A Stepwise Refinement Heuristic for Protocol Construction

A. UDAYA SHANKAR
University of Maryland
and
SIMON S. LAM
The University of Texas at Austin

A stepwise refinement heuristic to construct distributed systems is presented The heuristic is based on a conditional refinement relation between system specifications, and a "Marking." It is applied to construct four sliding window protocols that provide reliable data transfer over unreliable communication channels. The protocols use modulo-N sequence numbers. The first protocol is for channels that can only lose messages in transit. By refining this protocol, we obtain three protocols for channels that can lose, reorder, and duplicate messages in transit. The protocols herein are less restrictive and easier to implement than sliding window protocols previously studied in the protocol verification literature.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol verification*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time systems*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies*; D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.2.10 [**Software Engineering**]: Design—*methodologies*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions, invariants, pre- and post-conditions, specification techniques*

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Assertional reasoning, conditional refinement, cyclic sequence numbers, interfaces, message lifetimes, sliding window protocols, stepwise refinement

1. INTRODUCTION

The specification of a distributed system in our methodology consists of a state transition system and a set of requirements. A state transition system is defined by a set of state variables, a set of events, and an initial condition

The work of A. U. Shankar was supported by National Science Foundation grants ECS-8502113 and NCR-890450. The work of S. S. Lam was supported by National Science Foundation grants NCR-8613338 and NCR-9004464.

Authors' addresses: A. U. Shankar, Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742; S. S. Lam, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712.

on the state variables; each event is defined by a set of allowed state transitions. Requirements can be of three types: *invariant requirements*, *event requirements*, and *progress requirements*. Invariant requirements, events, and event requirements are used to specify desired safety properties of the distributed system. Progress requirements, stated using the temporal operator *leads-to* [5, 35], are used to specify desired progress properties.

The topology of a distributed system is, in general, a directed graph whose nodes are called *entities* and whose arcs are called *channels*.[1] To construct a distributed system with a given topology, the state variables and events of the state transition system are required to satisfy some constraints imposed by the topology.

To construct a distributed system using our stepwise refinement heuristic, we begin with a system specification consisting of a state transition system and a set of invariant, event and progress requirements. This very first state transition system is generally simple, with just enough resolution in its state space for *specifying* the desired safety and progress properties. The goal of our construction is a state transition system that *satisfies* all of the requirements in the system specification (given some fairness assumptions). To achieve this goal, a sequence of state transition systems is derived by applications of some system refinement steps. Requirements in the system specification are successively strengthened by applications of some requirement refinement steps; in doing so, new requirements may be generated. The objective of each refinement step is to increase the number of requirements that are *marked* (to be defined precisely below).

In applying our heuristic to construct a distributed system, the construction is not guaranteed to terminate. When it does terminate, however, there are two possible cases: (1) The construction terminates successfully when all requirements in the system specification are marked, and the state transition system satisfies all topology constraints of the distributed system. (2) The construction terminates unsuccessfully when a requirement is generated that is inconsistent with other requirements or with the initial condition of the system.

Our construction heuristic is influenced by Dijkstra's work on the formal derivation of programs using weakest preconditions [7]. A key element of our heuristic is the notion of one system specification being a *conditional refinement* of another system specification. It is adapted from our earlier work on refinement relations between state transition systems based on the use of projection mappings [17, 18, 33]. In Section 8 we give a more detailed comparison of our approach with other approaches in the literature and describe other applications of our heuristic.

1.1 Construction Examples

Our heuristic is illustrated by a rigorous exercise in constructing four sliding window protocols that provide reliable data transfer between a producer and

[1] We use terminology from the networking area.

a consumer connected by unreliable channels. All protocols use modulo-N sequence numbers.[2] The desired property that sequence numbers in data messages and acknowledgment messages are interpreted correctly is stated as invariant requirements. We first construct a basic protocol that satisfies these *correct interpretation requirements* for channels that can only lose messages in transit. This basic protocol is then refined to be used for channels that can lose, duplicate, and reorder messages arbitrarily. To satisfy the correct interpretation requirements for such channels, it is necessary that message lifetimes are bounded so that certain time constraints can be enforced in producing data blocks. We present three different ways of enforcing these time constraints, resulting in three protocols. The first and second of these protocols use $2N$ and N timers, respectively. The third protocol uses a single timer to enforce a minimum time interval between producing successive data blocks. The minimum time interval is a function of N, the receive window size, and the maximum message lifetimes. To construct these three protocols, we use the system model developed in [34] and [35] in which real-time constraints can be specified and verified as safety properties.

To our knowledge, this is the first verified construction of sliding window protocols that use modulo-N sequence numbers where N is arbitrary. Our first and second protocols for loss, duplication, and reordering channels appear to be novel. Our third protocol is best compared with the original Stenning's protocol [36], which has several unnecessary requirements. Stenning verified certain safety properties assuming unbounded sequence numbers. He then informally argued that modulo-N sequence numbers can be used provided that N satisfies a bound. His bound is similar to ours, but not as tight as ours. (A detailed comparison is presented in Section 6.6.)

Knuth [15] has analyzed a sliding window protocol using modulo-N sequence numbers. He gives the minimum value of N that ensures correct data transfer along channels that lose messages and also allow messages to overtake a limited number of previously sent messages. Because of this restriction on the reordering of messages, his protocol does not require timers and the assumption of bounded message lifetimes.

In [31] we have extended the protocol for loss-only channels and the third protocol for loss, duplication, and reordering channels to include the use of selective acknowledgment messages as well as variable windows for flow control.

1.2 Organization of this Report

In Section 2 we give an overview of our system model, proof rules, and a refinement relation between state transition systems. In Section 3 we give a brief description of our construction heuristic, including the conditional refinement relation between system specifications. In Section 4 we derive the

[2] In a real protocol, sequence numbers in data messages and acknowledgment messages are coded by a small number of bits.

basic protocol and show that for channels that can lose, duplicate, and reorder messages arbitrarily, its requirements are almost completely marked; only two invariant requirements concerning sequence numbers in channels remain unmarked. In Section 5 we show that, for channels that can only lose messages, the basic protocol in fact satisfies all of the requirements. In Section 6 we refine the basic protocol to obtain three different protocols that satisfy all of the requirements for channels that can lose, duplicate, and reorder messages arbitrarily. In Section 7 we review our heuristic and present a list of useful refinement steps; a proof of the soundness of these steps is presented in Appendix A. In Section 8 we describe other applications of our heuristic and discuss related work.

2. MODEL AND NOTATION

In this section we describe our notation for state transition systems, fairness requirements, safety and progress assertions, and present a refinement relation between state transition systems.

2.1 State Transition System and Fairness Requirements

A state transition system X is specified by (1) a set of state variables, $Variables_X$; (2) an initial condition on the state variables, $Initial_X$; (3) a set of events, $Events_X$; and (4) for every event $e \in Events_X$, an event formula $formula_X(e)$ that specifies a set of transitions (explained below).

The state variables define the state space of X. Associated with each state variable v is a set $domain(v)$ of allowed values. Each tuple $(d_v: v \in Variables_X)$, where $d_v \in domain(v)$, represents a state of X.

We use state formulas to specify sets of states. A *state formula* is a formula in $Variables_X$ that evaluates to *true* or *false* when $Variables_X$ is assigned s, for every state s of X.[3] The state formula specifies the set of states for which it evaluates to *true*. A state s satisfies a state formula P iff P evaluates to *true* for s. For example, the initial condition $Initial_X$ is specified by a state formula.

A *transition* is an ordered pair of states of X. Associated with each event $e \in Events_X$ is a set of transitions, referred to as the transitions of e. We use event formulas to specify these sets of transitions [18, 34]. An *event formula* is a formula in $Variables_X \cup Variables'_X$, where $Variables'_X = \{v': v \in Variables_X\}$ and $domain(v') = domain(v)$. The ordered pair (s, t) is a transition of an event formula iff the formula evaluates to *true* when $Variables_X$ is assigned s and $Variables'_X$ is assigned t.

Conventions. When defining events and their event formulas, we treat the event as a name for the formula, as in the example $e_1 \equiv x > 2 \wedge y' \in$

[3] We use *formula* to mean a *well-formed formula* in the language of predicate logic. In a formula, the logical operations, $\neg$, $\wedge$, $\vee$, and $\Rightarrow$, are assumed to have decreasing binding power. By "$Variables_X$ is assigned s", we mean the following: If s is the tuple $(d_v: v \in Variables_X)$ then for every variable v in $Variables_X$, each free appearance of v in the state formula is replaced by d_v.

$\{1, 2, 5\}$, where x and y are state variables. For every state variable v in $Variables_X$, if v' is not a free variable of $formula_X(e)$, the conjunct $v' = v$ is implicit in $formula_X(e)$; that is, the occurrence of e does not change the value of the state variable v. Also, we use parameters in event formulas as a convenient way to specify a group of related events; for example, $e_2(m) \equiv x > y \wedge x + y' = m$, where m is a parameter with a specified domain of allowed values.

A *behavior* of state transition system X is a sequence $\langle s_0, e_0, s_1, e_1, \ldots \rangle$ of alternating states and events such that s_0 is an initial state and, for all i, (s_i, s_{i+1}) is a transition of event e_i. Note that a behavior can be infinite or finite. By definition, a finite behavior ends in a state.

For every event $e \in Events_X$, the *enabling condition* of e, denoted $enabled_X(e)$, refers to the set of states $\{s$: for some state t, (s, t) is a transition of $e\}$; that is,

$$enabled_X(e) \equiv [\exists\ Variables'_X\colon formula_X(e)].$$

Event e is enabled (disabled) in state s iff s satisfies (does not satisfy) $enabled_X(e)$.

In order for a state transition system to satisfy progress properties, some fairness assumptions are needed. These assumptions are explicitly stated as fairness requirements for sets of events of X. For any event set $E \subseteq Events_X$, we say that E is enabled in state s iff, for some $e \in E$, e is enabled in s. In a behavior $\sigma = \langle s_0, e_0, s_1, e_1, \ldots \rangle$, we say that E occurs in state s_j iff $e_j \in E$. We say that an event set $E \subseteq Events_X$ has *weak fairness* to mean the following: if E is continuously enabled, then one of the events in E eventually occurs. Formally [18, 24], a behavior σ of X satisfies weak fairness for event set E iff (1) σ is finite and E is not enabled in the last state of σ, or (2) σ is infinite and either E occurs infinitely often or is disabled infinitely often in σ. Another type of fairness requirement will be introduced below for channels.

Given a set of fairness requirements, an *allowed behavior* of X is a behavior of X that satisfies every fairness requirement in the set.

Some of the state variables in $Variables_X$ may be *auxiliary variables*, that is, state variables that are needed for specification or verification only and do not have to be included in an implementation of X. For example, an auxiliary variable may be needed to record the history of certain event occurrences. Informally, a subset of $Variables_X$ is auxiliary if they do not affect the enabling condition of any event or the update of any state variable that is not auxiliary [28]. A more precise statement of this condition and a better explanation can be found in [18].

2.2 Safety and Program Properties

To state safety properties, we use assertions of the form $Invariant(P)$, where P is a state formula. $Invariant(P)$ is satisfied by a finite sequence $\sigma = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ of alternating states and events (or σ satisfies $Invariant(P)$)

iff P is satisfied by every state s_i in σ. *Invariant*(P) is satisfied by an infinite sequence of alternating states and events iff every finite prefix of the sequence satisfies *Invariant*(P). *Invariant*(P) is satisfied by a state transition system X (or X satisfies *Invariant*(P)) iff every finite behavior of X satisfies *Invariant*(P).[4] We say "P is an invariant of X" to mean that X satisfies *Invariant*(P).

To state progress properties, we use assertions of the form P *leads-to* Q, where P and Q are state formulas. P *leads-to* Q is satisfied by a sequence $\sigma = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ of alternating states and events iff the following holds: For every state s_i in σ that satisfies P, there is a state s_j in σ, $j \geq i$, that satisfies Q. P *leads-to* Q is satisfied by a state transition system X with a given set of fairness requirements iff every allowed behavior of X satisfies P *leads-to* Q.

We now present some inference rules that will be used in our construction heuristic (for a more complete treatment of inference rules, see [18] and [25]). To state these rules, we need the following notation: For an arbitrary state formula R, R' denotes the formula obtained from R by replacing every state variable v in it by v'. In the following rules, X denotes a state transition system:

Invariance Rule. X satisfies *Invariant*(P) if (1) $Initial_X \Rightarrow P$ and (2) for every event e of X, $P \wedge formula_X(e) \Rightarrow P'$.

Leads-to-via-Event Rule. Given an event set E with weak fairness, X satisfies P *leads-to* Q (via E) if (1) for every event $e \in E$, $P \wedge formula_X(e) \Rightarrow Q'$; (2) for every event $e \in Events_X - E$, $P \wedge formula_X(e) \Rightarrow P' \vee Q'$; and (3) X satisfies *Invariant*($[\exists\ e \in E\colon P \Rightarrow enabled(e)]$).

Leads-to-by-Closure Rules. P *leads-to* Q (by closure) if one of the following holds: (1) *Invariant*($P \Rightarrow Q$); (2) for some state formula R, P *leads-to* R and R *leads-to* Q; (3) $P = P_1 \vee P_2$, P_1 *leads-to* Q and P_2 *leads-to* Q; and (4) *Invariant*(R) and ($P \wedge R$) *leads-to* ($R \Rightarrow Q$).

If X satisfies *Invariant*(I), we can replace P by $P \wedge I \wedge I'$ in the antecedent of each of the above implications.

We use three types of preconditions in our heuristic. Consider state formulas P and Q, and an event e of state transition system X. We say that P is a *weakest precondition* of Q with respect to e iff P is logically equivalent to $[\forall\ Variables'_X\colon formula_X(e) \Rightarrow Q']$. Note that P is *false* precisely for those states where e is enabled and where its occurrence can cause Q to be falsified.[5] We say that P is a *sufficient precondition* iff P implies the

[4] This ensures that every infinite behavior of X also satisfies *Invariant*(P).

[5] This corresponds to Dijkstra's weakest liberal precondition [7]. Also, note that in our formalism P is *a* weakest precondition, and not *the* weakest precondition. Unlike in [7], if R is equivalent to P, then R is also a weakest precondition.

weakest precondition, that is, iff $[\forall\ Variables'_X\colon\ P \wedge formula_X(e) \Rightarrow Q']$ is true. We say that P is a *necessary precondition* iff P is implied by the weakest precondition, that is, $\neg P \Rightarrow [\exists\ Variables'_X\colon\ formula_X(e) \wedge \neg Q']$ is true.

We allow assertions to have parameters. For example, the assertion $x = k$ *leads-to* $x = k + 1$ has x as a state variable and k as a parameter. We follow the convention that such parameters are universally quantified. Thus, the above assertion is equivalent to $[\forall\ k\colon\ x = k$ *leads-to* $x = k + 1]$.

2.3 Distributed System

A distributed system is defined by a topology, a state transition system, and a set of fairness requirements. The topology is a directed graph whose nodes are entities and whose arcs are channels. The state transition system is required to satisfy certain topology constraints, which are given below.

For each channel, there is a state variable representing the sequence of messages traveling along the channel. For each entity, there is a set of nonauxiliary state variables. In addition, the system can have other state variables that are auxiliary.

Each event of the state transition system belongs to a channel or an entity. The events of a channel can access (read or write) only the channel state variable and auxiliary state variables. (Channel events model channel errors such as loss, duplication, and reordering of messages in transit.) The events of an entity can access auxiliary state variables, nonauxiliary state variables that belong to the entity, and state variables of channels connected to the entity. Furthermore, an entity event can access a channel state variable only by *send* and *receive primitives*. (Formulas for channel events and primitives are defined in Section 4.) We assume that entity events are well defined in the following sense: In every reachable system state, there is an entity event enabled to receive the message, if any, at the head of each channel.

The set of fairness requirements consists of weak fairness for specified sets of entity events, and the following *channel fairness requirements* for every unreliable channel: For any set of messages M, if messages from M are sent repeatedly along the channel, one of them is eventually received [12]. Formally, a behavior σ satisfies the channel fairness requirement iff the following holds: If σ is infinite and messages from M are sent infinitely often in σ, then messages from M are received infinitely often in σ [18].[6] This fairness requirement is generally needed to prove that a distributed system with unreliable channels has certain useful progress properties. Specifically, we have the following inference rule, where $count(M)$ is an auxiliary variable indicating the number of times messages in M have been sent since the beginning of system execution, and where $e_r(m)$ denotes an entity event whose occurrence results in reception of message m from the following channel:

Leads-to-via-Message Rule. P *leads-to* Q (via M) if (1) for every event $e_r(m)$, $[\forall\ m \in M\colon\ P \wedge e_r(m) \Rightarrow Q']$; (2) for every event $e \neq e_r(m)$, $P \wedge$

[6] Thus, every finite behavior satisfies the channel fairness requirement.

$e \Rightarrow P' \vee Q'$; and (3) for every natural number k, $P \wedge count(M) \geq k$ *leads-to* $Q \vee count(M) \geq k + 1$.

2.4 Refinement of a State Transition System

For two state transition systems X and Y, we next define the relation Y *is a refinement of* X. Let $Variables_Y \supseteq Variables_X$. Thus, there is a projection mapping from each state of Y to a state of X. Specifically, a state of Y denoted by $(d_v: v \in Variables_Y)$ is mapped to the state of X denoted by $(d_v: v \in Variables_X)$. With the projection mapping, state formulas in $Variables_X$ and event formulas in $Variables_X \cup Variables'_X$ can be interpreted directly in the state space of Y without translation.

Definition. Y *is a refinement of* X iff for some state formula P in $Variables_Y$ such that Y satisfies $Invariant(P)$

— $Variables_Y \supseteq Variables_X$ and $Events_Y \supseteq Events_X$;
— $Initial_Y \Rightarrow Initial_X$;
— $\forall\ e \in Events_X$: $P \wedge formula_Y(e) \Rightarrow formula_X(e)$; and
— $\forall\ e \in Events_Y - Events_X$: $P \wedge formula_Y(e) \Rightarrow [\forall\ v \in Variables_X: v = v']$.

The above definition is a special case of the one presented in [18]. Note that if Y is a refinement of X and X satisfies $Invariant(R)$ for some state formula R $Variables_X$, then Y satisfies $Invariant(R)$.

3. STEPWISE REFINEMENT HEURISTIC

We begin a construction with a topology and a state transition system that has just enough resolution in its state space to specify the safety and progress properties desired of the distributed system. The state transition system does not have to satisfy the topology constraints. Additionally, we use invariant and event requirements to specify desired safety properties that are not captured by the state transition system. We use progress requirements to specify desired progress properties. None of the requirements are marked initially.

To model interactions between the distributed system and its environment, each event is specified to be under either system control or environment control. Events that are under environment control are called *input events*.

Starting from this initial specification, a succession of state transition systems is derived by applications of some *system refinement steps*. These steps are used to increase the resolution of the system state space by adding new state variables, adding new messages, and refining a message into a set of messages. The set of state transitions is changed by refining existing events and adding new events. We also apply some *requirement refinement steps* that strengthen the three sets of requirements. (We postpone a detailed description of refinement steps to Section 7, after they have been motivated by our protocol construction exercises in Sections 4–6.) The objective of each

refinement step is to increase the set of marked requirements, which is introduced below.

Initially and at any point during a construction, we have the following:

—a topology.

—a state transition system specified by a state variable set *Variables*, an initial condition *Initial*, an event set *Events* (including input events), and an event formula *formula*(e) for each event e.

—a set of invariant requirements specified by state formulas $A_0, A_1, \ldots$. We use A to denote the conjunction of all of the state formulas that are in the set of invariant requirements; if there are no invariant requirements, then $A \equiv true$. $Initial \Rightarrow A$ holds. (We want a distributed system that satisfies *Invariant*(A).)

—a set of event requirements specified by state formulas $S_0, S_1, \ldots$. Each requirement is associated with an event. We use $S(e)$ to denote the conjunction of all of the S_i's that are associated with event e; if there are none, then $S(e) \equiv true$. (We want $S(e)$ to hold prior to any occurrence of e; that is, we want a distributed system that satisfies *Invariant*($enabled(e) \Rightarrow S(e)$).)

—a set of progress requirements $L_0, L_1 \ldots$, which are *leads-to* assertions.

—a *Marking*, consisting of the following:

(1) a subset of event requirements; each S_i in the subset is said to be marked.

(2) a subset of (A_i, e) pairs; each pair in the subset is said to be marked.

(3) a subset of progress requirements; each L_i in the subset is marked with a tag of one of the following forms: *via* E where E is an event set, *via* M *using* L_j where M is a message set, or *by closure using* $L_{j_1}, \ldots, L_{j_n}$.

(4) an ordering of the L_i's (to avoid circular reasoning).

The Marking indicates the extent to which we have established that the requirements are satisfied by the state transition system.

Parts (1) and (2) of the Marking are concerned with safety properties. An event requirement S_i of event e being marked means that S_i holds in any state where A holds and where e is enabled; that is, immediately prior to any occurrence of e, S_i holds if A holds. An (A_i, e) pair being marked means that, for any transition (s, t) of e, if s satisfies $A \wedge S(e)$ then t satisfies A_i; that is, A_i holds after any occurrence of e, assuming that A and $S(e)$ held immediately prior to the occurrence.

Parts (3) and (4) of the Marking are concerned with progress properties. A progress requirement L_i being marked with a tag *via* E means that the state transition system satisfies L_i, assuming that E has weak fairness and that the state transition system satisfies all of the safety requirements in the current specification. A progress requirement L_i being marked with a tag *via* M *using* L_j (or *by closure using* $L_{j_1}, \ldots, L_{j_n}$) means that the state transition system satisfies L_i, assuming that the state transition system

satisfies all of the progress requirements listed in the tag and all of the safety requirements in the current specification. To avoid circular reasoning, any progress requirement listed in the tag of L_i has to succeed L_i in the ordering of part (4).

Formally, we require the Marking to satisfying the following *consistency constraints*:

(C1) An event requirement S_i associated with event e is marked only if $A \wedge \mathit{enabled}(e) \Rightarrow S_i$ holds.

(C2) A pair (A_i, e) is marked only if $A \wedge S(e) \wedge \mathit{formula}(e) \Rightarrow A'_i$ holds.

(C3) A progress requirement P *leads-to* Q is marked with the tag *via* E only if the following hold:

(i) for every event $e \in E$, $P \wedge A \wedge A' \wedge S(e) \wedge \mathit{formula}(e) \Rightarrow Q'$;

(ii) for every event $e \notin E$, $P \wedge A \wedge A' \wedge S(e) \wedge \mathit{formula}(e) \Rightarrow P' \vee Q'$;

(iii) $[\exists\ e \in E\colon P \wedge A \wedge S(e) \Rightarrow \mathit{enabled}(e)]$; and

(iv) E does not contain an input event.

(C4) A progress requirement $L_i \equiv P$ *leads-to* Q is marked with the tag *via* M *using* L_j only if the following hold:

(i) for every event $e_r(m)$ that receives $m \in M$, $P \wedge A \wedge A' \wedge S(e_r) \wedge \mathit{formula}(e_r) \Rightarrow Q'$;

(ii) for every event $f \neq e_r(m)$, $P \wedge A \wedge A' \wedge S(f) \wedge \mathit{formula}(f) \Rightarrow P' \vee Q'$; and

(iii) $L_j \equiv P \wedge \mathit{count}(M) \geq k$ *leads-to* $Q \vee \mathit{count}(M) \geq k + 1$, and L_j is listed after L_i in the ordering.

(C5) A progress requirement $L_i \equiv P$ *leads-to* Q is marked with the tag *by closure using* $L_{j_1}, \ldots, L_{j_n}$ only if P *leads-to* Q can be derived from A and $L_{j_1}, \ldots, L_{j_n}$ using the closure rules, and each L_{j_k} is listed after L_i in the ordering.

Note that each constraint imposes a sufficient condition, and not a necessary condition, for a requirement to be marked. Therefore, a Marking does not have to be "maximal"; that is, it may not include all of the requirements satisfiable by the state transition system.

Example. Consider a state transition system defined by integer state variables x, y, both initially 0, and events $e_0 \equiv x' = x + 1$ and $e_1 \equiv y' = y + 1$. Assume an invariant requirement $A_0 \equiv x = y \vee x = y + 1$, a progress requirement $L_0 \equiv y \neq x \wedge x = n$ *leads-to* $y = n$, and an event requirement $S_0 \equiv x = y$ associated with e_0. We can mark (A_0, e_0) because $S_0 \wedge \mathit{formula}(e_0)$ implies A'_0. If e_1 is not an input event, we can mark L_0 with tag *via* e_1 because of the following: $y \neq x \wedge x = n \wedge A_0 \wedge \mathit{formula}(e_1)$ implies that $y' = n$ (thus, (C3i) holds); $y \neq x \wedge x = n \wedge S_0 \wedge \mathit{formula}(e_0)$ is *false* (thus, (C3ii) holds); and $\mathit{enabled}(e_1)$ is *true* (thus, (C3iii) holds). (A_0, e_1) and S_0 are unmarked.

Successful termination. The Marking is said to be *complete* if

(1) every S_j is marked,
(2) every (A_j, e) pair is marked, and
(3) every L_j is marked.

At any point in a construction, conditions (1) and (2) imply that the state transition system satisfies the safety requirements, as follows: Condition (1) implies that $A \wedge formula(e) \Rightarrow S(e)$ holds for every event e, which together with condition (2) imply that $A \wedge formula(e) \Rightarrow A'$ holds. At any point in the construction, we have $Initial \Rightarrow A$. Thus, A satisfies the invariance rule. This and condition (1) imply that $enabled(e) \Rightarrow S(e)$ is invariant.

Given that the safety requirements hold, condition (3) implies that the progress requirements hold, assuming that every event set E that appears in a *via* E tag has weak fairness. Specifically, each progress assertion L_i holds according to the rule indicated in its tag (*via event*, *via message*, or *by closure*). There is no circular reasoning in the proof of the L_k's, because there is a serial order of the L_k's such that if L_j appears in the tag of L_i then L_j follows L_i in the ordering. Note that (C3iv) ensures that an input event is never required to satisfy a fairness assumption.

A construction ends successfully when (1) the state transition system is a refinement of the initial state transition system; (2) the state transition system satisfies topology constraints; (3) the Marking is complete; and (4) for every input event e, $enabled_I(e) \Rightarrow enabled(e)$ is invariant, where $enabled_I(e)$ refers to the enabling condition of e as defined in the initial specification. The last condition is sufficient to ensure that the state transition system does not block users of the system from executing input events allowed by the initial specification [19].[7] Because of this requirement, vacuous implementations are eliminated [24].

Conditional refinement. The difference between the initial system specification and the final system specification (in a successful construction) is typically quite large and cannot be negotiated in one step. It is preferable to go through a succession of intermediate system specifications, $D_1, D_2, \ldots, D_n$. In our heuristic, we require that D_{i+1} is a "conditional" refinement of D_i, for every i. The conditional refinement relation, defined below, is weaker than the refinement relation. It ensures that the final state transition system is a refinement of the initial state transition system *provided that* the heuristic terminates successfully.

Let X and Y be two successive system specifications that are constructed using the heuristic. We require X and Y to satisfy the following conditions:

— $Variables_Y \supseteq Variables_X$ and $Events_Y \supseteq Events_X$,

[7] A weaker sufficient condition is obtained by replacing $enabled_I(e)$ with $possible_I(e) \equiv S_I(e) \wedge [\exists\ Variables_I'\colon formula_I(e) \wedge A_I']$ (where the subscript indicates that quantities are as defined in the initial specification). Users of the system being constructed are allowed to execute event e in a system state only if the state satisfies $possible_I(e)$. (Typically, an initial specification can be arranged such that $S(e) \equiv true$ and input event occurrences do not falsify A_I, in that case, $possible_I(e) \equiv enabled_I(e)$.)

$-Initial_Y \Rightarrow Initial_X$,
$-\forall\ e \in Events_Y \cap Events_X$: $A \wedge S(e) \wedge formula_Y(e) \Rightarrow formula_X(e)$, and
$-\forall\ e \in Events_Y - Events_X$: $A \wedge S(e) \wedge formula_Y\ (e) \Rightarrow [\forall\ v \in Variables_X$: $v = v']$.

where A and $S(e)$ are invariant and event requirements, respectively, of specification Y. If the above conditions are satisfied, we say that Y is a *conditional refinement* of X, that is, a refinement of X given that the invariant and event requirements of Y hold.

The system refinement steps used to derive Y from X may cause some requirements that are marked for X to become unmarked for Y. By requiring Y to be a conditional refinement of X, we limit the unmarking of requirements. Specifically, the Marking of X is preserved for Y, except in the following two cases: (1) An event requirement S_j of e that is marked for X becomes unmarked iff $A \wedge enabled_Y(e) \Rightarrow S_j$ does not hold for Y. (2) A progress requirement P *leads-to* Q that was marked *via* e for X becomes unmarked iff $P \wedge A \wedge S(e) \Rightarrow enabled_Y(e)$ does not hold for Y. (The advantages and disadvantages of allowing the Marking to decrease are discussed in Section 7.)

We also require that the system refinement steps do not strengthen enabling conditions of input events to the point where users are blocked from executing them in states allowed by X. Formally, for every input event e, we want $enabled_X(e) \Rightarrow enabled_Y(e)$ to be invariant (i.e., to be implied by A).[8]

Finally, we point out that, in some situations, enforcing this condition may result in a specification Y that is practically impossible to implement (e.g., because it requires unbounded memory). The alternative in that case is to backtrack to an earlier point in the construction and to attempt a different construction. Sometimes backtracking does not help, because the problem is with the initial specification. In that case, the only alternative is to modify the initial specification; this happens in the protocol construction below and is discussed further in Section 7.

4. SLIDING WINDOW PROTOCOL CONSTRUCTION: INITIAL PHASE

Consider the topology in Figure 1. Entity 1 is a producer of data blocks, and Entity 2 is a consumer of data blocks. The channels may lose, duplicate, or reorder messages in transit; these are the only errors in the channels. We want data blocks to be consumed in the same order as they were produced and within a finite time of being produced. We construct a sliding window protocol that uses modulo-N sequence numbers to achieve this objective.

Notation. If B is a set of values, then *sequence of* B denotes the set of finite sequences whose elements are in B, and *sequence*$(0 \cdot \cdot M-1)$ *of* B denotes the set of M-length sequences whose elements are in B. For any sequence y, let $|y|$ denote the length of y, and let $y(i)$ denote the ith element in y, with $y(0)$ being the leftmost element. Thus, $y =$

[8] A weaker sufficient condition is for $possible_X(e) \Rightarrow possible_Y(e)$ to be invariant.

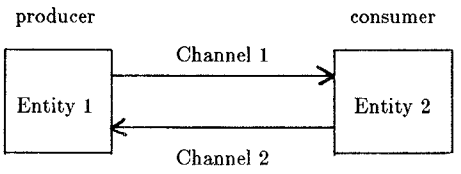

Fig. 1. The network topology.

$\langle y(0), \ldots, y(|\, y\,| - 1)\rangle$. We use $\langle\ \rangle$ to denote the null sequence. We use $y(i \cdot\cdot j)$ to denote $\langle y(i), y(i+1), \ldots, y(j)\rangle$ where $i, j < |\, y\,|$; it equals $\langle\ \rangle$ if $i > j$. We say "y *prefix-of* z" to mean $|\, y\,| \leq |\, z\,|$ and $y = z(0 \cdot\cdot |\, y\,| - 1)$. We define the function *Head*(y) to return $y(0)$ if $|\, y\,| > 0$, and *false* if $|\, y\,| = 0$. We define the function *tail*(y, i) to return $y(i \cdot\cdot |\, y\,| - 1)$ for any i, $0 \leq i < |\, y\,|$; that is, y with the leftmost i elements removed. We use @ as the concatenation operator for sequences. Given two sequences y and z, $y@z$ is the sequence $\langle y(0), \ldots, y(|\, y\,| - 1), z(0), \ldots, z(|\, z\,| - 1)\rangle$. Thus, the sequence obtained by appending an element b to (the right of) a sequence y is $y@\langle b\rangle$. Last, we use "wrt" as an abbreviation for "with respect to."

4.1 Initial System and Requirements

The initial system and requirements specify the services to be offered to the producer and the consumer. Let *DATA* denote the set of data blocks that can be sent in this protocol. We use a Pascal-like notation to define state variables and their domains.

At Entity 1, we have the following state variable and input event:

produced: sequence of *DATA*. Initially $\langle\ \rangle$.
Produce(*data*) $\equiv$ *produced'* = *produced*@$\langle$*data*$\rangle$

At Entity 2, we have the following state variable and event:

consumed: sequence of *DATA*. Initially $\langle\ \rangle$.
Consume(*data*) $\neq$ *consumed'* = *consumed*@$\langle$*data*$\rangle$

The state variables *produced* and *consumed* record the sequences of data blocks produced and consumed, respectively. In the sliding window protocols to be constructed, they will be auxiliary variables. The events *Produce* and *Consume* have a parameter *data* whose domain is *DATA*. *Produce* is the only input event of this construction. Observe that occurrence of input event *Produce* can be initiated by a protocol user in any state of Entity 1 and with any value of parameter *data*.

There is one invariant requirement and one progress requirement:

$A_0 \equiv$ *consumed prefix-of produced*
$L_0 \equiv$ | *produced* | $\geq n$ *leads-to* | *consumed* | $\geq n$

A_0 specifies that data blocks are consumed in the order they are produced. It holds initially. L_0 states that if a data block is produced then it is eventually consumed (parameter n is a natural number).

For each channel i shown in Figure 1, $i = 1, 2$, we define a state variable and events as shown in Table 1, where *MESSAGES* denotes a set of protocol

Table I. State Variables and Events of Channel i

$\mathbf{z}_i$: sequence of *MESSAGES*. Initial $\langle\ \rangle$.

$Loss_i \equiv [\exists\ n \in [0 \cdot\cdot |\mathbf{z}_i| - 1]: \mathbf{z}'_i = \mathbf{z}_i(0 \cdot\cdot n-1)@\mathbf{z}_i(n+1 \cdot\cdot |\mathbf{z}_i| - 1)]$

$Duplicate_i \equiv [\exists\ n \in [0 \cdot\cdot |\mathbf{z}_i| - 1]: \mathbf{z}'_i = \mathbf{z}_i(0 \cdot\cdot n)@\mathbf{z}_i(n \cdot\cdot |\mathbf{z}_i| - 1)]$

$Reorder_i \equiv [\exists\ n \in [1 \cdot\cdot |\mathbf{z}_i| - 1], \exists\ m \in [0 \cdot\cdot n-1]:$
$\mathbf{z}'_i = \mathbf{z}_i(0 \cdot\cdot m-1)@\mathbf{z}_i(n)@\mathbf{z}_i(m+1 \cdot\cdot n-1)@\mathbf{z}_i(m)@\mathbf{z}_i(n+1 \cdot\cdot |\mathbf{z}_i| - 1)]$

messages. The send and receive primitives for channel i are defined by the formulas

$$Send_i(m) \equiv z'_i = \mathbf{z}_i@\langle m\rangle$$
$$Rec_i(m) \equiv \mathbf{z}_i = \langle m\rangle@\mathbf{z}'_i$$

where m denotes a message. Note that $Rec_i(m)$ is *false* if $\mathbf{z}_i$ is empty.

4.2 The Sliding Window Mechanism

We want to refine the initial state transition system to a sliding window protocol. Let us review the basic features found in all sliding window protocols (see Figure 2). At any time at Entity 1, the data blocks in $produced(0 \cdot\cdot a-1)$ have been sent and acknowledged, while data blocks in $produced(a \cdot\cdot s-1)$ are unacknowledged, where $|\ produced\ | = s$. At any time at Entity 2, data blocks in $consumed(0 \cdot\cdot r-1)$ have been received and consumed in sequence, while data blocks in $consumed(r \cdot\cdot r+RW-1)$ 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.

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

Entity 1 sends $produced(n)$ accompanied by sequence number $\bar{n}$. When Entity 2 receives a data block with sequence number $\bar{n}$, if there is a number i in the receive window such that $\bar{i} = \bar{n}$, then the received data block is interpreted as $produced(i)$. Entity 2 sends acknowledgment messages containing $\bar{n}$, where n is the current value of r. When Entity 1 receives the sequence number $\bar{n}$, if there is a number i in the range $a+1$ to s such that $\bar{i} = \bar{n}$, then it is interpreted as an acknowledgment to data blocks a to $i-1$, and a is updated to i. Entity 1 increments s when a data block is produced. Entity 2 increments r when a data block is consumed.

Observe that each cyclic sequence number $\bar{n}$ corresponds to an *unbounded sequence number* n. When a cyclic sequence number is received at an entity, we require the entity to interpret correctly the value of the corresponding unbounded sequence number (which is not available in the message); that is, we require that $i = n$ in the preceding paragraph.

4.2.1 *Refinement of State Transition System and Requirements.* We next incorporate the above protocol features into the state transition system. Let the messages sent by Entity 1 be of type $(D, data, cn, n)$ where D is

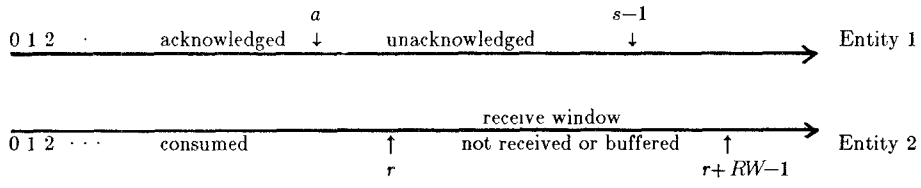

Fig. 2. Relationship between a, s, and r.

a constant that indicates the type of message, *data* is a data block, *cn* is a cyclic sequence number, and n is the corresponding unbounded sequence number. Let the acknowledgment messages sent by Entity 2 be of type (ACK, cn, n), where ACK is a constant that indicates the type of message, *cn* is a cyclic sequence number, and n is the corresponding unbounded sequence number. In both message types, n is an auxiliary field that is used to reason about correct interpretation only. Its value can never be used to update a nonauxiliary state variable. We have the following invariant requirements, each of which holds initially:

$$A_1 \equiv (D, data, cn, n) \in \mathbf{z}_1 \Rightarrow data = produced(n) \wedge cn = \overline{n},$$
$$A_2 \equiv (ACK, cn, n) \in \mathbf{z}_2 \Rightarrow cn = \overline{n}$$

At Entity 1 we add the following state variables:

s: $0 \cdot \cdot \infty$. Initially 0.
a: $0 \cdot \cdot \infty$. Initially 0.
sendbuff: sequence of *DATA*. Initially $\langle\ \rangle$.

s and a are as defined above. We ensure below that *sendbuff* always equals $produced(a \cdot \cdot s - 1)$, the unacknowledged data blocks. Entity 1 must retransmit these until they are acknowledged.

Conventions. For brevity in specifying events, we use the notation $P \rightarrow q$ to denote an action that does q if P holds and does nothing if $\neg P$ holds. Formally, $P \rightarrow q$ means that $(P \wedge q \vee (\neg P \wedge [\forall\ v \in Vars\colon v = v'])$, where *Vars* denotes those state variables updated in q. Similarly, $[\exists\ i\colon P \rightarrow q]$, where i is free in P and q, means $[\exists\ i\colon P \wedge q] \vee (\neg[\exists\ i\colon P] \wedge [\forall\ v \in Vars\colon v = v'])$.

At Entity 1 we refine the input event *Produce* to update *sendbuff* and s appropriately; note that this does not affect its enabling condition. We also add two events, one for sending data messages and one for receiving ack messages.

$$\begin{aligned}
Produce(data) &\equiv produced' = produced@\langle data\rangle \\
&\quad \wedge sendbuff' = sendbuff@\langle data\rangle \wedge s' = s + 1 \\
SendD(i) &\equiv i \in [0 \cdot \cdot s - a - 1] \\
&\quad \wedge Send_1(D, sendbuff(i), \overline{a + i}, a + i) \\
RecACK(cn, n) &\equiv Rec_2(ACK, cn, n) \\
&\quad \wedge [\exists\ i \in [1 \cdot \cdot s - a]\colon \overline{a + i} = cn \\
&\quad \rightarrow (a' = a + i \wedge sendbuff' = tail(sendbuff, i))]
\end{aligned}$$

At Entity 2 we add the following state variables, where *empty* is a constant not in *DATA*:

r: $0 \cdot\cdot \infty$. Initially 0.
recbuff: sequence($0 \cdot\cdot RW - 1$) of $DATA \cup \{empty\}$. Initially $recbuff(n) = empty$ for all n.

r equals $|\, consumed \,|$, as defined above. *recbuff* represents the buffers of the receive window. We ensure that, at any time, $recbuff(i)$ equals either *empty* or $produced(r + i)$.

At Entity 2, we refine *Consume* so that it passes $recbuff(0)$ only when the latter is not empty. We also add two events, one for sending ack messages and one for receiving data messages.

$$\begin{aligned} Consume(data) \equiv\ & recbuff(0) \neq empty \wedge data = recbuff(0) \\ & \wedge recbuff' = tail(recbuff, 1)@\langle empty\rangle \wedge r' = r + 1 \\ & \wedge consumed' = consumed@\langle data\rangle, \\ SendACK \equiv\ & Send_2(ACK, \bar{r}, r), \\ RecD(data, cn, n) \equiv\ & Rec_1(D, data, cn, n) \\ & \wedge [\exists\ i \in [0 \cdot\cdot RW - 1]: \overline{r + i} = cn \rightarrow recbuff(i)' = data] \end{aligned}$$

We add the following invariant requirements; each is a desired property mentioned in the discussion above:

$A_3 \equiv |\, produced \,| = s \wedge |\, consumed = r$
$A_4 \equiv 0 \leq a \leq r \leq s$
$A_5 \equiv sendbuff = produced(a \cdot\cdot s - 1)$
$A_6 \equiv i \in [0 \cdot\cdot RW - 1] \Rightarrow recbuff(i) = empty \vee recbuff(i) = produced(r + i)$

4.2.2 *Marking.* For the time being, we concentrate on marking the (A_i, e) pairs. We represent the Marking by a table that has a row for each A_i and a column for each e. If (A_i, e) is unmarked, its entry in the table is blank. If (A_i, e) is marked, its entry identifies a subset J of the A_j's and S_j's of e such that $J \wedge e \Rightarrow A_i'$ holds. Thus, the reader can easily check the validity of the Marking. Also, an (A_i, e) entry in the table contains *na* to indicate that e does *not affect* any of the state variables of A_i; thus, $A_i \wedge e \Rightarrow A_i'$ holds trivially. We use $A_{i,j}$ to denote $A_i \wedge A_j$ and A_{i-j} to denote $A_i \wedge A_{i+1} \wedge \cdots A_j$. The *LRD* column is for the loss, reordering, and duplication events of the channels; specifically: $LRD \equiv LRD_1 \vee LRD_2$, where $LRD_i \equiv Loss_i \vee Duplicate_i \vee Reorder_i$.

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*LRD*
A_0	A_0	*na*	*na*	$A_{6,3,0}$	*na*	*na*	*na*
A_1	*na*	$A_{1,5}$	*na*	*na*	*na*	A_1	A_1
A_2	*na*	*na*	A_2	*na*	A_2	*na*	A_2
A_3	A_3	*na*	*na*	A_3	*na*	*na*	*na*
A_4	A_4	*na*		$A_{6,3,4}$	*na*	*na*	*na*
A_5	$A_{5,3,4}$	*na*	A_5	*na*	*na*	*na*	*na*
A_6	$A_{6,3,4}$	*na*	*na*	A_6	*na*		*na*

The Marking can be easily checked as follows: Consider the entry for $(A_4, Consume)$, which indicates that $A_{6,3,4} \wedge formula(Consume) \Rightarrow A'_4$ holds. *Consume* occurs only if $recbuff(0) \neq empty$. This and A_6 imply that $recbuff(0) = produced(r)$, which together with A_3 imply that $r \leq s - 1$. This and A_4 imply that $a \leq r \leq s - 1$. *Consume* does the update $r' = r + 1$ and does not affect a or s. Thus, A'_4 holds. In the above proof, we used A_6 first, then A_3, and then A_4. To facilitate checking of the Marking, we have indicated this in the order of the subscripts in $A_{6,3,4}$.

Observe that the only (A_i, e) pairs that are unmarked are $(A_6, RecD)$ and $(A_4, RecACK)$. We can mark $(A_6, RecD)$ if we can ensure that *RecD* correctly interprets the cyclic sequence numbers in received data messages. Similarly, we can mark $(A_4, RecACK)$ if we can ensure that *RecACK* correctly interprets the cyclic sequence numbers in received acknowledgment messages. In the next two subsections, we generate invariant requirements on the sequence numbers that ensure correct interpretation.

4.3 Correct Interpretation of Data Messages

In this section we concentrate on marking $(A_6, RecD)$. Our general approach to marking an (A_i, e) pair is as follows: (1) Obtain a weakest precondition P of A_i with respect to e; (2) if $A \wedge S(e) \Rightarrow P$ does not hold, then introduce P as a new event requirement of e; and (3) mark (A_i, e). Sometimes we simplify the expression for P to either a sufficient or a necessary precondition; in the latter case, (A_i, e) remains unmarked. An alternative to introducing P as an event requirement is to introduce $enabled(e) \Rightarrow P$ as an invariant requirement, provided that $Initial \Rightarrow (enabled(e) \Rightarrow P)$ holds. We take such a step when we expect that the enabling condition of e will not be strengthened in future refinement steps. Finally, in our construction, P often has the form of an implication where the antecedent implies $enabled(e)$; in this case, we can introduce P, rather than $enabled(e) \Rightarrow P$, as an invariant requirement.

In practice, applying this approach requires insight into the particular problem being solved, such as in choosing which (A_i, e) pair to mark next, in deciding how to simplify a precondition, etc. In the case of marking $(A_6, RecD)$, we want to ensure that every data message received by Entity 2 is interpreted correctly. Intuitively, a $(D, data, cn, n)$ message is *incorrectly* interpreted if n lies so far outside the receive window that n mod N "wraps around" and matches some integer in the window. Thus, our first step is to determine a range of sequence numbers enclosing the receive window such that, for any sequence number n in this range, n mod N is correctly interpreted. Then, we determine constraints on the send window and on *Produce* such that the sequence number of messages in Channel 1 lie within this range.

The following is a weakest precondition of A_6 wrt *RecD*:

$$W \equiv Head(\mathbf{z}_1) = (D, data, cn, n) \wedge i \in [0 \cdot \cdot RW - 1] \wedge \overline{r + i} = \overline{n} \Rightarrow data = produced(r + i)$$

Instead of introducing W as an event requirement, we strengthen it to to obtain a simpler sufficient precondition. From A_1, we have $cn = \overline{n}$

and $data = produced(n)$. Thus, the consequent of W is equivalent to $produced(n) = produced(r + i)$. Let us strengthen this consequent to $n = r + i$. We do not expect this to lead to unsuccessful termination; indeed, unless $|DATA| = 1$, it appears necessary in order for $produced(n)$ and $produced(r + i)$ to be arbitrary entries from $DATA$. Next, let us weaken the antecedent of W by replacing $Head(\mathbf{z}_1) = (D, data, cn, n)$ by $(D, data, cn, n) \in \mathbf{z}_1$. In fact, this is necessary given that Channel 1 can lose messages arbitrarily. Thus, we arrive at the following sufficient precondition:

$$X \equiv (D, data, cn, n) \in \mathbf{z}_1 \wedge i \in [0 \cdot \cdot RW - 1] \wedge \overline{r + i} = \bar{n} \Rightarrow n = r + i$$

We could introduce X as an event requirement of $RecD$. However, we do not expect to strengthen the enabling condition of $RecD$ in future refinement steps, because we do not want Entity 2 to discard any received data message. Therefore, we decide to introduce $enabled(RecD) \Rightarrow X$ as an invariant requirement. But, because the antecedent of X implies $enabled(RecD)$, we can introduce X as an invariant requirement. Observe that X holds initially. We now proceed to generate further refinements from X.

Because $produced(r)$ is the data block to be consumed next, it is reasonable to expect that $(D, data, \bar{r}, r) \in \mathbf{z}_1$ may hold at any time. This would violate X with $i = N$ unless $RW \leq N$. We also know that $RW \geq 1$; otherwise, Entity 2 will never accept any data block, and the progress requirement L_0 will never hold. Thus, we have the following condition:

$$1 \leq RW \leq N$$

Observe that $i \in [0 \cdot \cdot RW - 1] \wedge \overline{r + i} = \bar{n}$ iff $i \in [0 \cdot \cdot RW - 1] \wedge \bar{i} = \overline{n - r}$ iff $\overline{n - r} \in [0 \cdot \cdot RW - 1] \wedge i = \overline{n - r}$, where we used $RW \leq N \Rightarrow i = \bar{i}$ to establish the last "iff." Thus, we can refine $RecD$ to the following, where we have also used the modulo arithmetic property $(n - r) \bmod N = (\bar{n} - r) \bmod N$:

$$RecD(data, cn, n) \equiv Rec_1(D, data, cn, n) \wedge \overline{cn - r} \in [0 \cdot \cdot RW - 1] \rightarrow recbuff(\overline{cn - r})' = data]$$

We can now refine X to the following invariant requirement:

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

Y is satisfied nonvacuously by $n - r \in [0 \cdot \cdot RW - 1]$, and satisfied vacuously by $n - r \in [RW + kN \cdot \cdot N - 1 + kN]$ for any integer k. We want every unbounded sequence number n in Channel 1 to be in the union of these intervals. Suppose that n_1 and n_2 are in Channel 1; let us assume that Channel 1 may contain any n between n_1 and n_2. We expect that an n equal to r may always be in Channel 1. The largest contiguous union of intervals containing r is $[r + RW - N \cdot \cdot r + N - 1]$, which is the union of $[r \cdot \cdot r + RW - 1]$ and $[r + RW + kN \cdot \cdot r + N - 1 + kN]$ for $k = 0$ and -1. Thus, we strengthen Y to the following invariant requirement:

$$A_7 \equiv (D, data, cn, n) \in \mathbf{z}_1 \Rightarrow n \in [r - N + RW \cdot \cdot r + N - 1]$$

We now proceed to mark $(A_7, SendD)$. A weakest precondition of A_7 wrt $SendD$ is $a \geq r - N + RW$. We make it an invariant requirement because we want $SendD$ to be always enabled to send outstanding data. Because $r \leq s$ (and we expect $r = s$ to be possible at any time), we strengthen it to the following invariant requirement:

$$A_8 \equiv s - a \leq N - RW$$

Produce is the only event that can falsify A_8. Because A_8 only involves variables of Entity 1, it can be enforced by strengthening the enabling condition of *Produce* with the conjunct $s - a \leq N - RW - 1$. However, *Produce* is an input event, and the initial specification allows the user to execute it in any state. The above refinement would block *Produce* in certain states.

Let us review our options. We are in this situation because when we first defined $SendD$, we allowed it to send any data in $[a \cdot \cdot s - 1]$. One option is to backtrack and redefine $SendD$ so that it can send only a subset of the data blocks in $[a \cdot \cdot s - 1]$, say, $[a \cdot \cdot t - 1]$; then A_8 would become $t - a \leq N - RW$, and data blocks in $[t \cdot \cdot s - 1]$, which have been produced but cannot yet be sent, would have to be buffered in Entity 1. However, this option requires Entity 1 to have unbounded memory capacity. In fact, it is easy to see that there is no way to avoid this with the given initial specification: Because channels can lose messages, Entity 1 must buffer a data block until it is acknowledged; because the user is never prevented from producing more data, Entity 1 must be prepared to buffer an unbounded amount of data.

So let us modify the initial specification as follows: We allow *Produce* to be blocked when $s - a = N - RW$, but we require that if *Produce* is blocked then it eventually becomes unblocked and stays unblocked at least until its next occurrence. We refine *Produce* as follows:

$$\begin{aligned} Produce(data) \equiv\ & s - a \leq N - RW - 1 \\ & \wedge\ produced' = produced@data \\ & \wedge\ sendbuff' = sendbuff@data \wedge s' = s + 1 \end{aligned}$$

We add the following progress requirement:

$$L_1 \equiv a = n \wedge s - a = N - RW \ leads\text{-}to\ a \geq n + 1$$

In order for *Produce* not to be permanently disabled (needed for L_1), we now require the following:

$$1 \leq RW \leq N - 1$$

Observe that the upper bound in A_7's consequent is implied by $n \leq s - 1$ (from $A_{1,3}$), A_4, and A_8. There is no need for A_7 to repeat this constraint. Thus, we can rewrite A_7 as follows:

$$A_7 \equiv (D, data, cn, n) \in \mathbf{z}_1 \Rightarrow n \geq r - N + RW$$

We can extend the previous Marking to the following, where * is used to indicate an old entry, and old A_i's marked wrt every event have been

aggregated into one row:

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*LRD*
$A_{0\sim3,5}$	*	*	*	*	*	*	*
A_4	*	*		*	*	*	*
A_6	*	*	*	*	*	$A_{7,1}$	*
A_7	na	$A_{8,4}$	na		na	A_7	A_7
A_8	A_8	na	A_8	na	na	na	na

4.4 Correct Interpretation of Acknowledgment Messages

In this section we concentrate on marking $(A_4, RecACK)$. The treatment is similar to the case of data messages above. The following is a weakest precondition of A_4 wrt $RecACK$:

$$W \equiv Head(\mathbf{z}_2) = (ACK, cn, n) \wedge i \in [1 \cdot\cdot s - a] \wedge \overline{a + i} = \bar{n} \Rightarrow a + i \leq r$$

Instead of W being an event requirement of $RecACK$, we decide to make it an invariant requirement because we do not want Entity 1 to discard any received ack message. We strengthen the consequent from $a + i \leq r$ to $a + i = n \wedge a + i \leq r$, where the first conjunct specifies correct interpretation. We weaken the antecedent by replacing $Head(\mathbf{z}_2) = (ACK, cn, n)$ by $(ACK, cn, n) \in \mathbf{z}_2$. Rewriting in two parts, we have

$$X \equiv (ACK, cn, n) \in \mathbf{z}_2 \wedge i \in [1 \cdot\cdot s - a] \wedge \overline{a + i} = \bar{n} \Rightarrow n \leq r$$
$$Y \equiv (ACK, cn, n) \in \mathbf{z}_2 \wedge i \in [1 \cdot\cdot s - a] \wedge \overline{a + i} = \bar{n} \Rightarrow a + i = n$$

Because an (ACK, cn, n) message is sent with $n = r$ and because r never decreases, X is satisfied invariantly by the current system; that is, we can add X as an invariant requirement and mark it wrt all events. In fact, the second and third conjuncts in the antecedent of X are not needed to justify the marking. Thus, with no additional work we can strengthen X to the following invariant requirement (which can be marked wrt all events):

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

We now consider Y. Observe that $i \in [1 \cdot\cdot s - a] \wedge \overline{a + i} = \bar{n}$ iff $i \in [1 \cdot\cdot s - a] \wedge \bar{i} = \overline{n - a}$ iff $\overline{n - a} \in [1 \cdot\cdot s - a] \wedge i = \overline{n - a}$, where we used $A_8 \Rightarrow i = \bar{i}$ in deriving the last "iff." Thus, we can refine $RecACK$ to the following:

$$RecACK(cn, n) \equiv Rec_2(ACK, cn, n)$$
$$\wedge [\overline{cn - a} \in [1 \cdot\cdot s - a]$$
$$\rightarrow (a' = a + \overline{cn - a} \wedge sendbuff' = tail(sendbuff, \overline{cn - a}))]$$

We can refine Y to the following invariant requirement:

$$Z \equiv (ACK, cn, n) \in \mathbf{z}_2 \wedge \overline{n - a} \in [1 \cdot\cdot s - a] \Rightarrow n - a = \overline{n - a}$$

Z is satisfied nonvacuously by $n - a \in [1 \cdot\cdot s - a]$ and vacuously by $\overline{n - a} \notin [1 \cdot\cdot s - a]$. Using the largest contiguous set of n satisfying these bounds and including $n = a$, we can refine Z to the following:

$$U \equiv (ACK, cn, n) \in \mathbf{z}_2 \Rightarrow n \in [s - N + 1 \cdot\cdot a + N].$$

The upper bound in U's consequent is implied by $n \leq r(A_9)$, $a \leq r \leq s(A_4)$, and $s \leq a + N - RW(A_8)$. Thus, we can refine U to the following invariant requirement:

$$A_{10} \equiv (ACK, cn, n) \in \mathbf{z}_2 \Rightarrow n \geq s - N + 1$$

We have the following Marking:

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*LRD*
$A_{0-3,5,6,8}$	*	*	*	*	*	*	*
A_4	*	*	A_{8-10}	*	*	*	*
A_7	*	*	*		*	*	*
A_9	*na*	*na*	A_9	A_9	A_9	*na*	A_9
A_{10}		*na*	A_{10}	*na*	$A_{10,8,4}$	*na*	A_{10}

The invariant requirements and the state transition system at this point are specified in Tables II and III. As shown in the Marking above, the only unmarked pairs are $(A_7, Consume)$ and $(A_{10}, Produce)$. To mark $(A_7, Consume)$ we have to ensure that, whenever $recbuff(0) \neq empty$, $\mathbf{z}_1$ does not contain unbounded sequence numbers less than N below the "top of the receive window," $r + RW$. Similarly, to mark $(A_{10}, Produce)$, we have to ensure that, whenever $s - a < N - RW$, $\mathbf{z}_2$ does not contain unbounded sequence numbers less than N below the "top of the send window," $s + 1$.

4.5 Progress Requirement Marking

We now try to mark L_0 and L_1. For the current system, we prove that L_0 and L_1 hold if $\{SendD(0)\}$, $\{SendACK\}$ and $\{Consume(data): data \in DATA\}$ have weak fairness. We then show that these properties continue to hold if Entity 2 sends an ack only in response to a received data message. For the progress markings in this section, we consider the L_i's to be ordered according to increasing subscripts. Hence, L_j is used in the tag of L_i only if $j > i$.

The following progress requirement implies L_0 and L_1:

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

We have the following Marking, where each tag also indicates the invariant requirements used to mark the progress requirement: L_0 by closure using L_2 and $A_{3,4}$, and L_1 by closure using L_2 and A_3 (and $N - RW \geq 1$). At this point, only L_2 is unmarked. L_2 follows from the closure of the following progress requirements:

$$L_3 \equiv s > r = a = n \textit{ leads-to } s \geq r > a = n$$
$$L_4 \equiv s \geq r > a = n \textit{ leads-to } a > n$$

L_3 and L_4 are implied by the following progress requirements, which hold for the current system. Here $(ACK, > n)$ denotes the message set

Table II. Invariant Requirements for the Basic Protocol

Properties relating state variables at the entities

$1 \le RW \le N - 1$

$A_0 \equiv$ *consumed prefix-of produced*
$A_3 \equiv |\ produced\ | = s \wedge |\ consumed\ | = r$
$A_4 \equiv 0 \le a \le r \le s$
$A_5 \equiv sendbuff = produced(a \cdot \cdot s - 1)$
$A_6 \equiv i \in [0 \cdot \cdot RW - 1] \Rightarrow recbuff(i) = empty \vee recbuff(\imath) = produced(r + \imath)$
$A_8 \equiv s - a \le N - RW$

Properties of D messages

$A_1 \equiv (D, data, cn, n) \in \mathbf{z}_1 \Rightarrow data = produced(n) \wedge cn = \bar{n}$
$A_7 \equiv (D, data, cn, n) \in \mathbf{z}_1 \Rightarrow n \ge r - N + RW$

Properties of *ACK* messages

$A_2 \equiv (ACK, cn, n) \in \mathbf{z}_2 \Rightarrow cn = \bar{n}$
$A_9 \equiv (ACK, cn, n) \in \mathbf{z}_2 \Rightarrow n \le r$
$A_{10} \equiv (ACK, cn, n) \in \mathbf{z}_2 \Rightarrow n \ge s - N + 1$

$\{(ACK, \bar{j}, j)\colon j > n\}$, and (D, n) denotes the message set $\{(D, data, cn, n)\}$:

$L_5 \equiv s > r = a = n$ *leads-to* $s \ge r > a = n \vee (recbuff(0) \ne empty \wedge s > r = a = n)$
$L_6 \equiv recbuff(0) \ne empty \wedge s > r = a = n$ *leads-to* $s \ge r > a = n$
$L_7 \equiv s > a = n \wedge count(D, n) \ge k$ *leads-to*
$\quad a > n \vee count(D, n) \ge k + 1$
$L_8 \equiv s \ge r > a = n \wedge count(ACK, > n) \ge k$ *leads-to*
$\quad a > n \vee count(ACK, > n) \ge k + 1$

The details are summarized in the following progress Marking: L_0 by closure using L_2, $A_{3,4}$; L_1 by closure using L_2, A_3; L_2 by closure using L_3, L_4, A_4; L_3 by closure using L_5, L_6; L_4 via $(ACK, > n)$ using L_8, $A_{4,8,10}$; L_5 via (D, n) using L_7, $A_{4,1,8}$; L_6 via $\{Consume(data)\colon data \in DATA\}$ using A_4; L_7 via $\{SendD(0)\}$ using A_{3-5}; and L_8 via $\{SendACK\}$ using A_4.

4.5.1 *Weaker Acknowledgment Policy.* Suppose Entity 2 sends an ack message only if it has received a data message following the last ack sent. We can model this by adding a Boolean variable *drecd* initially *false*, refining *RecD* by adding the conjunct *drecd'*, and refining *SendACK* to $drecd \wedge Send_2(ACK, \bar{r}, r) \wedge \neg drecd'$. The only effect of this refinement on the Marking is to unmark progress requirement L_8, which was marked *via* $\{SendACK\}$. However, L_8 still holds because Entity 1 retransmits (D, n) as long as $s \ge r > a = n$ holds. To prove this, we introduce the following progress requirements:

$L_9 \equiv drecd \wedge r > n \wedge count(ACK, > n) \ge k$ *leads-to* $count(ACK, > n) \ge k + 1$
$L_{10} \equiv s \ge r > a = n \wedge count(ACK, > n) \ge k$ *leads-to*
$\quad a > n \vee (drecd \wedge s \ge r > a = n \wedge count(ACK, > n) \ge k)$
$L_{11} \equiv s \ge r > a = n \wedge count(ACK, > n) \ge k \wedge count(D, n) \ge l$ *leads-to*
$\quad a > n \vee (drecd \wedge s \ge r > a = n \wedge count(ACK, > n) \ge k)$
$\quad\quad \vee count(D, n) \ge l + 1$

Table III. System Specification for the Basic Protocol

Entity 1
produced: sequence of *DATA*. Initially $\langle\ \rangle$.
s: $0 \cdot\cdot \infty$. Initially 0.
a: $0 \cdot\cdot \infty$. Initially 0.
sendbuff: sequence of *DATA*. Initially $\langle\ \rangle$.

Produce(*data*)	$\equiv s - a \leq N - RW - 1$ $\wedge sendbuff' = sendbuff@\langle data\rangle \wedge s' = s + 1$ $\wedge produced' = produced@\langle data\rangle$
SendD(i)	$\equiv i \in [0 \cdot\cdot s - a - 1] \wedge Send_1(D, sendbuff(i), \overline{a + i}, a + i)$
RecACK(cn, n)	$\equiv Rec_2(ACK, cn, n)$ $\wedge [\overline{cn - a} \in [1 \cdot\cdot s - a]$ $\rightarrow (a' = a + \overline{cn - a} \wedge sendbuff' = tail(sendbuff, \overline{cn - a}))]$

Entity 2
consumed: sequence of *DATA*. Initially $\langle\ \rangle$.
r: $0 \cdot\cdot \infty$. Initially 0.
recbuff: sequence $(0 \cdot\cdot RW - 1)$ of $DATA \cup \{empty\}$. Initially $recbuff = empty$.

Consume(*data*)	$\equiv recbuff(0) \neq empty \wedge data = recbuff(0)$ $\wedge recbuff' = tail(recbuff, 1)@\langle empty\rangle \wedge r' = r + 1$ $\wedge consumed' = consumed@\langle data\rangle$
SendACK	$\equiv Send_2(ACK, \bar{r}, r)$
RecD(*data*, *cn n*)	$\equiv Rec_1(D, data, cn, n)$ $\wedge [\overline{cn - r} \in [0 \cdot\cdot RW - 1] \rightarrow recbuff(\overline{cn - r})' = data]$

Channels 1 and 2 defined as in Table I.

We have a complete Marking by replacing "L_8 via $\{SendACK\}$ using A_4" in the above Marking with the following: L_8 by closure using L_9, L_{10}; L_9 via $\{SendACK\}$; L_{10} via (D, n) using L_{11}; and L_{11} via $\{SendD(0)\}$.

5. COMPLETING THE CONSTRUCTION FOR LOSS-ONLY CHANNELS

At this point, we have obtained a system with entities as specified in Table III. For channels that can lose, reorder, and duplicate messages, the construction is incomplete because $(A_{10}, Produce)$ and $(A_7, Consume)$ are not yet marked. As explained at the end of Section 4.4, to mark these entries, we have to ensure that $\mathbf{z}_1$ and $\mathbf{z}_2$ do not contain messages that are "too far below" the top of the receive window and the top of the send window, respectively. In this section we show that if the channels can only lose messages, then these pairs can be marked for the current system.

We start by considering $(A_7, Consume)$. The following is a weakest precondition of A_7 wrt *Consume*:

$$(D, data, cn, n) \in \mathbf{z}_1 \wedge recbuff(0) \neq empty \Rightarrow n \geq r + RW - N + 1$$

If instead of a single occurrence of *Consume* we consider $k + 1$ occurrences, then we obtain the following weakest precondition

$$(D, data, cn, n) \in \mathbf{z}_1 \wedge [\forall\ i \in [0 \cdot\cdot k]\colon recbuff(i) \neq empty] \Rightarrow n \geq r + RW - N + k + 1$$

Let us treat the above as an invariant requirement, rather than as an event requirement of *Consume* (because we do not expect to strengthen the enabling condition of *Consume* in later steps). Now, if there have been no channel errors for a while, then $[\forall\, i \in [0 \cdot\cdot k]$: $recbuff(i) \neq empty]$ holds when $recbuff(k) \neq empty$ holds. Thus, it is reasonable to strengthen the above weakest precondition to the following invariant requirement:

$$B_0 \equiv (D, data, cn, n) \in \mathbf{z}_1 \wedge recbuff(k) \neq empty \Rightarrow n \geq r + k + RW + N + 1$$

The following is a weakest precondition of B_0 wrt *RecD*:

$$B_1 \equiv (D, d_1, cn_1, n_1)@(D, d_2, cn_2, n_2)\ subseq\ \mathbf{z}_1 \Rightarrow n_2 \geq n_1 + RW - N + 1$$

We can see that B_0 is preserved by *SendD* as follows: $recbuff(k) \neq empty$ implies that $s > r + k$, which together with $a \geq s - N + RW(A_8)$ implies that $a > r + k - N + RW$. Thus, *SendD* preserves B_0, because it sends only $produced(n)$ where $n \geq a$. The argument that *SendD* preserves B_1 is similar. $(D, d_1, cn_1, n_1) \in \mathbf{z}_1$ implies that $s > n_1$, which implies that $a > n_1 - N + RW$.

We now consider marking $(A_{10}, Produce)$. Because Entity 2 sends nondecreasing n and Channel 2 does not reorder messages, we expect the following to be invariant

$$B_2 \equiv (ACK, cn, n) \in \mathbf{z}_2 \Rightarrow n \geq a$$

B_2 implies A_{10} because $n \geq s - N + 1$ if $n \geq a$ (from A_8). Thus, marking $(B_2, Produce)$ allows us to mark $(A_{10}, Produce)$. The following is a weakest precondition of B_2 wrt *RecACK* and is introduced as an invariant requirement:

$$B_3 \equiv (ACK, cn_1, n_1)@(ACK, cn_2, n_2)\ subseq\ \mathbf{z}_2 \Rightarrow n_1 \leq n_2$$

At this point, we have the following complete Marking, where $Loss \equiv Loss_1 \vee Loss_2$:

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*Loss*
$A_{0-6,8,9}$	*	*	*	*	*	*	*
A_7	*	*	*	B_0	*	*	*
A_{10}	B_2, A_8	*	*	*	*	*	*
B_0	*na*	B_0, $A_{8,6,3}$	*na*	B_0	*na*	$B_{0,1}$	B_0
B_1	*na*	B_1, $A_{8,1,3}$	*na*	*na*	*na*	B_1	B_1
B_2	*na*	*na*	B_3	*na*	B_2, A_4	*na*	B_2
B_3	*na*	*na*	B_3	*na*	$A_{9,4}$	*na*	B_3

6. COMPLETING THE CONSTRUCTION FOR LOSS, REORDERING, AND DUPLICATION CHANNELS

For loss, reordering, and duplication channels, we resume the protocol construction from the end of Section 4, that is, from the requirements and system shown in Tables II and III, respectively. Recall that only the pairs $(A_7, Consume)$ and $(A_{10}, Produce)$ are unmarked; to mark them, we need to ensure that the channels do not contain messages whose sequence numbers are "too far below" the send and receive windows.

Clearly, if the channels can reorder and duplicate arbitrarily, then A_7 and A_{10} cannot be enforced unless the channels impose an upper bound on the

lifetimes of messages in transit. Such bounds are enforced in many real-life protocols [29, 31]. Therefore, we assume that a message cannot stay in Channel i for longer than a specified $MAXLIFE_i$ time units. Given this, we show that A_7 and A_{10} are enforced if Entity 1 produces $produced(n)$ only after (1) $MAXLIFE_1$ time units have elapsed since $produced(n - N + RW)$ was last sent, and (2) $MAXLIFE_2$ time units have elapsed since $produced(n - N + 1)$ was first acknowledged. We then provide three ways to implement these two time constraints, using $2N$ timers, N timers, and 1 timer, respectively. Because (1) and (2) strengthen the enabling condition of *Produce*, an input event, we introduce a progress requirement guaranteeing that if *Produce* is disabled then it eventually becomes enabled and stays enabled at least until its next occurrence.

6.1 Real-Time System Model

For this 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 [35]. We now give a summary description of that model, adequate for our purposes here.

The system model presented in Section 2 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 $\{OFF, 0, 1, 2, \ldots\}$. 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 correspond to the timers and clocks implemented within entities of a distributed system. They need not be auxiliary. 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 i is upper bounded by a specified constant ϵ_i, for example, $\epsilon_i \approx 10^{-6}$ for a crystal oscillator driven clock.

Ideal timers are auxiliary variables that record the actual time elapsed. There is 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 used to measure the error in the rate of local time event occurrences. They are also convenient for relating elapsed times across different entities and channels.

A timer of an entity can be incremented by its time event. It can also be updated to either 0 or *OFF* by an event of that entity. Updating to the value 0 is referred to as *starting* the timer (similar to resetting the timer). Updating to the value *OFF* is referred to as *stopping* the timer. Thus, a timer that has been started by an entity event occurrence and has not yet been stopped measures the time elapsed, in ticks, since the entity event occurrence.

Given an ideal timer u and a local timer v of entity 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 since that instant, neither u nor v has been started or stopped. The maximum error in the rate of entity i's local time event occurrences is modeled by assuming the following condition, which we shall refer to as the *accuracy axiom*:

ACCURACY AXIOM. $\mathit{Started\text{-}together}(u, v) \Rightarrow |u - v| \leq \mathit{max}(1, \epsilon_i u)$.

An invariant requirement A_i can include *started-together* predicates. To mark (A_i, e), that is, to derive $e \wedge A \Rightarrow A_i'$, we use the following rules. Rules (1) and (2) are used if e is not a time event, and rule (3) is used if e is a time event:

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

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

(3) $u' \neq \mathit{OFF} \wedge v' \neq \mathit{OFF} \wedge \mathit{started\text{-}together}(u, v)$ implies $\mathit{started\text{-}together}(u, v)'$.

With timers and time events, time constraints between event occurrences can be specified by safety assertions. 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 can be specified by the invariant requirement $\mathit{enabled}(e_2) \Rightarrow v \geq T$. The time constraint that e_2 *must occur within* T time units of e_1's occurrence can be specified by the invariant requirement $v \leq T$. Note that to establish the invariance of an A_i involving timers, we have to show that it is preserved by the time events also.

We have the following progress property, assuming that time events have weak fairness and that the specified time constraints are implementable [35]:

Increasing timer property. For any timer v: $v = n \neq \mathit{OFF}$ *leads-to* $v = n + 1 \vee v = \mathit{OFF}$.

Specification of bounded message lifetime. 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). The following are *assumed* to be invariant:

$$TA_1 \equiv (D, data, \bar{n}, n, age) \in \mathbf{z}_1 \Rightarrow \mathit{MAXLIFE}_1 \geq age \geq 0$$
$$TA_2 \equiv (ACK, \bar{n}, n, age) \in \mathbf{z}_2 \Rightarrow \mathit{MAXLIFE}_2 \geq age \geq 0$$

6.2 A Time Constraint that Enforces A_7

In this section we concentrate on marking $(A_7, \mathit{Consume})$. We show that A_7 is enforced if Entity 1 produces $\mathit{produced}(n)$ only after $\mathit{MAXLIFE}_1$ ideal time units have elapsed since $\mathit{produced}(n - N + RW)$ was last sent.

Due to buffered data blocks, it is always possible for successive occurrences of *Consume* to increase r so that it equals s. Unlike in the case of loss-only channels, this does not allow us to infer constraints on the sequence numbers in Channel 1. Thus, to enforce A_7, we require the following stronger

invariant requirement to hold:

$$C_0 \equiv (D, data, cn, n) \in \mathbf{z}_1 \Rightarrow n \geq s - N + RW$$

Taking the weakest precondition of C_0 wrt *Produce*, we get the following event requirement of *Produce*:

$$S_0 \equiv (D, data, cn, n) \in \mathbf{z}_1 \Rightarrow n \geq s - N + RW + 1$$

Note that this is the first precondition in this construction that we have left as an event requirement. This is because S_0 has exactly the same form as the invariant requirement C_0 from which it was derived, with RW being replaced by $RW + 1$ (or, equivalently, with N being replaced by $N - 1$). Therefore, transforming S_0 into an invariant requirement would merely lead us to repeat the step with a larger RW (or smaller N). Repeated reductions like this would eventually lead to $N = RW$, at which point we would have a "dead" protocol because of A_8. We point out that recognizing this fact is nontrivial and is left to the user of the heuristic.

S_0 can be enforced by enabling *Produce* only after $MAXLIFE_1$ time units have elapsed since the last send of any data block in $produced(0 \cdot \cdot s - N + RW)$. With this motivation, we add ideal timers $t_D(n)$, $n \geq 0$, at Entity 1 to record the ideal time elapsed since $produced(n)$ was last sent. We also refine *SendD* and introduce an invariant requirement as follows:

t_D: sequence $(0 \cdot \cdot \infty)$ of ideal timer. Initially $t_D(n) = OFF$ for every n.

$$SendD(i) \equiv i \in [0 \cdot \cdot s - a - 1] \wedge Send_1(D, sendbuff(i), \overline{a + i}, a + i) \wedge t_D(a + i)' = 0$$

$$C_1 \equiv (D, data, cn, n, age) \in \mathbf{z}_1 \Rightarrow \text{Rage} \geq t_D(n) \geq 0$$

We can enforce S_0 by having $X \equiv n \in [0 \cdot \cdot s - N + RW] \Rightarrow t_D(n) > MAXLIFE_1 \vee t_D(n) = OFF$ as an event requirement of *Produce*. This would make the following invariant:

$$C_2 \equiv n \in [0 \cdot \cdot s - N + RW - 1] \Rightarrow t_D(n) > MAXLIFE_1 \vee t_D(n) = OFF$$

C_2 is preserved by *SendD* because $a > s - N + RW - 1$, and by *Produce* because of X. Because C_2 is an invariant requirement, we can enforce X by enforcing the following event requirement of *Produce*:

$$S_1 \equiv n = s - N + RW \geq 0 \Rightarrow t_D(n) > MAXLIFE_1 \vee t_D(n) = OFF$$

The above discussion is formalized in the following Marking, which now includes event requirements, and where *Ite* denotes the *Ideal time event*:

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*LRD*	*Ite*
$A_{0-6,8,9}$	*	*	*	*	*	*	*	*na*
A_7	*	*	*	C_0, A_4	*	*	*	*na*
A_{10}		*	*	*	*	*	*	*na*
C_0	S_0	A_8	*na*	*na*	*na*	C_0	C_0	*na*
C_1	*na*	C_1	*na*	*na*	*na*	C_1	C_1	TA_1
C_2	S_1, C_2	*na*	*na*	*na*	*na*	*na*	*na*	C_2

S_0 marked using $S_1, C_{1,2}, TA_1$	S_1 not marked

To enforce S_1, it is sufficient for Entity 1 to keep track of the ideal timers in $t_D(s - N + RW \cdot \cdot s - 1)$. This can be done with a bounded number of local timers, each of bounded capacity.

6.3 A Time Constraint that Enforces A_{10}

In this section we concentrate on marking $(A_{10}, Produce)$. We show that A_{10} is enforced if Entity 1 produces a data block for $produced(n)$ only after $MAXLIFE_2$ ideal time units have elapsed since $produced(n - N + 1)$ was acknowledged.

Taking the weakest precondition of A_{10} wrt *Produce*, we get the following event requirement of *Produce* (which, as in the case of S_0, should not be transformed into an invariant requirement):

$$S_2 \equiv (ACK, cn, n) \in \mathbf{z}_2 \Rightarrow n \geq s - N + 2$$

S_2 can be enforced only by ensuring that more than $MAXLIFE_2$ time units have elapsed since $(ACK, \overline{n}, n)$ was last sent, for any $n \in [0 \cdot \cdot s - N + 1]$. Unlike the previous case involving data messages, Entity 1 does *not* have access to the time elapsed since $(ACK, \overline{n}, n)$ was last sent. This is because *ACK* messages are sent by Entity 2 and not by Entity 1. However, Entity 1 can obtain a lower bound on this elapsed time because of the following considerations: $(ACK, \overline{n}, n)$ is not sent once r exceeds n; a exceeds n only after r exceeds n; and a and r are nondecreasing quantities. Thus, the time elapsed since a exceeded n is a lower bound on the ages of all $(ACK, \overline{n}, n)$ in Channel 2. Furthermore, this elapsed time *can* be measured by Entity 1.

With this motivation, we add ideal timers $t_R(n)$, $n \geq 0$, at Entity 2 to record the ideal time elapsed since r first exceeded n, and refine *Consume* appropriately (for brevity, we only indicate the addition to the previous definition given in Table III):

t_R: sequence$(0 \cdot \cdot \infty)$ of ideal timer. Initially $t_R(n) = OFF$ for every n
$Consume(data) \equiv <$ definition in Table III $> \wedge\ t_R(r)' = 0$

At Entity 1, we add ideal timers $t_A(n)$, $n \geq 0$, to record the ideal time elapsed since a first exceeded n, and refine *RecACK* appropriately:

t_A: sequence$(0 \cdot \cdot \infty)$ of ideal timer. Initially $t_A(n) = OFF$ for every n.
$RecACK(cn, n) \equiv Rec_2(ACK, cn, n)$
$\quad \wedge\ \overline{cn - a} \in [1 \cdot \cdot s - a]$
$\quad\quad \rightarrow (a' = a + \overline{cn - a}\ \wedge sendbuff' = tail(sendbuff, \overline{cn - a})$
$\quad\quad\quad \wedge [\forall\ i \in [a \cdot \cdot a' - 1]\colon t_A(i)' = 0])]$

We have the following invariant requirements:

$C_3 \equiv t_R(0) \geq t_R(1) \geq \cdots \geq t_R(r - 1) \geq 0 \wedge t_R(r \cdot \cdot \infty) = OFF$
$C_4 \equiv (ACK, \overline{n}, n, age) \in \mathbf{z}_2 \wedge n < r \Rightarrow age \geq t_R(n) \geq 0$
$C_5 \equiv t_A(0) \geq t_A(1) \geq \cdots \geq t_A(a - 1) \geq 0 \wedge t_A(a \cdot \cdot \infty) = OFF$
$C_6 \equiv n \in [0 \cdot \cdot a - 1] \Rightarrow t_A(n) \leq t_R(n)$

From A_8, C_{4-6}, TA_2, and $1 \le RW \le N - 1$, we see that the following implies S_2:

$$S_3 \equiv n = s - N + 1 \ge 0 \Rightarrow t_A(n) > MAXLIFE_2$$

We have the following Marking (using A'_4 to mark some entries is acceptable because A_4 has been proven invariant; equivalently, we can replace A'_4 with its tag A_{8-10}):

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*LRD*	*Ite*
A_{0-9}, C_{0-2}	*	*	*	*	*	*	*	*
A_{10}	S_2	*	*	*	*	*	*	*
C_3	*na*	*na*	*na*	C_3	*na*	*na*	*na*	C_3
C_4	*na*	*na*	C_4	C_4, TA_2	C_4	*na*	C_4	C_4
C_5	*na*	*na*	C_5	*na*	*na*	*na*	*na*	C_5
C_6	*na*	*na*	C_6, A'_4, C_3	C_6, A_4	*na*	*na*	*na*	C_6

S_0 marked using $S_1, C_{2,1}, TA_1$	S_1 unmarked	S_2 marked using S_3, A_8, C_{4-6}, TA_2	S_3 unmarked

Note that S_1 and S_3 have the effect of inhibiting *Produce*, which is an input event. Below, we add a progress requirement implying that if *Produce* is disabled because of S_1 or S_3 then it eventually becomes enabled and stays enabled at least until its next occurrence.

6.4 Protocol I: Implementation with 2N Timers

The only unmarked requirements are S_1 and S_3. In Table IV we provide a system specification in which Entity 1 enforces S_1 and S_3 using two circular arrays of N local timers, namely, $timer_D$ and $timer_A$. (It is possible for $timer_D$ to be of size $N - RW$ and $timer_A$ to be of size $N - 1$, but it involves notation for modulo $N - RW$ and $N - 1$ arithmetic.)

Given an ideal timer u and a local timer v of Entity 1 that are started together, from the accuracy axiom it is clear that $u > T$ holds if $v \ge 1 + (1 + \epsilon_1)T$, or, equivalently, if v is a timer of capacity $(1 + \epsilon_1)T$ and is *OFF*. With this motivation, define $MLIFE_i = (1 + \epsilon_1)MAXLIFE_i$ for $i = 1$ and 2.

$timer_D$ is an array $(0 \cdot \cdot N - 1)$ of local timers, each of capacity $MLIFE_1$. For $n \in [\max(0, s - N + RW) \cdot \cdot s - 1]$, $timer_D(\overline{n})$ tracks $t_D(n)$ up to $MLIFE_1$ local time units with an accuracy of ϵ_1. Thus S_1 is enforced by including $timer_D(\overline{s - N + RW}) = OFF$, or, equivalently, $timer_D(\overline{s + RW}) = OFF$, in the enabling condition of *Produce*, as shown in Table IV.

$timer_A$ is an array $(0 \cdot \cdot N - 1)$ of local timers, each of capacity $MLIFE_2$. For $n \in [\max(0, s - N + 1) \cdot \cdot a - 1]$, $timer_A(\overline{n})$ tracks $t_A(n)$ up to $MLIFE_2$ local time units with an accuracy of ϵ_1. Thus, S_3 is enforced by including $timer_A(\overline{s - N + 1}) = OFF$, or, equivalently, $timer_A(\overline{s + 1}) = OFF$, in the enabling condition of *Produce*, as shown in Table IV.

For brevity, we omit the formal proof that this protocol satisfies the event requirements S_1 and S_3. (It is contained in Appendix B.)

Table IV. System Specification for Protocol I

Entity 1

produced, *s*, *a*, *sendbuff* defined as in Table III.

t_D, t_A: sequence $(0 \cdot \cdot \infty)$ of ideal timer. Initially $t_D = t_A = OFF$.

$timer_D$: sequence $(0 \cdot \cdot N-1)$ of local timer of capacity $MLIFE_1$. Initially $timer_D = OFF$

$timer_A$: sequence $(0 \cdot \cdot N-1)$ of local timer of capacity $MLIFE_2$. Initially $timer_A = OFF$.

$Produce(data) \equiv timer_D(\overline{s+RW}) = OFF \wedge timer_A(\overline{s+1}) = OFF$
$\wedge$⟨definition in Table III⟩

$SendD(i) \equiv$ ⟨definition in Table III⟩
$\wedge timer_D(\overline{a+i})' = 0 \wedge t_D(a+i)' = 0$

$RecACK(cn, n) \equiv Rec_2(ACK, cn, n)$
$\wedge[\overline{cn-a} \in [1 \cdot \cdot s-a]$
$\rightarrow (a' = a + \overline{cn-a} \wedge sendbuff' = tail(sendbuff, \overline{cn-a})$
$\wedge[\forall\, i \in [a \cdot \cdot a'-1]: t_A(i)' = timer_A(\bar{i})' = 0])]$

Entity 2

consumed, *r*, *recbuff* defined as in Table III.

t_R: sequence $(0 \cdot \cdot \infty)$ of ideal timer. Initially $t_R = OFF$.

$Consume(data) \equiv$ ⟨definition in Table III⟩ $\wedge t_R(r)' = 0$

$SendACK \equiv$ ⟨definition in Table III⟩

$RecD(data, cn, n) \equiv$ ⟨definition in Table III⟩

Channels 1 and 2 defined as in Table I.

Because this protocol enforces S_1 and S_2, it blocks *Produce* as long as $MAXLIFE_1$ time units have not elapsed since data block $s - N + RW$ was last sent and as long as $MAXLIFE_2$ time units have not elapsed since data block $s - N + 1$ was first acknowledged. To establish that *Produce* is not blocked indefinitely, we introduce the following progress requirement, where *enabled*(*Produce*) denotes the enabling condition of *Produce*, as defined in Table IV.

$$L_{12} \equiv \neg\, enabled(Produce)\ \textit{leads-to}\ enabled(Produce)$$

L_{12} states that if *Produce* is disabled then it eventually becomes enabled; from *enabled*(*Produce*), we note that if *Produce* is enabled it stays enabled at least until its next occurrence. To see why L_{12} holds, suppose that *Produce* is not enabled. There are two cases. If it is not enabled because sufficient time has not elapsed, then, by merely waiting, sufficient time will elapse (and the relevant timers will become *OFF*). If *Produce* is not enabled because $s - a = N - RW$, then L_1 assures us that $s - a < N - RW$ will hold eventually. In either case, *Produce* will be enabled eventually.

To summarize, L_{12} can be marked by closure using L_1 and the increasing timer property. S_1 and S_3 can be marked as indicated above. The current system is a refinement of the basic protocol, and the only event whose enabling condition has changed is *Produce*. Hence, the previous Marking holds, specifically, strengthening the enabling condition of *Produce* does not unmark any progress requirement because no progress marking would have

Table V. System Specification for Protocol II

Entity 1

produced, *s*, *a*, *sendbuff*, t_D, t_A defined as in Table IV.

$timer_A$: sequence $(0 \cdot \cdot N-1)$ of local timer of capacity $\max(MLIFE_1, MLIFE_2)$.

Initially $timer_A = OFF$.

Produce(*data*)	$\equiv$ ⟨definition in Table III⟩ $\wedge timer_A(\overline{s+RW}) = OFF$ if $MLIFE_1 \geq MLIFE_2$
Produce(*data*)	$\equiv$ ⟨definition in Table III⟩ $\wedge timer_A(\overline{s+1}) = OFF$ if $MLIFE_1 < MLIFE_2$ $\wedge(timer_A(\overline{s+RW}) = OFF \vee timer_A(\overline{s+RW}) > MLIFE_1$
SendD(*i*)	$\equiv$ ⟨definition in Table III⟩ $\wedge t_D(a+i)' = 0$
RecACK(*cn*, *n*)	$\equiv$ ⟨definition in Table III⟩

Entity 2 defined as in Table IV.

Channels 1 and 2 defined as in Table I.

relied on the fairness of *Produce*, an input event. Thus, the Marking is complete, and this construction is over.

6.5 Protocol II: Implementation with N Timers

In Table V, we provide an implementation in which both S_1 and S_3 are enforced by the N local timers in $timer_A$. Unlike in the previous implementation with $timer_D$, the enforcement of S_1 is not tight; that is, Entity 1 takes more than the minimum time to detect that S_1 holds.

Because $produced(n)$ is not sent after it is acknowledged, we have $t_D(n) \geq t_A(n)$ for all $n \in [0 \cdot \cdot a-1]$. The proof of this is trivial and, therefore, omitted. Thus, an alternative way to enforce S_1 is to enforce the following:

$$S_4 \equiv n = s - N - RW \geq 0 \Rightarrow t_A(n) > MAXLIFE_1$$

S_4 is analogous to S_3 and can be enforced by including $timer_A(\overline{s+RW}) > MLIFE_1$ in the enabling condition of *Produce*. We have to combine this with the other condition $timer_A(\overline{s+1}) > MLIFE_2$ needed to enforce S_3, as shown in Table V.

The progress requirement L_{12} holds for this protocol also, where *enabled*(*Produce*) denotes the enabling condition of *Produce* as defined in Table V. The Marking is complete as in Protocol I.

6.6 Protocol III: Implementation with One Timer

In this section we prove that S_3 and S_4 can be enforced by imposing a minimum time interval δ between successive occurrences of *Produce*. This time constraint is of interest for two reasons. First, it can be implemented with a single local timer at Entity 1. Second, it corresponds to specifying a maximum rate of data transmission, if we assume that *Produce* also transmits the accepted data block. (There is no loss of generality here; Entity 1 need merely save in another buffer data blocks that are produced and not yet sent.) Note that if δ is sufficiently small, for example, the hardware clock period, then there is no need for Entity 1 to use a local timer explicitly. This would correspond to the situation in TCP [29] and to the original Stenning's protocol [36].

Table VI. System Specification for Protocol III

Entity 1

produced, *s*, *a*, *sendbuff*, t_D, t_A defined as in Table IV.

t_S: sequence $(0 \cdot \cdot \infty)$ of ideal timer. Initially $t_S = OFF$.

$timer_S$: local timer of capacity $(1 + \epsilon_1)\delta$. Initially $timer_S = OFF$.

$$
\begin{aligned}
Produce(data) \quad &\equiv s - a \leq SW - 1 \wedge timer_S = OFF \\
&\wedge timer_S' = 0 \wedge t_S(s)' = 0 \\
&\wedge sendbuff' = sendbuff@\langle data \rangle \wedge s' = s + 1 \\
&\wedge produced' = produced@\langle data \rangle \\
SendD(i) \quad &\equiv \langle \text{definition in Table V} \rangle \\
RecACK(cn, n) \quad &\equiv Rec_2(ACK, cn, n) \\
&\wedge [\overline{cn - a} \in [1 \cdot \cdot s - a] \\
&\quad \rightarrow (a' = a + \overline{cn - a} \wedge sendbuff' = tail(sendbuff, \overline{cn - a}, \\
&\qquad \wedge [\forall\, i \in [a \cdot \cdot a' - 1]\colon t_A(i)' = 0])]
\end{aligned}
$$

Entity 2 defined as in Table IV.

Channels 1 and 2 defined as in Table I.

The protocol is specified in Table VI. At Entity 1 we have $timer_S$ and t_S. $t_S(n)$ indicates the ideal time elapsed since *produced*(n) was produced. $timer_S$ is a local timer that tracks $t_S(s - 1)$ up to δ ideal time units.

We will obtain the minimum value of δ that enforces S_3 and S_4. Consider an occurrence of *Produce* that increments s from s_0 to $s_0 + 1$. Both S_3 and S_4 are of the form:

$$V \equiv s_0 \geq K \Rightarrow t_A(s_0 - K) > D,$$

that is, s_0 is produced only if D time units have elapsed after $s_0 - K$ was acked. For notational convenience, we assume below that $s_0 \geq K$. Thus, V is enforced if the following holds for some n_0:

$$W \equiv t_A(s_0 - K) \geq t_S(s_0 - n_0) > D.$$

The first inequality in W says that $s_0 - K$ is acknowledged before $s_0 - n_0$ is produced, or, equivalently, m is produced only after $m + n_0 - K$ is acknowledged. It can be enforced by including $s - a \leq K - n_0 - 1$ in the enabling condition of *Produce*. To avoid getting a dead protocol, we require that $n_0 \in [1 \cdot \cdot K - 1]$.

The second inequality in W says that more than D time units have elapsed since the production of $s_0 - n_0$ till the present moment, which is just before the production of s_0. Because successive occurrences of *Produce* are separated by at least δ time units, we can enforce this by having $n_0\delta > D$.

Thus, V is enforced if for some $n_0 \in [1 \cdot \cdot K - 1]$ we include $s - a \leq K - n_0 - 1$ in the enabling condition of *Produce*, and $n_0\delta > D$ holds. For S_3, these expressions specialize to (1) $s - a \leq N - RW - n_0 - 1$ and (2) $n_0\delta > MAXLIFE_1$, for some $n_0 \in [1 \cdot \cdot N - RW - 1]$. For S_4, these expressions specialize to (3)$s - a \leq N - 1 - m_0 - 1$ and (4) $m_0\delta > MAXLIFE_2$, for some $m_0 \in [1 \cdot \cdot N - 2]$. We want a solution that minimizes δ. For any n_0, any

$m_0 \in [1 \cdot \cdot RW - n_0 - 1]$ satisfies (3), while $m_0 = RW - n_0 - 1$ yields the smallest value of δ. Thus, we want n_0 that satisfies (1), (2), and $(RW - n_0 - 1)\delta > MAXLIFE_2$. In the literature, an upper bound on $s - a$ is referred to as the *send window size*, denoted by SW. Rephrasing these conditions in terms of $SW = N - RW - n_0$, we obtain

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

$$\delta \ge \max\left[\frac{MAXLIFE_1}{N - RW - SW}, \frac{MAXLIFE_2}{N - 1 - SW}\right]$$

We require that $s - a \le SW - 1$ in the enabling condition of *Produce*, as shown in Table VI. The progress requirement L_{12} holds for this protocol also, where *enabled*(*Produce*) denotes the enabling condition of *Produce* as defined in Table VI. The Marking is complete, as in Protocol I.

For the typical case of $MAXLIFE_1 = MAXLIFE_2 = MAXLIFE$, the above constraint on δ simplifies to $\delta \ge MAXLIFE/(N - SW - RW)$. 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 $\delta \ge MAXLIFE/N$.

Stenning [36] considered the case of $MAXLIFE_1 = MAXLIFE_2 = MAXLIFE$ and obtained the bound $N \ge SW + \max(M + RW, SW)$, where $M = MAXLIFE/\delta$. We get $N \ge SW + RW + M$, which is a tighter bound. Stenning's protocol also has some unnecessary restrictions: (1) Whenever the producer retransmits a data block with sequence number i, it also resends every outstanding data block with a sequence number larger than i; and (2) whenever the consumer receives a data message, it must send an acknowledgment message.

7. REFINEMENT STEPS

We have presented a stepwise refinement heuristic that maintains, at any point in a construction, a topology, a state transition system, a set of requirements, and a Marking. At the start of the construction, the topology, state transition system, and requirements specify the desired properties of the distributed system; the Marking is empty. The construction proceeds by applications of system refinement steps and requirement refinement steps. Successful termination of the construction is indicated by a complete Marking.

In this section we first present some useful refinement steps and then make some general observations about the heuristic. For readability, we say "event e is specified by formula p to mean that *formula*$(e) \equiv p$, and we use *enabled*(p) to mean $[\exists$ *Variables'*: $p]$.

7.1 System Refinement Steps

These steps are used to increase the resolution of the state space by adding new state variables, adding new messages, and refining a message into a set of messages. They are used to change the set of state transitions by refining events and by adding new events. Each step ensures that the resulting state transition system is a conditional refinement of the previous system. In

Appendix A we establish the soundness of each step by proving that it preserves the consistency constraints of the Marking.

Addition of new state variables and new events. We can augment the state variables set *Variables* with new state variables *Newvars*, and *Initial* with new conjuncts that define initial conditions for state variables in *Newvars*. The Marking is preserved. We can introduce a new event e_i that updates only state variables in *Newvars*. The Marking is preserved. In addition, (A_j, e_i) can be marked for every A_j.

Refinement of events. Let an existing event e be specified by the formula p. Let q be another formula such that $A \wedge S(e) \wedge q \Rightarrow p$ holds; that is, q is a conditional refinement of p. We can change the specification of e to q. We say that e has been *refined to* q.

The Marking is preserved, except for the following two cases: (1) A progress requirement P *leads-to* Q marked with tag *via* e becomes unmarked iff $A \wedge S(e) \wedge P \Rightarrow enabled(q)$ does not hold; and (2) an event requirement S_j of e that is marked becomes unmarked iff $A \wedge enabled(q) \Rightarrow S_j$ does not hold.

One application of the above refinement step is to incorporate event requirements into enabling conditions of events. Let S_i be an event requirement of e. Let every free variable of S_i be a nonauxiliary state variable accessible by e. Let e be specified by formula p. Then we can refine e to $S_i \wedge p$. In this case, no event requirement of e becomes unmarked, and S_i can be marked. For example, if e is specified by $x > 0 \wedge x' = x - 1$ and S_i by $x = y$, we can refine e to $x = y \wedge x > 0 \wedge x' = x - 1$ and mark S_i.

Another application of the above refinement step is to make state variables auxiliary. For example, let e be specified by the formula $x \bmod 2 = 0 \wedge x' = x + 1$, let y be a state variable with domain $\{0, 1\}$, and let $y = x \bmod 2$ be an event requirement of e. We can refine e to $y = 0 \wedge x' = x + 1$. Note that x satisfies the constraints of an auxiliary variable in this new specification of e. By similarly refining every event that involves x, we can make x into an auxiliary variable.

Often, we simultaneously apply the two refinement steps described above. Consider the previous example where e is specified by the formula $x \bmod 2 = 0 \wedge x' = x + 1$. We can introduce a new state variable z with domain $\{0, 1\}$, a new invariant requirement $z = x \bmod 2$, and refine e to $z = 0 \wedge x' = x + 1 \wedge z' = 1$.

Introduction of new messages. To introduce a new message n to be sent along a channel k from entity a to entity b, we introduce $Send_k(n)$ into a new or an existing event e_i of entity a, and introduce $Rec_k(n)$ into a new or existing event e_j of entity b.[9]

We assume that every existing invariant requirement refers to $\mathbf{z}_k$ only in formulas that are not affected by adding n to the tail of $\mathbf{z}_k$ or by removing n

[9] If e_i (or e_j) is an existing event, we assume that it does not access channel k prior to the refinement.

from the head of $\mathbf{z}_k$, for example, a formula indicating the number of m's in $\mathbf{z}_k$, where $m \neq n$, or the formula $\mathbf{m}$ *subseq* $\mathbf{z}_k$ for some specified sequence $\mathbf{m}$ that does not contain n. This assumption was valid for the construction in Sections 4–6.[10]

The Marking is preserved, except for the following: A progress requirement marked with tag *via* e_j becomes unmarked.

Introduction of new message fields. Suppose we want to add a new field d to a message m that is sent along a channel k. In every existing event e_s that sends m, replace every appearance of $Send_k(m)$ with $Send_k((m, d))$, where d can be restricted to satisfy some relationship involving the state variables accessible to e_s. In every event e_r that receives m, replace every appearance of $Rec_k(m)$ with $[\exists\ d\colon Rec_k((m, d))]$. Alternatively, we can replace $Rec_k(m)$ with $Rec_k((m, d))$ and introduce d as a parameter of e_r.

Observe that this step changes the domain of $\mathbf{z}_k$. To evaluate an existing requirement, we do the following: Each state in the new domain of $\mathbf{z}_k$ is mapped to the state in the old domain, obtained by deleting all appearances of the message field d. For example, the state formula $m \in \mathbf{z}_k$ is interpreted as $[\exists\ d\colon (m, d) \in \mathbf{z}_k]$. Although this mapping is appropriate for network protocols, it may be inadequate in other situations; for example, with a requirement that refers to the number of *fields* in channel k. The Marking is preserved.

7.2 Requirement Refinement Steps

Requirement refinement steps strengthen or reorganize the set of requirements so than an existing currently unmarked requirement can be marked. The state transition system is not changed. We now describe some requirement refinement steps. These steps always preserve the Marking. A proof of their soundness may be found in Appendix A.

Reorganization of safety requirements. The following steps can be used to simplify the formula specifying a requirement A_i or S_i. Below, P, Q, and R are state formulas. We say "*R is equivalent to P given Q*" to mean that $Q \Rightarrow (R \Leftrightarrow P)$ holds.[11]

(1) Let S_i be an event requirement of event e and let A_j be an invariant requirement. Let S_i be specified by P, and A_j by Q. We can change the specification of S_i to any R that is equivalent to P given Q.

(2) Let S_i and S_j, $i \neq j$, be event requirements of event e such that S_j is marked if S_i is marked. Let S_i be specified by P, and S_j by Q. We can change the specification of S_i to any R that is equivalent to P given Q.

(3) Let A_i and A_j, $i \neq j$, be two invariant requirements such that, for every event e, (A_j, e) is marked if (A_i, e) is marked. Let A_i be specified by P,

[10] If this assumption is not valid for an A_l, then a marked (A_l, e_l) or (A_l, e_j) may become unmarked.

[11] Examples of such R include $R \equiv P$, $R \equiv Q \Rightarrow P$, $R \equiv Q \wedge P$, and $R \equiv X \Rightarrow Y \wedge Q$ where $P \equiv X \Rightarrow Y$.

and A_j by Q. We can change the specification of A_j to any R that is equivalent to P given Q.

Generation of event requirement from invariant requirement. Let (A_i, e) be unmarked. Obtain a weakest precondition P of A_i with respect to e. If $A \wedge S(e) \Rightarrow P$ does not hold, then introduce P as a new event requirement of e. Mark (A_i, e).

If the expression for a weakest precondition is unmanageable (and this depends on our ingenuity and patience [7]), then we can obtain either a sufficient precondition or a necessary precondition. In the latter case, (A_i, e) remains unmarked; this is still a useful step because it increases the set of requirements.

Because of parts (1) and (2) of the safety requirements reorganization step, we can replace the precondition P by $D \Rightarrow P$ where D is a predicate implied by $A \wedge S(e)$ before the step. Thus, we can define a weakest precondition P as equivalent to $[\forall\ Variables'\colon D \wedge formula(e) \Rightarrow A_i']$. We can define a sufficient precondition P as satisfying $[\forall\ Variables'\colon P \wedge D \wedge formula(e) \Rightarrow A_i']$. These definitions often yield much simpler expressions than those obtained from the original definitions of weakest and sufficient preconditions in Section 2.

Example. Let e be specified by $y \geq 1 \wedge x' = x + 1$, and A_i by $x \in \{0, 1\}$. Then $y \geq 1 \Rightarrow x \in \{-1, 0\}$ is a weakest precondition of A_i with respect to e. If it is not implied by $A \wedge S(e)$, then introduce $y \geq 1 \Rightarrow x \in \{-1, 0\}$ as a new event requirement, say, S_j, of e. Mark (A_i, e). By applying part (1) of the safety requirements reorganization step to S_j and A_i, we can change the specification of S_j to $y \geq 1 \Rightarrow x = 0$.

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

Generation of invariant requirements. An invariant requirement can be introduced to mark an event requirement S_i of event e. If $Initial \Rightarrow (enabled(e) \Rightarrow S_i)$ holds, we can introduce $enabled(e) \Rightarrow P$ as a new invariant requirement and mark S_i. For example, if S_i is specified by $y > 0$ and e by $x \geq 1 \wedge y' = y + 1$, we get $x \geq 1 \Rightarrow y > 0$ as the new invariant requirement, provided it satisfies the initial condition.

An invariant requirement can be introduced to mark a progress requirement by closure. Let L_i be specified by P *leads-to* Q, and L_j by $P \wedge R$ *leads-to* Q. Let L_j follow L_i in the ordering. If $Initial \Rightarrow R$ holds, we can introduce R as an invariant requirement and mark L_i with the tag *closure using* L_j.

An invariant or event requirement can be introduced in order to refine an event. Suppose we want to change the specification of an event e from formula p to formula q, but we cannot use the event refinement step because $A \wedge S(e) \wedge q \Rightarrow p$ does not hold. If R is a state formula such that $A \wedge S(e) \wedge R \wedge q \Rightarrow p$ holds, then we can introduce R as a new event requirement of e. Alternatively, if $Initial \Rightarrow R$ also holds, we can introduce R

as a new invariant requirement. In either case, q is now a conditional refinement of p.

As described at the beginning of Section 4.3, it is usual to transform an event requirement S_i of e into an invariant requirement when we decide that the enabling condition of e is not to be strengthened in later refinement steps.

Generation of progress requirements. Given an unmarked progress requirement L_i, we can introduce new progress requirements, say $L_n, \ldots, L_{n+m}$, so that L_i can be marked by closure using $L_n, \ldots, L_{n+m}$. For example, if L_i is specified by P *leads-to* Q, we can introduce L_n specified by P *leads-to* R and L_{n+1} specified by R *leads-to* Q, and mark L_i by closure using L_n, L_{n+1}.

Progress requirements can also be introduced to mark an existing progress requirement via a message set. Let L_i be specified by P *leads-to* Q, and let M be a set of messages such that (1) for every event $e_r(m)$ that receives $m \in M$, $P \wedge e_r(m) \Rightarrow Q'$, and (2) for every event $e \neq e_r(m)$, $P \wedge formula(e) \Rightarrow P' \vee Q'$. Then we can introduce a new progress requirement L_n specified by $P \wedge count(M) \geq k$ *leads-to* $Q \vee count(M) \geq k + 1$, and mark L_i via M using L_n.

When new progress requirements $L_n, \ldots, L_{n+m}$ are introduced to mark L_i, they are included in the ordering of progress requirements after L_i; this ensures that the new ordering is compatible with all existing tags of the progress marking.

The ordering can be changed to facilitate progress marking. For example, suppose the current ordering is L_1, L_2, L_3 and L_1 is marked using L_2. If L_2 implies L_3, then we could change the ordering to L_1, L_3, L_2, and mark both L_1 and L_3 using L_2.

7.3 General Observations

The construction is not guaranteed to terminate. If it does terminate, there are two cases. It terminates successfully when all requirements in the system specification are marked, and the state transition system satisfies all topology constraints of the distributed system. It terminates unsuccessfully whenever we have an event requirement S_i of an event e that is inconsistent with the invariant requirements or with the other event requirements of e; that is, $S_i \Rightarrow \neg A \vee \neg S(e)$ holds. The only way to mark such an S_i will be to remove the event e.

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 variable set *Variables*). New state variables should be introduced whenever it is determined that the generation of an invariant requirement will cause unsuccessful termination.

Observe that the state space can be refined either by adding new state variables or by changing the domains of existing state variables. The latter introduces an additional translation step in the evaluation of existing state formulas whereas the former does not. The difficulty of the translation is an important consideration when choosing between the former and the latter. In this paper we have changed the domains of channel state variables only, and that too is restricted to the addition of new messages and message fields. The translation needed to evaluate an old state formula for a new state is trivial: Simply ignore all new messages and new message fields in the new state.

Events in the initial specification that are intended to be under the control of users of the distributed system are called input events. The distributed system should not block input event executions that are allowed by the initial specification. However, this may be practically impossible for some initial specifications, in which case the initial specification must be weakened. For example, in our data transfer construction, we first required input event *Produce*(*data*) to be enabled in any state. Such a requirement can only be satisfied by having an unbounded-capacity buffer at Entity 1 and is, thus, not realistic. Therefore, we changed the initial specification and allowed *Produce*(*data*) to be disabled in certain states, but required that whenever it is disabled it eventually becomes enabled and remains enabled at least until its next occurrence.

A related issue is that of "inconsistent" initial specifications, which cannot be satisfied by a distributed system even with unbounded resources. For example, consider an initial specification with state variables x and y associated with different entities, and invariant requirement $x = y$ and progress requirement $x = n$ *leads-to* $x = n + 1$. No distributed system can satisfy both requirements, given the topology, because y cannot equal x just after x is increased. Another example is the progress requirement $x = n$ *leads-to* $x = n + 1$, where x indicates the number of occurrences of an input event. This cannot be satisfied without imposing some fairness requirement on the input event.

The Marking is an important feature of our heuristic. At any point in the construction, each entry in the Marking indicates a part of an inference rule that is satisfied by the state transition system and a requirement. Thus, the Marking indicates the extent to which it has been established that the system satisfies the requirements. Note that we do not require the Marking to contain every item that can be marked.

We allow great flexibility in how our Marking can change during a construction. A refinement step that introduces new events or invariant requirements can increase the Marking in absolute terms, but decrease it relative to the number of unmarked items. We also allow refinement steps that can decrease the Marking in absolute terms. We believe that the added flexibility is more desirable than maintaining a monotonically nondecreasing Marking. There is no preferred order to mark the three types of requirements. It might appear from our sliding window protocol construction that we prefer to mark invariant requirements first, then progress requirements, and then event requirements. This is not true. There was no particular

ordering when we first constructed the protocols; the ordering came later, in our attempt to shorten the presentation.

The Marking allows us to introduce arbitrary modifications to a limited extent. For example, given an event e specified by a formula p, we can change the specification of e to an *arbitrary* formula q, where q is not a conditional refinement of p; every marking involving e becomes unmarked. Similarly, we can introduce a new event e that updates existing state variables in arbitrary ways; (A_t, e) is unmarked for every A_t. When introducing arbitrary modifications, we must be careful not to modify updates to state variables that were used to specify the desired properties at the start of a construction. Otherwise, these state variables may not have the meaning intended when they were used to specify the desired properties. For example, in our data transfer protocol, suppose 2 is a particular value in $DATA$; we should not modify $produced(s)' = data$ to $produced(s)' = 2$.

In summary, we emphasize that this is a heuristic. At any point in a construction, the user of the heuristic is attempting to find a solution from the space of all state transition systems that are conditional refinements of the current state transition system and that satisfy the current set of requirements. Not only is this space typically large (due to unbounded domains of state variables, etc.), but it can grow from the addition of new state variables. Thus, successful termination is not guaranteed, even if a solution exists in this space. However, if a solution exists and its basic features are known to the user, then he or she can guide the heuristic to converge to the solution.

Machine assistance in the form of proof checkers and theorem provers would be very helpful, especially to a user who is not comfortable with predicate logic or program verification. We envision an interactive environment in which the user is responsible for introducing events, state variables, and some requirements, and for choosing which unmarked item to mark next. The user would rely on machine assistance to obtain a precondition for a specified (A_t, e) pair, to verify that a specified state formula P is a precondition, to verify that a modification to an event is a conditional refinement, to verify that a marking satisfies the consistency constraints, etc.

8. DISCUSSIONS AND RELATED WORK

In our heuristic we start a construction with an initial specification, say, I, consisting of a topology, a state transition system, and a set of safety and progress requirements, which together specify the desired behavior. If the construction terminates successfully, we are left with a distributed system, say, D, that is a refinement of the initial state transition system and that satisfies the safety and progress requirements. Specifically, D and I satisfy the relation D *offers* I defined in [19]. Various other authors have defined relations with similar meanings, such as D *simulates* I of Lynch and Tuttle [24], and D *satisfies* I of Lamport [21–23] and Hoare [13]. The informal meaning of each of these relations is that every externally visible behavior allowed by D is allowed by I. (There are some differences in how behaviors and visible behaviors are defined.)

Our approach of constructing a system as a refinement of another system is based on our earlier work on the use of projection mappings to relate state transition systems [17]. In particular, a state transition system A is a refinement of a state transition system B iff B is an "image" of A. The derivation of B as an image of A was considered in [17]; the method was applied [33] to verify a version of the High-level Data Link Control (HDLC) protocol standard with functions of connection management and full-duplex data transfer. The derivation of A as a refinement of B was considered in [18].

Our stepwise refinement heuristic is influenced by Dijkstra's work on the derivation of programs by using weakest preconditions [7], and by his development of distributed programs by incrementally adding invariants and actions to preserve the invariants [8–11].

There are differences and similarities between our approach and the approach of Chandy and Misra [5, 6] to derive distributed programs by stepwise refinement. In both approaches, a distributed system is modeled by a set of state variables and events. Invariant and progress requirements are maintained throughout a construction. In the approach of Chandy and Misra, most of the effort is spent on refining the set of requirements; the distributed program is not shown until very detailed requirements have been obtained. In our approach much effort is spent on refining the state transition system; the detailed requirements are derived in order to satisfy our various conditions for one state transition system to be a refinement of another state transition system.

In many of the examples of Chandy and Misra, the topology of a network of processes is refined by breaking up an event into several events, which are subsequently associated with different processes. This type of refinement step has also been used by Back and Kurki-Suonio [2, 3]. We have not found use for such a refinement step in our examples, which are from the area of communication protocols.

Our approach has several unique features: a Marking, a conditional refinement relation between systems, and event requirements. A Marking provides a useful representation of the extent to which we have established that the current system satisfies a given set of requirements. The conditional refinement relation gives us flexibility in generating new state transition systems, while keeping any decrease in the Marking to a minimum. Event requirements play an interesting role in our heuristic, in that they allow us to state safety requirements that (1) cannot be included in enabling conditions of events (e.g., due to topology, atomicity, or memory constraints), and (2) cannot yet be made into invariant requirements without causing unsuccessful termination.

Another special feature of our heuristic is the extensive use of auxiliary variables.[12] In the course of a construction, a state variable is made auxili-

[12] We use the term *auxiliary variable* in the sense of Owicki and Gries [28], that is, to record some history of system execution. Abadi and Lamport have extended this notion of auxiliary variables with "prophecy variables" [1]. A formal explanation of auxiliary variables as they are used in this paper can be found in [18].

ary if we find that it requires too much memory, or cannot be updated atomically, or violates topology constraints, etc. This technique can be used to refine the atomicity of system execution, that is, to replace an atomic update of a set of variables by a nonatomic succession of updates to the individual variables in the set [19]. It can also be used to refine the topology.

We find the relational notation to be very convenient for expressing event refinement and for reasoning about the effect of an action on invariant requirements. However, our construction heuristic does not require it; events can be specified by guarded multiple-assignment statements, as in [5].

In our approach we make no attempt to reason on a per-process basis (in our case, a process is either an entity or a channel). An assertion can involve variables of different entities and channels. However, this does not mean that the heuristic cannot be used in conjunction with a composition approach. In [19] and [20], we have extended this work to a layered system of modules separated by interfaces. An interface is specified by a state transition system and a set of safety and progress requirements. A module is specified by a state transition system and a set of fairness requirements. We proved the following composition theorem: If each module using its lower interface offers its upper interface (defined precisely in [19] and [20]), then the layered system offers its topmost interface. In the context of this composition theorem, this paper provides a heuristic to construct a module for given upper and lower interfaces. In particular, the initial specification corresponds to the upper interface, and the (unreliable) message-delivery channels correspond to the lower interface.

Development of our system model and construction heuristic was motivated by communication protocols (however, we believe that they are applicable to distributed systems in general). We applied the heuristic presented herein to construct complete transport protocols with functions of connection management and full-duplex data transfer [26, 27, 32]. A unique feature of these constructions is the composition of protocols constructed separately for the individual functions. We have also used the heuristic to obtain two database modules, using a two-phase locking protocol and a multiversion time-stamp protocol, that offer a serializable interface [19]. The two-phase locking module uses a lower interface to access a physical database.

Other authors have used communicating finite-state machines (CFSMs), Petri nets, programming language models [12, 15, 36], and temporal logic [12] for the specification and verification of communication protocols. The advantage of CFSM and Petri net models is that they can be automatically verified (e.g., [30]). Their disadvantage is that they cannot handle unbounded variables, such as sequence numbers and timers, without suffering a state space explosion; consequently, they cannot adequately model many real-life protocols. Programming language and temporal logic approaches, of which ours is a special case, have the power to model any protocol, but cannot be solved automatically. Real-time features can be incorporated in each of these approaches, just as we have added real-time to our model [33–35]. Other real-time models outside the communications protocols area include [4], [14], and [16].

APPENDIX A. Soundness of Refinement Steps

We need to show that each refinement step preserves $Initial \Rightarrow A$ and the consistency constraints of the Marking. We first consider $Initial \Rightarrow A$. This is preserved by the first system refinement step (addition of new state variables and new events) and the third requirement refinement step (generation of invariant requirements). The other steps do not affect *Initial* or A.

We now consider the consistency constraints of the Marking. For each item in the Marking, the corresponding constraint specifies that certain implications hold; for example, if (A_i, f) is marked then $A \wedge S(f) \wedge formula(f) \Rightarrow A'_i$ must hold. These implications are of four types: (1) $A \wedge formula(f) \Rightarrow S_n$, (2) $A \wedge S(f) \wedge formula(f) \Rightarrow A'_n$, (3) $X \wedge A \wedge A' \wedge S(f) \wedge formula(f) \Rightarrow Y'$, and (4) $X \wedge A \wedge S(f) \Rightarrow enabled(f)$. We show that each step preserves all of the implications of a marking that it does not explicitly unmark.

Addition of new state variables and new events. Because the state variables in *Newvars* are new, they do not appear in any requirement or existing event. Therefore, all existing implications of types 1–4 are preserved. Because a new event e only updates *Newvars*, all type 2's with $f = e$ hold. No other implications are affected.

Refinement of events. Types 1–4 with $f \neq e$ are not affected. Types 2 and 3 with $f = e$ are preserved because $A \wedge S(e) \wedge q \Rightarrow A \wedge S(e) \wedge p$ holds. Types 1 and 4 with $f = e$ are covered by the exceptions.

Introduction of new messages. Type 1 is not affected. Types 2 and 3 are preserved because no requirement is affected by adding n to the tail of $\mathbf{z}_k$ or by removing n from the head of $\mathbf{z}_k$. Type 4 is not affected for f other than e_i or e_j. Type 4 for $f = e_i$ preserved because $Send_k(n)$ is never blocked. Type 4 for $f = e_j$ is covered by the exception.

Introduction of new message fields. Similar to the previous step, except that type 4 for $f = e_j$ is preserved because the new e_j receives message (m, d) regardless of the value of d.

Reorganization of safety requirements. *Part* (1): Type 1 with $S_n \neq S_i$ is not affected. Type 1 with $S_n = S_i$ is preserved because $A \wedge formula(e) \Rightarrow R$ follows from $A \wedge formula(e) \Rightarrow P$, $A \Rightarrow Q$ (which holds because A_j is a conjunct of A), and $Q \wedge P \Rightarrow R$ (which holds because R is equivalent to P given Q). Types 2–4 with $f \neq e$ are not affected. Types 2–4 with $f = e$ are preserved as follows: Let T denote $S(e)$ without S_i. Thus, $S(e)$ equals $T \wedge P$ before the step, and $T \wedge R$ after the step. It suffices to show that $A \wedge T \wedge P$ follows from $A \wedge T \wedge R$. This holds because $A \wedge P$ follows from $A \wedge R$, $A \Rightarrow Q$ (which holds because A_j is a conjunct of A), and $Q \wedge R \Rightarrow P$ (which holds because R is equivalent to P given Q).

Part (2): Type 1 with $S_n \neq S_i$ is not affected. Type 1 with $S_n = S_i$ is preserved because $A \wedge formula(e) \Rightarrow R$ follows from $A \wedge formula(e) \Rightarrow P$, $A \wedge formula(e) \Rightarrow Q$ (which holds because S_j is marked if S_i is marked), and $Q \wedge P \Rightarrow R$ (which holds because R is equivalent to P given Q). Types 2–4

with $f \neq e$ are not affected. Types 2–4 with $f = e$ are preserved as follows: Let T denote $S(e)$ without S_i. Thus, $S(e)$ equals $T \wedge P$ before the step, and $T \wedge R$ after the step. It suffices that $T \wedge P$ follows from $T \wedge R, T \Rightarrow Q$ (which holds because S_j is a conjunct of $S(e)$ and $i \neq j$), and $Q \wedge R \Rightarrow P$ (which holds because R is equivalent to P given Q).

Part (3): Let T be A without A_i. Thus A equals $T \wedge P$ before the step, and $T \wedge R$ after the step. Types 1, 3, 4, and 2 with $A_n \neq A_i$ are preserved because $T \wedge P$ follows from $T \wedge R, T \Rightarrow Q$ (which holds because A_j is a conjunct of A and $i \neq j$) and $Q \wedge R \Rightarrow P$ (which holds because R is equivalent to P given Q). Type 2 with $A_n = A_i$ is preserved because $T \wedge R \wedge S(e) \wedge formula(e) \Rightarrow R'$ follows from $T \wedge P \wedge S(e) \wedge formula(e) \Rightarrow P'$, $T \wedge P \wedge S(e) \wedge formula(e) \Rightarrow Q'$ (which holds because (A_j, e) is marked whenever (A_i, e) is marked), and $Q' \wedge P' \Rightarrow R'$ (which holds because R is equivalent to P given Q).

Generation of event requirement from invariant requirement. Only type 2 with $A_n = A_i$ and $f = e$ is affected. It holds by definition because P is a sufficient precondition and is a conjunct of $S(e)$ after the step.

Generation of invariant requirements. Let T denote A before the step. After the step, A is $T \wedge A_k$. Type 1 with $S_n = S_i$ and $f = e$ holds because $A_k \wedge formula(e) \Rightarrow S_i$. Other type 1's preserved because T follows from $T \wedge A_k$. Types 2–4 preserved because $T \wedge S(f)$ is implied by $T \wedge A_k \wedge S(f)$ for every f.

Generation of progress requirements. Obvious from the definition of progress marking.

APPENDIX B Safety Marking for Protocol I

To complete the Marking of Protocol I, we need the following invariant requirements that relate $timer_D$ to t_D, and $timer_A$ to t_A:

$$
\begin{aligned}
D_0 \equiv\ & n \in [\max(0, s - N + RW) \cdot\cdot s - 1] \Rightarrow started\text{-}together(timer_D(\bar{n}), t_D(n)) \\
& \vee (timer_D(\bar{n}) = OFF \wedge t_D(n) > MAXLIFE_1) \\
& \vee (timer_D(\bar{n}) = OFF \wedge t_D(n) = OFF \\
D_1 \equiv\ & n \in [s \cdot\cdot \max(s + RW - 1, N - 1)] \Rightarrow timer_D(\bar{n}) = OFF \\
D_2 \equiv\ & n \in [s \cdot\cdot \infty] \Rightarrow t_D(n) = OFF \\
D_3 \equiv\ & n \in [\max(0, s - N + 1) \cdot\cdot a - 1] \Rightarrow started\text{-}together(timer_A(\bar{n}), t_A(n)) \\
& \vee (timer_A(\bar{n}) = OFF \wedge t_A(n) > MAXLIFE_2) \\
D_4 \equiv\ & n \in [a \cdot\cdot \max(s, N - 1)] \Rightarrow timer_A(\bar{n}) = OFF.
\end{aligned}
$$

The Marking can be completed as follows, where *Lte* is the *Local time event* for Entity 1:

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*LRD*	*Ite*	*Lte*
A_{0-9}, C_{0-6}	*	*	*	*	*	*	*	*	*na*
D_0	D_{0-2}	A_8, D_0	*na*	*na*	*na*	*na*	*na*	D_0	D_0
D_1	D_1	D_1	*na*	*na*	*na*	*na*	*na*	*na*	D_1
D_2	D_2	D_2	*na*	*na*	*na*	*na*	*na*	D_2	*na*
D_3	D_3	*na*	$A_8, C_5, D_{3,4}$	*na*	*na*	*na*	*na*	D_3	D_3
D_4	D_4	*na*	D_4	*na*	*na*	*na*	*na*	*na*	D_4

S_0 marked using $S_1, C_{2,1}, TA_1$	S_1 marked using D_0	S_2 marked using S_3, A_8, C_{4-6}, TA_2	S_3 marked using D_3

We have used $\forall\ i, j \in [\max(0, s - N + RW) \cdot \cdot s - 1]$: $(i = j$ iff $\bar{i} = \bar{j})$ in marking D_{0-2}, and $\forall\ i, j \in [\max(0, s - N + 1) \cdot \cdot a - 1]$: $(i = j$ iff $\bar{i} = \bar{j})$ in marking D_{3-4}.

ACKNOWLEDGMENT

This paper has benefited greatly from the constructive criticisms and diligence of the anonymous referees.

REFERENCES

1. ABADI, M., AND LAMPORT, L. The existence of refinement mappings. Tech. Rep. 29, Digital Systems Research Center, Palo Alto, Calif., Aug. 1988.
2. BACK, R. J. R., AND KURKI-SUONIO, R. Decentralization of process nets with a centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGCOPS Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). ACM, New York, 1983, pp. 131–142.
3. BACK, R. J. R., AND KURKI-SUONIO, R. Distributed cooperation with action systems. *ACM Trans. Program Lang. Syst. 10*, 4 (Oct. 1988), 513–554.
4. BERNSTEIN, A., AND HARTER, P. Proving real-time properties of programs with temporal logic. In *Proceedings of the 8th ACM SIGCOPS*, ACM, New York, Dec. 1981, pp. 1–11.
5. CHANDY, K. M., AND MISRA, J. An example of stepwise refinement of distributed programs: Quiescence detection. *ACM Trans. Program Lang. Syst. 8*, 3 (July 1986), 326–343.
6. CHANDY, K. M., AND MISRA, J. *A Foundation of Parallel Program Design.* Addison-Wesley, Reading, Mass , 1988.
7. DIJKSTRA, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976
8. DIJKSTRA, E. W. The distribution snapshot of K. M. Chandy and L. Lamport. Tech. Rep. EWD-864, Univ. of Texas at Austin, Nov. 1983.
9. DIJKSTRA, E. W. Derivation of a termination detection algorithm for distributed computations. Tech. Rep. EWD-840, Univ. of Texas at Austin.
10. DIJKSTRA, E. W., AND SCHOLTEN C. S. Termination detection for diffusing computations. *Inf. Process Lett. 11*, 1 (Aug. 1980), 1–4.
11. DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., AND SCHOLTEN, C. S. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM 21*, 11, (Nov. 1978), 966–975.
12. HAILPERN, B. T., AND OWICKI, S. S Modular verification of computer communication protocols. *IEEE Trans. Commun. COM-31*, 1, (Jan. 1983), 56–68.
13. HOARE, C. A. R. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, N.J., 1985.
14. JAHANIAN, F., AND MOK, A. Safety analysis of timing properties in real-time systems. *IEEE Trans. Softw. Eng. SE-12, 9* (Sept. 1986), 890–904.
15. KNUTH, D. E. Verification of link-level protocols. *BIT 21* (1981), 31–36.
16. KOYMANS, R., SHYAMSUNDER, R. K., DE ROEVER, W. P., GERTH, R., AND ARUN-KUMAR, S. *Compositional Semantics for Real-Time Distributed Computing.* Lecture Notes in Computer Science, vol. 193. Springer-Verlag, New York, June, 1985, pp. 167–187.
17. LAM, S. S., AND SHANKAR, A U. Protocol verification via projections. *IEEE Trans. Softw. Eng. SE-10*, 4 (July 1984), 325–342.
18. LAM, S S., AND SHANKAR, A. U. A relational notation for state transition systems. *IEEE Trans. Softw. Eng. 16*, 7 (July 1990), 755–775 (An abbreviated version, entitled "Refinement and projection of relational specifications," appears in *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems* (Mook, The Netherlands). Lecture Notes in Computer Science, vol. 430. Springer-Verlag, New York, May/June 1989).
19. LAM, S. S., AND SHANKAR, A. U. Specifying modules to satisfy interfaces: A state transition system approach In *Distributed Computing*, Springer-Verlag. To be published. (Also Tech.

Rep. CS-TR-88-60.3, Dept. of Computer Science, Univ. of Maryland, College Park, Aug. 1988.)

20. LAM, S. S., AND SHANKAR, A. U. A composition theorem for layered systems. In *Proceedings of the 11th International Symposium on Protocol Specification, Testing, and Verification* (Stockholm, June 17-20), IFIP, 1991.
21. LAMPORT, L. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst. 5*, 2 (Apr. 1983), 190-222.
22. LAMPORT, L. What it means for a concurrent program to satisfy a specification: Why no one has specified priority. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages* (New Orleans, Jan. 1985). ACM, New York, 1985.
23. LAMPORT, L. A simple approach to specifying concurrent systems. *Commun. ACM 32*, 1 (Jan. 1989).
24. LYNCH, N. A., AND TUTTLE, M. R. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Aug. 1987). ACM, New York, 1987.
25. MANNA, Z., AND PNUELI, A. Adequate proof principles for invariance and liveness properties of concurrent programs. *Sci. Comput. Program. 4* (1984).
26. MURPHY, S. L. Service specification and protocol construction for a layered architecture. Ph.D. dissertation, Dept. of Computer Science, Univ. of Maryland, College Park, May 1990. (Also available as Tech. Rep. CS-TR 2583 (or UMIACS-TR-91-3), Computer Science Dept., Univ. of Maryland, College Park, Jan. 1991.)
27. MURPHY, S. L., AND SHANKAR, A. U. Connection management for the transport layer: Service specification and protocol verification. Tech. Rep. CS-TR-2051.1 (or UMIACS-TR-88-45.1), Computer Science Dept., Univ. of Maryland, College Park, June 1988. (Shortened version to appear in *IEEE Trans. Commun.* A preliminary abbreviated version, entitled "A verified connection management protocol for the transport layer," appeared in *Proceedings of the ACM SIGCOMM 87 Workshop* (Stowe, Vt., Aug. 1987). ACM, New York, 1987.)
28. OWICKI, S., AND GRIES, D. An axiomatic proof technique for parallel programs I. *Acta Inf. 6*, (1976), 319-340.
29. POSTEL, J., ED. Transmission control protocol: DARPA internet program protocol specification. RFC 793, Network Information Center, SRI International, 1981.
30. SABNANI, K. An algorithmic procedure for protocol verification. *IEEE Trans. Commun. 36*, 8 (Aug. 1988).
31. SHANKAR, A. U. Verified data transfer protocols with variable flow control. *ACM Trans. Comput. Syst. 7*, 3 (Aug. 1989). (An abbreviated version entitled "A verified sliding window protocol with variable flow control" appeared in *Proceedings of the ACM SIGCOMM 86 Symposium* (Stowe, Vt., Aug. 1986). ACM, New York, 1986.)
32. SHANKAR, A. U. Modular design principles for protocols with an application to the transport layer. *Proc. IEEE* (Dec. 1991), (Also available as Tech. Rep. CS-TR-2510.1, Computer Science Dept., Univ. of Maryland, College Park, July 1990.)
33. SHANKAR, A. U., AND LAM, S. S. An HDLC protocol specification and its verification using image protocols. *ACM Trans. Comput. Syst. 1*, 4 (Nov. 1983), 331-368.
34. SHANKAR, A. U., AND LAM, S. S. Time-dependent communication protocols. In *Principles of Communication and Networking Protocols*, S. S. Lam, Ed., IEEE Computer Society, New York, 1984.
35. SHANKAR, A. U., AND LAM, S. S. Time-dependent distributed systems: proving safety, liveness and real-time properties. *Distrib. Comput 2*, 2 (1987), 61-79.
36. STENNING, N. V. A data transfer protocol. *Comput. Networks 1* (Sept. 1976), 99-110.

Received 1987; revised April 1989, November 1990, and July 1991; accepted August 1991