A Stepwise Refinement Heuristic for Protocol Construction[1]

A. Udaya Shankar[2] and Simon S. Lam[3]

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-11 April 1987

[1]This report is also distributed as Technical Report CS-TR-1812, Department of Computer Science, University of Maryland, March 1987.

[2]Work supported by National Science Foundation Grant No. ECS 85-02113. Address: Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742.

[3]Work supported by National Science Foundation Grant No. ECS 83-04734.

ABSTRACT

We present a stepwise refinement heuristic to construct distributed systems. A distributed system in our model is specified by a set of state variables and a set of events. Each event is specified by a predicate that relates the values of the system state variables immediately before the event occurrence to their values immediately after the event occurrence. At any point during the construction, we have the following: A partially constructed distributed system referred to as an image system; a set of safety and progress requirements; and a marking that identifies the extent to which the requirements are satisfied by the image system. At each step of the construction, the image system and the set of requirements are refined and the marking is increased. We also have a reversal step to undo the construction to a limited extent. The construction ends when the image system satisfies all the requirements.

We provide two construction examples. The first, a distributed counter, is a small example for illustrating our heuristic. We then construct three sliding window protocols that use modulo-N sequence numbers (for any $N \geq 2$) to provide reliable data transfer over communication channels that can lose, reorder and duplicate messages in transit. These protocols utilize timers to enforce real-time constraints necessary for their correct operation, and are easier to implement than sliding window protocols previously studied in the protocol verification literature. This example illustrates a major application of the heuristic.

TABLE OF CONTENTS

1. INTRODUCTION

We present a heuristic for constructing a distributed system in successive steps. At any point in the construction, we have the following: an *image system*, a set of *requirements*, and a *marking* that identifies the extent to which the requirements are satisfied by the image system.

An image system is a fully specified distributed system in its own right; i.e., it can be implemented. Its state variables are a subset of the state variables of the distributed system, and its events are projections of the events of the distributed system [6].

There are three types of requirements: *invariant requirements*; *event requirements*, each associated with an event of the image system; and *progress requirements.* The invariant and event requirements represent the safety properties desired of the system. They are specified by predicates in the state variables of the image system. The progress requirements represent the progress properties desired of the system. They are specified using the temporal operator *leads*–*to* [1, 10].

Every invariant requirement is satisfied by the initial values of the state variables. An invariant requirement is *marked* with respect to an event if we have proved that it holds after any occurrence of the event, assuming that all invariant requirements and all event requirements associated with the event hold before the event occurrence. An event requirement is *marked* if we have proved that it is implied by the associated event's enabling condition and the invariant requirements. A progress requirement is *marked* if we have proved that it is satisfied by the image system, assuming that the image system satisfies all the invariant and event requirements, and has a fair implementation, i.e., every event that is continuously enabled will eventually occur.

We start each construction with an image system that has just enough structure to *specify* the safety and progress properties desired of the distributed system. The construction then proceeds by successive applications of *refinement* steps that refine the image system's events and state variables; they also increase the requirement sets and the marking set. In addition to the refinement steps, we also have a *reversal* step that allows us to "undo" the construction to a limited extent, at the expense of possibly reducing the marking set. The construction terminates successfully when all requirements are marked. The construction terminates unsuccessfully when a requirement is generated that is inconsistent with other requirements or with the initial conditions of the image system.

Our construction heuristic is strongly influenced by Dijkstra's pioneering work in the formal derivation of programs using weakest preconditions [2]. We are also influenced by the method of protocol projection [6]. In particular, we depend on the fact that when an image system is refined, its marking set expands.

Chandy and Misra [1] have presented a stepwise refinement heuristic, and used it to construct a general quiescence detection algorithm. In both our approach and theirs, a distributed system is modeled by a set of state variables and events. Both approaches maintain invariant and progress requirements throughout the construction. However, there are significant differences between the two approaches. Chandy and Misra construct a distributed system starting from a single-process topology and then successively refining the topology. During their construction, such topology refinements are incorporated into the events, invariant requirements, and progress requirements. Thus, at an arbitrary point in the construction, an event may not be implementable in the distributed system because it accesses variables that are not accessible in the final topology. In our approach, the network topology is constant and the image system at any time during the construction is implementable on the topology. We use the current image system and requirements to refine the requirements, the events and the state space of the next image system.

Our construction approach also places a strong emphasis on simultaneously generating a formal verification. Each step's application can be checked by automated techniques. To reduce verification effort, we have introduced the marking set, the event requirement set, the reversal step, and predicates for event specification. (These features are not present in the method of Chandy and Misra.) Our construction heuristic does not require events to be specified by predicates; they can be specified by guarded multiple-assignment commands as in [1]. We specify events by predicates primarily to facilitate the formal verification.

1.1. Construction examples

We provide two construction examples. The first one is small and serves to motivate the heuristic. In this example, we construct a distributed counter implemented in two processes connected by error-free channels. For each process, there is a local user who can increment the counter by 1. Each process maintains a local copy of the counter. We construct a system satisfying the invariant requirement that the copies differ by at most 1, and the progress requirement that user updates are never permanently disabled.

The second example is a major application of our heuristic. We construct three sliding window protocols that use modulo-N sequence numbers to achieve correct data transfer between a source process and a destination process connected by channels that can lose, duplicate, and reorder messages arbitrarily. The protocols are constructed in two phases. The first phase is common to all three protocols. In this phase, we show that the channels must impose an upper bound on message lifetimes, and the source process must enforce certain time constraints before accepting new data blocks from its user. Such time-dependent systems are common in networking [8, 4, 14]. To construct these protocols, we use the system model developed in [9, 10] in which real-time constraints can be specified and verified as safety properties. In the second phase of the construction, we present three different ways of enforcing the time constraint requirements on the source process, resulting in three protocols. The first and second protocols use $2N$ and N timers respectively. The third protocol uses a single timer to enforce a minimum time interval between accepting successive data blocks. The single timer can be dispensed with if the required minimum time interval is enforced by the hardware (as is assumed in [8, 4]). The minimum time interval is a function of N, the receive window size, and the maximum message lifetimes. Given any lower bound on the time interval between accepting successive data blocks, the function provides the minimum value of N that ensures correct data transfer without imposing any constraints on the transmission of messages.

To our knowledge, this is the first verified construction of sliding window protocols which use modulo-N sequence numbers where N is arbitrary and messages in channels can be reordered arbitrarily. The first two sliding window protocols appear to be novel. The third sliding window protocol is best compared with the original Stenning's protocol [15]. Like our protocols, the original Stenning's protocol considers arbitrary (but fixed) send and receive windows sizes, and channels that lose, reorder, and duplicate messages. Unlike our protocols, his protocol uses unbounded sequence numbers, requires the source to resend all outstanding data messages in FIFO order at each retransmission, and requires the destination to send an acknowledgement message in response to every received data message. Knuth [5] has considered a sliding window protocol using modulo-N sequence numbers, and obtained the minimum value of N that ensures correct data transfer assuming channels that lose messages and allow messages to overtake a limited number of previously sent messages.

Because of this limitation on the reordering of messages, his protocol does not require bounded message lifetimes and timers. (Knuth also allows the N for data messages to be different from the N for acknowledgement messages.) In [12, 13], we have extended the third protocol in several respects, e.g., variable windows for flow control, selective acks, etc.

1.2. Organization of this report

In Section 1.3, we describe our system model without real-time features. In Section 2, we present the distributed counter example first, and then describe the construction heuristic. In Section 3, we present our system model with real-time features. In Section 4, we present the initial phase of the sliding window protocol construction. In Section 5, we complete this construction and present the three sliding window protocols.

1.3. System model

We model a system by a finite set of state variables $\mathbf{v}=(v_1,v_2, \cdots)$ and a finite set of events $e_1,e_2, \cdots$. We refer to $\mathbf{v}$ as the *system state vector*. The initial conditions on the state variables are specified by a predicate *Initial*. We use the term "predicate" to refer to a well-formed formula of first-order logic augmented by appropriate mathematics for the variables. When we say that a predicate is *logically* valid (implies, equivalent, etc.), we are referring to derivations within this logic.

Each event has an enabling condition and an update. The enabling condition is a predicate in $\mathbf{v}$, i.e., its free variables are from $\mathbf{v}$. The event can occur only when the value of the state vector satisfies the enabling condition. The occurrence of the event updates the value of the state vector $\mathbf{v}$. Instead of using algorithmic code, we specify the update 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. The event is specified by the conjunction of its enabling condition predicate and its update predicate. Such predicates are referred to as *event predicates*. For example, an event e_1 that is enabled whenever the state variable v_2 is less than 5 and whose action increments the state variable v_1 by 1 is defined by $e_1 \equiv (v_2<5 \wedge v_1' = v_1+1)$. (In a procedural language such as [7], e_1 could be specified by *await* $v_2<5$ *then* $v_1:=v_1+1$.) For compactness in event specifications, we have adopted the convention that if a variable v' in $\mathbf{v}'$ does not occur in an event predicate then the state variable v is not affected by the event occurrence; i.e., the conjunct $v' = v$ is implicit in the event predicate.

We shall use $enabled(e)$ to denote the enabling condition of an event e; e.g., $enabled(e_1) \equiv v_2<5$. Formally, if event e is specified by event predicate p, then $enabled(e)$ is defined to be any predicate in $\mathbf{v}$ that is logically equivalent to the predicate $\exists \mathbf{v}' (p)$.

Given a predicate A in $\mathbf{v}$, we use A' to denote A with every free occurrence of v replaced by v'. A predicate A in $\mathbf{v}$ is *invariant* if the following are logically valid: (a) $Initial \Rightarrow A$; and (b) $A \wedge e \Rightarrow A'$ for every event e. Part (a) ensures that A holds initially; part (b) ensures that A is preserved by every event occurrence.

Given predicates A and B in $\mathbf{v}$ and an event e_1, we say that A *leads-to* B *via* e_1 [1, 10] if the following are logically valid: (a) $A \Rightarrow enabled(e_1) \wedge B'$, and (b) $A \wedge e \Rightarrow A' \vee B'$ for every event e. Whenever A holds, part (a) ensures that e_1 is enabled and its occurrence makes B hold; part (b) ensures that no event can violate A without establishing B. Thus, in any fair implementation B will hold at some point. We use *leads-to* to denote the closure of the *leads-to-via* relation [1, 10]; e.g., A *leads-to* B

if A *leads-to* $B \vee C$ and C *leads-to* B.

Consider predicates A and B in $\mathbf{v}$, and an event e. We say that A is a *weakest precondition* of B with respect to e if it is logically equivalent to $\forall \mathbf{v}' \ (e \Rightarrow B')$. Note that A is False over only those states where e is enabled and its occurrence can cause B to be violated. (This corresponds to Dijkstra's weakest liberal precondition [2].) We say that A is a *sufficient* precondition if it implies $\forall \mathbf{v}' \ (e \Rightarrow B')$; i.e., satisfies $\forall \mathbf{v}' \ (A \wedge e \Rightarrow B')$. We say that A is a *necessary* precondition if it is implied by $\forall \mathbf{v}' \ (e \Rightarrow B')$; i.e., satisfies $\neg A \Rightarrow \exists \mathbf{v}' \ (e \wedge \neg B')$. Finally, we will use "wrt" as an abbreviation of "with respect to".

Distributed system model

We specialize the above model to represent a distributed system of entities P_1, P_2, $\cdots$, P_I and one-way channels C_1, C_2, $\cdots$, C_K connected in some arbitrary network topology. Both entities and channels are processes. Each process has a set of state variables and a set of events. Let $\mathbf{v}_i$ denote the set of state variables of P_i. Let $\mathbf{z}_i$ denote the sequence of messages in transit in channel C_i. The system state vector is $\mathbf{v}=(\mathbf{v}_1, \cdots, \mathbf{v}_I, \mathbf{z}_1, \cdots, \mathbf{z}_K)$.

The events of entity P_i can access the state variables in $\mathbf{v}_i$ and only those $\mathbf{z}_j$'s of channels connected to P_i. Entity events model message sends and receptions, 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.

Further, entity events access $\mathbf{z}_j$'s only via *send* and *receive primitives*. The send primitive for channel C_i is defined by $Send_i(m) \equiv (\mathbf{z}_i' = \mathbf{z}_i @ m)$; i.e., append the message value m to the tail of $\mathbf{z}_i$. We use @ as the concatenation operator. The receive primitive for channel C_i is defined by $Rec_i(m) \equiv (\mathbf{z}_i = m @ \mathbf{z}_i')$; 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 $Rec_i(m)$ is 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.

2. CONSTRUCTION OF DISTRIBUTED SYSTEMS

In Section 2.1, we present the distributed counter example and use it to motivate the construction heuristic described in Section 2.2.

2.1. Distributed counter example

Consider the network of entities P_1 and P_2 connected by error-free channels C_1 and C_2 as shown in Figure 1. At each entity, there is a local user who can update the counter by adding 1 to it. Each entity P_i has an integer state variable x_i which is a local copy of the

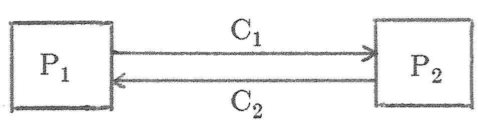

Figure 1. The network topology

counter. Initially, $x_1 = x_2 = 0$. The desired safety property is that x_1 and x_2 must not differ by more than 1 at any time. The desired progress property is that user updates should not be permanently disabled.

Clearly, an update by the local user at P_i has to be communicated to $P_j, (j \neq i)$. Let *INC* be the message used to communicate the update. To specify the progress property, we need at each P_i a state variable y_i that counts the number of local user updates. Initially, $y_1 = y_2 = 0$. The progress properties cannot be specified in terms of x_1 and x_2, because these can be increased, for example, by user updates at P_1 alone, without any user updates at P_2. We shall restrict ourselves to constructing symmetric protocols, where the specification of P_2 is identical to that of P_1 with the subscripts 1 and 2 interchanged. We start the construction with the image system defined by the following events at P_i, for $i = 1$ and 2:

$$\begin{array}{lcl} local_i & \equiv & x_i' = x_i + 1 \wedge y_i' = y_i + 1 \wedge Send_i(INC) \\ remote_i & \equiv & Rec_j(INC) \wedge x_i' = x_i + 1 \end{array}$$

The desired safety property is specified by the following invariant requirement:

$$A_0 \quad \equiv \quad x_1 - x_2 \in \{-1, 0, 1\}$$

The desired progress property is specified by the following progress requirements:

$$\begin{array}{lcl} L_0 & \equiv & y_1 = n \;\; leads\text{-}to \;\; y_1 = n+1 \\ L_{\bar{0}} & \equiv & y_2 = n \;\; leads\text{-}to \;\; y_2 = n+1 \end{array}$$

For any requirement X_i, we use $X_{\bar{i}}$ to denote X_i with the variable subscripts 1 and 2 interchanged. Because we shall consider only symmetric protocols, whenever we introduce a requirement X_i, we shall also introduce the corresponding symmetric requirement $X_{\bar{i}}$, unless they happen to be the same (as in the case of A_0).

Observe that $x_2 - y_2$ counts the number of occurrences of $remote_2$. Because C_1 is initially empty, we expect $y_1 \geq x_2 - y_2$ to hold. We shall go one step further and restrict our construction to protocols that satisfy the following invariant requirement (the corresponding symmetric requirement is also stated):

$$\begin{array}{lcl} A_1 & \equiv & y_1 - (x_2 - y_2) \in \{0, 1\} \\ A_{\bar{1}} & \equiv & y_2 - (x_1 - y_1) \in \{0, 1\} \end{array}$$

Note that $A_{1,\bar{1}}$ implies A_0. (We use $A_{i,j}$ to denote $A_i \wedge A_j$.) If $local_1$ is to preserve A_1, then the following must hold prior to $local_1$'s occurrence (more formally, the following is a weakest precondition of A_1 wrt $local_1$):

$$S_0 \quad \equiv \quad y_1 = x_2 - y_2$$

S_0 is an event requirement associated with $local_1$. How can we enforce S_0 prior to every occurrence of $local_1$? We could insist that S_0 holds whenever $local_1$ is enabled. However, because the current $local_1$ is always enabled, this corresponds to requiring S_0 to hold invariantly. Such a step would inhibit $local_1$ from ever occurring, thereby invalidating L_0. The only alternative is to increase the resolution of the system state space.

We introduce at P_1 a boolean state variable b_1 and let b_1=True be the enabling condition of $local_1$. Initially, b_1=True. Obviously, S_0 cannot hold after $local_1$ occurs. Therefore, $local_1$ must be disabled after each occurrence. This is accomplished by setting b_1 to False in $local_1$. Thus, we have the following *refined* version of $local_1$:

$$local_1 \quad \equiv \quad b_1 \wedge x_1' = x_1+1 \wedge y_1' = y_1+1 \wedge Send_1(INC) \wedge \neg b_1'$$

We can now enforce event requirement S_0 with the following invariant requirement:

$$A_2 \quad \equiv \quad b_1 \Rightarrow y_1 = x_2 - y_2$$

Event $remote_2$ preserves A_1 iff $y_1=x_2-y_2+1$ holds prior to its occurrence. Let $|\mathbf{z}_{1,INC}|$ denote the number of *INC* messages in $\mathbf{z}_1$. Because $remote_2$ is enabled only if $|\mathbf{z}_{1,INC}| \geq 1$, we can enforce the event requirement S_0 by having $|\mathbf{z}_{1,INC}| \geq 1 \Rightarrow y_1=x_2-y_2+1$ as an invariant requirement. Because $remote_2$ invalidates the consequent $y_1=x_2-y_2+1$, we require it to also invalidate the antecedent $|\mathbf{z}_{1,INC}| \geq 1$; i.e., C_1 must contain at most one *INC* message. This is summarized in the following invariant requirement:

$$A_3 \quad \equiv \quad |\mathbf{z}_{1,INC}| = 0 \vee (|\mathbf{z}_{1,INC}| = 1 \wedge y_1 = x_2 - y_2 + 1)$$

The corresponding modifications of adding state variable b_2 and refining $local_2$ are made to P_2. We also have the corresponding symmetric requirements $S_{\bar{0}}$ and $A_{\bar{2},\bar{3}}$. Henceforth, we shall not explicitly list these corresponding symmetric requirements and modifications. The following marking now holds (proof below), where A_{i-j} denotes $A_i \wedge A_{i+1} \wedge \cdots \wedge A_j$, and $e{:}A$ denotes that e is marked wrt A:

$local_1$: $A_{0\text{-}3,\bar{1}\text{-}\bar{3}}$, S_0	$remote_1$: $A_{0\text{-}3,\bar{1}\text{-}\bar{3}}$	$local_2$: $A_{0\text{-}3,\bar{1}\text{-}\bar{3}}$, $S_{\bar{0}}$	$remote_2$: $A_{0\text{-}3,\bar{1}\text{-}\bar{3}}$

Proof. Given $local_1$, we have $A_2 \Rightarrow A_1'$, A_2', $A_{2,3} \Rightarrow A_3'$, $A_i \Rightarrow A_i'$ for $i=\bar{1}\text{-}\bar{3}$, and $A_2 \Rightarrow S_0$. Given $remote_2$, we have $A_3 \Rightarrow A_{1,2}'$, $A_3 \Rightarrow A_3'$, and $A_{\bar{1}\text{-}\bar{3}}$ is *not affected* (i.e., no variable in $A_{\bar{1}\text{-}\bar{3}}$ is updated by $remote_2$). The markings for $local_2$ and $remote_1$ follow from symmetry; i.e., $local_1$ is marked wrt to A_i ($A_{\bar{i}}$) iff $local_2$ is marked wrt $A_{\bar{i}}$ (A_i). Also, recall that A_0 is the same as $A_{\bar{0}}$. **End of proof**.

We still need a marking for L_0. To preserve L_0, we require an event at P_1 that resets b_1 to True. Denote this event by $reset_1$. If $reset_1$ is to preserve A_2, then $y_1=x_2-y_2$ must hold prior to its occurrence. From A_3, we know that $remote_2$ establishes $y_1=x_2-y_2$. Thus, we refine $remote_2$ to send a new message *ACK*, and refine $reset_1$ to occur only upon the reception of an *ACK*:

$$\begin{aligned} remote_2 \quad &\equiv \quad Rec_1(INC) \wedge x_2' = x_2+1 \wedge Send_2(ACK) \\ reset_1 \quad &\equiv \quad Rec_2(ACK) \wedge b_1' \end{aligned}$$

The earlier marking of $remote_2$ wrt $A_{0\text{-}3,\bar{1}\text{-}\bar{3}}$ continues to hold because $A_{0\text{-}3,\bar{1}\text{-}\bar{3}}$ does not involve $\mathbf{z}_2$, while the modification to $remote_2$ updates only $\mathbf{z}_2$.

We can now enforce the event requirement $y_1 = x_2 - y_2$ of $reset_1$ by requiring $|\mathbf{z}_{2,ACK}| \geq 1 \Rightarrow y_1 = x_2 - y_2$ to be invariant, where $|\mathbf{z}_{2,ACK}|$ is the number of ACK messages in $\mathbf{z}_2$. Because $local_1$ violates the consequent $y_1 = x_2 - y_2$, we require $|\mathbf{z}_{2,ACK}| \geq 1 \Rightarrow \neg b_1$ to be invariant. Because $reset_1$ violates the consequent $\neg b_1$, we require $|\mathbf{z}_{2,ACK}| \leq 1$ to be invariant. This is summarized in the following invariant requirement:

A_4	$\equiv$	$|\mathbf{z}_{2,ACK}| = 0 \vee (\, |\mathbf{z}_{2,ACK}| = 1 \wedge y_1 = x_2 - y_2 \wedge \neg b_1)$

The previous marking can now be extended to the following:

$local_1$: $A_{0\text{–}4,\bar{1}\text{–}\bar{4}}$, S_0	$remote_1$: $A_{0\text{–}4,\bar{1}\text{–}\bar{4}}$	$reset_1$: $A_{0\text{–}4,\bar{1}\text{–}\bar{4}}$
$local_2$: $A_{0\text{–}4,\bar{1}\text{–}\bar{4}}$, $S_{\bar{0}}$	$remote_2$: $A_{0\text{–}4,\bar{1}\text{–}\bar{4}}$	$reset_2$: $A_{0\text{–}4,\bar{1}\text{–}\bar{4}}$

Proof. Given $local_1$, we have $A_4 \Rightarrow A_4'$ and $A_{\bar{4}}$ not affected. Given $remote_2$, we have $A_{3,4} \Rightarrow A_4'$ and $A_{\bar{4}}$ not affected. Given $reset_1$, we have $A_4 \Rightarrow A_{2,4}'$, and $A_{0,1,3,\bar{1}\text{–}\bar{4}}$ not affected. Markings for $local_2$, $remote_1$, $reset_2$ follow from symmetry. **End of proof**.

We have $b_1 \wedge y_1 = n$ *leads-to* $y_1 = n+1$ *via* $local_1$. Thus, to establish L_0 it is sufficient to establish $\neg b_1$ *leads-to* b_1. We have $|\mathbf{z}_{2,ACK}| \geq 1$ *leads-to* b_1 *via* $reset_1$, and $|\mathbf{z}_{1,INC}| \geq 1$ *leads-to* $|\mathbf{z}_{2,ACK}| \geq 1$ *via* $remote_2$. Thus, we can mark L_0 by introducing the following invariant requirement:

A_5	$\equiv$	$\neg b_1 \Rightarrow |\mathbf{z}_{1,INC}| \geq 1 \vee |\mathbf{z}_{2,ACK}| \geq 1$

Both $local_1$ and $remote_2$ imply A_5' because they send a message. Event $reset_1$ implies b_1' which implies A_5' . These events do not affect $A_{\bar{5}}$.

At this point, all of the invariant requirements are marked wrt all events, and all event and progress requirements are marked. The construction is terminated successfully. Each P_i is specified by state variables x_i, y_i, b_i with initial values 0, 0, True, respectively, and the latest versions of the events $local_i$, $remote_i$, $reset_i$.

2.2. Construction heuristic

We now provide a description of the construction heuristic. The construction of a distributed system proceeds in successive steps. At any point during the construction, we have the following:

(a) An image system specified by a finite set of state variables $\mathbf{v}$, initial condition predicate *Initial*, and a finite set of events $e_1, e_2, \cdots$.

(b) A finite set of invariant requirements $A_0, A_1, \cdots$. Each A_i is a predicate in $\mathbf{v}$. We use the notation A to denote the conjunction of all the A_i's that are currently defined. *Initial* $\Rightarrow A$ is always logically valid. (We want A to hold at all times.)

(c) A finite set of event requirements $S_0, S_1, \cdots$. Each S_i is a predicate in $\mathbf{v}$ that is associated with an event. We use the notation S_e to denote the conjunction of all the S_i's that are currently associated with event e. (We want S_e to hold prior to any occurrence of e.)

(d) A finite set of progress requirements $L_0, L_1, \cdots$. Each L_i is a *leads-to* or a *leads-to-via* statement.

(e) Marking:

An event e is *marked* with respect to an invariant requirement A_i if $A \wedge S_e \wedge e \Rightarrow A_i'$ is logically valid.

An event requirement S_i of event e is *marked* if $A \wedge enabled(e) \Rightarrow S_i$ is logically valid.

A progress requirement B *leads-to* C *via* e_1 is *marked* if the following are logically valid: $B \wedge A \wedge S_{e_1} \Rightarrow enabled(e_1) \wedge C'$, and $B \wedge A \wedge S_e \wedge e \Rightarrow B' \vee C'$ for every event e.

A progress requirement B *leads-to* C is *marked* if it can be derived from the closure of the marked *leads-to-via* requirements.

We begin with an image system that has just enough structure to specify the safety and progress properties desired of the distributed system to be constructed. The marking is initially empty. Typically, the set of event requirements is also empty because the desired safety properties can be specified in terms of invariant requirements alone.

The construction terminates successfully when the following conditions hold: (a) every S_i is marked; (b) every event e is marked with respect to every A_i; (c) every L_i is marked. Condition (a) implies that $A \wedge e \Rightarrow S_e$ holds. This and condition (b) imply that $A \wedge e \Rightarrow A'$ holds for every event e. We always have $Initial \Rightarrow A$. Thus, A is invariant. This and condition (c) imply that the image system satisfies all the L_i's.

Let us formalize the notion of *event refinement.* Recall that every event e is specified by an event predicate. Let the specification of event e be changed from event predicate p to event predicate q. We say that e is *refined* if $A \wedge S_e \wedge q \Rightarrow p$ holds. In other words, the effect that the new e can have on the state vector $\mathbf{v}$ is a special case of the effect that the old e can have on the state vector $\mathbf{v}$. (This definition of event refinement can be extended to include sequential composition of events [10].)

We have the following rather obvious *replacement property.* Given a predicate B that is logically implied by A, we can replace an invariant requirement A_i by $B \Rightarrow A_i$ or $B \wedge A_i$. Recall that S_e is the conjunction of all event requirements associated with an event e. Given a predicate B that is logically implied by $A \wedge S_e$, we can replace an event requirement S_i of e by $B \Rightarrow S_i$ or $B \wedge S_i$.

We now describe some *refinement* steps that can be applied during a construction. We shall refer to steps in the distributed counter construction for illustration.

1. **Refinement of event requirements**

Let event e be unmarked wrt invariant requirement A_i. Obtain a weakest precondition B of A_i wrt e. If B is not logically implied by $A \wedge S_e$, then include B as a new event requirement of e. Mark e wrt A_i. (This is how S_0 is obtained in the distributed counter example.)

If the predicate expression for a weakest precondition is unmanageable (and this depends on our ingenuity and patience [2]), then we can obtain either a sufficient precondition or a necessary precondition. In the latter case, e remains unmarked wrt A_i; this is still a useful step because it increases the set of requirements.

In any case, because of the replacement properties, we can always replace the precondition B with a predicate equivalent to $C \Rightarrow B$ or $C \wedge B$, where C is logically implied by $A \wedge S_e$. Thus, we can define a weakest precondition B to be a predicate that is logically equivalent to $\forall \mathbf{v}' \ (A \wedge S_e \wedge e \Rightarrow A_i')$. Similarly, we can define a sufficient precondition B to be a predicate that satisfies $\forall \mathbf{v}' \ (A \wedge S_e \wedge B \wedge e \Rightarrow A_i')$. Predicate expressions obtained from these definitions are often much simpler than expressions obtained from the original definitions of weakest and sufficient preconditions in Section 1.3.

2. **Refinement of invariant requirements**

   Given an event requirement S_i associated with event e, we can introduce *enabled*$(e) \Rightarrow S_i$ as a new invariant requirement. As a result, S_i is marked. (Illustrated in the distributed counter example by the derivation of A_2 from S_0.)

   Similarly, an invariant requirement can be introduced in order to mark a progress requirement. (Illustrated in the distributed counter example by the derivation of A_5 from L_0.)

3. **Refinement of event enabling conditions**

   Let event e of entity P_i have an event requirement S_i which refers only to $\mathbf{v}_i$. Let e be defined by the predicate p. Then we can refine e to be $S_i \wedge p$. As a result, S_i is marked. The marking of e with respect to invariant requirements is not affected by this refinement. However, this refinement step can potentially unmark a *leads-to-via* progress requirement involving e, which in turn can unmark *leads-to* progress requirements.

4. **Refinement of system state vector and events**

   Let the state vector $\mathbf{v}$ be augmented with new state variables $\mathbf{u}$, and *Initial* be augmented with new conjuncts that define initial conditions for state variables in $\mathbf{u}$.

   An existing event e can be modified with the addition of updates to the new state variables in $\mathbf{u}$. If the new predicate defining e is an *event refinement* of the old predicate definition of e, no marking is affected (otherwise, see step 6 below).

   A completely new event e that updates only state variables in $\mathbf{u}$ can be introduced; such an event is marked with respect to all existing A_i.

   The above refinements are illustrated in the distributed counter example by the addition of state variable b_i and the refinement of event $local_i$.

5. **Introduction of new messages**

   To introduce a new message m that is sent by P_i to P_j via C_k, we introduce $Send_k(m)$ in a new or existing event e_i of P_i, and introduce $Rec_k(m)$ in a new or existing event e_j of P_j.

   If e_i (or e_j) was previously marked wrt to an A_i, then that marking can remain only if A_i does not refer to $\mathbf{z}_k$ or refers to $\mathbf{z}_k$ only in functions or predicates whose values are not affected by adding m to the tail of $\mathbf{z}_k$ (or removing m from the head of $\mathbf{z}_k$); e.g., a function that returns the number of m_0's in $\mathbf{z}_k$ where m_0 is a previously defined message, and a predicate that is True iff a given sequence of messages is in $\mathbf{z}_k$.

   This step is illustrated in the distributed counter example by the addition of the message ACK.

6. **Modification of system events**

   An existing event e defined by predicate p can be redefined to be a predicate q, where q is not an event refinement of p. Every marking involving e has to be reexamined, and unmarked if it no longer holds.

A new event e that updates existing state variables can be introduced. Such an event is unmarked wrt to every existing A_i. (Illustrated in the distributed counter example by the addition of event $reset_i$.)

This is the reversal step that allows us to undo the construction to a limited extent. We must be careful not to modify the updates to state variables that were used to specify the desired properties at the start of the construction. Otherwise, these state variables may not have the meaning intended when they were used to specify the desired properties.

General observations

The construction terminates unsuccessfully whenever we have a requirement that is logically inconsistent with the other requirements or with the initial conditions; e.g., an A_i such that $A_i \Rightarrow \neg A \vee \neg Initial$, or an S_i of event e such that $S_i \Rightarrow \neg A \vee \neg S_e \vee \neg Initial$.

Generating a precondition that is only sufficient (and not necessary) and including it as an event requirement may cause unsuccessful termination later on. Generating an invariant requirement from an event or progress requirement may have a similar effect if it is done without an adequate resolution in the system state space (as defined by the state vector $\mathbf{v}$). New state variables should be introduced whenever it is determined that the generation of an invariant requirement will cause unsuccessful termination.

It is often very convenient to generate a precondition wrt a sequence of events, rather than just one event. For example, B_0 is a necessary precondition of A_i wrt to a sequence of events $e_1, \cdots, e_n$ if there exists $B_1, B_2, \cdots, B_n$ such that $\neg B_{i-1} \Rightarrow e_i \wedge \neg B_i'$ for $i=1,..,n$, and $\neg B_n \Rightarrow \neg A_i$.

3. REAL-TIME SYSTEM MODEL

For the sliding window protocol construction, we require a system model in which real-time constraints can be formally specified and verified. Such a real-time model has been presented in [10]. We now give a summary description of that model, adequate for our purposes here.

The system model presented in Section 1.3 is augmented with special state variables, referred to as *timers*, and with *time events* to age the timers. A timer takes values from the domain {Off,0,1,2, $\cdots$ }. Define the function *next* on this domain by $next$(Off)=Off and $next(i)=i+1$ for $i \neq$Off. A timer can also have a maximum capacity M, for some positive integer M; in this case, $next(M)$=Off.

There are two types of timers: *local timers* and *ideal timers*. Local timers are ones implemented within individual entities of a distributed system. 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 in the system is affected. Thus, local timers in different entities are decoupled. We assume that the error in the ticking rate of the local time event of entity P_i is upper bounded by a specified constant ϵ_i; e.g., $\epsilon_i \approx 10^{-6}$ for a crystal oscillator driven clock.

We also include in our model an *ideal time event* whose occurrence updates every ideal timer in the system. The ideal time event is a hypothetical event that is assumed to occur at a *constant* rate. Ideal timers are *not* available to the implementation. Rather they are auxiliary variables that record the actual time elapsed, and are used to measure errors in the rates of local time event occurrences.

In addition to being affected by its time event, a timer of an entity can be updated to either 0 or Off by an event of that entity. The former is referred to as *starting* the timer, and the latter as *stopping* the timer. Thus, a timer that is started by an event occurrence measures the time elapsed since that event occurrence.

Given an ideal timer u and a local timer v of entity P_i, we define the predicate $started\text{-}together(u,v)$ 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 maximum error in the rate of P_i's local time event occurrences is modeled by assuming the following condition, which we shall refer to as the *accuracy axiom*:

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

An invariant requirement A_i can include *started-together* predicates. To mark such an A_i wrt to an event e (i.e., to derive $e \wedge A \Rightarrow A_i'$), we use the following two rules provided that e is not a time event:

(a) $u' = 0 \wedge v' = 0$ implies $started\text{-}together(u,v)'$.

(b) $u' = u \wedge v' = v \wedge started\text{-}together(u,v)$ implies $started\text{-}together(u,v)'$.

We use the following rule if e is a time event:

(c) $u' \neq \text{Off} \wedge v' \neq \text{Off} \wedge started\text{-}together(u,v)$ implies $started\text{-}together(u,v)'$.

Time constraints

With timers and time events, time constraints between event occurrences can be specified as safety properties. For example, let e_1 and e_2 be two events, and let v be a timer that is started by e_1 and stopped by e_2. The time constraint that e_2 *does not occur within* T time units of e_1's occurrence is modeled by having $v \geq T$ as an event requirement of e_2, or by transforming it to the invariant requirement $enabled(e_2) \Rightarrow v \geq T$. The time constraint that e_2 *must occur within* T time units of e_1's occurrence is modeled by having $v \leq T$ as an invariant requirement.

A time constraint is either implementable or derived. An *implementable* time constraint is one that is enforced by an individual process of the protocol system without any cooperation from the rest of the protocol system. In this case, the corresponding safety property is referred to as a *timer axiom*, and is assumed to hold. Obviously, arbitrary time constraints are not implementable, and hence cannot be treated as timer axioms. If the two examples above are to be implementable, then both e_1 and e_2 have to be events of the same process. Furthermore, in the second example, the enabling condition of e_2 must depend entirely on that process; e.g., e_2 cannot require the reception of a message. (See [10] for a formal definition of implementable time constraints.)

A *derived* time constraint is one that holds for the protocol system because of the interaction between the processes. In this case, it is a safety property and has to be verified, just like any other safety property. Note that our system model now has time events, in addition to the usual communication and internal events. Thus, to establish an invariant requirement A_i that involves timers, we have to ensure that it is preserved by the time events also. Because time events update only timers, we can automatically mark them wrt invariant requirements that do not refer to timers.

4. SLIDING WINDOW PROTOCOL CONSTRUCTION: INITIAL PHASE

We consider the network topology of Figure 1, where the channels C_1 and C_2 can lose, reorder and duplicate messages in transit. There is a source at P_1 that produces new data blocks, and a destination at P_2 that consumes data blocks. We want to construct a sliding window protocol that delivers data blocks to the destination in a timely manner and in the same order as they were produced.

We start by considering the most basic features found in any sliding window protocol. Refer to Figure 2. At any time, let data block 0, data block 1, ..., data block $s-1$, denote the sequence of data blocks that have been produced by the source at P_1. Of these, data blocks 0 to $a-1$ have been sent and acknowledged, while data blocks a to $s-1$ are unacknowledged. At any time at P_2, data blocks 0 to $r-1$ have been received and forwarded to the destination in sequence, while data blocks in r to $r+RW-1$ may have been received (perhaps out-of-sequence) and are temporarily buffered. The numbers r to $r+RW-1$ constitute the *receive window*; RW is its constant size.

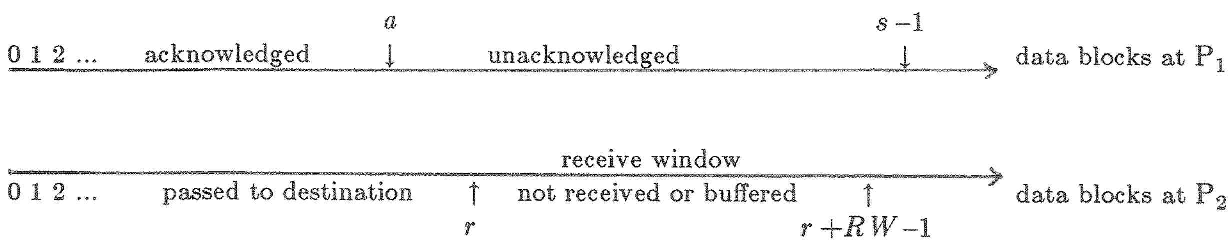

Figure 2. Relationship between a , s , r

A sliding window protocol uses modulo-N sequence numbers to identify data blocks, where $N \geq 2$. We use $\overline{n}$ to denote n mod N for any integer value n . We use $\oplus$ and $\ominus$ to denote modulo-N addition and subtraction respectively.

P_1 sends data block n accompanied by sequence number $\overline{n}$. When P_2 receives a data block with sequence number $\overline{n}$, if there is a number i in the receive window such that $\overline{i}=\overline{n}$, then the received data block is interpreted as data block i . P_2 sends acknowledgement messages containing $\overline{n}$, where n is the current value of r . When P_1 receives the sequence number $\overline{n}$, if there is a number i in the range $a+1$ to s such that $\overline{i}=\overline{n}$, then it is interpreted as an acknowledgement to data blocks a to $i-1$, and a is updated to equal i . P_1 increments s when a new data block is produced. P_2 increments r when data block r is forwarded to the destination.

Observe that each cyclic sequence number $\overline{n}$ corresponds to an *unbounded sequence number* n . When a cyclic sequence number is received at an entity, we require the entity to correctly interpret the value of the corresponding unbounded sequence number (which is not available in the message); i.e., i must equal n above. To reason about correct interpretation, we include the unbounded sequence number as an auxiliary field in the message.

4.1. Initial image system

We now formally specify the image system. Let the data messages sent by P_1 be of the type $(D, data, cn, n)$, where D is a constant that indicates the type of the message, *data* is a data block, cn is its identifying cyclic sequence number, and n is the corresponding unbounded sequence number. Let the acknowledgement messages sent by P_2 be of the type (ACK, cn, n), where cn is a cyclic sequence number, and n is the corresponding unbounded sequence number. Here, cn takes values from $[0..N-1]$, and n takes non-negative integer values. The notation $[i..j]$ denotes the sequence of integers $[i, i+1, \ldots, j]$; the sequence is empty if $i > j$.

Specification of P_1

We next list the state variables of P_1 using a Pascal-like notation for specifying their domain. *DATA* denotes the set of data blocks that can be sent in this protocol; *empty* is a constant that is not in *DATA*.

$Source$: array$[0..\infty]$ of $\{empty\} \cup DATA$; $\{Source[n]$ will record the nth data block produced by the source. Initially, $Source[0..\infty] = empty\}$

s : $0..\infty$; $\{Source[0..s-1]$ have been produced by the source. Initially, $s = 0\}$

a : $0..\infty$; $\{Source[0..a-1]$ have been acknowledged. Initially, $a = 0\}$

We now specify the events of P_1. Given an array S, we use the notation $S[n]' = d$ as an abbreviation for $S[n]' = d \wedge \forall i (i \neq n \Rightarrow S[i]' = S[i])$. ($S[n]' = d$ corresponds to the procedural statement $S[n] := d$.)

$$
\begin{array}{lcl}
sourcedata & \equiv & Source[s]' \in DATA \wedge s' = s+1 \\
senddata & \equiv & \exists n (n \in [a..s-1] \wedge Send_1(D, Source[n], \overline{n}, n)) \\
recack & \equiv & \exists cn, n (Rec_2(ACK, cn, n) \\
 & & \quad \wedge (\exists i (i \in [a+1..s] \wedge \overline{i} = cn \wedge a' = i) \\
 & & \quad \vee (\neg \exists i (i \in [a+1..s] \wedge \overline{i} = cn) \wedge a' = a)))
\end{array}
$$

Event *sourcedata* is always enabled to accept any data block. *senddata* is always enabled to send any unacknowledged data block. *recack* is always enabled to receive any (ACK, cn, n) message; the message is ignored if there is no i matching cn; the auxiliary field n is not accessed.

Specification of P_2

The state variables of P_2 are as follows:

$Sink$: array$[0..\infty]$ of $\{empty\} \cup DATA$; $\{Sink[n]$ will record the received data block interpreted as the nth data block. Initially, $Sink[0..\infty] = empty\}$

r : $0..\infty$; $\{Sink[0..r-1]$ have been passed on to the destination. Initially, $r = 0\}$

The events of P_2 are as follows:

$$
\begin{array}{lcl}
sinkdata & \equiv & Sink[r] \neq empty \wedge r' = r+1 \\
sendack & \equiv & Send_2(ACK, \overline{r}, r) \\
recdata & \equiv & \exists data, cn, n (Rec_1(D, data, cn, n) \\
 & & \quad \wedge (\exists i (i \in [r..r+RW-1] \wedge \overline{i} = cn \wedge Sink[i]' = data) \\
 & & \quad \vee (\neg \exists i (i \in [r..r+RW-1] \wedge \overline{i} = cn) \wedge Sink' = Sink)))
\end{array}
$$

sinkdata is always enabled to sink any insequence data. *sendack* is always enabled. *recdata* is always enabled to receive any $(D, data, cn, n)$ message; the message is ignored if there is no i matching cn; the auxiliary field n is not accessed.

For the sake of brevity, we have assumed that *Source*, *Sink*, s, a, r are available to the implementation. Later, we will refine the events so that these become auxiliary variables.

Specification of channels

Each C_i has a state variable $\mathbf{z}_i$ that denotes the sequence of messages in transit. C_i has an event that can lose, duplicate and reposition any message in $\mathbf{z}_i$. We will make sure that every occurrence of $\mathbf{z}_i$ in the invariant requirements is in a predicate of the form $m \in \mathbf{z}_i$ for some message m. Thus, every invariant requirement can automatically be marked wrt channel event.

4.2. Requirements

We want the protocol to satisfy the following invariant requirements:

$$A_0 \equiv 0 \leq a \leq r \leq s$$
$$A_1 \equiv n \in [0..r-1] \Rightarrow Sink[n] = Source[n]$$

A_0 specifies that data is acknowledged at P_1 only after it has been delivered to the destination at P_2, which in turn happens only after it was accepted from the source at P_1. A_1 specifies insequence delivery of data to the destination.

We want the protocol to satisfy the following progress requirement:

$$L_0 \equiv a = n \text{ leads-to } a \geq n+1$$

L_0 specifies that any data block n will eventually be acknowledged. Because of the safety properties $A_{0,1}$, this will occur only after data block n is produced and delivered to the destination. L_0 requires the following:

$$RW \geq 1$$

Otherwise, *recdata* will never place any data into $Sink[r]$; therefore *sinkdata* will never increase r; because of A_0, a will never increase.

The following invariant requirements formalize the meaning of s, $r+RW$, and the unbounded sequence number field in messages:

$$A_2 \equiv n \geq s \Leftrightarrow Source[n] = empty$$
$$A_3 \equiv n \geq r+RW \Rightarrow Sink[n] = empty$$
$$A_4 \equiv (D, data, cn, n) \text{ in } \mathbf{z}_1 \Rightarrow data = Source[n] \land cn = \overline{n}$$
$$A_5 \equiv (ACK, cn, n) \text{ in } \mathbf{z}_2 \Rightarrow cn = \overline{n}$$

Because of $A_{4,5}$, henceforth we assume that $cn = \overline{n}$ for any message in the channels.

The following invariant requirement, which states that data blocks buffered at P_2 have been correctly interpreted, is a sufficient precondition of A_1 wrt *sinkdata*:

A_6	$\equiv$	$n \in [r..r+RW-1] \wedge Sink[n] \neq empty \Rightarrow Sink[n] = Source[n]$

In fact, A_6 is a necessary precondition because a succession of *sourcedata*, *sendata*, and *sinkdata* occurrences can take $A_0 \wedge \neg A_6$ to $\neg A_1$.

Marking

The following table indicates the (e, A_i) pairs that can be marked (proof below):

sourcedata : $A_{0\text{–}6}$	*senddata* : $A_{0\text{–}6}$	*recack* : $A_{1\text{–}6}$
sinkdata : $A_{0\text{–}6}$	*sendack* : $A_{0\text{–}6}$	*recdata* : $A_{0\text{–}5}$

In the proof of the marking, we use the following conventions for the sake of brevity. We say $A \Rightarrow A_i'$ holds given an event e to mean that $e \wedge A \Rightarrow A_i'$ holds; thus, e can be marked wrt A_i. We say that A_i is *not affected* given event e if e does not update any of the variables in A_i, with the exception of a channel state variable $\mathbf{z}_i$ from which e removes a message. Because $\mathbf{z}_i$ can occur in A_i only in the form $m \in \mathbf{z}_i$, this means that e can be marked wrt A_i.

Proof of marking.
Given *sourcedata* we have $A_0 \Rightarrow A_0'$, $A_{0,1} \Rightarrow A_1'$, $A_2 \Rightarrow A_2'$, $A_{6,2} \Rightarrow A_6'$, and $A_{3\text{–}5}$ not affected.
Given *sendata* we have $A_4 \Rightarrow A_4'$, and $A_{0\text{–}3,5,6}$ not affected.
Given *recack*, we have $A_{1\text{–}6}$ not affected.
Given *sinkdata* we have $A_{6,2,0} \Rightarrow A_0'$, $A_{1,6} \Rightarrow A_1'$, $A_3 \Rightarrow A_3'$, $A_{6,3} \Rightarrow A_6'$, and $A_{2,4,5}$ not affected.
Given *sendack* we have $A_5 \Rightarrow A_5'$, and $A_{0\text{–}4,6}$ not affected.
Given *recdata* we have $A_1 \Rightarrow A_1'$, $A_3 \Rightarrow A_3'$, and $A_{0,2,4,5}$ not affected.
End of proof.

4.3. Correct interpretation of messages

Recall that every message in C_i has a cyclic sequence number $\overline{n}$ corresponding to an unbounded sequence number n. When P_j receives a message, it either ignores the message or interprets $\overline{n}$ as equal to some i. Thus, we define the *correct interpretation requirements* as follows: for any message in C_i that can be immediately received and will not be ignored, i must equal n. Note that if C_i can reorder, then any message in it can be immmediately received by P_j.

The correct interpretation requirement for data messages is specified by the following invariant requirement:

A_7	$\equiv$	$(D, data, \overline{n}, n) \in \mathbf{z}_1 \wedge i \in [r..r+RW-1] \wedge \overline{i} = \overline{n} \Rightarrow i = n$

A_7 can also be obtained directly as a necessary precondition of A_6 wrt *recdata*. Its necessity follows because $Source[i]$ and $Source[n]$ are arbitrary entries from $DATA$, and therefore $Source[i] = Source[n]$ iff $i = n$.

The correct interpretation requirement for acknowledgement messages is specified by the following invariant requirement:

A_8	$\equiv \quad (ACK, \overline{n}, n) \in \mathbf{z}_2 \wedge i \in [a+1..s] \wedge \overline{i} = \overline{n} \Rightarrow i = n$

Because of A_8, a weakest precondition of A_0 wrt *recack* is $(ACK, \overline{n}, n) \in \mathbf{z}_2 \wedge i \in [a+1..s] \wedge \overline{i} = \overline{n} \Rightarrow n \leq r$. However, it is obvious that we expect any $(ACK, \overline{n}, n)$ message in $\mathbf{z}_2$ to satisfy $n \leq r$. We specify this in the following invariant requirement:

A_9	$\equiv \quad (ACK, cn, n) \in \mathbf{z}_2 \Rightarrow n \leq r$

The corresponding requirement for data messages is $(D, data, cn, n) \in \mathbf{z}_1 \Rightarrow n \leq s-1$, which can be derived from $A_{2,7}$ and $data \neq empty$.

The previous marking can be extended to the following:

sourcedata : $A_{0\text{-}7,9}$	*senddata* : $A_{0\text{-}6,8\text{-}9}$	*recack* : $A_{0\text{-}9}$
sinkdata : $A_{0\text{-}6,8\text{-}9}$	*sendack* : $A_{0\text{-}7,9}$	*recdata* : $A_{0\text{-}9}$

Proof of extension of marking
sourcedata does not affect $A_{7,9}$. *senddata* does not affect $A_{8,9}$.
Given *recack* we have $A_{8,9} \Rightarrow A_0'$, $A_8 \Rightarrow A_8'$, and $A_{7,9}$ not affected.
Given *sinkdata* we have $A_9 \Rightarrow A_9'$, and A_8 not affected.
Given *sendack* we have $A_9 \Rightarrow A_9'$, and A_7 not affected.
Given *recdata* we have $A_{7,4} \Rightarrow A_6'$, and $A_{7,8,9}$ not affected.
End of proof.

4.4. Refining the requirement to interpret data correctly

We now determine the values of r, RW, and the unbounded sequence numbers n in C_1 that satisfy A_7. First, observe that a succession of *sourcedata* followed by a *senddata* can always take A_0 to $(D, data, \overline{r}, r) \in \mathbf{z}_1$. If $RW > N$, then $(D, data, \overline{r}, r) \in \mathbf{z}_1$ violates A_7 with $i = r + N$. Thus, $RW \leq N$ is a necessary condition. Combining this with $RW \geq 1$, we have

$1 \leq RW \leq N$

Second, observe that $i \in [r..r+RW-1] \wedge \overline{i} = \overline{n}$ iff $i-r \in [0..RW-1] \wedge \overline{i-r} = \overline{n-r}$ iff $\overline{n-r} \in [0..RW-1] \wedge i = r + \overline{n-r}$. The last equivalence is because $RW \leq N$ and $i-r \in [0..RW-1]$ imply $i-r = \overline{i-r}$. Thus, we can refine *recdata* to the following, where we have also used the modulo arithmetic property $\overline{n-r} = \overline{n} \ominus \overline{r}$:

$$\begin{aligned} recdata \quad \equiv \quad & \exists\, data, cn, n\, (\, Rec_1(D, data, cn, n) \\ & \wedge (\, (cn \ominus \overline{r} \in [0..RW-1] \wedge Sink[r + cn \ominus \overline{r}]' = data\,) \\ & \vee (cn \ominus \overline{r} \notin [0..RW-1] \wedge Sink' = Sink\,))) \end{aligned}$$

We can rewrite A_7 as the following invariant requirement:

$$(D, data, \overline{n}, n) \in \mathbf{z}_1 \wedge \overline{n-r} \in [0..RW-1] \Rightarrow n = r + \overline{n-r}$$

This specifies that every n in $\mathbf{z}_1$ must satisfy either of the following:

(i) $\overline{n-r} \in [0..RW-1] \wedge n = r + \overline{n-r}$, which is iff $n \in [r..r+RW-1]$ (because $RW \leq N$).

(ii) $\overline{n-r} \in [RW..N-1]$, which is iff $n \in [r+RW+kN..r+N-1+kN]$ for some k.

In addition to satisfying (i) and (ii), it seems reasonable to assume that the n's in C_1 satisfy the following: if C_1 contains n_1 and n_2, then it is possible for C_1 to contain any n in $[n_1..n_2]$. Finally, because of A_0 and *sendata*, it is always possible for C_1 to contain n equal to r. Thus, we are looking for a contiguous set of integers that satisfies either (i) or (ii) at each integer, and includes r. The largest set that meets these requirements is $[r+RW-N..r+N-1]$, which is obtained by taking the union of $[r..r+RW-1]$ and $[r+RW+kN..r+N-1+kN]$ for $k=0$ and -1. Thus, we have the following invariant requirement:

$$A_{10} \quad \equiv \quad (D, data, \overline{n}, n) \in \mathbf{z}_1 \Rightarrow n \in [r-N+RW..r+N-1]$$

4.5. Refining the requirement to interpret acknowledgements correctly

Proceeding as in the case of data sequence numbers above, we now determine the values of a, s, and the unbounded sequence numbers n in C_2 that satisfy A_8. First, observe the following: $\langle sendack, recack, sendack \rangle$ can take the system from A_0 to $a = r \wedge (ACK, \overline{a}, a) \in \mathbf{z}_2$; a succession of *sourcedata* can take the latter to $s - a \geq N \wedge (ACK, \overline{a}, a) \in \mathbf{z}_2$, which violates A_8 with $i = a+N$. Thus, we obtain the following invariant requirement:

$$A_{11} \quad \equiv \quad s-a \leq N-1$$

Second, observe that $i \in [a+1..s] \wedge \overline{i} = \overline{n}$ iff $i-a \in [1..s-a] \wedge \overline{i-a} = \overline{n-a}$ iff $\overline{n-a} \in [1..s-a] \wedge i = a + \overline{n-a}$. The last equivalence is because A_{11} and $i-a \in [1..s-a]$ imply $i-a = \overline{i-a}$. Thus, we can refine *recack* to the following, where we have used $\overline{n-a} = \overline{n} \ominus \overline{a}$:

$$\begin{aligned} recack \quad \equiv \quad & \exists\, cn, n\, (\, Rec_2(ACK, cn, n) \\ & \wedge (\, (cn \ominus \overline{a} \in [1..s-a] \wedge a' = a + cn \ominus \overline{a}) \\ & \vee (cn \ominus \overline{a} \notin [1..s-a] \wedge a' = a\,))) \end{aligned}$$

We can rewrite A_8 as the following invariant requirement:

$$(ACK, \overline{n}, n) \in \mathbf{z}_2 \wedge \overline{n-a} \in [1..s-a] \Rightarrow n = a + \overline{n-a}$$

This specifies that every n in $\mathbf{z}_2$ must satisfy either of the following:

(i) $\overline{n-a} \in [1..s-a] \wedge n = a + \overline{n-a}$, which is iff $n \in [a+1..s]$ (because of A_{11}).

(ii) $\overline{n-a} \in [s-a+1..N]$, which is iff $n \in [s+1+kN..a+N+kN]$ for some k.

Because of *sendack*, C_1 can always contain n equal to r. Therefore, as in the case of data sequence numbers, we are looking for a contiguous set of integers that satisfies either (i) or (ii) at each integer, and includes r. The largest set that meets these requirements is $[s+1-N..a+N]$, which is obtained by taking the union of $[a+1..s]$ and $[s+1+kN..a+N+kN]$ for $k=0$ and -1. Because of $A_{0,9,11}$, we can replace the upper bound $a+N$ by r. Thus, we have the following invariant requirement:

$$A_{12} \equiv (ACK, \bar{n}, n) \in \mathbf{z}_2 \Rightarrow n \in [s-N+1..r]$$

4.6. Constraints on accepting new data

Observe that A_{10} can be violated by *senddata* and by *sinkdata*. The *sinkdata* event can violate A_{10} by increasing r to a value m such that there is an n in C_1 that is less than $m-N+RW$. Because of A_0, we know that $m \le s$ holds. Because the channels can reorder and duplicate messages, we observe that it is always possible for $m=s$ to hold. Thus, we shall require every n in C_1 to satisfy $n \ge s-N+RW$, rather than the weaker bound $n \ge r-N+RW$ in A_{10}. Finally, observe that the upper bound $r+N-1$ in A_{10} can be replaced by $s-1$, because of $A_{0,11}$. Thus, we have the following invariant requirement:

$$A_{13} \equiv (D, data, \bar{n}, n) \in \mathbf{z}_1 \Rightarrow n \in [s-N+RW..s-1]$$

A_{13} can be violated by *senddata* and by *sourcedata*. The *senddata* event can introduce any $n \in [a..s-1]$ into C_1. This always preserves the upper bound $s-1$ in A_{13}. The lower bound is preserved iff $a \ge s-N+RW$. Thus, we have the following invariant requirement:

$$A_{14} \equiv s-a \le N-RW$$

Observe that A_{14} implies that $RW \le N-1$. Otherwise *sourcedata* can never occur and L_0 will never hold. Combining this with $1 \le RW \le N$, we have

$$1 \le RW \le N-1$$

The previous marking can be extended to the following:

sourcedata : $A_{0-7,9-10}$	*senddata* : A_{0-14}	*recack* : A_{0-14}
sinkdata : A_{0-14}	*sendack* : A_{0-14}	*recdata* : A_{0-14}

Proof of extension of marking

Given *sourcedata*, we have A_{10} not affected.
Given *senddata*, we have $A_{13,14} \Rightarrow A_{13}'$, and $A_{11,12,14}$ not affected. We can also mark $A_{7,10}$ because $A_{0,11,13} \Rightarrow A_{7,10}$.
Given *recack* we have $A_{14} \Rightarrow A_{14}'$, and $A_{10,12,13}$ not affected. We can mark A_{11} because $A_{14} \Rightarrow A_{11}$.
Given *sinkdata* we have $A_{12} \Rightarrow A_{12}'$, $A_{11,13-14}$ not affected. We can mark $A_{7,10}$ because $A_{0,11,13} \Rightarrow A_{7,10}$.
Given *sendack* we have $A_{12} \Rightarrow A_{12}'$, $A_{0,8,14} \Rightarrow A_8'$, and $A_{10,11,13,14}$ not affected.
Given *recdata* we have A_{10-14} not affected.

End of proof.

All that is left is to ensure that *sourcedata* preserves A_{12-14}. This would also mark $A_{8,11}$ because $A_{0,12,14} \Rightarrow A_{8,11}$. We now obtain the following three necessity requirements as the weakest preconditions of A_{14}, A_{13}, and A_{12} respectively wrt *sourcedata*:

S_0	$\equiv$	$s-a \leq N-RW-1$
S_1	$\equiv$	$(D, data, \overline{n}, n) \in \mathbf{z}_1 \Rightarrow n \geq s-N+RW+1$
S_2	$\equiv$	$(ACK, \overline{n}, n) \in \mathbf{z}_2 \Rightarrow n \geq s-N+2$

At this point, *sourcedata* can be marked wrt $A_{8,11-14}$. We have left the preconditions S_{0-2} as event requirements because they have exactly the same form as the invariant requirements $A_{14,13,12}$ from which they were derived, with N being replaced by $N-1$. Therefore, transforming the above S_i's into invariant requirements would merely lead us to repeat the construction with a smaller N. In fact, repeated reductions like this would eventually lead to $N=RW$, at which point we would have a dead protocol because of A_{14}.

S_0 is a requirement involving only variables of P_1. Hence it can be incorporated into the enabling condition of *sourcedata*, which is now refined to

sourcedata	$\equiv$	$s-a \leq N-RW-1 \wedge Source[s]' \in DATA \wedge s' = s+1$

This marks *sourcedata* wrt S_0. At this point, we have the following marking:

sourcedata : A_{0-14}, S_0	*senddata* : A_{0-14}	*recack* : A_{0-14}
sinkdata : A_{0-14}	*sendack* : A_{0-14}	*recdata* : A_{0-14}

4.7. Bounded message lifetime channels

All that is left is to enforce S_1 and S_2 before every occurrence of *sourcedata*. Unlike S_0, these requirements cannot be included in the enabling condition of *sourcedata* because they involve messages in $\mathbf{z}_i$ that are not accessible to P_1. Because C_1 and C_2 can reorder and duplicate messages to an arbitrary extent, it is obvious that S_1 and S_2 can only be enforced if the channels impose an upper bound on the lifetimes of messages in transit. Therefore, we assume a message cannot stay in channel C_i for longer than a specified $MaxDelay_i$ time units.

To formally specify this real-time constraint, we augment our existing image system with timers and time events. As usual, every addition we make will be a refinement of the image system. To every message in a channel, we add an auxiliary ideal timer field, denoted by *age*, that indicates the ideal time elapsed since the message was sent. The *age* field is started at 0 when the message is sent (this update is specified in the send primitive). Like any ideal timer, the *age* fields are updated to their *next* values by the ideal time event. The maximum message lifetime property is specified by the following timer axioms, which are *assumed* to be invariant for the system:

TX_1	$\equiv$	$(D, data, \overline{n}, n, age)\, in\, \mathbf{z}_1 \Rightarrow MaxDelay_1 \geq age \geq 0$
TX_2	$\equiv$	$(ACK, \overline{n}, n, age)\, in\, \mathbf{z}_2 \Rightarrow MaxDelay_2 \geq age \geq 0$

Recall that predicates of the form $(M, \mathbf{f}) \in \mathbf{z}_i$ occur in our A_i's, where $\mathbf{f}$ denote the fields of message type M. We now treat $(M, \mathbf{f}) \in \mathbf{z}_i$ as corresponding to $\exists age\, ((M, \mathbf{f}, age) \in \mathbf{z}_i)$. Because A_{0-14} do not involve timers, the ideal time event can be marked wrt them. Thus, the previous marking can be extended to the following:

$sourcedata : A_{0-14}, S_0$	$senddata : A_{0-14}$	$recack : A_{0-14}$
$sinkdata : A_{0-14}$	$sendack : A_{0-14}$	$recdata : A_{0-14}$
$ideal\ time\ event : A_{0-14}$		

4.8. An implementable time constraint that enforces S_1

In this section, we prove that S_1 is enforced if P_1 produces $Source[n]$ only after $MaxDelay_1$ ideal time units have elapsed since $Source[n-N+RW]$ was last sent. In Section 5, we implement this ideal time constraint in terms of local timers.

Define the following array of ideal timers at P_1:

T_D : array[0..∞] of ideal timer; {$T_D[n]$ will record the ideal time elapsed since $Source[n]$ was last sent. Initially, $T_D[0..\infty]$ = Off}

T_D is started in *senddata*, which is now refined to the following:

$$senddata \equiv \exists n (n \in [a..s-1] \wedge Send_1(D, Source[n], \bar{n}, n) \wedge T_D[n]' = 0)$$

The following invariant requirement specifies that $T_D[n]$ records the ideal time elapsed since $Source[n]$ was last sent:

$$A_{15} \equiv (D, data, \bar{n}, n, age) \in \mathbf{z}_1 \Rightarrow age \geq T_D[n] \geq 0$$

It is obvious from A_{15} and TX_1 that a $(D, data, \bar{n}, n, age)$ message is not in C_1 if $T_D[n]$ is either greater than $MaxDelay_1$ or Off. Thus, S_1 holds if the following time constraint holds:

$$S_3 \equiv n \in [0..s-N+RW] \Rightarrow T_D[n] > MaxDelay_1 \vee T_D[n] = \text{Off}$$

S_3 is an implementable time constraint. It implies the following invariant requirement:

$$A_{16} \equiv n \in [0..s-N+RW-1] \Rightarrow T_D[n] > MaxDelay_1 \vee T_D[n] = \text{Off}$$

A_{16} is preserved by *sendata* because $a > s-N+RW-1$, and by *sourcedata* because of S_3. Because A_{16} is an invariant requirement, P_1 can enforce S_3 by enforcing the following time constraint:

$$S_4 \equiv s \geq N-RW \Rightarrow T_D[s-N+RW] > MaxDelay_1 \vee T_D[s-N+RW] = \text{Off}$$

The above discussion is formalized in the following marking:

$sourcedata : A_{0-16}, S_{0,1,3}$	$senddata : A_{0-16}$	$recack : A_{0-16}$
$sinkdata : A_{0-16}$	$sendack : A_{0-16}$	$recdata : A_{0-16}$
$ideal\ time\ event : A_{0-16}$		

Proof of extension of marking
Given *senddata*, we have $A_{15} \Rightarrow A_{15}'$, and $A_{16,14} \Rightarrow A_{16}'$.
Given *sourcedata*, we have $S_4 \wedge A_{16} \Rightarrow A_{16}'$, $S_4 \wedge A_{16} \wedge TX_1 \Rightarrow S_{1,3}$, and A_{15} not affected.
Given *ideal time event*, we have $A_{15} \Rightarrow A_{15}'$, and $A_{16} \Rightarrow A_{16}'$.
A_{15} and A_{16} are not affected by any other event.
End of proof.

To enforce S_4, it is sufficient if P_1 tracks the ideal timers in $T_D[s-N+RW..s-1]$. This can be done with a bounded number of local timers, each of bounded counter capacity. For example, a circular array of $N-RW$ local timers, where local timer $n \bmod N-RW$ tracks $T_D[n]$ for $n \in [\max(0, s-N+RW)..s-1]$. Each local timer can be stopped once it indicates that the corresponding ideal timer has exceeded $MaxDelay_1$. (See Section 5. for several different implementations.)

4.9. An implementable time constraint that enforces S_2

In this section, we prove that S_2 is enforced if P_1 produces $Source[n]$ only after $MaxDelay_2$ ideal time units have elapsed since $Source[n-N+1]$ was acknowledged. In Section 5, we enforce this time constraint in terms of local timers.

S_2 can be enforced only by ensuring that more than $MaxDelay_2$ time units have elapsed since $(ACK, \bar{n}, n)$ was last sent, for any $n \in [0..s-N+1]$. Unlike the previous case involving data messages, P_1 does *not* have access to the time elapsed since $(ACK, \bar{n}, n)$ was last sent. This is because ACK messages are sent by P_2 and not by P_1. However, P_1 can obtain a lower bound on this elapsed time because of the following considerations: $(ACK, \bar{n}, n)$ is not sent once r exceeds n; a exceeds n only after r exceeds n; a and r are nondecreasing quantities. Thus, the time elapsed since a exceeded n is a lower bound on the ages of all $(ACK, \bar{n}, n)$ in C_2. Furthermore, this elapsed time *can* be measured by P_1.

With this motivation, define the following array of ideal timers at P_2:

T_R : array[0..∞] of ideal timer; {$T_R[n]$ will record the ideal time elapsed since r first exceeded n. Initially, $T_R[0..\infty] = \text{Off}$}

T_R is started in *sinkdata*, which is now refined to the following:

$$sinkdata \equiv Sink[r] \neq empty \wedge r' = r+1 \wedge T_R[r]' = 0$$

The following invariant requirements specify that r is nondecreasing, and that $T_R[n]$ for $n < r$ is a lower bound to the age of any $(ACK, \bar{n}, n)$ message in C_2:

$$A_{17} \equiv T_R[0] \geq T_R[1] \geq \cdots \geq T_R[r-1] \geq 0 \wedge T_R[r..\infty] = \text{Off}$$
$$A_{18} \equiv (ACK, \bar{n}, n, age) \in z_2 \wedge n < r \Rightarrow age \geq T_R[n] \geq 0$$

We define the following array of ideal timers at P_1:

T_A : array[0..∞] of ideal timer; {$T_A[n]$ will record the ideal time elapsed since a first exceeded n. Initially, $T_A[0..\infty] = \text{Off}$}

T_A is started in *recack*, which is now refined to the following:

recack	$\equiv$	$\exists cn, n\,(Rec_2(ACK, cn, n)$ $\wedge\,((cn \ominus \bar{a} \in [1..s-a] \wedge a' = a + cn \ominus \bar{a} \wedge T_A[a..a'-1]' = 0)$ $\vee\,(cn \ominus \bar{a} \notin [1..s-a] \wedge a' = a \wedge T_A' = T_A)))$

The following invariant requirements specify that a is nondecreasing, and that $T_A[n]$ is a lower bound to $T_R[n]$:

A_{19}	$\equiv$	$T_A[0] \geq T_A[1] \geq \cdots \geq T_A[a-1] \geq 0 \wedge T_A[a..\infty] = \text{Off}$
A_{20}	$\equiv$	$n \in [0..a-1] \Rightarrow T_A[n] \leq T_R[n]$

From $A_{14,18-20}$ and TX_2, we see that the following time constraint implies S_2:

S_5	$\equiv$	$s \geq N-1 \Rightarrow T_A[s-N+1] > MaxDelay_2$

The above discussion is formalized in the following marking:

sourcedata : A_{0-20}, S_{0-3}	*senddata* : A_{0-20}	*recack* : A_{0-20}
sinkdata : A_{0-20}	*sendack* : A_{0-20}	*recdata* : A_{0-20}
ideal time event : A_{0-20}		

Proof of extension of marking
Given *ideal time event*, we have $A_i \Rightarrow A_i'$ for $i = 17,18,19,20$.
Given *sourcedata*, we have $S_5 \wedge A_{14,18-20} \Rightarrow S_2$, and A_{17-20} not affected.
Given *senddata*, we have A_{17-20} not affected.
Given *recack*, we have $A_{19} \Rightarrow A_{19}'$, $A_0' \wedge A_{17,20} \Rightarrow A_{20}'$ (we can use A_0' because A_0 is marked wrt *recack*), and $A_{17,18}$ not affected.
Given *sinkdata*, we have $A_{17} \Rightarrow A_{17}'$, $A_{18} \wedge TX_2 \Rightarrow A_{18}'$, $A_{0,20} \Rightarrow A_{20}'$, and A_{19} not affected.
Given *sendack*, we have $A_{18} \Rightarrow A_{18}'$, and $A_{17,19-20}$ not affected.
Given *recdata*, we have A_{17-20} not affected.
End of proof.

P_1 can enforce S_5 by tracking the ideal timers in $T_A[s-N+1..a-1]$. This can be done, for example, with a circular array of $N-1$ local timers where local timer n mod $N-1$ tracks $T_A[n]$ for $n \in [\max(0, s-N+1)..a-1]$. Note that each local timer can be stopped once it indicates that the corresponding ideal timer has exceeded $MaxDelay_2$.

5. SLIDING WINDOW PROTOCOL CONSTRUCTION: FINAL PHASE

In this section, we complete the construction by providing three different implementations of S_4 and S_5 using local timers. For the sake of readability, we have summarized the current state of the construction in Tables 1, 2, and 3. Table 1 lists the current invariant and event requirements. Tables 2 and 3 list the current specifications of P_1 and P_2. Recall from the previous marking that the only requirements that the current image system does not satisfy are the requirements S_4 and S_5 for *sourcedata*.

5.1. Protocol implementation with 2N timers

In Sections 4.8 and 4.9, we outlined how P_1 can implement S_4 and S_5 with two circular arrays of $N-RW$ and $N-1$ local timers, respectively. We now provide a formal specification and verification of that implementation. For the sake of notational convenience, we use two circular arrays of size N, rather than of sizes $N-RW$ and $N-1$. This allows us to avoid modulo $N-RW$ and $N-1$ arithmetic.

Given an ideal timer u and a local timer v of P_1 which are started together, from the accuracy axiom it is clear that $u > T$ holds if $v \geq 1+(1+\epsilon_1)T$, 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.

Enforcing S_4

Define the following array of local timers at P_1:

$Timer_D$: array[0..$N-1$] of local timer of capacity $MDelay_1$; {Initially, $Timer_D[n]=\text{Off}$}

For $n \in [\max(0, s-N+RW)..s-1]$, $Timer_D[\overline{n}]$ will be started together with $T_D[n]$. Therefore, it will track $T_D[n]$ upto $MDelay_1$ local time units with an accuracy of ϵ_1. The *send-data* event is now refined to the following:

$$senddata \equiv \exists n (n \in [a..s-1] \wedge Send_1(D, Source[n], \overline{n}, n) \wedge T_D[n]' = 0 \wedge Timer_D[\overline{n}]' = 0)$$

The relationship between $Timer_D$ and T_D is formalized in the following invariant requirement:

$$\begin{aligned} B_0 \equiv\ & n \in [\max(0, s-N+RW)..s-1] \Rightarrow \\ & \quad started\text{-}together(Timer_D[\overline{n}], T_D[n]) \\ & \quad \vee (Timer_D[\overline{n}] = \text{Off} \wedge T_D[n] > MaxDelay_1) \\ & \quad \vee (Timer_D[\overline{n}] = \text{Off} \wedge T_D[n] = \text{Off}) \end{aligned}$$

From B_0 and S_0, we easily derive $Timer_D[\overline{s-N+RW}] = \text{Off} \Rightarrow S_4$. Also, observe that $\overline{s-N+RW} = \overline{s} \oplus RW$. Thus, we can enforce S_4 by including $Timer_D[\overline{s}\oplus RW] = \text{Off}$ in the enabling condition of *sourcedata*, which is now refined to the following:

$$sourcedata \equiv s-a \leq N-RW-1 \wedge Timer_D[\overline{s}\oplus RW] = \text{Off} \wedge Source[s]' \in DATA \wedge s' = s+1$$

The previous marking can be extended to the following:

$sourcedata : A_{0-20}, B_{0-2}, S_{0-4}$	$senddata : A_{0-20}, B_{0-2}$	$recack : A_{0-20}, B_{0-2}$
$sinkdata : A_{0-20}, B_{0-2}$	$sendack : A_{0-20}, B_{0-2}$	$recdata : A_{0-20}, B_{0-2}$
$ideal\ time\ event : A_{0-20}, B_{0-2}$		

where

B_1	$\equiv$	$n \in [s..\max(s+RW-1,\ N-1)] \Rightarrow Timer_D[\overline{n}]=\text{Off}$
B_2	$\equiv$	$n \in [s..\infty] \Rightarrow T_D[n]=\text{Off}$

Proof of extension of marking
We use the following modulo arithmetic property in the proof:
(*) $\forall i,j \in [\max(0,\ s-N+RW)..s-1]\ (i=j \text{ iff } \overline{i}=\overline{j})$
Given *sourcedata*, we have $B_{0\text{-}2} \Rightarrow B_0'$, $B_1 \Rightarrow B_1'$, and $B_2 \Rightarrow B_2'$.
Given *senddata*, we have $A_{14} \wedge B_0 \wedge (*) \Rightarrow B_0'$, $B_1 \Rightarrow B_1'$, and $B_2 \Rightarrow B_2'$.
Given *ideal time event*, we have $B_0 \Rightarrow B_0'$, $B_2 \Rightarrow B_2'$, and B_1 not affected.
Given *local time event* of P_1, we have $B_0 \Rightarrow B_0'$, $B_1 \Rightarrow B_1'$, and B_2 not affected.
$B_{0\text{-}2}$ is not affected by any other event.
End of proof.

Enforcing S_5

Define the following array of local timers at P_1:

$Timer_A$: array[0..N-1] of local timer of capacity $MDelay_2$; {Initially, $Timer_A[0..N-1] = \text{Off}$}

For $n \in [\max(0,\ s-N+1)..a-1]$, $Timer_A[\overline{n}]$ is started together with, and will track $T_A[n]$ upto $MDelay_2$ local time units with an accuracy of ϵ_1. The *recack* event is now refined to the following:

$$recack \equiv \exists cn,n(\ Rec_2(ACK,cn,n)\ \wedge ((cn\ominus\overline{a} \in [1..s-a] \wedge a' = a+cn\ominus\overline{a} \wedge \forall i \in [a..a'-1](T_A[i]' = Timer_A[\overline{i}]' = 0)) \vee (cn\ominus\overline{a} \notin [1..s-a] \wedge a' = a \wedge T_A' = T_A \wedge Timer_A' = Timer_A)))$$

The relationship between $Timer_A$ and T_A is formalized in the following invariant:

$$B_3 \equiv n \in [\max(0,\ s-N+1)..a-1] \Rightarrow started\text{-}together(Timer_A[\overline{n}], T_A[n]) \vee (Timer_A[\overline{n}] = \text{Off} \wedge T_A[n] > MaxDelay_2)$$

From B_3, S_0, and the fact that $\overline{s-N+1} = \overline{s}\oplus 1$, we easily derive that $Timer_A[\overline{s}\oplus 1] = \text{Off}$ implies S_5. Thus, we enforce S_5 by refining *sourcedata* to the following:

$$sourcedata \equiv s-a \leq N-RW-1 \wedge Timer_D[\overline{s}\oplus RW] = \text{Off} \wedge Timer_A[\overline{s}\oplus 1] = \text{Off} \wedge Source[s]' \in DATA \wedge s' = s+1$$

The previous marking can be extended to the following:

sourcedata : $A_{0\text{-}20}, B_{0\text{-}4}, S_{0\text{-}5}$	*senddata* : $A_{0\text{-}20}, B_{0\text{-}4}$	*recack* : $A_{0\text{-}20}, B_{0\text{-}4}$
sinkdata : $A_{0\text{-}20}, B_{0\text{-}4}$	*sendack* : $A_{0\text{-}20}, B_{0\text{-}4}$	*recdata* : $A_{0\text{-}20}, B_{0\text{-}4}$
ideal time event : $A_{0\text{-}20}, B_{0\text{-}4}$		

where

$$B_4 \quad \equiv \quad n \in [a..\max(s, N-1)] \Rightarrow Timer_A[\overline{n}]=\text{Off}$$

Proof of extension of marking
We use the following modulo arithmetic property which holds because of A_{14}:
(*) $\forall i, j \in [\max(0, s-N+1)..a-1]$ ($i=j$ iff $\overline{i} = \overline{j}$)
Given *sourcedata*, we have $B_3 \Rightarrow B_3'$, and $B_4 \Rightarrow B_4'$.
Given *recack*, we have $A_{14,19} \wedge B_{3\text{-}4} \wedge (*) \Rightarrow B_3'$, and $B_4 \Rightarrow B_4'$.
Given *ideal time event*, we have $B_3 \Rightarrow B_3'$, and B_4 not affected.
Given *local time event* of P_1, we have $B_3 \Rightarrow B_3'$, $B_4 \Rightarrow B_4'$.
$B_{3\text{-}4}$ is not affected by any other event.
End of proof.

At this point, every invariant requirement is marked wrt every event, and every event necessity requirement marked. This completes the construction of our first protocol, except for the marking of the progress requirements which is in Section 5.5. The system specification is exactly as in Tables 2 and 3, except that in Table 2 the state variables $Timer_D$ and $Timer_A$ are added, and *sourcedata*, *recack*, and *senddata* are as specified above.

5.2. Protocol implementation with N timers

In this section, we provide an implementation in which both S_4 and S_5 are enforced by the N local timers in $Timer_A$. Unlike in the previous implementation with $Timer_D$, the enforcement of S_4 is not tight.

Because $Source[n]$ is not sent after it is acknowledged, we have $T_D[n] \geq T_A[n]$ for all $n \in [0..a-1]$ (the proof of this is trivial). Thus, an alternative way to enforce S_4 is to enforce the following:

$$S_6 \quad \equiv \quad s \geq N-RW \Rightarrow T_A[s-N+RW] > MaxDelay_1$$

S_6 is analogous to S_5 and can be enforced by including $Timer_A[\overline{s}\oplus RW] > MDelay_1$ in the enabling condition of *sourcedata*. We have used the fact that $Timer_A[\overline{s-N+RW}]$ tracks $T_A[s-N+RW]$ (from S_0 and B_1) and $\overline{s-N+RW} = \overline{s}\oplus RW$. We have to combine this with the other condition $Timer_A[\overline{s}\oplus 1] > MDelay_2$ needed to enforce S_5. There are two cases. If $MaxDelay_1 < MaxDelay_2$, then we refine *sourcedata* (in Table 2) as follows:

$$\begin{aligned} sourcedata \quad \equiv \quad & s-a \leq N-RW-1 \wedge Timer_A[\overline{s}\oplus 1]=\text{Off} \\ & \wedge (Timer_A[\overline{s}\oplus RW]=\text{Off} \vee Timer_A[\overline{s}\oplus RW] > MDelay_1) \\ & \wedge Source[s]' \in DATA \wedge s' = s+1 \end{aligned}$$

If $MaxDelay_1 \geq MaxDelay_2$, then we define each $Timer_A[i]$ to have a capacity $MDelay_1$, and refine *sourcedata* as follows:

$$\begin{aligned} sourcedata \quad \equiv \quad & s-a \leq N-RW-1 \wedge Timer_A[\overline{s}\oplus RW]=\text{Off} \\ & Source[s]' \in DATA \wedge s' = s+1 \end{aligned}$$

There is no need to include $Timer_A[\overline{s}\oplus 1]=\text{Off}$ because $T_A[s-N+1] > MaxDelay_2$ follows from $T_A[s-N+RW] > MaxDelay_1$ and A_{19}.

This completes the construction of our first protocol. The marking of the progress requirements is in Section 5.5. The specification of this protocol is exactly as in Tables 2 and 3, except that the state variable $Timer_A$ is added, and *sourcedata* and *recack* are as specified above.

5.3. Protocol implementation with one timer

In this section, we prove that S_5 and S_6 can be enforced by imposing a minimum time interval δ between successive occurrences of *sourcedata*. This time constraint is of interest for two reasons. First, it can be implemented with a single local timer at P_1. Second, it corresponds to specifying a maximum rate of data transmission, if we assume that *sourcedata* also transmits the accepted data block. (There is no loss of generality here; P_1 need merely save in another buffer data blocks that are produced and not yet sent.) Note that if δ is sufficiently small, e.g. the hardware clock period, then there is no need for P_1 to explicitly use a local timer. This would correspond to the situation in TCP [8] and the original Stenning's protocol [15].

Define the following timers at P_1, where $\delta_M = (1+\epsilon_1)\delta$:

$Timer_S$: local timer of capacity δ_M; {indicates the local time elapsed upto δ_M, since the last occurrence of *sourcedata*. Initially, $Timer_S$ = Off}

T_S : array[0..∞] of ideal timer; {$T_S[n]$ will record the ideal time elapsed since $Source[n]$ was produced. Initially, $T_S[0..\infty]$ = Off}

We refine the *sourcedata* event to the following:

sourcedata	$\equiv$	$s-a \leq N-RW-1 \wedge Timer_S = \text{Off} \wedge Timer_S' = 0 \wedge T_S[s]' = 0$ $Source[s]' \in DATA \wedge s' = s+1$

The following invariant requirements state that $Timer_S$ tracks $T_S[s-1]$, and that successive occurrences of *sourcedata* are separated by at least δ ideal time units:

B_5	$\equiv$	$s \geq 1 \Rightarrow started\text{-}together(Timer_S, T_S[s-1]) \vee (Timer_S = \text{Off} \wedge T_S[s-1] > \delta)$
B_6	$\equiv$	$n, m \in [1..s-1] \Rightarrow T_S[m] - T_S[n] > (n-m)\delta$

Proof that $B_{5,6}$ can be marked with respect to all events

Given *sourcedata*, we have B_5', and $B_{5,6} \Rightarrow B_6'$.

Given *ideal time event*, we have $B_5 \Rightarrow B_5'$, and $B_6 \Rightarrow B_6'$.

Given *local time event* of P_1, we have $B_5 \Rightarrow B_5'$, and B_6 not affected.

B_{5-6} is not affected by any other event.

End of proof.

Consider an occurrence of *sourcedata* that increments s from s_0 to s_0+1. Just prior to an occurrence of *sourcedata*, we have $T_S[s_0-1] > \delta$ because of $Timer_S$=Off and B_5. This and B_6 imply

$$(*) \quad n \in [1..s_0] \Rightarrow T_S[s_0-n] > n\,\delta$$

Both S_5 and S_6 are of the form $s_0 \geq K \Rightarrow T_A[s_0-K] > D$. Obviously, this is enforced by (*) if there is some $n_0 \in [1..s_0]$ that satisfies

$$(**) \quad s_0 \geq K \Rightarrow T_A[s_0-K] > T_S[s_0-n_0] > D$$

The first inequality above states that data block s_0-K is acknowledged before data block s_0-n_0 is produced, or equivalently, that a exceeded s_0-K before s exceeded s_0-n_0. This can be enforced simply by requiring $s-a \le K-n_0-1$ to hold prior to any occurrence of *sourcedata*. The second inequality above, $T_S[s_0-n_0] > D$, can be enforced for any $n_0 \in [1..s_0]$ simply by having δ large enough so that it satisfies $n_0\,\delta > D$.

Stating this in terms of $m_0 = K-n_0$, we have the following: (**) is enforced if there is an $m_0 \in [1..K-1]$ such that $(K-m_0)\,\delta > D$ and $s-a \le m_0-1$ holds prior to any occurrence of *sourcedata*. Note that $m_0=K$ would require $s-a=0$ to be invariant, resulting in a dead protocol.

S_6 is (**) with $D = MaxDelay_1$ and $K = N-RW$. Thus, it is enforced if there is an $m_0 \in [1..N-RW-1]$ such that $(N-RW-m_0)\,\delta > MaxDelay_1$ and $s-a \le m_0-1$ holds prior to an occurrence of *sourcedata*. S_5 is (**) with $D = MaxDelay_2$ and $K = N-1$. It is enforced if there is an $m_0 \in [1..N-2]$ such that $(N-1-m_0)\,\delta > MaxDelay_2$ and $s-a \le m_0-1$ holds prior to an occurrence of *sourcedata*.

Therefore, both S_5 and S_6 are enforced for any $m_0 \in [1..N-RW-1]$ if δ is large enough to satisfy $(N-RW-m_0)\,\delta > MaxDelay_1$ and $(N-1-m_0)\,\delta > MaxDelay_2$, and $s-a \le m_0-1$ is enforced prior to every occurrence of *sourcedata*. Observe that m_0 is an upper bound on $s-a$. In the literature, such an upper bound is referred to as a *send window size*, and is often denoted by SW. Rephrasing the above conditions in terms of SW, we require SW and δ to satisfy the following:

$$1 \le SW \le N-RW-1$$
$$\delta \ge \max\left(\frac{MaxDelay_1}{N-RW-SW}, \frac{MaxDelay_2}{N-1-SW}\right)$$

We require $s-a \le SW-1$ to hold prior to every occurrence of *sourcedata*, which is now refined to the following:

$$sourcedata \;\equiv\; s-a \le SW-1 \wedge Timer_S = \text{Off} \wedge Timer_S{}' = 0 \wedge T_S[s]' = 0$$
$$Source[s]' \in DATA \wedge s' = s+1$$

For the typical case of $MaxDelay_1 = MaxDelay_2 = MaxDelay$, the above constraint on δ simplifies to $\delta \ge \dfrac{MaxDelay}{N-SW-RW}$. If in addition, N is very large compared to SW or RW (e.g. in TCP, $N=2^{32}$ while $SW, RW \le 2^{16}$), then the bound simplifies to $\delta \ge \dfrac{MaxDelay}{N}$.

The specification of this image system is exactly as in Tables 2 and 3, except that the state variables $Timer_S$ and T_S are added, and *sourcedata* is as specified above. We have proved that every event in this protocol can be marked with respect to the invariant requirements $A_{0\text{-}20} \wedge B_{5\text{-}7}$ and the requirements $S_{0\text{-}6}$ for *sourcedata*, where

$$B_7 \;\equiv\; s-a \le SW$$

Recall that the image system of Tables 2 and 3 was marked with respect to the invariant requirements $A_{0\text{-}20}$ and the requirements $S_{0\text{-}3}$ for *sourcedata*. The marking of the progress requirements is in Section 5.5.

5.4. Transforming variables to auxiliary variables

We now show how *Source*, s, a, *Sink*, and r can be transformed into auxiliary variables. These are minor modifications that make the protocol system more realistic.

P_1 must save unacknowledged data blocks in local buffers for retransmission purposes. Therefore, even though *Source*, a, s are auxiliary variables, the implementation has access to the value $s-a$ and the data blocks $Source[a+i]$ for any $i \in [0..s-a-1]$. From the events of P_1, we see that the only other value needed by the implementation is $\overline{a}$. Because of A_{14}, we also have $s-a = \overline{s-a} = \overline{s} \ominus \overline{a}$. For the sake of symmetry, we shall implement $\overline{s}$ instead of $s-a$. Thus, we define the implemented variables cs and ca at P_1 which track $\overline{s}$ and $\overline{a}$ respectively. The events of P_1 would be redefined as follows. In *sourcedata*, replace $s-a$ by $cs \ominus ca$, s by $a + cs \ominus ca$, and include $cs' = cs \oplus 1$. In *sendata*, replace $n \in [a..s-1]$, n, and $\overline{n}$ by $i \in [0..cs \ominus ca-1]$, $a+i$, and $ca \oplus i$ respectively. In *recack*, replace $\overline{a}$ and $s-a$ by ca and $cs \ominus ca$ respectively, include $ca' = cn$ as a conjunct to $a' = a + cn \ominus \overline{a}$, and include $ca' = ca$ as a conjunct to $a' = a$.

P_2 has to maintain buffers corresponding to the receive window. Thus, $Sink[r+i]$ for $i \in [0..RW-1]$ are available to the implementation. From the events of P_2, we see that the only other value needed is $\overline{r}$. Thus, we define the implemented variable cr at P_2 which tracks $\overline{r}$. In *sinkdata*, include $cr' = cr \oplus 1$. In *sendack*, replace $\overline{r}$ by cr. In *recdata*, replace $\overline{r}$ by cr.

It is obvious that $cs = \overline{s} \wedge ca = \overline{a} \wedge cr = \overline{r}$ is invariant. Because of this invariant, the new definitions of the events are refinements of the previous definitions. Hence, all the old markings continue to hold.

5.5. Progress marking update

We will prove that $L_0 \equiv a = n \ \textit{leads-to} \ a \geq n+1$ can be marked for the above image system, provided that the following progress assumptions hold:

(a) P_1 eventually retransmits the next outstanding data block. Formally, $s > a = n$ leads to either $a \geq n+1$ or P_1 sending $Source[n]$.

(b) P_2 eventually sinks insequence data. Formally, $r = n \wedge Sink[n] \neq empty$ leads to $r \geq n+1$.

(c) P_2 eventually responds to receptions of data messages. Formally, let *unacked* be an auxiliary state variable that is true iff an acknowledgement message has not been sent since the last data message reception. Then, we have *unacked* leads to P_2 sending an acknowledgement message.

(d) The channels eventually deliver a message that is repeatedly sent [3, 10]. Formally, let the auxiliary state variable α_n (β_n) denote the number of times that data block n (an acknowledgement to data block n) has been sent by P_1 (P_2) since the last time that it was received by P_2 (P_1). Then, we have that α_n and β_n do not grow unboundedly.

We first prove the following:

$$\boxed{L_1 \quad \equiv \quad s > a = r = n \ \textit{leads-to} \ s \geq r > a = n}$$

Proof of L_1

From assumption (a) and A_0, we have

$s > a = r = n \wedge \alpha_n \geq i$ *leads-to*

$(s > a = r = n \wedge \alpha_n \geq i+1) \vee (s > a = r = n \wedge Sink[n] \neq empty)$

Applying induction over i to this, and using assumption (d), we have

$s > a = r = n$ *leads-to* $s > a = r = n \wedge Sink[n] \neq empty$

Using assumption (b) on this, we have L_1.

End of proof.

We next prove the following:

$$L_2 \quad \equiv \quad s \geq r > a = n \text{ leads-to } a \geq n+1$$

Proof of L_2

From assumption (a) and A_0, we have

(*) $s \geq r > a = n \wedge \alpha_n \geq i \wedge \beta_n \geq j$ *leads-to* $a \geq n+1$

$\vee (s \geq r > a = n \wedge \alpha_n \geq i+1 \wedge \beta_n \geq j) \vee (s \geq r > a = n \wedge unacked \wedge \beta_n \geq j)$

From assumption (c), we have

$s \geq r > a = n \wedge unacked \wedge \beta_n \geq j$ *leads-to* $(s \geq r > a = n \wedge \beta_n \geq j+1) \vee a \geq n+1$

Substituting this in (*) and regrouping, we get

$s \geq r > a = n \wedge \alpha_n \geq i \wedge \beta_n \geq j$ *leads-to* $a \geq n+1$

$\vee (s \geq r > a = n \wedge (\beta_n \geq j+1 \vee (\beta_n \geq j \wedge \alpha_n \geq i+1))$

Applying lexicographic induction over (j, i) on this, and using assumption (d), we have L_2.

End of proof

From L_1, L_2, and A_0, we have $s > a = n$ *leads-to* $a \geq n+1$. To establish L_0, all that is left is to show $s = a = n$ *leads-to* $s > a = n$, i.e., to show that *sourcedata* will eventually be enabled whenever $s = a$ holds. We now list the enabling condition of *sourcedata* in the three implementations:

(1) $s - a \leq N - RW - 1 \wedge Timer_D[\bar{s} \oplus RW] = \text{Off} \wedge Timer_A[\bar{s} \oplus 1] = \text{Off}$

(2a) $s - a \leq N - RW - 1 \wedge Timer_A[\bar{s} \oplus 1] = \text{Off}$

$\wedge (Timer_A[\bar{s} \oplus RW] = \text{Off} \vee Timer_A[\bar{s} \oplus RW] > MDelay_1)$

(2b) $s - a \leq N - RW - 1 \wedge Timer_A[\bar{s} \oplus RW] = \text{Off}$

(3) $s - a \leq SW - 1 \wedge Timer_S = \text{Off}$

Observe that the conjuncts $s - a \leq N - RW - 1$ and $s - a \leq SW - 1$ hold whenever s equals a. Thus, all we need to show is that the conjuncts involving the timers eventually hold whenever $s = a$ holds.

Because the timer axioms are implementable, the time events are never deadlocked (see [10] for a proof). Therefore, the value of any timer that is not Off keeps increasing. In particular, if u is a bounded capacity timer that is only started when it is Off, then $u \neq \text{Off}$ *leads-to* $u = \text{Off}$ holds. From *sourcedata*, we see that $Timer_S$ is started only when it is Off. From B_4 and *recack*, we see that $Timer_A[n]$ is started only when it is Off for all $n \in [0..N-1]$. From B_1, $s = a$, and *senddata*, we see that $Timer_D[\bar{s} \oplus RW]$ is started only when it is Off. Therefore, in each case, $s - a$ leads to $s > a$ or all the local timers becoming Off, at which point *sourcedata* is enabled. Thus, L_0 holds.

Table 1: Invariant and event requirements of the protocol

		{Properties relating *Source*, *Sink*, s, a, r, N, RW}
		$1 \leq RW \leq N-1$
A_0	$\equiv$	$0 \leq a \leq r \leq s$
A_{14}	$\equiv$	$s-a \leq N-RW$
A_1	$\equiv$	$n \in [0..r-1] \Rightarrow Sink[n] = Source[n]$
A_6	$\equiv$	$n \in [r..r+RW-1] \wedge Sink[n] \neq empty \Rightarrow Sink[n] = Source[n]$
A_2	$\equiv$	$n \geq s \Leftrightarrow Source[n] = empty$
A_3	$\equiv$	$n \geq r+RW \Rightarrow Sink[n] = empty$
		{Properties relating D messages, *Source*, T_D, *Sink*, s}
A_4	$\equiv$	$(D, data, cn, n) \in \mathbf{z}_1 \Rightarrow data = Source[n] \wedge cn = \overline{n}$
A_{13}	$\equiv$	$(D, data, \overline{n}, n) \in \mathbf{z}_1 \Rightarrow n \in [s-N+RW..s-1]$
A_{15}	$\equiv$	$(D, data, \overline{n}, n, age) \in \mathbf{z}_1 \Rightarrow age \geq T_D[n] \geq 0$
A_{16}	$\equiv$	$n \in [0..s-N+RW-1] \Rightarrow T_D[n] > MaxDelay_1$
		{Properties relating ACK messages, r, s, T_R}
A_5	$\equiv$	$(ACK, cn, n) \in \mathbf{z}_2 \Rightarrow cn = \overline{n}$
A_9	$\equiv$	$(ACK, cn, n) \in \mathbf{z}_2 \Rightarrow n \leq r$
A_{12}	$\equiv$	$(ACK, \overline{n}, n) \in \mathbf{z}_2 \Rightarrow n \in [s-N+1..r]$
A_{18}	$\equiv$	$(ACK, \overline{n}, n, age) \in \mathbf{z}_2 \wedge n < r \Rightarrow age \geq T_R[n] \geq 0$
A_{17}	$\equiv$	$T_R[0] \geq T_R[1] \geq \cdots \geq T_R[r-1] \geq 0$
		{Properties relating s, a, r, T_A, T_R}
A_{19}	$\equiv$	$T_A[0] \geq T_A[1] \geq \cdots \geq T_A[a-1] \geq 0$
A_{20}	$\equiv$	$n \in [0..a-1] \Rightarrow T_A[n] \leq T_R[n]$
A_{21}	$\equiv$	$n \in [0..s-N] \Rightarrow T_A[n] > MaxDelay_2$
		{Requirements for *sourcedata*}
S_0	$\equiv$	$s-a \leq N-RW-1$
S_4	$\equiv$	$s \geq N-RW \Rightarrow T_D[s-N+RW] > MaxDelay_1$
S_5	$\equiv$	$s \geq N-1 \Rightarrow T_A[s-N+1] > MaxDelay_2$

Table 2: Specification of P_1

State variables:

$Source$: array[0..∞] of $\{empty\} \cup DATA$; {Initially, $Source[0..\infty] = empty$}

s, a : 0..∞; {Initially, $s = a = 0$}

T_D, T_A : array[0..∞] of ideal timer; {Initially, $\forall n \in [0..\infty](TD[n] = T_A[n] = \text{Off})$}

Events:

$$sourcedata \equiv s-a \le N-RW-1 \wedge Source[s]' \in DATA \wedge s' = s+1$$

$$senddata \equiv \exists n\,(n \in [a..s-1] \wedge Send_1(D, Source[n], \overline{n}, n) \wedge T_D[n]' = 0)$$

$$recack \equiv \exists cn, n\,(\, Rec_2(ACK, cn, n) \wedge (\,(cn \ominus \overline{a} \in [1..s-a] \wedge a' = a + cn \ominus \overline{a} \wedge T_A[a..a'-1]' = 0) \vee (cn \ominus \overline{a} \notin [1..s-a] \wedge a' = a \wedge T_A' = T_A)))$$

Table 3: Specification of P_2

State variables:

$Sink$: array[0..∞] of $\{empty\} \cup DATA$; {Initially, $Sink[0..\infty] = empty$}

r : 0..∞; {Initially, $r = 0$}

T_R : array[0..∞] of ideal timer; {Initially, $\forall n \in [0..\infty](TR[n] = \text{Off}\}$

Events:

$$sinkdata \equiv Sink[r] \ne empty \wedge r' = r+1 \wedge T_R[r]' = 0$$

$$sendack \equiv Send_2(ACK, \overline{r})$$

$$recdata \equiv \exists data, cn, n\,(\, Rec_1(D, data, cn, n) \wedge (\,(cn \ominus \overline{r} \in [0..RW-1] \wedge Sink[r + cn \ominus \overline{r}]' = data) \vee (cn \ominus \overline{r} \notin [0..RW-1] \wedge Sink' = Sink)))$$

REFERENCES

[1] 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.

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

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

[4] 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,

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

[6] Lam, S. S. and A. U. Shankar, "Protocol verification via projections," *IEEE Trans. on Soft. Engg.*, Vol. SE-10, No. 4, July 1984, pp. 325-342.

[7] Owicki, S. and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica*, Vol. 6, 1976, pp. 319-340.

[8] Postel, J. (ed.), "Transmission Control Protocol: Darpa internet program protocol specification," Defense Advanced Research Projects Agency, Information Processing Techniques Office, RFC 793, September 1981.

[9] 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.

[10] 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, revised October 1986, to appear in *Distributed Computing*.

[11] 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.

[12] 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.

[13] Shankar, A. U., "Verified Data Transfer Protocols with Variable Flow Control," Tech. Rep. CS-TR-1746, UMIACS-TR-86-25, Computer Science Dept., Univ. of Maryland.

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

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