ to denote n mod 2.

An informal description of the protocol follows. Let data block 0, data block 1, ..., data block n, denote the sequence of data blocks produced by the source. P_1 sends data block n accompanied by sequence number $\underline{n}$. P_2 accepts (and passes to the destination) received data blocks with successively increasing sequence numbers. P_2 sends ACK messages with *nr* equal to the next expected value of *ns*. P_1 considers data block n to be acknowledged when it receives an (ACK,*nr*) message with *nr* equal to $\underline{n+1}$. To achieve correct data transfer, it is necessary that P_1 has at most one unacknowledged data block at any time. Otherwise, if data blocks n-1 and n are both outstanding and P_1 receives an (ACK,$\underline{n+1}$) message, then it has no way of knowing whether the ACK message acknowledges data block n or data block n-2. We allow P_1 to retransmit an outstanding data block at any time, and P_2 to send an ACK message at any time.

We next formally specify the variables and events of P_1 and P_2. For brevity in defining event predicates, we shall use the guarded command [4] notation $g(\mathbf{v}) \rightarrow h(\mathbf{v};\mathbf{v}')$ to mean that the action in h is done only if the guard g is true. Formally, if $\mathbf{u}' \subseteq \mathbf{v}'$ is the set of primed variables that occur in the body of h, then $g \rightarrow h \equiv (g \Rightarrow h)$ **and** (not $g \Rightarrow \mathbf{u}'=\mathbf{u}$).

Entity P_1

Source : array[0..∞] of DataSet ∪ {empty}; {History variable. Initially, *Source*[0..∞]=empty}

s : 0..∞; {*Source*[0..*s*-1] is the sequence of data blocks generated by the source. Initially, *s*=0}

Acked : Boolean; {*Acked*=True iff all data blocks are acknowledged. Initially, *Acked*=True}

P_1 has three events: accepting a data block from the local source, sending a D message, and receiving an ACK message. (To keep the example small, we have assumed that *Source*[*s*-1] is available to the implementation.)

AcceptData ≡ *Acked*=True	{if no unacknowledged data}
and *Source*[*s*]′ in DataSet	{then accept data}
and *s*′=*s*+1 **and** *Acked*′=False	{and update state}
Send_D ≡ *Acked*=False **and** Send_1 ((D,*Source*[*s*-1],$\underline{s-1}$))	
Rec_ACK ≡ (for some *nr*)[Rec_2 ((ACK,*nr*)) **and** ((*Acked*=False **and** $\underline{s}$ = *nr*) → *Acked*′=True)]	

Entity P_2

Sink : array[0..∞] of DataSet ∪ {empty}; {History variable. Initially, *Sink*[0..∞]=empty}

r : 0..∞; {*Sink*[0..r-1] is the sequence of data blocks that have been passed on to the destination, and data block r is the one next expected. Initially, r=0}

P_2 has two events: sending an ACK message, and receiving a D message.

Rec_D ≡ (for some *data*, *ns*)[Rec_1((D,*data*,*ns*)) **and** ($\underline{r} = ns \rightarrow (Sink[r]' = data$ **and** $r' = r+1$))]

Send_ACK ≡ Send_2((ACK,$\underline{r}$))

3.1 Safety verification

We desire the following to be invariant:

$A_0 \equiv$ $r \leq s$ **and** (for all n in [0..r-1])[*Sink*[n] = *Source*[n]]
$A_1 \equiv$ *Acked*=True ⇒ r=s

A_0 states that data blocks are delivered to the destination at P_2 in the same order as they were accepted from the source at P_1. A_1 states that if P_1 believes that a data block is acknowledged, then indeed the data block has been delivered to the destination at P_2.

We shall use our heuristic (described in Section 2.2) to generate an assertion A that satisfies the safety inference rule. Note that we do not have any known invariant for this example; thus I≡True.

The lossy nature of the channels simplifies the task of obtaining a weakest precondition of A_i wrt a receive event (wrt is abbreviation of "with respect to"). Every receive event has the form Rec_M ≡ (for some **f**)[Rec_j((M,**f**)) **and** ($g \rightarrow h$)], where **f** denotes the fields of message type M. If Rec_M occurs with g=False, then the only effect of the event occurrence is to delete message (M,**f**) from $\mathbf{z}_j$. Exactly the same effect is achieved by the loss event of C_j. At some point in the verification, A_i will be marked wrt this loss event. Consequently, when generating a weakest precondition of A_i wrt Rec_M, it is sufficient to consider only those occurrences of Rec_M where g holds. Therefore, the weakest precondition (for all **v**′)[(A **and** Rec_M) ⇒ A_i') can be strengthened to

$$\text{(for all } \mathbf{v}')\text{(for some } \mathbf{f})[(A \textbf{ and } \mathbf{z}_j = ((\text{M},\mathbf{f}),\mathbf{z}'_j) \textbf{ and } g \textbf{ and } h) \Rightarrow A_i'] \qquad (*)$$

Further simplification occurs if A_i does not involve $\mathbf{z}_j$. Whenever a message (M,**f**) is in transit in C_j, it is possible via successive loss event occurrences for that message to reach the head of $\mathbf{z}_j$. If A_i does not involve $\mathbf{z}_j$, then it is not affected by this sequence of message deletions. Therefore, the weakest precondition (*) must be strengthened to

$$\text{(for all } \mathbf{v}')\text{(for some } \mathbf{f})[(A \textbf{ and } (\text{M},\mathbf{f}) \text{ in } \mathbf{z}_j \textbf{ and } g \textbf{ and } h) \Rightarrow A_i'] \qquad (**)$$

We start the heuristic by considering the unmarked pair (Rec_ACK,A_1). From (**), we have the following weakest precondition of A_1 wrt Rec_ACK:

$$\text{(for all } \mathbf{v}')\text{(for some } nr)[(A \textbf{ and } (\text{ACK},nr) \text{ in } \mathbf{z}_2 \textbf{ and } Acked{=}\text{False} \textbf{ and } nr{=}\underline{s} \textbf{ and } Acked'{=}\text{True} \textbf{ and } r{=}r' \textbf{ and } s{=}s') \Rightarrow A_1']$$

(The conjuncts $r=r'$ and $s=s'$ are implicit in Rec_ACK.) This predicate can be easily simplified to

$$(A \textbf{ and } (\text{ACK},\underline{s}) \text{ in } \mathbf{z}_2 \textbf{ and } Acked{=}\text{False}) \Rightarrow \text{r=s}$$

which is equivalent to

$$A_2 \equiv (Acked{=}\text{False} \textbf{ and } (\text{ACK},\underline{s}) \text{ in } \mathbf{z}_2) \Rightarrow r{=}s$$

A_2 is clearly a precondition. To see its necessity, suppose that A_2 is False; i.e. *Acked*=False, message (ACK,$\underline{s}$) is in $\mathbf{z}_2$, and $r \neq s$. Then the following can happen: due to message losses, (ACK,$\underline{s}$) comes to the head of $\mathbf{z}_2$; Rec_ACK is now enabled and its occurrence invalidates A_1. Thus, A_2 is indeed a weakest precondition of A_1 wrt Rec_ACK and the loss events.

We next obtain a weakest precondition of A_2 wrt Send_ACK. (Henceforth, we shall be more brief in our derivations.) The weakest precondition (for all $\mathbf{v}'$)[(A **and** Send_ACK)$\Rightarrow A_2'$] is easily shown to be equivalent to

$$(Acked{=}\text{False} \textbf{ and } \underline{r} = \underline{s}) \Rightarrow r{=}s$$

We next obtain a weakest precondition of this wrt Rec_D. Applying (**), and after a little manipulation, we obtain

$$(\text{(for some } data)[(\text{D},data,\underline{r}) \text{ in } \mathbf{z}_1] \textbf{ and } Acked{=}\text{False} \textbf{ and } \underline{r} \neq \underline{s}) \Rightarrow r{=}s\text{-}1$$

A weakest precondition of this wrt Send_D is easily obtained as

$(Acked=\text{False} \textbf{ and } \underline{r}\neq\underline{s}) \Rightarrow r=s\text{-}1$

Combining this with the first assertion in this paragraph, we have

$$A_3 \equiv \quad Acked=\text{False} \Rightarrow (r=s \textbf{ or } r=s\text{-}1)$$

We next obtain a weakest precondition of A_2 wrt AcceptData. As usual, we start with the weakest precondition (for all $\mathbf{v}'$)[(A **and** AcceptData)$\Rightarrow A_2'$] and massage it into a simpler equivalent expression. Assume that the antecedent (A **and** AcceptData) holds (otherwise the weakest precondition holds trivially). Then, from AcceptData we derive $Acked$=True, $Acked'$=False, $s'=s+1$, and $r'=r$. From the last two equalities and A_1, we derive $s'=r'+1$, which invalidates the consequent of A_2'. Therefore, the weakest precondition requires that the antecedent of A_2' be false; i.e., C_2 should not contain any (ACK,$\underline{s'}$) messages. Because $s'=s+1$, this can be stated as follows:

$$A_4 \equiv \quad Acked=\text{True} \Rightarrow \mathbf{z}_2 = (\text{ACK},\underline{s})^*$$

where m^* denotes a sequence of zero or more m's.

We next obtain a weakest precondition of A_4 wrt Rec_ACK. Applying (*), we obtain the weakest precondition

$$(\text{for all } \mathbf{v}')[(A \textbf{ and } \mathbf{z}_2=((\text{ACK},\underline{s}),\mathbf{z}_2') \textbf{ and } Acked=\text{False} \textbf{ and } Acked'=\text{True} \textbf{ and } s'=s) \Rightarrow A_4']$$

which is equivalent to

$$(\text{for all } \mathbf{v}')[(A \textbf{ and } \mathbf{z}_2=((\text{ACK},\underline{s}),\mathbf{z}_2') \textbf{ and } Acked=\text{False}) \Rightarrow \mathbf{z}_2' = (\text{ACK},\underline{s})^*]$$

which is equivalent to

$$(A \textbf{ and } \mathbf{z}_2=((\text{ACK},\underline{s}),\mathbf{z}_2') \textbf{ and } Acked=\text{False}) \Rightarrow (\text{ACK},\underline{s+1}) \text{ not in } \mathbf{z}_2'$$

A weakest precondition of this wrt the loss event for C_2 is

$$(A \textbf{ and } (\text{ACK},\underline{s}) \text{ in } \mathbf{z}_2 \textbf{ and } Acked=\text{False}) \Rightarrow ((\text{ACK},\underline{s}),(\text{ACK},\underline{s+1})) \text{ not a subsequence of } \mathbf{z}_2$$

Note that if (m_1,m_2) is a subsequence of $\mathbf{z}_i$, then m_1 is closer than m_2 to the head of $\mathbf{z}_i$; thus m_1 would be deleted before m_2 can be at the head. This weakest precondition is equivalent to

$$A_5 \equiv \quad Acked=\text{False} \Rightarrow \mathbf{z}_2 = ((\text{ACK},\underline{s+1})^*, (\text{ACK},\underline{s})^*)$$

A_5 is obviously a weakest precondition. Its necessity becomes obvious when we assume its negation; then P_1 can receive the (ACK,$\underline{s}$) and invalidate A_4.

We next obtain a weakest precondition of $A_{0,1,3}$ wrt Rec_D. (The notation $A_{i,j}$ denotes A_i **and** A_j.) Applying (**), we obtain

$$(\mathrm{D},data,\underline{r}) \text{ in } \mathbf{z}_1 \Rightarrow (data{=}Source[r] \textbf{ and } r{=}s\text{-}1)$$

We next obtain a weakest precondition of this wrt to Rec_D. Applying (*), we obtain

$$(\text{for all } \mathbf{v}')[(\mathbf{z}_1{=}((\mathrm{D},data_1,\underline{r}),\mathbf{z}'_1) \textbf{ and } r'{=}r{+}1) \Rightarrow$$
$$((\mathrm{D},data_2,\underline{r'}) \text{ in } \mathbf{z}'_1 \Rightarrow (data_2{=}Source[r'] \textbf{ and } r'{=}s\text{-}1))]$$

But $r'{=}s\text{-}1{=}r{+}1$ implies $s{=}r{+}2$, which violates $A_{1,3}$. Therefore, $\mathbf{z}'_1$ cannot contain (D,$data_2$,$\underline{r'}$), or equivalently $\mathbf{z}_1$ cannot contain the subsequence ((D,$Source[r]$,$\underline{r}$),(D,$data$,$\underline{r+1}$)). Therefore, $\mathbf{z}_1$ = ((D,$data_1$,$\underline{r+1}$)*, (D,$data_2$,$\underline{r}$)*). Incorporating this with the first assertion in this paragraph, we have

$A_6 \equiv$ (for some $data$)[$\mathbf{z}_1$=(D,$data$,$\underline{r+1}$)*] **or** (for some $data$)[$\mathbf{z}_1$ = ((D,$data$,$\underline{r+1}$)*, (D,$Source[r]$,$\underline{r}$)*) **and** $s{=}r{+}1$]

At this point, we have obtained the desired A; i.e., any further precondition that we obtain will equal True. We can now complete the marking of all unmarked event-assertion pairs. We say that an event e *does not affect* A_i if A_i does not contain any state variable changed by e; clearly, such an (e,A_i) pair can be considered marked.

AcceptData: It is already marked wrt A_2. (A_1 **and** AcceptData) implies the following: $X \equiv Acked'$=False and $Y \equiv s'\text{-}1{=}s{=}r{=}r'$. ($A_0$ **and** AcceptData **and** Y) $\Rightarrow A_0'$. $Y \Rightarrow A_3'$. $X \Rightarrow A_{1,4}'$. (Y **and** A_4) $\Rightarrow A_5'$. (Y **and** A_6) $\Rightarrow \mathbf{z}_1$=(D,$data$,$\underline{r+1}$)* $\Rightarrow A_6'$.

Send_D: Does not affect $A_{0,1,2,3,4,5}$. Send_D implies $X \equiv Acked'{=}Acked$=False, and $Y \equiv s'{=}s$ **and** $r'{=}r$. (X **and** Y **and** A_3) implies $Z \equiv s{=}r$ **or** $s{=}r{+}1$. (Z **and** A_6) implies A_6'.

Rec_ACK: Already marked wrt $A_{1,4}$. Does not affect $A_{0,6}$. We need only consider the case when $\mathbf{z}_2$=((ACK,$\underline{s}$),$\mathbf{z}'_2$), $Acked$=False, $nr{=}\underline{s}$, and $Acked'$=True; thus, $A_{2,3,5}'$ holds vacuously. Otherwise, Rec_ACK is not enabled or its effect is exactly that of C_2's loss event, which is handled below; this is precisely the same argument used to derive (*)

Send_ACK: Already marked wrt A_2. Does not affect $A_{0,1,3,6}$. ($A_{4,1}$ **and** Send_ACK) $\Rightarrow A_4'$. ($A_{2,3,5}$ **and** Send_ACK) $\Rightarrow A_5'$.

Rec_D: Already marked wrt A_0. Does not affect $A_{4,5}$. Need only consider when $X \equiv r'{=}r{+}1$ **and** $\mathbf{z}_1$=((D,$Source[r]$,$\underline{r}$),$\mathbf{z}'_1$) holds. (X **and** A_6) implies $Acked{=}Acked'$=False, $s{=}r{+}1$, and $\mathbf{z}_1$=(D,$Source[r]$,$\underline{r}$)*. From these, we derive $A_{1,2,3,6}'$.

Channel loss events: We have (A_i **and** message loss) $\Rightarrow A_i'$, for all A_i.

3.2 Liveness verification

We would like to prove that once a data block is accepted from the source at P_1, then it will be acknowledged eventually. More formally, we wish to prove

L_0	($Acked$=False **and** s=n) leads-to ($Acked$=True **and** s=n)

We shall prove this property for a restricted version of the protocol in which P_2 sends an ACK message only if it has received a D message since it last sent an ACK message. If L_0 holds for this restricted protocol, then clearly it holds for the original protocol as well. We shall introduce the boolean state variable *SendACK* at P_2, include ($SendACK'$=True) as a conjunct in Rec_D, and include ($SendACK$=True **and** $SendACK'$=False) as a conjunct in Send_ACK. *SendACK* is initially False. Note that any safety property verified earlier continues to hold because each event e_m in the modified protocol is a *refinement* of the corresponding event e in the earlier protocol such that $e_m \Rightarrow e$.

We will prove the liveness property L_0 assuming the channel liveness axiom (Section 2.4): i.e., even though the channels are lossy, they will eventually deliver a message that is sent repeatedly. Let α_1 and α_2 denote the send counts of (D,$Source[n]$,$\underline{n}$) and (ACK,$\underline{n+1}$) respectively. Let

$B \equiv Acked$=False **and** s-1=r=n
$C \equiv Acked$=False **and** s=r=n+1
$D \equiv Acked$=True **and** s=r=n+1

We prove L_0 in two stages. First, we prove that $Source[n]$ will be received at P_2; i.e., B leads-to C.

Proof of B leads-to C. Applying the leads-to inference rule with $I \equiv A_{1,3}$, we obtain

(B **and** $\alpha_1 \geq i$) leads-to ((B **and** $\alpha_1 \geq i+1$) **or** C) via Send_D

The details of the inference rule application are as follows: Send_D is enabled in (B **and** $\alpha_1 \geq i$) and leads-to (B **and** $\alpha_1 \geq i+1$). The reception of (D,$Source[n]$,$\underline{n}$) in (B **and** $\alpha_1 \geq i$) leads-to C. Any other event occurrence in (B **and** $\alpha_1 \geq i$) leads-to (B **and** $\alpha_1 \geq i$).

Applying the induction rule to the above inductive leads-to statement, with $u=\alpha_1$ and noting that $\alpha_1 \geq 0$, we have (B leads-to ((B **and** $\alpha_1 \geq i$) **or** C). Applying the channel liveness axiom to this, we have B leads-to C. **End of proof**

We next prove that once P_2 has received $Source[n]$, then P_1 eventually receives the acknowledgement (ACK,$\underline{n+1}$); i.e., C leads-to D.

Proof of C **leads-to** D. Applying the leads-to inference rule with $I \equiv A_{1,3}$, we obtain the following:

(C **and** $\alpha_1 \geq i$ **and** $\alpha_2 \geq j$) leads-to
((C **and** $\alpha_1 \geq i+1$ **and** $\alpha_2 \geq j$) **or** (C **and** $\alpha_2 \geq j$ **and** *SendACK*=True) **or** D) via Send_D

(C **and** $\alpha_2 \geq j$ **and** *SendACK*=True) leads-to ((C **and** $\alpha_2 \geq j+1$) **or** D) via Send_ACK

Applying leads-to rule 2 to the above yields the following inductive leads-to statement:

(C **and** $\alpha_1 \geq i$ **and** $\alpha_2 \geq j$) leads-to (D **or** (C **and** ($\alpha_2 \geq j+1$ **or** ($\alpha_2 \geq j$ **and** $\alpha_1 \geq i+1$))))

Applying the induction rule to this with $\mathbf{u}=(\alpha_2,\alpha_1)$, we have ($C$ leads-to ((C **and** $(\alpha_2,\alpha_1) \geq (j,i)$) **or** D). We can infer C leads-to D from the above and the channel liveness axiom, because $(\alpha_2,\alpha_1) \geq (j,i)$ implies that either $\alpha_2 \geq j$ or $\alpha_1 \geq i$. **End of proof.**

Applying leads-to rule 1 to A_3, we get (*Acked*=False **and** s-1=n) leads-to (B **or** C). Applying leads-to rules 2 and 3 to this, B leads-to C, and C leads-to D, we get L_0.

4. REAL-TIME SYSTEM MODEL

In Section 4.1, we define timers and time events. In Section 4.2, we model implementable time constraints that are enforced by individual processes of a distributed system. In Section 4.3, we model bounded-delay communication channels. In Section 4.4, we model time constraints that are enforced due to the cooperation of the processes. The discussion in Sections 4.1 and 4.2 applies to both entity processes and channel processes. For the sake of brevity, we shall refer only to entity processes.

4.1 Measures of Time

We use the term *local timers* for those timers that are implemented within individual entities of a distributed system. In our model, local timers are discrete-valued state variables: the interval between successive ticks of a local timer is not infinitesimally small. Local timers in different entities are not coupled: the ticks of one timer do not coincide in time with the ticks of another timer. The ticking rate of local timers of entity P_i is not constant but can vary within a specified error bound ϵ_i. Typically, $\epsilon_i \ll 1$; for a crystal oscillator driven system, $\epsilon_i \approx 10^{-6}$.

A local timer can take values from the domain {Off,0,1,2,...}. For this domain, define the successor function *next* as follows: *next*(Off)=Off and *next*(i)=i+1 for $i \neq$Off. 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. In this case, *next*(M)=Off.

For each entity, there is a *local time event* (corresponding to a clock tick) whose occurrence updates every local timer within that entity to its *next* value. No other timer is affected.

In addition to being affected by its time event, a local timer can be reset to either 0 or Off by an event of that entity. (Unless otherwise indicated, the term *events* will refer only to the events of the entity other than the time event; i.e., the communication and internal events.) Resetting to 0 is referred to as *starting* the timer, and resetting to Off is referred to as *stopping* the timer. Thus, a local timer that is started by an event occurrence measures the time elapsed (in number of occurrences of its local time event) since that event occurrence.

To keep the rates of time event occurrences in different entities 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 a *constant* rate. We allow the system model 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 event occurrences (not necessarily of the same process).

Given an ideal timer u and a local timer v of entity P_i, we say: (u,v) *started-together* to mean that at some instant in the past u and v were simultaneously started, and after that instant neither u nor v has been started or stopped. The accuracy of local timer v is modeled by assuming that occurrences of the ideal time event and the local time event of P_i preserve the following condition, which we shall refer to as the *accuracy axiom*:

Accuracy Axiom. (u,v) started-together $\Rightarrow |u\text{-}v| \leq \max(1, \epsilon_i u)$

Note that the accuracy axiom would be enforced if the occurrence rate of the local time event of entity P_i differs from the occurrence rate of the ideal time event by at most ϵ_i. The accuracy axiom is a discrete version of the condition $| 1 - \frac{dv}{du} | \leq \epsilon_i$ used for continuous clocks [12].

Proving safety assertions with started-together statements

Safety assertions can contain started-together statements; e.g., $x>y \Rightarrow ((u,v)$ started-together). We next present rules to be used when applying the safety inference rule to such assertions. In part (i) of the safety inference rule, we can use the rule

(ST1) (u=0 **and** v=0) $\Rightarrow$ ((u,v) started-together)

In part (ii) of the safety inference rule, we use the following two rules, if the event e being considered is not a time event:

(ST2) (u'=0 **and** v'=0) $\Rightarrow$ ((u',v') started-together)
(ST3) (u'=u **and** v'=v **and** ((u,v) started-together)) $\Rightarrow$ ((u',v') started-together)

A started-together statement is preserved by a time event occurrence, unless one of the timers is a bounded capacity timer that is exceeding its capacity and is stopped by the time event occurrence. Thus, the following rule applies in part (ii) of the safety inference rule, if the event e being considered is a time event:

(ST4) ((u,v) started-together **and** $next(u) \neq$Off **and** $next(v) \neq$Off) $\Rightarrow$ ((u',v') started-together)

4.2 Implementable Time Constraints

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

Let e_1 and e_2 be two events of entity P_i, and let v be a local timer in $\mathbf{v}_i$ that is started by e_1, and stopped by e_2. Consider the following three time constraints:

(E1) e_2 will not occur within T time units of e_1's occurrence: This is modeled by adding the conjunct $v \geq T$ to e_2; in other words, including $v \geq T$ in the enabling condition of e_2.

(E2) If e_2 occurs, then it occurs within T time units of e_1's occurrence: This is modeled by including $v \leq T$ in the enabling condition of e_2.

(E3) e_2 *must occur* within T time units of e_1's occurrence: This is modeled by specifying that the time events keep the condition $v \leq T$ invariant. Note that this constraint *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 general form "event e *will occur only if* the elapsed times satisfy a condition *TC*," where *TC* is a predicate in $\mathbf{v}_i$. They are modeled by including *TC* in the enabling condition of event e. Such a time constraint is always implementable.

E3 is an example of time constraints of the general form "event e *must occur while* the elapsed times satisfy a condition *TA*." *They are modeled by requiring that time events keep TA invariant*. We refer to such a *TA* as a *timer axiom*.

Not every predicate in $\mathbf{v}_i$ can be a timer axiom. Consider the example E3 where e_1 and e_2 are both events of P_i but e_2 involves the reception of a message (we shall refer to this example as E4). We do not allow E4 to be modeled by the timer axiom $v \leq T$ because P_i cannot by itself enforce E4: the cooperation of other processes is needed to ensure that the required message is present in the channel. Notice that without this cooperation the time events will eventually deadlock; i.e. no time event will be able to occur without violating some timer axiom.

An event e of entity P_i is said to be *controlled* by P_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. Let **TA** be the conjunction of all the timer axioms of entity P_i. We require **TA** to be a predicate in $\mathbf{v}_i$ that satisfies the following *implementable conditions*:

(IC1) **TA** holds initially.

(IC2) For every value of $\mathbf{v}_i$ where $\mathbf{TA}(\mathbf{v}_i)$=True, there is no event of P_i which sets $\mathbf{v}_i$ to a value such that $\mathbf{TA}(\mathbf{v}_i)$=False.

(IC3) **TA** cannot refer to both ideal and local timers.

(IC4) If **TA** refers to local (ideal) timers, then let $next_i(\mathbf{v}_i)$ denote $\mathbf{v}_i$ with every local (ideal) timer updated to its *next* value. For every value of $\mathbf{v}_i$ such that $\mathbf{TA}(\mathbf{v}_i)$=True **and** $\mathbf{TA}(next_i(\mathbf{v}_i))$=False, there is a sequence of enabled events $e_1, e_2, \ldots, e_n$ controlled by P_i whose occurrence will set $\mathbf{v}_i$ to a value such that $\mathbf{TA}(next_i(\mathbf{v}_i))$=True.

IC1 and IC2 ensure that events do not violate the invariance of the timer axioms. IC3 ensures that **TA** can not specify an accuracy for local timers higher than that specified by the accuracy axiom. Indeed, without IC3 the accuracy axiom would be a special case of a timer axiom. IC4 is the key condition; it ensures that P_i can implement its timer axioms without cooperation from the rest of the distributed system. In example E4, IC4 is violated because the receive event e_2 is not controlled by process i. We have shown in [22] that if the timer axioms of each process in a distributed system satisfy the implementable conditions, then the time events will never deadlock. Thus, every running timer will be either reset by an event or incremented:

Theorem 1. Given a system with implementable timer axioms, the following holds for any timer u: u=n leads-to (u=n+1 **or** u=Off **or** u=0)

4.3 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. Such behavior is not only common in communication networks, but is crucial for the correct operation of communication protocols [20, 26].

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 timer axiom (for every <*message,age*> in $\mathbf{z}_i$)$[0 \leq age \leq MaxDelay_i]$.

The send and receive primitives are modified to the following: $\text{Send}_i(m) \equiv (\mathbf{z}'_i = (\mathbf{z}_i, <m,0>))$, i.e., append message m with an age of 0 to the tail of $\mathbf{z}_i$. $\text{Rec}_i(m) \equiv$ (for some age)$[(<m,age>, \mathbf{z}'_i) = \mathbf{z}_i]$, i.e., receive the message m from the head of $\mathbf{z}_i$ irrespective of its age provided that $\mathbf{z}_i$ is not empty.

In practice, the check that the above timer axiom 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 IC4. 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. IC3 holds because $\mathbf{z}_i$ has only ideal timers.

4.4 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. 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 B" is logically equivalent to the statement that B holds whenever e is enabled. It is established by proving that the assertion $enabled(e) \Rightarrow B$ is invariant.

Consider a derived time constraint of the form: system event e *must occur while* the elapsed times satisfy a condition B. This time constraint is logically equivalent to the statement that no time event occurrence violates B. It is established by proving that for every time event e_t that affects B, the assertion (for every $\mathbf{v}'$)[(B **and** e_t)$\Rightarrow B'$] is invariant. Typically, the timers can be chosen such that B holds initially and is preserved by events other than the time events. In this case, the time constraint corresponds to the requirement that B is invariant (e.g. property D_0 in Section 6.1).

The problem of analyzing the relationships between time constraints enforced within processes and the resulting system-wide time constraints may be handled in two phases: a global analysis phase involving only ideal timers, followed by a local analysis phase during which implementable time constraints expressed with ideal timers are realized using local timers (see example in Section 5.1).

5. A TIME-DEPENDENT PROTOCOL

We reconsider the data transfer protocol example of Section 3, where now each channel C_i can *lose, duplicate and reorder* messages in transit. Moreover, messages in channel C_i have a maximum lifetime of $MaxDelay_i$, for i=1 and 2. These maximum message lifetimes are necessary for correct operation; without them, it is always possible for an old data block with a currently active cyclic sequence number to be mistaken for a new data block.

To achieve correct operation over these channels, it is sufficient that P_1 sends a new data block *Source*[n] only when the following two conditions hold: First, at least $MaxDelay_1$ time has elapsed since the last send of data block *Source*[n-1]. Second, at least $MaxDelay_2$ time has elapsed since P_1 first received the acknowledgement of *Source*[n-1]. These two time constraints are in addition to the earlier requirement that *Source*[n] can be sent only after *Source*[n-1] has been acknowledged. Informal justifications of the time constraints are given below. The verification in Section 5.1 provides a formal proof that they ensure correct operation. (The reader is referred to [24] for the general case of ns and nr taking values from $\{0,1,\ldots,N\text{-}1\}$ for any $N \geq 2$.)

The first time constraint ensures that there is no (D,*Source*[n-1],$\underline{n\text{-}1}$) message in C_1 when P_1 sends (D,*Source*[n],$\underline{n}$). Otherwise, because C_1 can duplicate and reorder messages, the following can happen: P_2 receives a (D,*Source*[n],$\underline{n}$) message where r=n, followed by a (D,*Source*[n-1],$\underline{n\text{-}1}$) message. Because $\underline{n\text{-}1} = \underline{n+1}$, P_2 would incorrectly interpret the second D message as containing *Source*[n+1].

The second time constraint ensures that there are no (ACK,$\underline{n\text{-}1}$) messages in C_2 when (D,*Source*[n],$\underline{n}$) is sent. The constraint achieves this because (ACK,$\underline{n\text{-}1}$) is not sent after *Source*[n-1] has been received at P_2; this happens before acknowledgement of *Source*[n-1] is received at P_1. It is necessary that (ACK,$\underline{n\text{-}1}$) not be present in C_2 because such an ACK message if received by P_1 would be incorrectly interpreted as an acknowledgement for *Source*[n] (recall $\underline{n\text{-}1} = \underline{n+1}$).

Observe that neither of the above time constraints applies to the retransmission of an outstanding data block. The time to wait before a retransmission can thus be chosen on the basis of performance goals and the probability distributions of channel delays, channel loss, etc. (further discussions in Section 6). 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 specify the variables and events of P_1, as well as the time events. The variables *Source*, s and *Acked* of P_1, as well as the entire specification of P_2 are exactly the same as in the earlier example in Section 3. Given an ideal timer u and local timer v of P_1 which are started together, from the accuracy axiom it is clear that $u > T$ holds if $v \geq 1+(1+\epsilon_1)T$ holds, or equivalently if v is a timer of capacity $(1+\epsilon_1)T$ and is Off. With this motivation, we define $MDelay_i = (1+\epsilon_1)\ MaxDelay_i$ for i=1 and 2.

Variables of P_1

Source : array[0..∞] of DataSet ∪ {empty}; {History variable. Initially, *Source*[0..∞]=empty}

s : 0..∞; {Initially, s=0}

Acked : Boolean; {Initially, *Acked*=True}

DTimeG : ideal timer; {indicates the time elapsed since the last D message sent. Initially, *DTimeG*=Off}

DTimer : local timer of capacity $MDelay_1$; {started together with *DTimeG*. Initially, *DTimer*=Off}

ATimeG : ideal timer; {indicates the time elapsed since the reception of the first ACK message that acknowledged *Source*[*s*-1]. Initially, *ATimeG*=Off}

ATimer : local timer of capacity $MDelay_2$; {started together with *ATimeG*. Initially, *ATimer*=Off}

Events of P_1

AcceptData ≡ *Acked*=True **and** {if no unacknowledged data} *DTimer*=Off **and** *ATimer*=Off **and** {and time constraints met} *Source*[*s*]′ in DataSet **and** *s*′=*s*+1 **and** *Acked*′=False {accept data}
Send_D ≡ *Acked*=False **and** Send_1((D,*Source*[*s*-1],$\underline{s\text{-}1}$)) {send outstanding data} **and** *DTimer*′=0 **and** *DTimeG*′=0 {start *DTimer*}
Rec_ACK ≡ (for some *nr*)[Rec_2((ACK,*nr*)) **and** ((*Acked*=False **and** $\underline{s}$ = *nr*) → (*Acked*′=True **and** *ATimer*′=0 **and** *ATimeG*′=0))]

Time events

The only timer axioms in this example are those for specifying the maximum message lifetime property. Formally, for i=1 and 2:

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

There are two time events: the ideal time event and P_1's local time event. They are defined exactly as in Section 4.1 and 4.2. The ideal time event affects all ideal timers; it can be formally specified by the predicate

$$\mathbf{z}'_1 = next(\mathbf{z}_1) \textbf{ and } \mathbf{z}'_2 = next(\mathbf{z}_2)$$
$$\textbf{and } DTimeG' = next(DTimeG) \textbf{ and } ATimeG' = next(ATimeG)$$

where $next(\mathbf{z}_i)$ returns $\mathbf{z}_i$ with every *age* variable in it incremented by 1.

P_1's local time event affects all local timers in $\mathbf{v}_1$; it can be specified by the predicate

$$DTimer'{=}next(DTimer)\ \textbf{and}\ ATimer'{=}next(ATimer)$$

Note that these events will occur *only if* their occurrence preserves the invariance of every time and accuracy axiom. In this respect, the above predicates specify just the actions of the time events. Note also that the time event predicates are fully specified once all the timer axioms, if any, are known. In future examples, we shall not write out explicitly the time event predicates.

5.1 Safety verification

As in Section 3.1, we shall use our heuristic to generate an assertion A that establishes the invariance of the following:

$$A_0 \equiv\ r{\le}s\ \textbf{and}\ (\text{for all } n \text{ in } [0..r\text{-}1])[Sink[n] = Source[n]]$$
$$A_1 \equiv\ Acked{=}\text{True} \Rightarrow r{=}s$$

The following weakest precondition of A_1 wrt Rec_ACK and C_2 loss event is generated exactly as in Section 3.1:

$$A_2 \equiv\ (Acked{=}\text{False}\ \textbf{and}\ (\text{ACK},\underline{s}) \text{ in } \mathbf{z}_2) \Rightarrow r{=}s$$

The following weakest precondition of A_2 wrt Send_ACK, Rec_D, and Send_D is generated exactly as in Section 3.1:

$$A_3 \equiv\ Acked{=}\text{False} \Rightarrow (r{=}s\ \textbf{or}\ r{=}s\text{-}1)$$

The following weakest precondition of A_2 wrt AcceptData is generated exactly as in Section 3.1:

$$A_4 \equiv\ (Acked{=}\text{True}\ \textbf{and}\ DTimer{=}\text{Off}\ \textbf{and}\ ATimer{=}\text{Off}) \Rightarrow \mathbf{z}_2 = (\text{ACK},\underline{s})^*$$

We now obtain a precondition of A_4 wrt the local time event of P_1. The time event occur-

rence will cause *DTimer*=*ATimer*=Off to become true if and only if the following held before its occurrence:

$$\begin{aligned} X \equiv\ & (\mathit{ATimer}=\mathit{MDelay}_2 \ \textbf{and}\ \mathit{DTimer}=\text{Off}) \\ & \textbf{or}\ (\mathit{ATimer}=\text{Off}\ \textbf{and}\ \mathit{DTimer}=\mathit{MDelay}_1) \\ & \textbf{or}\ (\mathit{ATimer}=\mathit{MDelay}_2\ \textbf{and}\ \mathit{DTimer}=\mathit{MDelay}_1) \end{aligned}$$

Thus, a weakest precondition would be $(\mathit{Acked}=\text{True}\ \textbf{and}\ X) \Rightarrow \mathbf{z}_2 = (\text{ACK},\underline{s})^*$. However, from the informal justification of the second time constraint (fourth paragraph in Section 5 above), we know that *DTimer* is not relevant to enforcing the right hand side of A_4. Thus, we can strengthen the above weakest precondition to the following precondition:

$$(\mathit{Acked}=\text{True}\ \textbf{and}\ \mathit{ATimer}=\mathit{MDelay}_2) \Rightarrow \mathbf{z}_2 = (\text{ACK},\underline{s})^*$$

We also know that the right hand side is enforced by exploiting the maximum message lifetime property of C_2. Specifically, *ATimeG* maintains a lower bound on the ages of $(\text{ACK},\underline{s+1})$ messages in $\mathbf{z}_2$, and *ATimer* is started together with *ATimeG*. This leads us to the following precondition:

$$\begin{aligned} A_5 \equiv\ & (\mathit{Acked}=\text{True}\ \textbf{and}\ <(\text{ACK},\underline{s+1}),\mathit{age}>\ \text{in}\ \mathbf{z}_2) \\ & \Rightarrow ((\mathit{ATimer},\mathit{ATimeG})\ \text{started-together}\ \textbf{and}\ \mathit{ATimeG} \leq \mathit{age}) \end{aligned}$$

We now obtain a precondition of A_0 wrt to Rec_D. As noted in the informal justification of the first time constraint (third paragraph in Section 5 above), because C_1 can reorder and duplicate messages in transit, it is necessary that all D messages in $\mathbf{z}_1$ have the same value of *ns*. If that *ns* equals $\underline{r}$, then the data fields should all equal *Source*[*r*] and *r* must equal *s*-1 (to ensure $A_{1,3}$ after the reception).

$$A_6 \equiv\ (\text{for some}\ \mathit{data})[\mathbf{z}_1=(\text{D},\mathit{data},\underline{r+1})^*]\ \textbf{or}\ (\mathbf{z}_1=(\text{D},\mathit{Source}[r],\underline{r})^*\ \textbf{and}\ r=s\text{-}1)$$

A weakest precondition of $A_{6,3}$ wrt to Send_D is

$$\begin{aligned} A_7 \equiv\ & \mathit{Acked}=\text{False} \Rightarrow (((\text{for some}\ \mathit{data})[\mathbf{z}_1=(\text{D},\mathit{data},\underline{r+1})^*]\ \textbf{and}\ r=s) \\ & \textbf{or}\ (\mathbf{z}_1=(\text{D},\mathit{Source}[r],\underline{r})^*\ \textbf{and}\ r=s\text{-}1)) \end{aligned}$$

We next obtain a weakest precondition of A_7 wrt to AcceptData. From AcceptData and A_T, we have $r'=r=s=s'$-1 and Acked'=False. Thus, in order that A_7' hold, we require that $\mathbf{z}_1=(\text{D},\mathit{Source}[r],\underline{r})^*$ before AcceptData occurs. But $(A_6\ \textbf{and}\ r=s) \Rightarrow \mathbf{z}_1=(\text{D},\mathit{data},\underline{r+1})^*$. The only way that $\mathbf{z}_1$ can satisfy both requirements is if there are no D messages in $\mathbf{z}_1$.

$A_8 \equiv$ (*Acked*=True **and** *DTimer*=Off **and** *ATimer*=Off) $\Rightarrow \mathbf{z}_1$ = null

We next obtain a precondition of A_8 wrt time events. From the informal justification of the first time constraint, it is obvious that the enforcement of A_8 does not depend on *ATimer* or on *Acked*. It does depend on the maximum message lifetime property of C_1. Specifically, *DTimeG* maintains a lower bound on the ages of D messages in $\mathbf{z}_1$, and *DTimer* is started together with *DTimeG*. This leads us to the following:

$A_9 \equiv$ <(D,*data*,*ns*),*age*> in $\mathbf{z}_1$)
$\Rightarrow$ ((*DTimer*,*DTimeG*) started-together **and** $DTimeG \leq age$)

The assertions $A_{0\text{-}9}$ satisfy the safety inference rule, as shown below.

AcceptData: AcceptData implies *Acked′*=False, which implies $A_{1,4,5,8}'$. (A_1 **and** AcceptData) $\Rightarrow A_{0,3}'$. (Henceforth, for brevity, we will omit listing the event name in the antecedent of all implications in this proof; i.e., the last derivation would be stated as $A_1 \Rightarrow A_{0,3}'$.) $A_4 \Rightarrow A_2'$. $A_8 \Rightarrow A_{6,7,9}'$.

Send_D: $A_{0,1,2,3,5}$ is not affected. Send_D implies *DTimer′*=0 which implies $A_{4,8}'$ vacuously. A_9, *DTimer′*=0, and timer axiom for C_1 together imply A_9'. $A_7 \Rightarrow A_{6,7}'$.

Rec_ACK: We only need consider the case when *Acked*=False, *Acked′*=True, *ATimer′*=*ATimeG′*=0, $\mathbf{z}_2$=((ACK,$\underline{s}$),$\mathbf{z}_2'$). $A_{0,6,9}$ is not affected. $A_{2,3,7}'$ holds vacuously. $A_2 \Rightarrow A_1'$. $A_{4,8}'$ holds vacuously because *ATimer′*=0. A_5' holds from A_5, *ATimer′*=0, and timer axiom for C_2.

Rec_D: $A_{4,5,8}$ is not affected. $A_9 \Rightarrow A_9'$. We only need consider the case of $ns=\underline{r}$, $r'=r+1$, and $Sink[r]'=data$. From A_6, we have $r=s\text{-}1$. $r=s\text{-}1$ and $A_{0,6}$ imply $A_{0,1,2,3}'$. $r=s\text{-}1$ and A_6 imply $A_{6,7}'$.

Ideal time event: We have $A_{5,8} \Rightarrow A_{5,8}'$ from the started-together rules (Section 4.1). No other A_i is affected.

Local time event of P_1: Only $A_{4,5,8,9}$ are affected. We first consider $A_{4,5}$. We only need to consider the case of *Acked*=True. If $ATimer \neq MDelay_2$ then A_4 is not affected and $A_5 \Rightarrow A_5'$ from started-together rules. If $ATimer = MDelay_2$ then from A_5, accuracy axiom, and definition of $MDelay_2$, we have (ACK,$\underline{s+1}$) in $\mathbf{z}_2 \Rightarrow ATimeG > MaxDelay_2 \Rightarrow$ (ACK,$\underline{s+1}$) not in $\mathbf{z}_2$ (from the timer axiom for C_2). Thus, $\mathbf{z}_2$=(ACK,$\underline{s}$)* and then A_5' holds vacuously and A_4' nonvacuously. $A_{8,9}'$ is similarly derived from $A_{8,9}$.

5.2 Liveness verification

The liveness verification of the earlier example (Section 3.2) serves as a verification for this protocol too. Specifically, there are two types of leads-to statements in that verification. Statements of the first type have the form "X leads-to Y via event e," and were derived from the system specifications by applying the leads-to inference rule. Statements of the second type have the form "X leads-to Y," and were derived from the leads-to statements of the first type by applying the leads-to rules 1-3.

It is easy to check that the derivation of each leads-to statement of the first type continues to be valid for this protocol example. Because these statements do not involve any timers, they are not affected by the time events. The only safety property used in their derivation was $A_{1,3}$, which also holds in this protocol example. The derivations of the second type of leads-to statements continue to be valid, because these derivations require only the leads-to statements of the first type; indeed, these derivations are independent of the system specifications.

6. A PROTOCOL WITH REAL-TIME PROGRESS

We now modify the time-dependent protocol in Section 5 so that unacknowledged data blocks are acknowledged within a bounded response time T, provided that the channels do not consistently perform badly. Such a real-time progress property is more realistic than the liveness property L_0 shown in Sections 3.2 and 5.2. In practice, if progress is not achieved within time T, then P_1 aborts the connection with P_2.

Let $Delay_i$ ($\leq MaxDelay_i$) be the delay that a message is *expected* to encounter in channel C_i. We say that a message m in C_i is *overdelayed* if it is not received within $Delay_i$ time of its send. ($Delay_i \ll MaxDelay_i$ for a realistic channel.) Note that if $Delay_i = MaxDelay_i$ then overdelaying message m corresponds to losing m and any of its duplicates.

Entity P_2 will now send an ACK message within a specified *MaxResponseTime* of receiving a D message.

Define $RoundTripDelay = (Delay_1 + Delay_2 + MaxResponseTime)$. Define $RTripDelay = 1 + (1+\epsilon_1)\times RoundTripDelay$. P_1 transmits a data block as soon as it is accepted, and retransmits it once every *RTripDelay* local time units until it is acknowledged. Separating successive transmissions by *RoundTripDelay* time ensures that a separate acknowledgement is sent for each new data block transmission that is received at P_2. As a result, a single channel failure affects at most one data block transmission.

We note that the proposed ISO transport protocol standard [9] includes exactly such real-time behaviour and delay parameters.

Before stating the real-time progress property, we define the following auxiliary variables:

OutTimeG : ideal timer; {If *Acked*=False then *OutTimeG* indicates the time elapsed since *Acked* last became False. *OutTimeG*=Off initially and whenever *Acked*=True}

β_1 : integer; {If *Acked*=False then β_1 indicates the number of times that C_1 has over-delayed (D,*Source*[*s*-1],$\underline{s\text{-}1}$) since *Acked* last became False. β_1=0 initially and whenever *Acked*=True}

β_2 : integer; {If *Acked*=False then β_2 indicates the number of times that C_2 has over-delayed (ACK,$\underline{s}$) since *Acked* last became False. β_2=0 initially and whenever *Acked*=True}

We assume that the channels satisfy the following axiom (analogous to the channel liveness axiom in Section 2.4.):

Channel real-time progress axiom: $\beta_1 + \beta_2 < M$ for some $M \geq 1$.

Given this axiom, we will verify that every data block is acknowledged within $T = M \times RTDelay$ seconds of it being accepted, where $RTDelay = 1 + (1/(1\text{-}\epsilon_1)) \times RTripDelay$. This worst-case progress property is formally expressed by the following safety property:

$$D_0 \equiv Acked{=}\text{False} \Rightarrow OutTimeG < M \times RTDelay$$

6.1 Protocol specifications

We now refine the specifications of P_1 and P_2 in Section 5 to enforce the required real-time behavior. The channel timer axioms TA_1 and TA_2 continue to apply.

At P_2, there is now a local time event of accuracy ϵ_2, and the following variables:

ACKTimer : local timer; {indicates the time elapsed since the earliest D message reception for which an ACK has not yet been sent. *ACKTimer*=Off initially and whenever there is no unacknowledged D message}

ACKTimeG : ideal timer; {started together with *ACKTimer*. Initially, *ACKTimeG*=Off}

Both *ACKTimer* and *ACKTimeG* are stopped in the Send_ACK event, and started, if they were Off, in the Rec_D event. *ACKTimer* is constrained by the timer axiom

$$TA_3 \equiv (ACKTimer \neq \text{Off} \Rightarrow ACKTimer < MResponseTime)$$

where $MResponseTime = (1\text{-}\epsilon_2) \times MaxResponseTime$ - 1. This timer axiom is implementable because of the Send_ACK event.

We have the following additional variables at P_1:

TryTimer : local timer; {If *Acked*=False then *TryTimer* indicates the time elapsed since the last send of *Source*[*s*-1]. *TryTimer*=Off initially and whenever *Acked*=True}

TryTimeG : ideal timer; {started together with *TryTimer*. Initially, *TryTimeG*=Off}

TryCount : integer; {Indicates the number of transmissions of *Source*[*s*-1] that have already occurred. *TryCount*=0 initially and whenever *Acked*=True}

When *Source*[*s*-1] is transmitted, *TryCount* is incremented by 1 and *TryTimer* is reset to 0. This transmission occurs whenever *TryTimer*=*RTripDelay*. This time constraint is formally specified by the following timer axiom:

$$TA_4 \equiv (Acked{=}\text{False } \textbf{and } TryTimer \neq \text{Off}) \Rightarrow TryTimer \leq RTripDelay$$

This timer axiom is implementable because of the Send_D event.

We now specify auxiliary variables needed to update β_1 and β_2. In each D message, we include an auxiliary *tn* field which indicates the value that *TryCount* had immediately after the message transmission. A D message with *tn* equal to the current value of *TryCount* is referred to as a *current* D message. An ACK message sent in response to a current D message is referred to as a *current* ACK message.

TryRecd : boolean; {True iff *Acked*=False and a current D message has been received at P_2. Initially, *TryRecd*=False}

TryAckedG : ideal timer; {If *Acked*=False and P_2 has sent current ACK messages, then *TryAckedG* indicates the time elapsed since the first such ACK message was sent. *TryAckedG*=Off otherwise and initially}

The events of this protocol are obtained by refining the corresponding events of the protocol from Section 5.

AcceptData is the conjunction of the previous AcceptData event and the following:

NEWTRY $\equiv$ ($Send_1$((D,*Source*[*s*],$\underline{s}$,*TryCount'*))
and *TryTimer'*=*TryTimeG'*=0 **and** *TryCount'*=1
and *OutTimeG'*=0 **and** *TryRecd'*=False **and** Try*Acked*G'=Off
and $\beta_1'=\beta_2'=0$)

Send_D is obtained by replacing $Send_1$((D,*Source*[*s*-1],$\underline{s\text{-}1}$)) with

RETRY $\equiv$ ($Send_1$((D,*Source*[*s*-1],$\underline{s\text{-}1}$,*TryCount'*)) **and** *TryTimer*=*RTripDelay*
and *TryTimer'*=*TryTimeG'*=0 **and** *TryCount'*=*TryCount*+1
and *TryRecd'*=False **and** *TryAckedG'*=Off)

Rec_ACK is obtained by including the following as a conjunct to *Acked'*=True:

ENDTRY $\equiv$ (*TryTimer'*=*TryTimeG'*=*OutTimeG'*=Off)

Rec_D $\equiv$ (for some *data, ns, tn*)[Rec_1((D,*data*,*ns*,*tn*))
and ($\underline{r}$ = *ns* $\rightarrow$ (*Sink*[*r*]' = *data* **and** *r'* = *r*+1))
and (*ACKTimer*$\neq$Off $\rightarrow$ *ACKTimer'*=*ACKTimeG'*=0)
and ((*Acked*=False **and** *tn*=*TryCount*) $\rightarrow$ *TryRecd'*=True)]

Send_ACK $\equiv$ $Send_2$((ACK,$\underline{r}$)) **and** *ACKTimer'*=*ACKTimeG'*=Off **and**
((*Acked*=False **and** *TryRecd*=True **and** *r*=*s* **and** *TryAckedG*=Off)
$\rightarrow$ *TryAckedG'*=0)

There are now three time events: the ideal time event which affects all ideal timers, the local time event of P_1 which affects all local timers in $\mathbf{v}_1$, and the local time event of P_2 which affects all local timers in $\mathbf{v}_2$. Each time event occurrence preserves the accuracy axioms and the four timer axioms TA_1, TA_2, TA_3, and TA_4.

In addition to affecting the timers, the ideal time event has the following conjunct for updating β_1 and β_2:

((*Acked*=False **and** *TryTimeG*=$Delay_1$ **and** *TryRecD*=False) $\rightarrow$ $\beta_1'=\beta_1+1$)
and (*Acked*=False **and** *TryAckedG*=$Delay_2$) $\rightarrow$ $\beta_2'=\beta_2+1$)

Note that β_2 actually indicates the overdelay count for current (ACK,$\underline{s}$) messages, and not for all (ACK,$\underline{s}$) messages.

6.2 Safety and liveness verification

Because each event in this protocol is a refinement of some event of the earlier protocol in Section 5, the safety properties $A_{1\text{-}9}$ continue to hold for this protocol.

Because all the timer axioms in this protocol are implementable, Theorem 1 implies that every running timer increases until it is either stopped or restarted. This together with the liveness verification of the protocol in Section 5 provides a liveness verification of this protocol.

6.3 Real-time progress verification

The following is obviously invariant (proof in Appendix B):

$$D_1 \equiv \; Acked{=}\text{False} \Rightarrow (OutTimeG \leq TryTimeG + (TryCount\text{-}1) \times RTDelay \;\; \textbf{and} \;\; (TryTimer,\, TryTimeG) \text{ started-together } \textbf{and} \;\; TryCount \geq 1)$$

Just after AcceptData occurs, we have $TryCount{=}1$ and $\beta_1+\beta_2{=}0$. In general, just after a try, we expect $TryCount{=}\beta_1+\beta_2+1$ to hold, and we have $TryTimeG{=}TryTimer{=}0$. Let us assume that *Acked* is still False when *TryTimer* attains *RTripDelay*. At this point *TryTimeG* has exceeded *RoundTripDelay* and the next try will occur. We expect $\beta_1+\beta_2$ to be incremented by 1 some time between the two tries. Specifically, at any instant between the two tries we expect either $TryCount{=}\beta_1+\beta_2$ or one of the following to hold:

(1) $TryTimeG \leq Delay_1$ and the current D message has not yet been received at P_2.

(2) P_2 received a current D message when *TryTimeG* had the value $t_1 \leq Delay_1$, and P_2 has not yet responded to it, and $TryTimeG \leq t_1 + MaxResponseTime$.

(3) P_2 sent a current ACK when *TryTimeG* had the value $t_2 \leq Delay_1 + MaxResponseTime$, and $TryTimeG \leq t_2 + Delay_2$.

Formally, we have

$D_2 \equiv Acked$=False $\Rightarrow$ ($D_{2.1}$ **or** $D_{2.2}$ **or** $D_{2.3}$ **or** $D_{2.4}$)
where
$D_{2.1} \equiv$ *TryTimeG* in [0..$Delay_1$] **and** *TryRecd*=False **and** $TryCount=\beta_1+\beta_2+1$

$D_{2.2} \equiv$ (for some t_1 in [0..$Delay_1$])
[*TryTimeG* in [t_1..t_1+*MaxResponseTime*] **and** *TryRecd*=True **and** $r=s$
and *ACKTimeG* $\geq$ *TryTimeG*-t_1
and ((*ACKTimeG*, *ACKTimer*) started-together)
and $TryCount=\beta_1+\beta_2+1$]

$D_{2.3} \equiv$ (for some t_2 in [0..$Delay_1$+*MaxResponseTime*])
[*TryTimeG* in [t_2..t_2+$Delay_2$] **and** *TryAckedG* = *TryTimeG*-t_2
and $TryCount=\beta_1+\beta_2+1$]

$D_{2.4} \equiv$ *TryTimeG* $\geq$ min($Delay_1$, $Delay_2$) + 1 **and** $TryCount=\beta_1+\beta_2$

D_2 satisfies the safety inference rule with $I \equiv D_1$ **and** A_7 (proof in Appendix B). D_2 implies that ($TryCount=\beta_1+\beta_2+1$) is invariant. This and the channel real-time progress axiom imply $TryCount \leq M$. The timer axiom for *TryTimer* and D_1 imply *TryTimeG*<*RTDelay*. D_0 follows by plugging the last two inequalities into D_1.

REFERENCES

[1] Bartlett, K. A. et al, "A note on reliable full-duplex transmission over half-duplex links," *Commun. of the ACM*, May 1980.

[2] Chandy, K. M. and J. Misra, "An Example of Stepwise Refinement of Distributed Programs," *ACM Trans. on Prog. Lang. and Syst.*, Vol. 8, No. 3, July 1986.

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

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

[5] Francez, N. and A. Pnueli, "A Proof Method for Cyclic Programs," *Acta Informatica*, September 1978, pp. 133-157.

[6] Hailpern, B. T. and S. S. Owicki, "Modular verification of computer communication protocols," *IEEE Trans. on Commun.*, COM-31, 1, January 1983.

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

[8] 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.

[9] International Standards Organization, "Information Processing Systems—Open Systems Interconnection—Transport Protocol Specifications," Ref. No. ISO/TC 97/SC 16 N 1990, DIS 8073 Rev., September 1984,

[10] Knuth, D. E., *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition,* pp. 20, Addison-Wesley, 1973.

[11] Knuth, D. E., "Verification of Link-Level Protocols," *BIT*, Vol. 21, pp. 31-36, 1981.

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

[13] Lamport, L. "An assertional correctness proof of a distributed algorithm," *Science of Computer Programming,* Vol. 2, North-Holland, Amsterdam, 1982, pp. 175-206.

[14] Lamport, L., "Specifying Concurrent Program Modules," *ACM Trans. on Prog. Lang. and Syst.,* Vol. 5, No. 2, April 1983, pp. 190-222.

[15] Lamport, L., "What Good is Temporal Logic?," *Proc. IFIP 9th World Congress,* IFIP, North Holland, Paris, September 1983.

[16] Lamport, L., "Specification Simplified," Preliminary Draft, Digital Equipment Corporation, May 1986.

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

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

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

[20] 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.

[21] 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.

[22] Shankar, A. U. and S. S. Lam, "Time-dependent communication protocols," *Tutorial: Principles of Communication and Networking Protocols*, S. S. Lam (ed.), IEEE Computer Society, 1984.

[23] Shankar, A. U. and S. S. Lam, "Time-dependent distributed systems: proving safety, liveness and real-time properties," Tech. Rep. CS-TR-1586, Computer Science Dept., Univ. of Maryland, also TR-85-24, Computer Science Dept., Univ. of Texas, October 1985.

[24] Shankar, A. U. and S. S. Lam, "Construction of Sliding Window Protocols," Tech. Rep. CS-TR-1647, Computer Science Dept., Univ. of Maryland, also TR-86-09, Computer Science Dept., Univ. of Texas, March1 1986.

[25] Shankar, A. U., "A Verified Sliding Window Protocol with Variable Flow Control," *Proc. ACM SIGCOMM '86*, Stowe, Vermont, August 1986, also Tech. Rep. CS-TR-1638, Computer Science Dept., Univ. of Maryland.

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

[27] Stenning, N. V., "A data transfer protocol," *Computer Networks*, Vol. 1, pp. 99-110, September 1976.

Appendix A

Modeling a variety of channels

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

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 2.4.
   $\text{Send}_i(m) \equiv (\mathbf{z}_i' = (\mathbf{z}_i, m))$

2. *Finite-buffer blocking channel.* Send is blocked if the channel is full.
   $\text{Send}_i(m) \equiv (|\mathbf{z}_i| < J \textbf{ and } \mathbf{z}_i' = (\mathbf{z}_i, m))$,
   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.
   $\text{Send}_i(m) \equiv$ (for some message sequences $\mathbf{a},\mathbf{b},\mathbf{c}$)(for some message n)
   $[(\mathbf{a}=(\mathbf{z}_i, m)) \textbf{ and } (|\mathbf{a}| \leq J \Rightarrow \mathbf{z}_i' = \mathbf{a})$

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 receive 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

$\mathrm{Rec}_i(m) \equiv$ (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 ChannelError. 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).

Loss $\equiv$ $(\mathbf{z}_i{=}(\mathbf{a},m,\mathbf{b})$ **and** $\mathbf{z}_i{}'{=}(\mathbf{a},\mathbf{b}))$

Duplicate $\equiv$ $(\mathbf{z}_i{=}(\mathbf{a},m,\mathbf{b})$ **and** $\mathbf{z}_i{}' = (\mathbf{a},m,m,\mathbf{b}))$

Reorder $\equiv$ $((\mathbf{z}_i{=}(\mathbf{a},m,\mathbf{b},\mathbf{c})$ **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.,

ChannelError $\equiv$ (Loss **or** Duplicate **or** Reorder)

Appendix B

Proof of D_1's invariance

Initial $\Rightarrow$ *Acked*=True $\Rightarrow$ D_1.

AcceptData: NEWTRY $\Rightarrow$ $D_1{}'$.

Send_D: From *Acked*=False and D_1, we get the right hand side of D_1; from this, RETRY, the accuracy axiom, and definition of *RTDelay*, we get $D_1{}'$.

Rec_ACK: We need only consider when *Acked'*=True which implies $D_1{}'$.

Any time event and D_1 implies $D_1{}'$.

No other event affects D_1.

End of proof

Proof of D_2's invariance

. The proof uses the invariance of D_1 and A_7.

Initial $\Rightarrow$ *Acked*=True $\Rightarrow$ D_2.

Channel events do not affect D_2. Local time events preserve the started-together statement in $D_{2.2}$; otherwise, they do not affect D_2.

AcceptData: NEWTRY implies $TryTimeG'$=0, $TryCount'$=1, $TryRecd'$=False, and $\beta_1'=\beta_2'=0$. These together imply $D_{2.1}'$.

Send_D: Send_D implies $X \equiv$ *Acked*=False **and** *TryTimerG*=*RTripDelay*. (X **and** D_1) implies $Y \equiv$ (*TryTimer*, *TryTimeG*) started-together. Y, together with the definition of *RTripDelay* and the accuracy axiom, implies that *TryTimeG* > *RoundTripDelay*. This and D_2 imply that $D_{2.4}$ holds, i.e., $Z \equiv TryCount=\beta_1+\beta_2$ holds. RETRY implies $TryTimeG'$=0, $TryCount'$=$TryCount$+1, and $TryRecd'$=False. These together with Z implies $D_{2.1}'$.

Rec_ACK: We only need consider the case $Acked'$=True, which implies D_2' vacuously.

Send_ACK: Does not affect $D_{2.1}$, $D_{2.3}$, $D_{2.4}$. Therefore, let us assume that $D_{2.2}$ holds and $D_{2.1}$, $D_{2.3}$, and $D_{2.4}$ do not hold. Then we have *TryRecd*=True, $r=s$, *Acked*=False, and *TryAckedG*=Off (because $D_{2.1}$, $D_{2.3}$, and $D_{2.4}$ do not hold). In this case, $D_{2.2}$ implies $D_{2.3}'$ with $TryAckedG'$=0 and $t_2=t_1+TryTimeG$.

Rec_D: $D_{2.2}$, $D_{2.3}$, and $D_{2.4}$ are not affected. (Note that $D_{2.2}\Rightarrow$(*ACKTimer*$\neq$Off **and** *TryRecd*=True).) ($D_{2.1}$ **and** $TryRecd'$=False) $\Rightarrow D_{2.1}'$. From $D_{2.1}$ and $TryRecd'$=True, we have $D_{2.2}'$ with $t_1=TryTimeG$: from A_7, we have $s=r'$; if *ACKTimer*=Off then we have $ACKTimer'=ACKTimeG'$=0.

Ideal time event: ($D_{2.1}$ and $TryTimeG<Delay_1$) $\Rightarrow D_{2.1}'$. ($D_{2.1}$ and $TryTimeG=Delay_1$) $\Rightarrow D_{2.4}'$. ($D_{2.2}$ and timer axiom for *ACKTimer*) $\Rightarrow D_{2.2}'$. ($D_{2.3}$ and $TryAckedG<Delay_2$) $\Rightarrow D_{2.3}'$. ($D_{2.3}$ and $TryAckedG=Delay_2$) $\Rightarrow D_{2.4}'$. $D_{2.4}\Rightarrow D_{2.4}'$.

End of proof