A Little Knowledge Goes a Long Way: Knowledge-based derivations and correctness proofs for a family of protocols[*]

Joseph Y. Halpern
IBM Almaden Research Center
halpern@almvma (BITNET); halpern@ibm.com (ARPA/CSNET)

Lenore D. Zuck[†]
Department of Computer Science, Yale University
zuck@yalecs (BITNET); zuck@yale (ARPA)

Abstract: We use a high-level, knowledge-based approach for deriving a family of protocols for the *sequence transmission* problem. The protocols of Aho, Ullman, and Yannakakis [AUY79, AUWY82], the Alternating Bit protocol [BSW69], and Stenning's protocol [Ste76] are all instances of one knowledge-based protocol that we derive. Our derivation leads to transparent and uniform correctness proofs for all these protocols.

[*]This paper is essentially identical to one that appears in *Journal of the ACM* **39**:3, 1992, pp. 449–478. A preliminary version appears in *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, 1987, pp. 269–280.

[†]This author was supported in part by NSF grant DCR-8405478

1 Introduction

Designing and proving the correctness of protocols in distributed systems is a notoriously difficult problem. The potential for faulty behavior makes the problem even more difficult. Subtle bugs are often found in seemingly correct protocols (see, for example, [GS80, SH86]). Consequently, researchers have looked for good tools to analyze distributed systems. Temporal logic [OL82, Pnu77], the state machine approach [BG77, BS80, Mer76, Sun79], Floyd-Hoare-style methods [HO83], model checking [CES86, QS82], and interval logic [SMS82] have all been advocated, and indeed, have been used successfully to verify a number of distributed protocols.

While the proofs in the cited papers do indeed demonstrate correctness, they do not usually help the reader to understand *why* the protocol is correct. Reading the step-by-step details of these proofs, one loses the global picture of what is happening in the protocol. It is not obvious which of the protocol's features are important and what is the role of each of the steps in the protocol. This understanding is crucial if we want to redesign the protocol so that it still works correctly in a slightly different environment or if we want to optimize the protocol in some way. Ideally, the design and verification of a protocol would be closely related and one could straightforwardly derive correctness proofs from the design methodology. In practice, design and verification are often done separately, in fact by different groups of people.

It has been suggested [HM90] that a useful way to analyze distributed systems is in terms of knowledge and how communication changes the processes' state of knowledge. The role of knowledge in distributed systems has by now been extensively studied [CM86, DM90, FHV92, FI86, Had87, Hal87, HF85, HV86, LR86, MT88, NT93, PR85] (see [Hal87] for an overview). In this paper, we use reasoning about knowledge to help design and verify a family of protocols that deal with a standard problem of data communication that we call the *sequence transmission* problem.

The problem is easily stated: Consider two processes, called the *sender* and the *receiver*. The sender S has an input tape with an infinite sequence $X = \langle x_0, x_1, \ldots \rangle$ of data elements. S reads these data elements and tries to transmit them to the receiver R. R must write these data elements onto an output tape. We require that (a) at any time the sequence of data elements written by R is a prefix of X (this is the *safety* property) and (b) if the communication medium satisfies appropriate *fairness* conditions, then every data element x_i in the sequence X is eventually written by R (this is the *liveness* property).

The sequence transmission problem clearly has a trivial solution if we assume that messages sent by S cannot be lost, corrupted, duplicated, or reordered. S simply sends $x_0, x_1, \ldots$ in order, and R writes them out as it receives them. However, once we consider a faulty communication medium, the problem becomes far more complicated. A number of different communication models have been extensively studied in the literature. For example, [AUWY82, AUY79] describe protocols that solve the problem in the case of a completely synchronous communication channel which allows only one-bit messages (we call this communication model the *AUY model*). They consider various types of faulty behavior, including message deletion and corruption. The famous *Alternating Bit protocol* [BSW69] is a solution to the sequence transmission problem for an asynchronous channel where messages cannot be reordered or duplicated, but may be lost or (detectably) corrupted. *Stenning's protocol* [Ste76] deals with the problem in the case where messages may be duplicated, lost, (detectably) corrupted, or reordered.

The solutions to the sequence transmission problem that appear in the literature were all designed individually, on an *ad hoc* basis. We attempt to provide a more uniform framework here. We start with a *knowledge-based* protocol that we prove solves the sequence transmission problem, where a knowledge-based protocol is one with explicit tests for knowledge [HF85, HF89]. A knowledge-based protocol can be viewed as a program written in a high-level programming language. Once we have a knowledge-based protocol and have verified that it solves a given problem, we still want to find a protocol—preferably a finite-state protocol, since it is more easily implementable and potentially mechanically verifiable [CES86, QS82]—that solves the problem and does not have explicit tests for knowledge. Although we do not have a general methodology for going from a knowledge-based protocol to a standard protocol, we show that derivations of standard protocols from the knowledge-based protocols for the sequence transmission problem that we provide are quite straightforward. Our expectation is that this will be the case for many other protocols of interest.

In order to prove the correctness of a standard protocol given the correctness of the knowledge-based protocol, we show that the standard protocol is an *implementation* of the knowledge-based protocol, where protocol P is an implementation of protocol P' if there is a mapping from the runs of P to the runs of P' that preserves relevant properties (in this case, what elements are read and written) (cf. [HF89]). We can extend this idea to get a sequence of standard protocols, each of which is an implementation of the previous one, thereby getting a top-down proof of correctness of the final protocol. This approach is in the same spirit as that advocated in [Gaf86, LT89], although these papers do not start with knowledge-based protocols at the top level. We are not the first to use knowledge-based protocols in this way; Moses and Tuttle [MT88] also start with a knowledge-based protocol (for Byzantine agreement) and derive efficient implementations of it.

We believe that the knowledge-based viewpoint gives us a unifying framework for understanding, verifying, and designing protocols. For example, we show that all the protocols for the sequence transmission problem that have appeared in the literature can be viewed as straightforward implementations of one high-level knowledge-based protocol. Of course, this allows us to give a uniform treatment of correctness. By using the idea of implementations, we are able to give a particularly transparent proof of correctness for the protocols of Aho, Ullman, and Yannakakis [AUY79, AUWY82]. We urge the reader to compare our proofs with those found in [AUY79, AUWY82, Gou85, Hai85].

It is interesting that the idea of thinking about such protocols in terms of knowledge appears in an informal way quite early in the literature. For example, in [BG77] it says that "Verification ...will correspond ...to finding out whether and in which circumstances the sender subsystem (and its user) can 'know' that all data obtained from the user have been received correctly and in sequence to the user in the receiver subsystem." We view this paper as providing a formalization of these intuitions.

The rest of the paper is organized as follows: In the next section we describe our formal model of distributed systems. We also discuss formally the notion of knowledge in distributed systems and knowledge-based protocols. In Section 3 we present a knowledge-based protocol for the sequence transmission model, and use it to derive standard protocols for the problem. The correctness of the protocols we present is proved in Section 4. In Section 5 we extend our results to the AUY model. The knowledge-based approach, by cutting out extraneous implementation details, may also facilitate the process of finding different solutions to a problem. We demonstrate this process in the case of the sequence transmission problem in Section 6. We conclude in Section 7 with further discussion of the knowledge-based viewpoint and some directions for further research.

2 The formal model

A detailed description of the model we use can be found in [HF89], so we only sketch the necessary details here, and refer the reader to [HF89] for further motivation and examples.

We assume that our processes are state machines and that the relevant features of a system at a given time are described by the *global state* of the system, where a global state is a tuple describing the local state of each of the processes and the state of the *environment.* We take the environment to consist of everything in the system that is relevant to the analysis that is not part of the state of the processes. (Exactly what is relevant will of course depend on the particular system being analyzed.) For simplicity, we assume that time ranges over the non-negative integers; the definitions can easily be extended to other time models. A *run* (or *execution*) of the system is defined to be a function from the non-negative integers to global states. Intuitively, a run is a description of the relevant features of the system over time. We occasionally refer to a pair (r, m) consisting of a run r and a time m as a *point.* As has been done in numerous previous papers (e.g., [HF85, HM90, Mos86, PR85]), we identify a distributed system with a set $\mathcal{R}$ of runs. We say (r, m) is a point in system $\mathcal{R}$ if $r \in \mathcal{R}$.

For protocols solving the sequence transmission problem, the processes are S and R. Thus a global state s is a tuple of the form (s_e, s_S, s_R), where s_e is the environment state, s_S is S's local state, and s_R is R's local state. The details of the states will depend on how we choose to analyze the system; we will discuss this in more detail when we formally analyze the protocols presented in the sequel. We denote j's local state in the global state $r(m)$ by $r_j(m)$, for $j \in \{S, R\}$.

We define a *protocol for process j* to be a (possibly nondeterministic or probablistic) function from j's local states to *actions.* Thus a process' protocol describes what actions the process takes as a function of its local state. Usually we think of these actions as coming from a small set of basic actions, such as reading a data element, writing a value, sending a message, or receiving a message. We find it useful to think of the environment as also running a protocol. In the introduction we discussed assumptions on the communication model such as "messages cannot be reordered or duplicated, but may be lost or (detectably) corrupted." Such assumptions, which implicitly describe the environment's behavior, can be captured by the environment's protocol. We take a *joint protocol P* to consist of protocols P_e, P_S, P_R for e, S, and R respectively. (We remark that for all the cases we consider in this paper, both P_S and P_R are deterministic, while P_e is nondeterministic or probabilistic.)

We would like to associate with every (joint) protocol a particular set of runs. To do this, we first must specify the possible local states for each of e, S, and R. Call these sets of states L_e, L_S, and L_R respectively. Let $\mathcal{G} = L_e \times L_S \times L_R$ be the set of possible global states. (Not all the global states in $\mathcal{G}$ will necessarily be reachable when we run the protocol.) The next step is to specify the subset $\mathcal{G}_0$ of $\mathcal{G}$ which consists of the possible initial global states. Finally, we must specify how the actions performed by e, S, and R change the global state. Let Act_e,

Act_S, and Act_R be the actions performed in P_e, P_S, and P_R respectively. A *transition function* τ associates with every *joint action* $(a_e, a_S, a_R) \in Act_e \times Act_S \times Act_R$ a global state transformer $\tau(a_e, a_S, a_R)$, i.e., a deterministic function from $\mathcal{G}$ to $\mathcal{G}$. (We could allow $\tau(a_e, a_S, a_R)$ to be nondeterministic, but we do not need this level of generality in this paper.) Thus we can think of $\tau(a_e, a_S, a_R)$ as describing the effect of simultaneously having e perform action a_e, S perform a_S, and R perform a_R.

Fix $\mathcal{G}$, $\mathcal{G}_0$ ($\mathcal{G}_0 \subseteq \mathcal{G}$), and τ as above. We say that a run r is *consistent with protocol P (with respect to $\mathcal{G}$, $\mathcal{G}_0$, and τ)* if

1. $r(0) \in \mathcal{G}_0$ (so $r(0)$ is a legal initial state).
2. For all $m \geq 0$, if $r(m) = (s_e, s_S, s_R)$, then there is a joint action $(a_e, a_S, a_R) \in P_e(s_e) \times P_S(s_S) \times P_R(s_R)$ such that $r(m+1) = \tau(a_e, a_S, a_R)(r(m))$ (so $r(m+1)$ is the result of transforming $r(m)$ by a joint action that could have been performed from $r(m)$ according to P).

We use $\mathcal{R}(P)$ to denote the set of all runs consistent with P (with respect to $\mathcal{G}$, $\mathcal{G}_0$, and τ).

Besides the type of protocol defined above (which we occasionally call a *standard* protocol), we will also be interested in a more high-level notion called a *knowledge-based protocol* ([HF85]), where we allow explicit tests for knowledge. To define such protocols formally, we find it convenient to assume that there is some set Φ of *basic facts* about the system. The set Φ can include facts of the type "$x_6 = 0$", "R sent m to S", etc. We define an *interpreted system $\mathcal{I}$* to be a pair $(\mathcal{R}, \pi)$ consisting of a system $\mathcal{R}$ and an assignment π of truth values to the basic facts for each point in $\mathcal{R}$, so that for every $p \in \Phi$ and point (r, m) in $\mathcal{R}$, we have $\pi(r, m)(p) \in \{\mathbf{true}, \mathbf{false}\}$. We say that the point (r, m) is in the interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ if $r \in \mathcal{R}$.

Given an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ and a point (r, m) in $\mathcal{I}$, we define a satisfiability relation $\models$ between the tuple $(\mathcal{I}, r, m)$ and a formula φ. For a basic fact $p \in \Phi$, we have

$$(\mathcal{I}, r, m) \models p \text{ iff } \pi(r, m)(p) = \mathbf{true}.$$

We extend the $\models$ relation to conjunctions and negations in the obvious way:

$$\begin{array}{rll} (\mathcal{I}, r, m) \models \neg\varphi & \text{iff} & (\mathcal{I}, r, m) \not\models \varphi \\ (\mathcal{I}, r, m) \models \varphi \wedge \psi & \text{iff} & (\mathcal{I}, r, m) \models \varphi \text{ and } (\mathcal{I}, r, m) \models \psi. \end{array}$$

We want to extend our language to allow formulas of the form $K_j\varphi$, which is read "process j *knows* φ". We ascribe knowledge to processes in a distributed system using ideas first discussed in [HM90], and later amplified in numerous other papers (see [Hal87] for an overview and references). Again, we state our definitions under the assumption that the only processes in the system are S and R, although they clearly extend to the case with an arbitrary (but fixed) set of processes.

Given two global states $s = (s_e, s_S, s_R)$ and $s' = (s'_e, s'_S, s'_R)$, we say s and s' are *indistinguishable to process j* (where j is either S or R) if j has the same state in both s and s', i.e., if $s_j = s'_j$. We say two points (r, m) and (r', m') are *indistinguishable to j*, and write

$(r,m) \sim_j (r',m')$, if the global states $r(m)$ and $r'(m')$ are indistinguishable to j. We then define

$$(\mathcal{I},r,m) \models K_j\varphi \quad \text{iff} \quad (\mathcal{I},r',m') \models \varphi \text{ for all } r' \text{ and } m' \text{ such that } (r,m) \sim_j (r',m').$$

This definition is designed to capture the intuition that processor j *knows* φ at $r(m)$ if φ is true at time m' in run r' for all points (r',m') indistinguishable to j from (r,m).

An important property of this definition of knowledge is that $K_j\varphi$ implies φ; i.e., if an agent knows φ, then φ is true. Thus, if $(\mathcal{I},r,m) \models K_j\varphi$, then $(\mathcal{I},r,m) \models \varphi$; this easily follows from the observation that $(r,m) \sim_j (r,m)$.

A knowledge-based protocol allows explicit tests for knowledge. Unlike the tests that appear in a standard protocol, the truth value of the test in a conditional statement of the form "**if** $K_S\varphi$ **then** ..." cannot be determined by looking at the local state in isolation. Its truth depends on the truth of φ at other points (all the ones with global states that S cannot distinguish from its current global state). Thus, whereas a protocol for S is a function from S's local states to actions, a knowledge-based protocol for S is is a function from a pair consisting of a local state for S *and an interpreted system* to actions. For example, suppose that in local state ℓ process S is at the step in P_S with an instruction of the form "**if** $K_S\varphi$ **then** send m **else** send m'". The action performed by S in state ℓ is "send m" if S knows φ, and "send m'" otherwise. Thus we have

$$P_S(\ell,\mathcal{I}) = \begin{cases} \text{send } m & \text{if } (\mathcal{I},r,m) \models \varphi \quad \text{for all points } (r,m) \text{ where } r_S(m) = \ell \\ \text{send } m' & \text{otherwise.} \end{cases}$$

Note that the only difference between the formal definition of knowledge-based protocols and standard protocols is that a knowledge-based protocol takes an interpreted system as one of its arguments. Of course, a standard protocol can be viewed as a special case of a knowledge-based protocol where the function is independent of the interpreted system.

Fix a set $\mathcal{G}$ of global states, a subset $\mathcal{G}_0 \subseteq \mathcal{G}$ of initial states, a transition function τ on G, and an interpreted system $\mathcal{I}$ with global states in $\mathcal{G}$. We define a run r to be *consistent with (knowledge-based) protocol P relative to $\mathcal{I}$* (with respect to $\mathcal{G}$, $\mathcal{G}_0$, and τ) just as we defined the notion of a run being consistent with a standard protocol P, except that now the joint action (a_e,a_S,a_R) in clause 2 is in $P_e(s_e,\mathcal{I}) \times P_S(s_S,\mathcal{I}) \times P_R(s_R,\mathcal{I})$ rather than $P_e(s_e) \times P_S(s_S) \times P_R(s_R)$.

An interpreted system $\mathcal{I} = (\mathcal{R},\pi)$ is *consistent* with a knowledge-based protocol P (with respect to $\mathcal{G}$, $\mathcal{G}_0$, and τ) if every run $r \in \mathcal{R}$ is consistent with P relative to $\mathcal{I}$. In general, there is not a unique interpreted system that is consistent with a given protocol. Thus, in order to prove that a knowledge-based protocol P has certain properties (for example, that is is satisfies certain specifications), we typically show that the required properties hold for *all* interpreted systems $\mathcal{I}$ consistent with P.

3 A knowledge-based protocol and implementations of it

In this section we present a knowledge-based protocol for the sequence transmission problem and show how it can be implemented by standard protocols. As we show below, Stenning's protocol and the Alternating Bit protocol are in fact straightforward implementations of this knowledge-based protocol.

Intuitively, the knowledge-based protocol is quite simple. S reads the i^{th} data element, and repeatedly sends it to R until S knows that R has received it and that R knows that it is the i^{th} element. At that point, S reads the $(i+1)^{\text{st}}$ element, and so on. R writes the data elements as it "learns" about them, and tells S which values it has learnt about so far. We present an informal description of a knowledge-based protocol A that captures these intuitions in Figure 1 below. For simplicity, we assume here and throughout this paper that the input sequence consists only of 0s and 1s, although we could easily extend our techniques to deal with any finite data domain.

The INIT statement that begins both S's protocol and R's protocol describes how certain variables are to be initialized. In particular, S initializes a counter i to 0, and R initializes a counter j to 0; we thus implicitly assume that i is part of S's local state, and j is part of R's local state. In the description of the protocol, we take $K_R(x_k)$ as an abbreviation for $K_R(x_k = 0) \vee K_R(x_k = 1)$. Thus, if $K_R(x_k)$ holds, then R knows the value of x_k. Recall that S sends the i^{th} data element until it knows that R knows the value of this data element. Intuitively, this suggests that S should test if $K_S K_R(x_i)$ holds; if it does not, then S should continue to send the i^{th} data element; otherwise S can increment i and read the next data element. However, there is a subtle problem with this intuition. Roughly speaking, it arises because we cannot substitute equals for equals inside the scope of a K operator. For example, suppose we are at a point where $i = 3$. What S really wants to do is to continue sending the value of x_3 until $K_S K_R(x_3)$ holds. This is not the same as sending it the value of x_3 until $K_S K_R(x_i)$ holds. Put another way, $(i = 3) \wedge K_S K_R(x_i)$ is not equivalent to $(i = 3) \wedge K_S K_R(x_3)$. The problem is that R may know the value of x_3 without knowing the value of x_i (or, for that matter, without even knowing the value of i), since the variable i may take on different values in the global states that R considers possible. In a state where $i = 3$, we want S to continue sending x_i until $K_S K_R(x_3)$ holds, not until $K_S K_R(x_i)$ holds. In order to achieve this effect, we define $K_R(x_{@i})$ to be an abbreviation for $K_R(x_k)$, where k is the value of i in the current state. We could similarly define $K_R(x_{@j})$ as an abbreviation for $K_R(x_\ell)$, where ℓ is the value of j in the current state, but there is no need. Since j is part of R's local state, it is easy to see that $K_R(x_{@j})$ and $K_R(x_j)$ are equivalent.[1]

We take $\neg\mathtt{K}_\mathtt{R}(\mathtt{x}_i)$ in the description of the protocol to be some string of symbols (distinct for each value of i). The effect of "receive z" is that S checks its message buffer for the message delivered on that round, and stores that message in the z register. If no message is delivered, then it stores λ in its register. The effect of "receive z'" is similar. Note that in the protocol S sends the data element x_i only if necessary; that is, if $\neg K_S K_R(x_{@i})$ holds. If it is common knowledge that the first 10 elements of every sequence are 0s, then S would not need to send these elements; R could write them without receiving any message from S.

We would like to show that the knowledge-based protocol A is correct in a wide variety of

[1]The fact that we needed to extend the logic by adding $K_R(x_{@i})$ indicates a nontrivial lack of expressive power in what is an essentially propositional logic of knowledge in terms of dealing with scoping. For those readers familiar with first-order modal logics, note that using first-order constructs, we can easily express $K_R(x_{@i})$ as $\forall k(k = i \Rightarrow K_R(x_k))$. Since we did not want the overhead of a lengthy discussion of first-order modal logics, we used the approach presented above. For further discussion of the subtleties of scoping and how this affects the expressive power of modal logic, the interested reader is encouraged to consult [Fit91, GH91].

S's protocol:

```
INIT: $i := 0$; read $y$
do forever
    if $\neg K_S K_R(x_{@i})$
        then send $\langle i, y \rangle$; receive $z$
        else $i := i + 1$; read $y$; receive $z$
end
```

R's protocol:

```
INIT: $j := 0$
do forever
    if $\neg K_R(x_j)$
        then send $\neg \mathtt{K_R}(\mathtt{x}_j)$; receive $z'$
        else if $K_R(x_j = 0)$
                then write 0; $j := j + 1$; receive $z'$
                else write 1; $j := j + 1$; receive $z'$
end
```

Figure 1: Protocol A

settings, precisely because it abstracts away the details of how a state of knowledge such as $K_S K_R(x_1)$ is attained. In particular, it is correct even if

1. we allow messages to be deleted, duplicated, reordered, or detectably corrupted;
2. we have an asynchronous system, where S and R perform an action only when they are scheduled (rather than performing an action at every round). Of course, in order to assure the liveness property, we must assume that S and R are scheduled infinitely often (although this by itself may not suffice);
3. there is some *a priori* knowledge about the sequence X (e.g., it is common knowledge that the first 10 elements of X are 0's).

The presentation of protocol A in Figure 1 is informal, in that it is given in terms of **if-then-else** statements. In order to provide a formal proof of correctness, we need to be able to view A as a function from states and interpreted systems to actions along the lines discussed in Section 2. Thus we must specify local states for S, R, and the environment e, view S's and R's protocols as functions from local states to actions, define a protocol for the environment, and define a transition function that associates with each joint action a global state transformer. We briefly sketch the details here.

There is no unique way to represent the local states of S, R, and e. Indeed, finding the appropriate representation is often a very difficult problem. Fortunately, in this case, the text of the program suggests some reasonable choices. Since S has commands which update a counter i, we expect this counter to be part of its local state. S's local state also has a variable (or *register*) y which contains the value of the last data element read. Similarly, R's local state has a variable j. S's local state and R's local state contain variables z and z' respectively that store the last message received. Finally, we require that S keeps track of the values it has read and R keeps track of the values it has written. To see where we use this, suppose R sends S a message of the form $\neg\mathtt{K_R}(\mathtt{x}_{j+1})$ at the point (r,m) and S receives this message at some later point (r,m'). As we shall show (Lemma 4.2), R knows the value of x_j when it sends this message; i.e., $K_R(x_j)$ holds at (r,m). Let k be the value of j at the point (r,m). We would like to be able to conclude that $K_S K_R(x_k)$ holds when S receives the message. (Note that we do not expect that $K_S K_R(x_j)$ will hold when S receives the message, because by the time that S receives the message, j may have a different value.) However, if R "forgets" the value of x_k earlier, then $K_R(x_k)$ would no longer hold at (r,m'), and hence neither would $K_S K_R(x_k)$. By having R keep track of all the values it has written, we assure that such forgetting does not occur. (We return to this point later.)

Motivated by this discussion, we take L_S, the set of local states for S, to consist of states of the form (y,i,z,X'), where $y \in \{0,1\}$ is the last data element read, i records the value of the counter i used in A_S, z is the last message received by S, and X' is the sequence of values read by S. Similarly, we take L_R, R's local states, to consist of states of the form (j,z',Y), where j is the value of R' counter, z' is the last message received by R, and Y is the sequence of values written by R.

In the environment state we want to keep track of the input sequence, the elements read by S, the elements written by R, all the messages sent by S and R (since we allow message duplication, it does not suffice to just keep track of messages sent but not yet received); we also need some mechanism for allowing the environment to determine which of S and R is scheduled. Thus, we take the set L_e of environment states to consist of states of the form $(X,c,Y,b_S,b_R,go_S,go_R)$, where X is the input sequence, c is a counter describing which element of X was last read, Y, as before, is the sequence of elements written, b_S (resp. b_R) is a sequence containing all the messages sent by S (resp. R). We take go_S (resp. go_R) to be a Boolean variable whose value is 0 or 1 depending on whether S (resp. R) is scheduled to move on that round. As we shall see, by using go_S and go_R we can capture asynchrony, where a process moves only when it is scheduled.

We take $\mathcal{G}$, the set of possible global states for A, to be $L_e \times L_S \times L_R$. $\mathcal{G}_0$, the set of initial global states of A, consists of all global states of the form

$$((X,0,\langle\rangle,\langle\rangle,\langle\rangle,go_S,go_R),(x_0,0,\lambda,\langle\rangle),(0,\lambda,\langle\rangle)),$$

where X is an infinite sequence of 0s and 1s, x_0 is the first element in the sequence X, and λ denotes the empty message. Thus, in an initial state, S starts out reading the first element of the sequence X, R has not written any elements, and neither S nor R has sent or received any messages.

We define $\mathcal{R}^p(A)$, the set of *potential runs of* A, to be the set of all runs r over $\mathcal{G}$ such that $r(0) \in \mathcal{G}_0$. When we consider interpreted systems that are consistent with A, all the runs will be potential runs of A.

It is now easy to view S's protocol A_S as a function from its local state and an interpreted system $\mathcal{I}$ to actions. We have:

$$A_S((y,i,z,X'),\mathcal{I}) = \begin{cases} i := i+1; \text{read } y & \text{if } (\mathcal{I},r,m) \models K_R(x_{@i}) \text{ whenever } r_S(m) = (y,i,z,X') \\ \text{send } \langle i,y\rangle & \text{otherwise.} \end{cases}$$

Note that there is a slight overloading of notation above. The i that appears in the local state (y,i,z,X') as an argument to A_S refers to the value of the register i in S's local state, the i that appears in the left-hand side of the action $i := i+1$ refers to the register, while the one in the right-hand side again refers to the value of the register. Similar comments hold for the occurrences of y. We hope the reader will be able to disambiguate without any difficulty. We also leave it to the reader to provide the analogous definition for A_R.

We want to prove that A is correct even if messages can be reordered, duplicated, and detectably corrupted. We capture these possibilities in the environment's protocol. At each step, the environment can nondeterministically choose to perform a "send$_S$ m" action, for some m in the sequence b_R or in $\{*,\lambda\}$, or a "send$_S$ current" action. Thus, the environment delivers either a message previously sent by R (i.e., one in b_R), a corrupted message, denoted by $*$, an empty message (one that may have been deleted), denoted by λ, or the message that R is currently sending. If S is scheduled (i.e., if $go_S = 1$), the result of this action is that z is updated appropriately, since S performs a "receive z" action whenever it is scheduled. Similarly, at each step the environment nondeterministically chooses to perform a "send$_R$ m" action for some m in the sequence b_S or in $\{*,\lambda\}$, or a "send$_R$ current" action. (We could of course assume that the environment can deliver more than one message at a time. The resulting model would be similar to the one we use.) Duplication of messages is possible since the same message in b_R can be delivered several times. We capture the possibility of message corruption by allowing the environment to send the message $*$ (other more sophisticated ways of capturing message corruption are clearly possible); message deletion is captured by allowing the environment to send λ. Finally, at each step, the environment can perform actions of the form $go_p := 0$ or $go_p := 1$ for $p \in \{S,R\}$; this has the effect of determining which of S and R will be scheduled at the next step.

All that remains is to define the transition function τ. The definition is completely straightforward, although tedious to write down. For example, if we take $a_e = \text{send}_S *;\ go_R := 1$, $a_S = \text{send } \langle i,y\rangle;\text{ receive } z$, and $a_R = \text{send } \neg\mathtt{K_R(x_{\mathit{j}})};\text{ receive } z'$, then we have

$$((X,c,Y,b_S,b_R,1,0),(y,i,z,X'),(j,z',Y)) \xrightarrow{\tau(a_e,a_S,a_R)} ((X,c,Y,b_S\cdot\langle i,y\rangle,b_R,1,1),(y,i,*,X'),(j,z',Y)).$$

Thus, as a result of the environment scheduling S when S sends $\langle i,y\rangle$ to R and the environment sends $*$ to S, $\langle i,y\rangle$ is appended to b_S (we use $\cdot$ to indicate the operation of appending to a sequence) and z is set to $*$. R's action is disabled since $go_R = 0$ (although go_R does get set to 1 in the new global state as a result of the environment's $go_R := 1$ action). If go_R were 1, then the result of the "send $\neg\mathtt{K_R(x_{\mathit{j}})}$" message would be to extend b_R. The effect of τ is similar for other joint actions. We leave further details to the reader.

Although up to now we have implicitly been associating A with the pair (A_S,A_R), we now formally take it to be the joint protocol (A_e,A_S,A_R), where A_e is the nondeterministic protocol for the environment described above.

We have given all the details necessary to determine whether a given interpreted system is consistent with the knowledge-based protocol A (with respect to $\mathcal{G}$, $\mathcal{G}_0$, and τ). Note that there might be many interpreted systems consistent with A. We allow systems where there is no message corruption (so no message of the form $*$ is ever delivered), or systems where there is some *a priori* knowledge of the input sequence (so that, for example, the first element of the input sequence in every run of the system is 0). We do make one restriction on the interpreted systems we consider: we want them to give the natural meaning to the relevant basic facts. In this case, we are only interested in basic facts of the form $x = 0$ and $x = 1$, so we restrict our attention to interpreted systems $\mathcal{I} = (\mathcal{R}, \pi)$ where the truth assignment π is such that the formula $x_i = 0$ (resp. $x_i = 1$) is true exactly at the points where x_i (the i^{th} component of the sequence X in the environment's local state) really does have the value 0 (resp. 1). For the purposes of this paper, we call such systems *appropriate*.

We can now state the correctness result. We say a run r is *fair* if both S and R are scheduled infinitely often in R (i.e., if $go_S = 1$ at infinitely many points in r, and similarly for go_R), and every message sent by S (resp. R) infinitely often after a given point is eventually delivered uncorrupted after that point (when R (resp. S) is scheduled). Note that if a message is delivered at a point when S is scheduled, then the semantics of A ensures that the message is read (stored in the register z) at that point; similarly for R. We remark that our fairness condition is easily seen to be equivalent to the condition that any message that is sent infinitely often by S is delivered uncorrupted infinitely often to R when R is scheduled, and similarly with the roles of S and R reversed.

Recall that a run of a protocol for the sequence transmission problem has the safety property if the sequence of elements written is always a prefix of the input sequence, and it has the liveness property if every element in the input sequence is eventually written.

Theorem 3.1: *Let $\mathcal{I}$ be an appropriate interpreted system consistent with A (with respect to $\mathcal{G}$, $\mathcal{G}_0$, and τ). Then every run of $\mathcal{I}$ has the safety property, and the fair runs of $\mathcal{I}$ have the liveness property.*

The fact that the theorem holds for *every* appropriate interpreted system consistent with A shows, as we claimed, that A is correct in a wide variety of settings. In particular, A is correct in systems where messages can be deleted, duplicated, reordered, and detectably corrupted (since the environment can perform all these actions), in completely asynchronous system, where the scheduling of S and R is determined by a possibly adversary scheduler (since the environment can perform an arbitrary sequence of go_S and go_R actions), and in systems where there is *a priori* knowledge about the input sequence.

We defer the proof of Theorem 3.1 to the next section. We continue here with our discussion of A. As we mentioned above, the only reason that we need R to keep track of the values it has written is to ensure that $K_S K_R(x_{@i})$ holds when S receives $\neg\mathtt{K_R}(\mathtt{x}_{i+1})$. However, it should be clear that S does not actually need to know that R currently knows the value of x_i before it reads the next data element. It suffices for S to know that R wrote x_i at some time in the past. Thus, we can replace the test $\neg K_S K_R(x_{@i})$ in S's protocol by $\neg K_S(R \text{ wrote } x_i)$, and then delete the X' and Y from the local states of S and R. The resulting protocol is still correct. (We sketch the minor modifications necessary to prove this after our proof of Theorem 3.1 in the next section.)

This remark suggests how we can implement protocol A as a standard protocol A^{st}. The protocol is informally described in Figure 2 below. As the reader can see, the protocols A and A^{st} are very similar syntactically. Tests for knowledge in A are replaced by tests on z and z' in A^{st}, and instead of sending messages of the form $\neg \mathtt{K_R}(\mathtt{x}_j)$, R now sends j, since it carries the same information. As we mentioned above (and prove in the next section), in the case of protocol A, when S receives a message $\neg \mathtt{K_R}(\mathtt{x}_{i+1})$, then S knows that R knows the value of x_i. Similarly, in this case, when S receives a message $i+1$ (i.e., when $z = i+1$), then $K_S K_R(x_{@i})$. Thus, we replace the test $\neg K_S K_R(x_{@i})$ in protocol A by $z \neq i+1$ in protocol A^{st}. Similarly, $K_R(x_j)$ holds if R receives a message of the form $\langle j, y\rangle$ (i.e., when $proj_1(z') = j$, where $proj_1$ returns the first component of its argument). In this case, R writes $proj_2(z')$, the second component of z'.

```
$S$'s protocol:

INIT: $z := \lambda$; $i := 0$; read $y$
do forever
    if $z \neq i+1$
        then send $\langle i, y\rangle$; receive $z$
        else $i := i+1$; read $y$; receive $z$
end

$R$'s protocol:

INIT: $z' := \lambda$; $j := 0$
do forever
    if $proj_1(z') \neq j$
        then send $j$; receive $z'$
        else write $proj_2(z')$; $j := j+1$; receive $z'$
end
```

Figure 2: Protocol A^{st}

The formal semantics of protocol A^{st} is very similar to that of A. L_S^{st} and L_R^{st}, the sets of local states for S and R, are identical to L_S and L_R respectively, except that the components X' and Y are deleted. L_e remains unchanged (except that the form of the messages recorded in b_S and b_R is slightly different). A_S^{st}, A_R^{st}, and A_e^{st}, the protocols for S, R, and e, are defined in a manner analogous to that for A; again, A_e^{st} captures the fact that messages can be deleted, reordered, duplicated, and detectably corrupted. We now formally identify the protocol A^{st} with the joint protocol $(A_S^{st}, A_R^{st}, A_e^{st})$. We leave the details to the reader.

Because of the close syntactic similarity between A^{st} and A, it is quite straightforward to show that the correctness of A^{st} follows from the correctness of A. Formally, we show that A^{st} is an implementation of A by constructing a function Ψ^{st} which maps a run r of A^{st} to a potential run of A such that S and R read and write the same data elements at (r,m) and $(\Psi^{st}(r), m)$. Moreover, if r is a fair run of A^{st}, then $\Psi^{st}(r)$ is also fair. We can then show that the system $\mathcal{I} = (\Psi^{st}(\mathcal{R}(A^{st})), \pi)$ is consistent with respect to A, where π is defined so as to make $\mathcal{I}$ an appropriate system. From the correctness of A, it follows that safety holds for every run of $\Psi^{st}(\mathcal{R}(A^{st}))$, and liveness holds for the fair runs. Since Ψ^{st} preserves fairness and reading and writing, it follows that safety holds for every run in $\mathcal{R}(A^{st})$ and liveness holds for the fair runs of $\mathcal{R}(A^{st})$. We provide the details of the construction in the next section. This gives us the following analogue to Theorem 3.1:

Theorem 3.2: *Every run of A^{st} has the safety property and every fair run of A^{st} has the liveness property.*

Thus, protocol A^{st} solves the sequence transmission problem in systems where messages can be deleted, reordered, duplicated, and detectably corrupted.

Note that A^{st} is still an infinite-state protocol, since z (resp. z'), which is recorded in S's (resp. R's) local state, can take on infinitely many values. This is not just an artifact of our solution; it is a necessary requirement for sufficiently general solutions. In [AFWZ88] it is shown that if we allow messages to be reordered and duplicated (and there is no upper bound on message delivery time for messages that do get delivered), then in *any* solution to the sequence transmission problem S must be able to transmit infinitely many distinct messages; thus there can be no finite state solution to the problem. A somewhat more involved argument given in [AFWZ88] shows that if messages can be deleted and reordered then there is also no finite state solution.

There are finite-state solutions if we restrict the environment's actions. In particular, in Figure 3, we present a finite-state protocol A^{fs} that solves the sequence transmission problem if we do not allow messages to be reordered (although they can be deleted, detectably corrupted, and duplicated). The informal description of A^{fs} is identical to that of A^{st}, except that $+$ is replaced by $\oplus$ (addition mod 2).

The formal description of A^{fs} is, not surprisingly, also very similar to that of A^{st}. The set L_S^{fs} of local states for S is identical to L_S^{st} except that now i and z only take on the values 0 and 1. Similar remarks hold for L_R^{fs}. However, we must now capture the fact that we do not allow messages to be reordered. There are a number of ways that this could be done. We have chosen to do it by adding components $last_S$ and $last_R$ to the environment's state. These are natural numbers that point to the last message in b_S (resp. b_R) that was delivered; i.e., if $last_S = k$, then the k^{th} message in b_S was the last message sent to R by the environment. Initially, we take $last_S = last_R = 0$, and require that they be nondecreasing over time. This guarantees that messages cannot be reordered.

In the next section we describe a function Ψ^{fs} which maps runs of A^{fs} to runs of A^{st} in the obvious way so as to preserve reading and writing of data elements, and fairness. Correctness of A^{fs} then follows from the correctness of A^{st}. We thus get:

Theorem 3.3: *Every run of A^{fs} has the safety property and every fair run of A^{fs} has the liveness property.*

Thus, protocol A^{fs} solves the sequence transmission problem in systems where messages can be deleted, detectably corrupted, and duplicated (but not reordered).

```
S's protocol:

INIT: $z := \lambda$; $i := 0$; read $y$
do forever
    if $z \neq i \oplus 1$
        then send $\langle i, y \rangle$; receive $z$
        else $i := i \oplus 1$; read $y$; receive $z$
end
```

```
R's protocol:

INIT: $z' := \lambda$; $j := 0$
do forever
    if $proj_1(z') \neq j$
        then send $j$; receive $z'$
        else write $proj_2(z')$; $j := j \oplus 1$; receive $z'$
end
```

Figure 3: Protocol A^{fs}

We remark that protocol A^{st} is essentially Stenning's protocol, while protocol A^{fs} is essentially the Alternating Bit protocol.[2] Thus, these well-known protocols are simply implementations of the knowledge-based protocol A.

4 Correctness proofs

In this section we prove the correctness of protocols A, A^{st}, and A^{fs}. We begin with A.

4.1 Correctness of A

Let $\mathcal{I} = (\mathcal{R}, \pi)$ be an appropriate interpreted system consistent with A. Let $r \in \mathcal{R}$. For every $k \geq 0$, for every component α of $r(k)$, we let $\alpha^{r(k)}$ denote the value of α at the global state $r(k)$. For example, $i^{r(k)}$ denotes the value of i at $r(k)$ (recall that i is in fact in the s_S component of $r(k)$). Since $X^{r(k)} = X^{r(k')}$ for all $k, k' \geq 0$, we often omit the k and write X^r for $X^{r(k)}$.

Intuitively, safety for A is obvious since R writes a data element only if R knows its value. A formal proof follows from the next lemma.

Lemma 4.1: *For all runs $r \in \mathcal{R}$ and all times $m \geq 0$, $|Y^{r(m)}| = j^{r(m)}$ and $Y^{r(m)} \preceq X^r$.*

[2]Typically, these protocols are assumed to run in an event-driven system; thus, for example, rather than S sending a message $\langle i, y \rangle$ every time it is scheduled and $z \neq i$, S would only send the message periodically, as determined by an internal timer. We omit the straightforward modifications of the protocol required to deal with the timer here.

Proof: Let $r \in \mathcal{R}$. We proceed by induction on m. For $m = 0$, the claim follows from our characterization of $\mathcal{G}_0$, the set of initial global states. To see that for every $m > 0$, $|Y^{r(m)}| = j^{r(m)}$, note that, from the semantics of A, it is immediate that $|Y|$ increments iff j does. For the inductive step of the second part, assume the claim is established for every $\ell < m$. If $j^{r(m)} = j^{r(m-1)}$, then the semantics of A shows that either $(\mathcal{I}, r, m-1) \models \neg K_R(x_j)$ or $go_R^{r(m-1)} = 0$. In either case it is easy to see that $Y^{r(m)} = Y^{r(m-1)}$, so that the claim trivially follows from the induction hypothesis. If $j^{r(m)} \neq j^{r(m-1)}$, then $(\mathcal{I}, r, m-1) \models K_R(x_\ell)$, where $\ell = j^{r(m-1)}$, $go_R^{r(m-1)} = 1$ and $j^{r(m)} = j^{r(m-1)}+1$. If $x_\ell = 0$ then, since $(\mathcal{I}, r, m-1) \models K_R((x_\ell = 0) \vee K_R(x_\ell = 1))$ and $(I, r, m-1) \models (x_\ell = 0)$, it follows that $(\mathcal{I}, r, m-1) \models K_R(x_\ell = 0)$, so that $Y^{r(m)} = Y^{r(m-1)} \cdot 0$. Similarly, if $x_\ell = 1$ then $(\mathcal{I}, r, m-1) \models \neg K_r(x_\ell = 0)$, so that $Y^{r(m)} = Y^{r(m-1)} \cdot 1$. In either case, it is immediate that the claim holds. (Note that in our semantics, we regard the actions of increasing j and writing either 0 or 1 as being performed at the same time step, i.e., as one atomic action. Without this assumption, the lemma does not hold. However, we can modify A in a straightforward way to get a protocol whose correctness does not depend on such atomicity assumptions; we leave details to the reader.) ▮

In order to prove liveness, we need two preliminary lemmas.

Lemma 4.2: *For every $\ell \geq 0$, $d \in \{0,1\}$, and appropriate interpreted system $\mathcal{I}$, the following all hold:*

1. *If $\langle \ell, d\rangle$ is in $b_S^{r(m)}$ then $(\mathcal{I}, r, m) \models (x_\ell = d)$.*
2. *if $\neg\mathtt{K_R(x_\ell)}$ is in $b_R^{r(m)}$ and $\ell > 1$, then $(\mathcal{I}, r, m) \models \bigwedge_{k=0}^{\ell-1} K_R(x_k)$.*
3. *If $(z')^{r(m)} = \langle \ell, d\rangle$, then $(\mathcal{I}, r, m) \models K_R(x_\ell = d)$.*
4. *if $z^{r(m)} = \neg\mathtt{K_R(x_\ell)}$ and $\ell > 1$, then $(\mathcal{I}, r, m) \models \bigwedge_{k=0}^{\ell-1} K_S K_R(x_k)$.*

Proof:

1. Suppose $\langle \ell, d\rangle$ is in $b_S^{r(m)}$ and let $k \leq m$ be the least integer such that $\langle \ell, d\rangle$ is in $b_S^{r(k)}$. From the semantics of A it follows that the message was sent at the point $(r, k-1)$; moreover, $i^{r(k-1)} = \ell$ and $d = x_\ell$. Hence, $(\mathcal{I}, r, k-1) \models (x_\ell = d)$. Since $\mathcal{I}$ is an appropriate interpreted system, it follows that $(\mathcal{I}, r, m) \models (x_\ell = d)$.

2. Suppose $\neg\mathtt{K_R(x_\ell)}$ is in $b_R^{r(m)}$, $\ell > 1$, and $(r, m) \sim_R (r', m')$. We want to show that the values of $x_0, \ldots, x_{\ell-1}$ are the same in both $r(m)$ and $r'(m')$. Note from the semantics of A it follows that there exists $n \leq m$ such that $\neg\mathtt{K_R(x_\ell)}$ was sent by R at the point (r, n) and $j^{r(n)} = \ell$. By Lemma 4.1, we have that $|Y^{r(n)}| = \ell$ and $Y^{r(n)} \prec X^r$. We clearly have $Y^{r(n)} \preceq Y^{r(m)}$. Since R records the sequence of values it has written in its local state, we must have $Y^{r'(m')} = Y^{r(m)}$. By Lemma 4.1 again, $Y^{r'(m')} \prec X^{r'}$. It follows that the values of $x_0, \ldots, x_{\ell-1}$ are the same in both $r(m)$ and $r'(m')$. Thus, $(\mathcal{I}, r, m) \models \bigwedge_{k=0}^{\ell-1} K_R(x_k)$.

3. Suppose that $(z')^{r(m)} = \langle \ell, d\rangle$ and that $(r', m') \sim_R (r, m)$. Then $(z')^{r'(m')} = \langle \ell, d\rangle$. Thus, we must have that $\langle \ell, d\rangle$ is in $b_S^{r'(m')}$, since $\langle \ell, d\rangle$ must have been sent at some time $n' \leq m'$ in run r'. From part 1, it follows that $(\mathcal{I}, r', m') \models (x_\ell = d)$. Thus $(\mathcal{I}, r, m) \models K_R(x_\ell)$.

4. This proof follows the same lines as that for part 3, using part 2 instead of part 1; we leave details to the reader. ∎

Lemma 4.3: *Let r be a fair run. Then for every $k \geq 0$ there exists an $m_k \geq 0$ such that $j^{r(m_k)} = k$.*

Proof: We prove the lemma by induction on k. For the base case we can take $m_0 = 0$; the result follows since $j^{r(0)} = 0$ by our assumptions about the form of the initial state. For the inductive case assume that for some $m_k \geq 0$, we have $j^{r(m_k)} = k$. We want to show that there exists $m > m_k$ such that $j^{r(m)} = k + 1$. Assume, by way of contradiction, that for all $m \geq m_k$, we have $j^{r(m)} = k$. The semantics of A implies that for all $m \geq m_k$, if R is scheduled at (r, m), we must have $(\mathcal{I}, r, m) \models \neg K_R(x_k)$. Since R's local state does not change when it is not scheduled, it follows that $(\mathcal{I}, r, m) \models \neg K_R(x_k)$ for all $m \geq m_k$. The run r is assumed to be fair, so R sends infinitely many $\neg\mathtt{K_R}(\mathtt{x}_k)$ messages to S in r, and these are received at infinitely many points by S at times when S is scheduled. From Lemma 4.2 we get that $(\mathcal{I}, r, n) \models \bigwedge_{\ell=0}^{k-1} K_S K_R(x_\ell)$ if $z^{r(n)} = \neg\mathtt{K_R}(\mathtt{x}_k)$. Since S is scheduled infinitely often in r, it follows from the semantics of A that i takes on all the values up to and including k in r. In particular, there is some point (r, n') such that $i^{r(n')} = k$. Since $(\mathcal{I}, r, m) \models \neg K_R(x_k)$ for all $m \geq m_k$, it follows from the properties of knowledge that $(\mathcal{I}, r, m) \models \neg K_S K_R(x_k)$ for all $m \geq m_k$ (recall that for any formula φ, if $(\mathcal{I}, r, m) \models K_S\varphi$ then $(\mathcal{I}, r, m) \models \varphi$). Thus, every time S is scheduled after the point (r, n'), it sends R the message $\langle k, y\rangle$. Since r is a fair run, there must be some $m' \geq 0$ such that $z^{r(m')} = \langle k, y\rangle$. By Lemma 4.2, this implies that $(\mathcal{I}, r, m') \models K_R(x_k)$, which is a contradiction. ∎

The proof of Theorem 3.1 now follows easily. Safety follows from Lemma 4.1. For liveness, suppose r is a fair run. We want to show that every data element x_k in X^r is eventually written. By Lemma 4.3, there is some m_k such that $j^{r(m_k)} = k$. By Lemma 4.1, we have that $|Y^{r(m_k)}| = j^{r(m_k)}$ and that $Y^{r(m_k)} \prec X^r$, so that x_k is written by time m_k.

A straightforward modification of this proof shows the correctness of the modified version of A discussed in the previous section, where the test $K_S K_R(x_{@i})$ in line S2 is replaced by $K_S(R \text{ wrote } x_i)$, and the X' and Y components of S's and R's local states are deleted. All we need to do is replace the $K_R(x_k)$ in parts 2 and 4 of Lemma 4.2 by $(R \text{ wrote } x_k)$. We leave details to the reader.

4.2 Correctness of A^{st}

As we said before, we prove correctness of A^{st} by showing that A^{st} is an implementation of A. We define a mapping Ψ^{st} from runs of A^{st} to potential runs of A that preserves reading and writing of data elements, and preserves fairness. We want a function $\Psi^{st}: \mathcal{R}(A^{st}) \to \mathcal{R}^p(A)$ such that for every run $r \in \mathcal{R}(A^{st})$ and every time m we have:

I1. $X^{\Psi^{st}(r)} = X^r$, $c^{\Psi^{st}(r)(m)} = c^{r(m)}$, $Y^{\Psi^{st}(r)(m)} = Y^{r(m)}$,

I2. if r is fair then $\Psi^{st}(r)$ is fair.

Note that I1 says that reading and writing of data elements are preserved.

The definition of Ψ^{st} is the obvious one. Intuitively, we just replace every message j sent by R by the message $\neg\mathtt{K_R}(\mathtt{x}_j)$. We now define $\Psi^{st}(r)(m)$ by induction on m, so that not only does I1 hold, but we also have

- $go_S^{\Psi^{st}(r)(m)} = go_S^{r(m)}$ and $go_R^{\Psi^{st}(r)(m)} = go_R^{r(m)}$
- $i^{\Psi^{st}(r)(m)} = i^{r(m)}$ and $j^{\Psi^{st}(r)(m)} = j^{r(m)}$
- $y^{\Psi^{st}(r)(m)} = y^{r(m)}$
- $b_S^{\Psi^{st}(r)(m)} = b_S^{r(m)}$ and $b_R^{\Psi^{st}(r)(m)}$ is the result of replacing every message ℓ in $b_R^{r(m)}$ by $\neg\mathtt{K_R}(\mathtt{x}_\ell)$
- $(z')^{\Psi^{st}(r)(m)} = (z')^{r(m)}$, and $z^{\Psi^{st}(r)(m)}$ is like $z^{r(m)}$ except that if $z^{r(m)} = \ell$, then $z^{\Psi^{st}(r)(m)} = \neg\mathtt{K_R}(\mathtt{x}_\ell)$.

This completely specifies Ψ^{st}. It is easy to see that with this definition, I2 also holds.

Let $\mathcal{R} = \Psi^{st}(\mathcal{R}(A^{st}))$, and let $\mathcal{I} = (\mathcal{R}, \pi)$ be an appropriate interpreted system. (Recall this means that the formulas $x_c = 1$ and $x_c = 0$ are true exactly where they ought to be.) Our goal is to show that $\mathcal{I}$ is consistent with A. Safety of A^{st} then follows from safety of A, and liveness of A^{st} then follows from liveness of A. By the definition of Ψ^{st}, it is easy to see that if $r \in A^{st}$, then $r(0) = \Psi^{st}(r)(0)$, and that initial states of runs of A^{st} are legal initial states of runs of A. Thus it only remains to show that $\Psi^{st}(r)(m+1)$ is the result of transforming $\Psi^{st}(r)(m)$ by a joint action that could have been performed by A in $\mathcal{I}$.

Proposition 4.4: *Let $r \in \mathcal{R}(A^{st})$. Then, for every $m, k \geq 0$, if $j^{r(m)} = k$ and $j^{r(m+1)} = k+1$ then $i^{r(m)} = k$, and if $i^{r(m)} = k$ and $i^{r(m+1)} = k+1$ then $j^{r(m)} = k+1$*

Proof: The proof is by induction on m. If $m = 0$, then $j^{r(0)} = i^{r(0)} = 0$. Moreover, $z^{r(0)} = \lambda$, so the semantics of A^{st} guarantees that $i^{r(1)} = 0$. The claim for the case $m = 0$ trivially follows. For the inductive step, assume the claim holds for every $m' < m$. If $j^{r(m)} = k$ and $j^{r(m+1)} = k+1$, then, by the semantics of A^{st}, $proj_1((z')^{r(m)}) = k$. It follows that $i^{r(n)} = k$ for some $n < m$. Since i is clearly nondecreasing, we must have $i^{r(m)} \geq k$. To see that $i^{r(m)} = k$, assume that $i^{r(m)} > k$. From the semantics of A^{st} it follows that for some $m' < m$, $i^{r(m')} = k$ and $i^{r(m'+1)} = k+1$. Hence, by the induction hypothesis, $j^{r(m')} = k+1$. Since j is nondecreasing, this contradicts the assumption that $j^{r(m)} = k$. It therefore follows that $i^{r(m)} = k$. The inductive step for the second half is similar. ∎

Proposition 4.5: *Let $r \in \mathcal{R}(A^{st})$. Then, for every $m \geq 0$:*

1. *$z^{r(m)} \neq i^{r(m)} + 1$ iff $(\mathcal{I}, \Psi^{st}(r), m) \models \neg K_S K_R(x_{@i})$.*
2. *$proj_1(z'^{r(m)}) \neq j^{r(m)}$ iff $(\mathcal{I}, \Psi^{st}(r), m) \models \neg K_R(x_j)$.*

Proof: Assume that $z^{r(m)} \neq i^{r(m)} + 1$. Suppose that $i^{r(m)} > 0$ (the proof is similar, but somewhat easier, if $i^{(r,m)} = 0$; we leave details to the reader). From the semantics of A^{st}, it follows that there exists some $m'' < m$ such that $i^{r(m'')} = i^{r(m)} - 1$, $z^{r(m'')} = i^{r(m)}$, and $i^{r(m''+1)} = i^{r(m)}$. Thus there must exist $m' < m''$ such that $j^{r(m')} = z^{r(m'')}$ and $go_R^{r(m')} = 1$. We now show that every message delivered to S between times m' and m was actually sent by R at or before time m'. From Proposition 4.4 and the fact that both j and i are nondecreasing, we have that $j^{r(n)}$ is either $i^{r(m)}$ or $i^{r(m)} + 1$ for all n with $m' \leq n \leq m$, and for $n \leq m'$, we have $j^{r(n)} \leq i^{r(m)} + 1$. Moreover, if $j^{r(n)} = i^{r(m)} + 1$, then $i^{r(n)} = i^{r(m)}$. It follows that we cannot have $z^{r(n)} = i^{r(m)+1}$ for $n \leq m$. For if $z^{r(n)} = i^{r(m)} + 1$, then we must have $i^{r(n)} = i^{r(m)}$. If S is not scheduled between n and m, then $z^{r(m)} = i^{r(m)} + 1$, contradicting our initial assumption, and if S is at n' between n and m, then we would have $i^{r(n')} = i^{r(m)} + 1$, also a contradiction. Thus, if $n \leq m$, then $z^{r(n)}$, if it is an integer, must be at most $i^{r(m)}$. In particular, every message delivered to S at or before time m was already sent by R at or before time m'.

Let r' be a run identical to r up to time m', such that R is not scheduled (and hence no messages to R are delivered) from time m' to m, such that $r_S(n) = r'_S(n)$ for all $n \leq m$ (such a run r' exists since all the messages received by S in r at or before time m were already sent by R at or before time m'). By the definition of Ψ^{st}, it follows that $(\Psi^{st}(r), m) \sim_R (\Psi^{st}(r'), m)$. Let $r'' \in \mathcal{R}^{st}$ be such that

1. $X^{r''}$ is identical to $X^{r'}$ (and X^r) except that $x^{r''}_{i^{r(m)}} \neq x^{r'}_{i^{r(m)}}$
2. For $n \leq m$, we have $r''_R(n) = r'_R(n)$.

It is easy to see that such a run r'' exists. For example, we can take r'' so that it is identical to r' up to time m' (except for the difference between $X^{r''}$ and $X^{r'}$) such that neither S nor R is scheduled between m' and m in r''. It immediately follows that $(\Psi^{st}(r''), m) \sim_R (\Psi^{st}(r'), m)$. Since $x^{r''}_{i^{r(m)}} \neq x^{r'}_{i^{r(m)}}$, we must have $(\mathcal{I}, \Psi^{st}(r'), m) \models \neg K_R(x_{@i})$. Consequently, $(\mathcal{I}, \Psi^{st}(r), m) \models \neg K_S K_R(x_{@i})$, as desired.

If $z^{r(m)} = i^{r(m)}+1$, then choose m' to be the last time before m that S is scheduled (i.e., such that $go_S^{r(m')} = 1$). Note that there must be such a time, otherwise we would have $z^{r(m)} = \lambda$; z never changes if S is not scheduled. We must have $z^{r(m'+1)} = i^{r(m)} + 1$ (again, because z does not change when S is not scheduled). By definition, we have $(z)^{\Psi^{st}(r)(m'+1)} = \neg \mathtt{K_R}(\mathtt{x}_{i^{r(m)}+1})$. From part 4 of Lemma 4.2, it follows that $(\mathcal{I}, \Psi^{st}(r), m) \models K_S K_R(x_{@i})$, as desired.

The proof of part 2 of this lemma is similar and left to the reader. ▮

From Proposition 4.5 we can derive:

Corollary 4.6: $\mathcal{I} = (\Psi^{st}(\mathcal{R}(A^{st})), \pi)$ *is consistent with* A.

Since the same read and write actions are performed at the same times in r and $\Psi^{st}(r)$, and since $\mathcal{I}$ is consistent with A, it follows from Theorem 3.1 that every run in $\mathcal{R}(A^{st})$ has the safety property. And since Ψ^{st} also preserves fairness, it follows from Theorem 3.1 that fair runs of $\mathcal{R}(A^{st})$ have the liveness property.

4.3 Correctness of A^{fs}

We prove the correctness of A^{fs} by showing that it is an implementation of A^{st}. Again we define a mapping from runs of A^{fs} to runs of A^{st} that preserves reading and writing of data elements, and preserves fairness. That is, we want $\Psi^{fs}: \mathcal{R}(A^{fs}) \to \mathcal{R}(A^{st})$ to have the following properties:

I1. $X^{\Psi^{fs}(r)} = X^r$, $c^{\Psi^{fs}(r)(m)} = c^{r(m)}$, $Y^{\Psi^{fs}(r)(m)} = Y^{r(m)}$,

I2. if r is fair then $\Psi^{fs}(r)$ is fair.

Again, the definition of Ψ^{fs} is straightforward. We define Ψ^{fs} so that I1 holds and:

- $go_S^{\Psi^{fs}(r)(m)} = go_S^{r(m)}$, $go_R^{\Psi^{fs}(r)(m)} = go_R^{r(m)}$
- $y^{\Psi^{fs}(r)(m)} = y^{r(m)}$
- $i^{\Psi^{fs}(r)(m)} = c^{r(m)}$ and $j^{\Psi^{fs}(r)(m)} = |Y^{r(m)}|$.
- The definitions of $z^{\Psi^{fs}(r)(m)}$, $z'^{\Psi^{fs}(r)(m)}$, $b_S^{\Psi^{fs}(r)(m)}$, and $b_R^{\Psi^{fs}(r)(m)}$, are slightly more complicated, since in protocol A^{st}, arbitrary integers can be sent, not just 0 and 1. Thus we have:

$$z^{\Psi^{fs}(r)(m)} = \begin{cases} i^{\Psi^{fs}(r)(m)} & \text{if } z^{r(m)} = i^{r(m)} \\ i^{\Psi^{fs}(r)(m)} + 1 & \text{if } z^{r(m)} = i^{r(m)} \oplus 1 \\ z^{r(m)} & \text{otherwise} \end{cases}$$

$$b_S^{\Psi^{fs}(r)(m)} = \begin{cases} \langle\rangle & \text{if } m = 0 \\ b_S^{\Psi^{fs}(r)(m-1)} & \text{if } m > 0,\ b_S^{r(m)} = b_S^{r(m-1)} \\ b_S^{\Psi^{fs}(r)(m-1)} \cdot \langle i^{\Psi^{fs}(r)(m-1)}, y^{\Psi^{fs}(r)(m-1)} \rangle & \text{otherwise;} \end{cases}$$

$z'^{\Psi^{fs}(r)(m)}$ and $b_R^{\Psi^{fs}(r)(m)}$ are similarly defined.

Let $\mathcal{R} = \Psi^{fs}(\mathcal{R}(A^{fs}))$. Our goal is to show that every run $r \in \mathcal{R}$ is consistent with protocol A^{st}. It is easy to see that $\Psi^{fs}(r)(0) \in \mathcal{G}_0^{st}$. Thus, we have to show that for all $m \geq 0$, $r(m+1)$ is the result of transforming $r(m)$ by a joint action that could have been performed from $r(m)$ according to A^{st}. To show this, we first establish some invariant properties of runs in $\mathcal{R}(A^{fs})$:

Lemma 4.7: *Let $r \in \mathcal{R}(A^{fs})$. Then for every $m \geq 0$, the following all hold:*

1. $c^{r(m)} = i^{r(m)}$ *(mod 2) and* $j^{r(m)} = |Y^{r(m)}|$ *(mod 2).*
2. (a) *If $c^{r(m)} = z^{r(m)} = 0$, then there exists $m' < m$ such that $|Y^{r(m')}| = 0$ and $b_R^{r(m'+1)} \neq b_R^{r(m')}$.*

   (b) *If $|Y^{r(m)}| = proj_1(z'^{r(m)}) = 0$, then there exists $m' < m$ such that $c^{r(m')} = 0$ and $b_S^{r(m'+1)} \neq b_S^{r(m')}$.*

3. *(a)* *If $c^{r(m)} = k > 0$ (resp. and in addition $z^{r(m)} = k+1 \pmod 2$), then there exist times $m_1 < \ldots < m_k < m$ (resp. $m_1 < \ldots < m_{k+1} < m$) such that for all ℓ with $1 \leq \ell \leq k$ (resp. $1 \leq \ell \leq k+1$), we have $|Y^{r(m_\ell)}| = \ell$ and $b_R^{r(m_\ell+1)} \neq b_R^{r(m_\ell)}$ (so that a message was sent by R at the point (r, m_ℓ)).*

(b) *If $|Y^{r(m)}| = k$, $k > 0$ (resp. and in addition $proj_1((z')^{r(m)}) = k+1 \pmod 2$) then there exist times $m_0 < \ldots < m_{k-1} < m$ (resp. $m_0 < \ldots m_k < m$) such that for all ℓ with $0 \leq \ell \leq k-1$ (resp. $0 \leq \ell \leq k$), we have $c^{r(m_\ell)} = \ell$ and $b_S^{r(m_\ell+1)} \neq b_S^{r(m_\ell)}$ (so that a message was sent by S at the point (r, m_ℓ)).*

(c) $|Y^{r(m)}| \leq c^{r(m)} \leq |Y^{r(m)}| + 1$.

Proof:

1. For $m = 0$ we have $c^{r(0)} = i^{r(0)} = j^{r(0)} = |Y^{r(0)}| = 0$, so that the claim holds by definition of the initial states of A^{fs}, while for $m > 0$, the claim holds since i (resp. j) is incremented mod 2 exactly when a new data element is read (resp. written).

2. (a) From the semantics of A^{fs} it follows that if $z^{r(m)} = 0$, then there must have been some time $m' < m$ such that R was first scheduled at m'. From the semantics of A^{fs}, it follows that $|Y^{r(m')}| = 0$, $b_R^{r(m')} = \langle\rangle$, $j^{r(m')} = 0$, $(z')^{r(m')} = \lambda$, and hence that $b_R^{r(m'+1)} = \langle 0 \rangle$. The result follows.

   (b) Similar to the previous case.

3. We prove all the parts simultaneously by induction on m. The case $m = 0$ is trivial. Suppose $m > 0$. For part (a), consider the case $c^{r(m)} = k$, $z^{r(m)} = k+1 \pmod 2$ and $k > 0$. (The proof in the other case is easier, and left to the reader.) Let n_ℓ, $\ell = 1, \ldots, k$ be such that $c^{r(n_\ell)} = \ell - 1$, $c^{r(n_\ell+1)} = \ell$. The semantics of A^{fs} guarantees that it must be the case that $z^{r(n_\ell)} = i^{r(n_\ell)} \oplus 1$. By part (1) it follows that $z^{r(n_\ell)} = \ell \pmod 2$. Since messages are not reordered, and the messages received at times $n_1, \ldots, n_\ell$ alternate values between 0 and 1, it must be the case that there exist times $m_1 < \ldots < m_{k+1}$ such that $m_\ell < n_\ell$ for $\ell = 1, \ldots, k$, $m_{k+1} < m$, and the value $z^{r(n_\ell)}$ is sent at time m_ℓ and $j^{r(n_\ell)} = z^{r(n_\ell)}$, $\ell = 1, \ldots, k+1$. By part (1), it follows that the value of $|Y|$ has changed at least $\ell - 1$ times by the point (r, n_ℓ), so that $|Y^{r(n_\ell)}| \geq \ell$; moreover, by (1) again, $|Y^{r(n_\ell)}| = z^{r(n_\ell)} \pmod 2$. Since the value of c is nondecreasing, by the induction hypothesis applied to (c), we have that $|Y^{r(n_\ell)}| \leq \ell + 1$. Since $|Y^{r(n_\ell)}| = \ell \pmod 2$, it follows that in fact $|Y^{r(n_\ell)}| = \ell$. This argument also shows that $|Y^{r(m)}| \geq |Y^{r(n_k)}| \geq k$, proving the first half of (c).

   The proof of (b) and the second half of (c) is similar and is left to the reader. ∎

We can now show that $\Psi^{fs}(r) \in \mathcal{R}(A^{st})$ if $r \in \mathcal{R}(A^{fs})$. That is, we can show that the actions performed by the processors and the environment in $\Psi^{fs}(r)$ are really the ones dictated by A^{st}.

Lemma 4.8: *Let $r \in \mathcal{R}(A^{fs})$. Then for every $m \geq 0$ we have:*

1. *For every $k \geq 0$,*

   (a) *If $z^{\Psi^{fs}(r)(m)} = k$ then $k \in b_R^{\Psi^{fs}(r)(m)}$.*

(b) *If* $z'^{\Psi^{fs}(r)(m)} = \langle k, \ell \rangle$, *then* $\langle k, \ell \rangle \in b_S^{\Psi^{fs}(r)(m)}$.

2. (a) *If* $go_S^{\Psi^{fs}(r)(m)} = 1$ *then*

   i. *If* $z^{\Psi^{fs}(r)(m)} \neq i^{\Psi^{fs}(r)(m)} + 1$ *then* $b_S^{\Psi^{fs}(r)(m+1)} = b_S^{\Psi^{fs}(r)(m)} \cdot \langle i^{\Psi^{fs}(r)(m)}, y^{\Psi^{fs}(r)(m)} \rangle$,

   ii. *If* $z^{\Psi^{fs}(r)(m)} = i^{\Psi^{fs}(r)(m)} + 1$ *then* $i^{\Psi^{fs}(r)(m+1)} = i^{\Psi^{fs}(r)(m)} + 1$ *and* $c^{\Psi^{fs}(r)(m+1)} = c^{\Psi^{fs}(r)(m)} + 1$.

   (b) *If* $go_R^{\Psi^{fs}(r)(m)} = 1$ *then*

   i. *If* $proj_1(z'^{\Psi^{fs}(r)(m)}) \neq j^{\Psi^{fs}(r)(m)}$ *then* $b_R^{\Psi^{fs}(r)(m+1)} = b_R^{\Psi^{fs}(r)(m)} \cdot j^{\Psi^{fs}(r)(m)}$,

   ii. *If* $z'^{\Psi^{fs}(r)(m)} = j^{\Psi^{fs}(r)(m)}$ *then* $j^{\Psi^{fs}(r)(m+1)} = j^{\Psi^{fs}(r)(m)} + 1$ *and* $Y^{\Psi^{fs}(r)(m+1)} = Y^{\Psi^{fs}(r)(m)} \cdot proj_2(z'^{\Psi^{fs}(r)(m)})$.

Proof:

1. (a) If $z^{\Psi^{fs}(r)(m)} = k$, then either (a) $z^{\Psi^{fs}(r)(m)} = i^{\Psi^{fs}(r)(m)}$ and (b) $z^{r(m)} = i^{r(m)}$ or $z^{\Psi^{fs}(r)(m)} = i^{\Psi^{fs}(r)(m)} + 1$ and $z^{r(m)} = i^{r(m+1)} \oplus 1$. Suppose case (a) obtains. Since, by definition of Ψ^{fs}, we have $i^{\Psi^{fs}(r)(m)} = c^{r(m)}$, and by part (1) of Lemma 4.8 we have $i^{r(m)} = c^{r(m)}$ (mod 2), it follows from part (3) of Lemma 4.7 (or part (2) in case $c^{r(m)} = 0$) that there exist times $m_0 < \ldots < m_k < m$ such that for all $0 \leq \ell \leq k$, we have $|Y^{r(m_\ell)}| = \ell$ and $b_R^{r(m_\ell+1)} \neq b_R^{r(m_\ell)}$. From the definition of Ψ^{fs}, it follows that $\ell \in b_R^{\Psi^{fs}(r)(m_\ell+1)}$, and, in particular $k \in b_R^{\Psi^{fs}(r)(m_k+1)}$. The proof if case (b) obtains is similar and is left to the reader.

   (b) Similar to the previous case.

2. We prove only part (i) of (a); the other cases are similar and left to the reader. Observe that if $z^{\Psi^{fs}(r)(m)} \neq i^{\Psi^{fs}(r)(m)}+1$ and $go_S^{\Psi^{fs}(r)(m)} = 1$, then, by the definition of Ψ^{fs}, we have $z^{r(m)} \neq i^{r(m)} + 1$ and $go_S^{r(m)} = 1$. The semantics of A^{fs} guarantees that $b_S^{r(m+1)} \neq b_S^{r(m)}$. The claim now follows immediately from the definition of $b_S^{\Psi^{fs}(r)(m+1)}$. ∎

We now immediately get:

Lemma 4.9: *Let* $r \in \mathcal{R}(A^{st})$. *Then* $\Psi^{fs}(r)$ *is consistent with protocol* A^{st}.

Since the definition of Ψ^{fs} guarantees that for every run r in $\mathcal{R}(A^{fs})$, the same read and write actions are performed at the same times in r and $\Psi^{fs}(r)$, it follows from Theorem 3.2 that every run in $\mathcal{R}(A^{fs})$ has the safety property. And since Ψ^{fs} preserves fairness, it also follows from Theorem 3.2 that every fair run of A^{fs} has the liveness property. This completes the proof of Theorem 3.3.

5 A solution in the AUY model

In the AUY model [AUWY82, AUY79], message transmission is assumed to proceed in synchronous clocked rounds. Processes are always scheduled. We can conceptually think of a round or step as consisting of three phases: a send phase, a receive phase (where all messages sent in that step that are not deleted are received, possibly after being corrupted), and a local computation phase (during which data elements may be read from the input sequence or written onto the output sequence). Since a message is received on the same round in which it is sent (if it is received at all), messages cannot be reordered or duplicated. As in [AUWY82, AUY79], we assume that the symbols transmitted over the channel are 0, 1, and λ (again λ denotes that nothing is sent), and that the input sequence X consists of 0s and 1s. We consider three types of errors in the communication medium:

- *Deletion* errors: 0 or 1 is sent, but λ (nothing) is received.
- *Mutation* errors: 0 (resp. 1) is sent, but 1 (resp. 0) is received.
- *Insertion* errors: λ is sent, but 0 or 1 is received.

As observed in [AUWY82, AUY79], the sequence transmission problem has no solution if all three error types are present. To see this, observe that if all error types are present, then any sequence σ of messages in $\{0, 1, \lambda\}^*$ transmitted by S can be altered by the channel to any other sequence σ' of the same length. Thus R can gain no information from the messages it receives about the messages that S actually transmitted. It was also shown in [AUWY82, AUY79] that for any combination of two out of the three possible types of errors, the problem is solvable. However, the informal correctness proofs presented in [AUWY82, AUY79] are difficult to follow, and some effort has been expended in constructing more formal proofs [Gou85, Hai85]. Unfortunately, these proofs are also far from transparent.

We provide solutions to the sequence transmission problem in the AUY model by implementing protocol A^{fs}. Note that A^{fs} as it stands does not provide a solution, since messages such as $\langle i, y\rangle$ are illegal in the AUY model. We overcome this difficulty by encoding messages of the form $\langle i, y\rangle$ using only λ, 0, and 1. This is straightforward under our assumption that the system is synchronous. Suppose we first restrict attention to deletion and insertion faults. A message of the form $\langle i, y\rangle$ can then be transmitted by first sending i, then on the next round sending y. This leads us to the protocol A^{di}, described in Figure 4 below (the *di* stands for deletion and insertion since, as we shall show, the protocol tolerates deletion and insertion errors). Note that each iteration of the loop corresponds to two time steps (each one begun by a "send" statement), since each step consists of a send phase, receive phase, and a local computation phase. S now has two variables for receiving, z_1 and z_2. Messages are received in z_1 in the first step of the loop, and in z_2 in the second step.

We give semantics to A^{di} along the same lines as A^{fs}. The set L_S^{di} of local states for S is identical to L_S^{st} except that now we have variables z_1 and z_2 instead of just z, and we have a counter *step* taking values in $\{1, 2\}$ keeping track of which step in the loop S is about to perform. Similar remarks hold for L_R^{di}. The set L_e^{di} of local states for the environment is now considerably simpler. It just has the form $\langle X, c, Y\rangle$. We can omit the b_S and b_R components, since the only messages that can be delivered are the messages sent in that round; there is no point in keeping track of all the messages sent by S and R. And we can omit the go_S and go_R components in the environment's state, since S and R are always scheduled. The actions performed by S and R are analogous to those performed when running protocol A^{fs}. However, we must allow the environment to perform actions corresponding to deletion and insertion

S's protocol:

```
INIT: $z_1 := \lambda$; $z_2 := \lambda$; $i := 0$; read $y$
do forever
  send $i$; receive $z_1$;
  send $y$; receive $z_2$; if $z_1 = i \oplus 1$ then $i := i \oplus 1$; read $y$ end
end
```

R's protocol:

```
INIT: $z_1' := \lambda$; $z_2' := \lambda$; $j := 0$
do forever
  send $j$; receive $z_1'$;
  send $\lambda$; receive $z_2'$; if $\langle z_1', z_2' \rangle = \langle j, \lambda \rangle$ then write $z_2'$; $j := j \oplus 1$ end
end
```

Figure 4: Protocol A^{di}

errors. Thus, we take the environment's actions to have the form (a_S, b_R), where a and b are one of $ins0$, $ins1$, del, or Λ. The effect of $ins0_S$ is that R receives 0 if S sent λ; otherwise R receives whatever S sent. The effect of $ins1_S$ is similar. The effect of del_S is that R receives λ (i.e., S's message, if there is one, is deleted), while the effect of Λ_S is that R receives whatever S sent. The effect of actions of the form b_R is analogous. The environment nondeterministically performs one of these actions at each step. We leave further details to the reader.

We prove correctness of A^{di} by defining a mapping Ψ^{di} mapping runs of A^{di} to runs of A^{fs}. Since the sending of a message in A^{fs} corresponds to two steps in A^{di}, we conceptually divide runs of A^{di} into *2-blocks*. The first 2-block consists of times 0 and 1, the next one consists of times 2 and 3, and so on. Each 2-block corresponds to one iteration of the loop. If both messages i and y sent by S in a 2-block of a run r of A^{di} are delivered uncorrupted (i.e., undeleted) to R, then we say that $\langle i, y \rangle$ is delivered in $\Psi^{di}(r)$. If either part of the 2-block is corrupted, then we say that λ is delivered to S in $\Psi^{di}(r)$. Note that the messages sent by R in a 2-block are of the form j followed by λ. As long as both of these messages are delivered uncorrupted, then we say j is delivered in the corresponding run $\Psi^{di}(r)$.[3] Note that either the message that S intended to send is actually delivered to R, or else R can tell that the message sent by S was corrupted (and thus R can treat it as if no message at all was sent); identical comments hold with the roles of S and R reversed.

We say a run of A^{di} is fair if there are infinitely many 2-blocks where both of S's transmissions are delivered, and infinitely many 2-blocks where both of R's transmissions are delivered.[4]

[3]In the case of R's messages, we could have just focused attention on the first component of the 2-block, since it is the only one that gives information to S. We have chosen to present the protocol in this way for uniformity with the protocols we present later.

[4]Note that if we assume a fixed non-zero probability of any given message arriving, then the set of runs satisfying our fairness criterion has measure 1. Our fairness condition is equivalent to that used in [AUWY82,

With this notion of fairness, it is easy to check that the mapping Ψ^{di} sketched above maps fair runs of A^{di} to fair runs of A^{fs}. We leave the details to the reader. Using this mapping, we get:

Theorem 5.1: *Protocol A^{di} solves the sequence transmission problem in the AUY model with deletion and insertion errors. Every run of A^{di} has the safety property, and the fair runs have the liveness property.*

Although A^{di} solves the sequence transmission problem in the AUY model for the case of deletion and insertion errors, it cannot deal with mutation errors. If 0 and 1 can be mutated, then it is possible that, for example, S sends $\langle 0,1\rangle$ and R receives $\langle 0,0\rangle$ (i.e., 0 followed by 0). The point is that R cannot tell when it receives a message from S whether or not it has been corrupted. We solve this problem by using a different encoding function when dealing with mutation errors. We need an encoding function e that has the following two properties (with respect to the faulty behavior allowed) for each message m that can be sent by either S or R (so that $m \in \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle, 0, 1\}$):

1. (*Unique decodability*) If $e(m)$ is received uncorrupted, then the recipient knows that it is uncorrupted, and that it is an encoding of m.
2. (*Corruption detectability*) If $e(m)$ is corrupted, then the recipient knows it is corrupted.

An encoding with these properties essentially allows us to treat any type of faulty behavior as a deletion error. If corruption is detected, a message can be ignored and treated as if it were lost.

In the case of deletion and insertion errors, we took the trivial encoding function e^{di} such that:

$$e^{di}(\langle i,y\rangle) = \langle i,y\rangle \text{ and } e^{di}(i) = \langle i,\lambda\rangle, \text{ for } i,y \in \{0,1\}.$$

It is easy to check that e^{di} has the properties of unique decodability and corruption detectability in the case of deletion and insertion errors. In order to implement A^{fs} in the presence of deletion and mutation errors, we must find an encoding that preserves these properties in the presence of deletion and mutation errors. One encoding that works is e^{dm}, defined by:

$$\begin{aligned}
&e^{dm}(\langle 0,0\rangle) = e^{dm}(0) = \langle 1,\lambda,\lambda,\lambda\rangle,\\
&e^{dm}(\langle 0,1\rangle) = e^{dm}(1) = \langle \lambda,1,\lambda,\lambda\rangle,\\
&e^{dm}(\langle 1,0\rangle) = \langle \lambda,\lambda,1,\lambda\rangle,\\
&e^{dm}(\langle 1,1\rangle) = \langle \lambda,\lambda,\lambda,1\rangle.
\end{aligned}$$

Time is now divided into 4-blocks rather than 2-blocks. The intention here is that if, for example, S wants to send a message $\langle 0,0\rangle$, then it sends a 1 followed by three λs over the next four rounds. It is easy to check that e^{dm} does indeed have the properties of unique decodability and corruption detectability in the case of deletion and mutation errors. Clearly any uncorrupted sequence of messages received in a 4-block is uniquely decodable. A mutation error is easily detectable and, indeed, easily corrected: Since 0 does not appear in any encoded message, if 0 is received it must be the case that a 1 was sent and mutated. If a non-λ message sent in a 4-block is deleted, then the recipient will get all λs in that 4-block, and thus know

AUY79].

that the message was corrupted. These ideas lead us to the protocol A^{dm} described in Figure 5. The protocol is a straightforward variant of A^{di}. We now use e^{dm} to do the encoding. Each iteration of the loop now consists of four time steps, corresponding to a 4-block. S now has four receive variables, z_1, z_2, z_3, and z_4, such that z_k is used in the k^{th} step of a 4-block. As before, we take $proj_j$ to be the function which projects onto the j^{th} component of its argument.

```
S's protocol:

INIT: $z_1 := \lambda$; $z_2 := \lambda$; $z_3 := \lambda$; $z_4 := \lambda$; $i := 0$; read $y$
do forever
    send $proj_1(e^{dm}(\langle i, y\rangle))$; receive $z_1$;
    send $proj_2(e^{dm}(\langle i, y\rangle))$; receive $z_2$;
    send $proj_3(e^{dm}(\langle i, y\rangle))$; receive $z_3$;
    send $proj_4(e^{dm}(\langle i, y\rangle))$; receive $z_4$; if $(e^{dm})^{-1}(\langle z_1, z_2, z_3, z_4\rangle) = i \oplus 1$
                                              then $i := i \oplus 1$; read $y$
end

R's protocol:

INIT: $z'_1 := \lambda$; $z'_2 := \lambda$; $z'_3 := \lambda$; $z'_4 := \lambda$; $j := 0$
do forever
    send $proj_1(e^{dm}(j))$; receive $z'_1$;
    send $proj_2(e^{dm}(j))$; receive $z'_2$;
    send $proj_3(e^{dm}(j))$; receive $z'_3$;
    send $proj_4(e^{dm}(j))$; receive $z'_4$; if $proj_1((e^{dm})^{-1}(\langle z'_1, z'_2, z'_3, z'_4\rangle)) = j$
                                        then write $proj_2((e^{dm})^{-1}(\langle z'_1, z'_2, z'_3, z'_4\rangle))$; $j := j \oplus 1$
end
```

Figure 5: Protocol A^{dm}

In order to deal with mutation and insertion errors, we define an encoding e^{mi} which is just like e^{dm} except that the roles of 1 and λ are interchanged:

$$\begin{aligned} e^{mi}(\langle 0,0\rangle) &= e^{mi}(0) = \langle \lambda, 1, 1, 1\rangle, \\ e^{mi}(\langle 0,1\rangle) &= e^{mi}(1) = \langle 1, \lambda, 1, 1\rangle, \\ e^{mi}(\langle 1,0\rangle) &= \langle 1, 1, \lambda, 1\rangle, \\ e^{mi}(\langle 1,1\rangle) &= \langle 1, 1, 1, \lambda\rangle. \end{aligned}$$

It is easy to check that e^{mi} has the properties of unique decodability and corruption detectability in the presence of mutation and insertion errors. We can then obtain a protocol A^{mi} which is just like A^{dm} except that occurrences of e^{dm} are replaced by e^{mi}.

We say a run of A^{dm} or A^{mi} is fair if there are infinitely many 4-blocks where all of S's transmissions are delivered uncorrupted, and infinitely many 4-blocks where both of R's transmissions are delivered uncorrupted. Again, using an implementation mapping, we can show the correctness of A^{dm} and A^{mi}. We state the theorem below, leaving details of the proof to the reader.

Theorem 5.2: *Protocol A^{dm} (resp. A^{mi}) solves the sequence transmission problem in the AUY model with deletion and mutation (resp., mutation and insertion) errors. Every run of A^{dm} (resp. A^{mi}) has the safety property, and the fair runs have the liveness property.*

We remark that we could deal with other error types too, provided we could find appropriate encoding functions. There is, however, no encoding function with the properties of unique decodability and corruption detectability that can deal simultaneously with deletion, mutation, and insertion errors.

Note that as we move further away from our knowledge-based protocol, we make more and more use of the details of the underlying model. In particular, A^{fs} makes heavy use of the assumption that messages are not reordered, and the protocols A^{di}, A^{dm}, and A^{mi} make heavy use of the synchronous nature of the AUY model, so that messages are received in the same round that they are sent (if they are received at all).

6 Another solution to the sequence transmission problem

In this section we construct another family of protocols that solves the sequence transmission problem.

Protocol A can be viewed as a *sender-driven* protocol: S sends x_i if S does not know that R knows x_i; i.e., S sends x_i when $\neg K_S K_R(x_{@i})$ holds. We can also consider *receiver-driven* protocols, where S sends x_i only if S knows that R does *not* know x_i; i.e., when $K_S \neg K_R(x_{@i})$ holds. Receiver-based protocols might be quite practical if S is relatively busy, or if S's messages are quite long rather than only consisting of one bit. Indeed, a receiver-based protocol is implemented as part of the NETBLT protocol at MIT [CLZ87].

It is easy to modify protocol A to get a receiver-driven protocol. Indeed, the tests for knowledge allow us to get to the heart of the problem immediately. We simply add a test to protocol A to ensure that S only sends $\langle i, y\rangle$ if $K_S \neg K_R(x_{@i})$ holds. The resulting knowledge-based protocol B is described in Figure 6 below. The formal semantics of B are essentially the same as those for A; we leave details to the reader.

We can prove that B has the safety property by applying the same arguments as for A. However, B in general does not have the liveness property, even in fair runs. To see the problem here, consider an asynchronous system where messages may take an arbitrary number of rounds to be delivered. Suppose that S has sent the message $\langle 0, 1\rangle$ once, and this message is not received by R. Without further assumptions on the system, there is no way for S ever to know that $K_S \neg K_R(x_0)$ holds. Even if S gets a message from R saying $\neg \mathtt{K_R(x_0)}$, it is not the case that $K_S \neg K_R(x_0)$ holds when the message is received, since it is possible that R received S's message $\langle 0, 1\rangle$ after it sent the $\neg \mathtt{K_R(x_0)}$ message. Although $K_R(x_{@i})$ is a stable formula (once true it remains true), $\neg K_R(x_{@i})$ is not.

B does have the liveness property in fair runs of systems where we make the additional assumption that messages have finite lifetimes, that is, systems where there is some constant T such that messages are delivered within time T if they are delivered at all. Note the AUY

S's protocol:

```
INIT: $i := 0$; read $y$
do forever
    if $\neg K_S K_R(x_{@i})$
        then if $K_S \neg K_R(x_{@i})$
                then send $\langle i, y \rangle$; receive $z$
                else receive $z$
        else $i := i + 1$; read $y$; receive $z$
end
```

R's protocol:

```
INIT: $j := 0$
do forever
    if $\neg K_R(x_j)$
        then send $\neg K_R(x_j)$; receive $z'$
        else if $K_R(x_j = 0)$
                then write 0; $j := j + 1$; receive $z'$
                else write 1; $j := j + 1$; receive $z'$
end
```

Figure 6: Protocol B

model is a special case of a system where messages have finite lifetimes; in this case the lifetime is one round.

If we further restrict to systems where messages can be deleted, duplicated, or detectably corrupted, but not reordered, we can obtain a finite state implementation B^{fs} of B analogous to A^{fs}. In general, the implementation requires a timer which keeps track of whether the lifetime of S's previous message has expired. We can avoid this complication in the AUY model, since the lifetime of a message is one round. For example, in Figure 7 we describe B^d, a straightforward implementation of protocol B in the AUY model with deletion errors only.

The idea behind B^d is quite simple. As in protocol A^{fs} and A^{di}, S uses a variable i to keep track of the parity of the data element it last sent, and R uses a variable j to keep track of the parity of the data element it next wants to receive. R keeps sending j to S (thus telling S the parity of the next data element it wants to receive). If S receives a non-λ value z, then it knows which data element R would currently like to know; if $z = i$, then R still doesn't know the value of the data element S is currently reading, so S sends it; if $z = i \oplus 1$, then R does know the value of the current data element, so S reads the next data element and sends it. If R receives a non-λ value z', then R knows that this is the value of the next data element, so R writes z'. We can easily find protocols B^{di}, B^{dm}, and B^{mi} that deal with any pair of error types in the AUY model by doing encoding just as in the previous section.

```
*S*'s **protocol:**

INIT: $z := \lambda$; $i := 0$; read $y$
do forever
    if $z \neq i \oplus 1$
      then if $z = i$
              then send $y$; receive $z$
              else receive $z$
      else $i := i \oplus 1$; read $y$; receive $z$
end

*R*'s **protocol:**

INIT: $z' := \lambda$; $j := 0$
do forever
    if $z \neq j$
      then send $j$; receive $z'$
      else write $z'$; $j := j \oplus 1$; receive $z'$
end
```

Figure 7: Protocol B^d

7 Conclusions

We have described high-level knowledge-based protocols for the sequence transmission problem, and have shown that several well-known protocols are simply implementations of one of our knowledge-based protocols. These observations allowed us to provide well-motivated and relatively straightforward correctness proofs for all these protocols. We feel that such an approach—starting with a knowledge-based protocol that is almost model-independent, and then implementing the knowledge acquisition using particular properties of the model—leads to a better understanding of the protocol and its correctness than other approaches that have been used. Although in the case of the Alternating Bit and Stenning's protocols (essentially protocols A^{st} and A^{fs}), a direct proof of correctness might be somewhat shorter than the proof we gave, given our overhead of first having to verify the knowledge-based protocol A, we believe that the understanding gained by using the knowledge-based protocol is itself of benefit. We expect that in other cases it will also result in shorter proofs. Certainly our proofs of correctness of the protocols of Aho, Ullman, and Yannakakis, obtained by viewing these protocols as an implementation of A^{fs}, seem substantially simpler than the others in the literature.

Although we feel that our correctness proofs are simpler than others, there is no question that they are still somewhat long and tedious. This seems to be an inescapable feature of formal correctness proofs. The advantage of the knowledge-based approach used here is that it at least allows the reader to obtain a clear high-level picture of what is going on, even if the low-level details are still tedious to check.

There are other advantages of the knowledge-based approach which are perhaps not so apparent from the analysis we have done here. In particular, it should be easier to develop correct protocols and change them to meet new requirements by starting with knowledge-based protocols, since this should make it clearer what the role of each command in the protocol is. As well, thinking at the knowledge level may often simplify matter. We remark that we discovered the receiver-driven protocol B by thinking at the knowledge level and trying to understand if we could replace the test $\neg K_S K_R(x_{@i})$ in S's protocol by $K_S \neg K_R(x_{@i})$.

The knowledge-based protocols we designed assumed processes with infinitely many distinct states and required that infinitely many distinct messages could be sent. In retrospect, this is perhaps not surprising. It is often the case that a high-level solution to a problem is inefficient. Protocols such as the AUY protocols and the Alternating Bit protocol show that under some circumstances (i.e., under appropriate restrictions on the actions of the environment) we can find finite-state solutions to the sequence transmission problem. However, as we observed at the end of Section 3, if we allow messages to be reordered and lost (or reordered and duplicated), and there is no upper bound on message delivery time (for messages that are delivered), then there is no finite-state solution. (We remark that if there is an upper bound on message delivery time, then the modified Stenning protocol [Ste76] gives a finite-state solution.) In fact, in [AFWZ88] it is shown that if we allow messages to be reordered and duplicated, then there is no solution using a finite message alphabet, and both [AFWZ88] and [LMF88] show that under most natural requirements, if we allow messages to be reordered and deleted, then there is no solution using a finite message alphabet.

If we allow undetectable corruption of messages (so that any message can be converted to any other), then there is no solution at all, either finite state or infinite state (since any message sequence can then be converted to any other). It would be interesting to characterize the assumptions required to guarantee that a solution to the sequence transmission problem exists, and the assumptions required to guarantee a finite-state solution, and then to generalize these results to the case where we have a whole network rather than just two processes (see [AG88] for preliminary work along these lines).

Finally, we should acknowledge that at this point the verification of protocols is still far more an art than a science. We do not have a general methodology for deriving a knowledge-based protocol, nor a general methodology for implementing it as a standard protocol once we have found it. Nevertheless, we feel that the knowledge-based approach will be an aid in designing and verifying protocols. Recent papers, such as [Had87, HMT88, HMW90, ML90, Maz90, NT93], that have used the knowledge-based approach lend credence to our feelings.

Acknowledgements: We would particularly like to thank Brent Hailpern, who started us thinking about the AUY protocols. We also thank Ron Fagin, Mike Fischer, Brent Hailpern, Yoni Malachi, Murray Mazer, Yoram Moses, and Moshe Vardi for their insightful comments on earlier drafts of this paper, Micah Beck for his thoughtful referee's report and for catching a subtle bug in an earlier presentation of protocol A, Eli Gafni and Laura Haas for pointing out references [GS80] and [SH86], Moshe Vardi for pointing out reference [BG77] (and the quote about the use of knowledge that appears there), and Josh Cohen Benaloh for LaTeX help.

References

[AFWZ88] C. Attiya, M. J. Fischer, D. Wang, and L. D. Zuck. Reliable communication using unreliable channels. Manuscript. 1988.

[AG88] Y. Afek and E. Gafni. End-to-end communication in unreliable networks. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 117–130, 1988.

[AUWY82] A. V. Aho, J. D. Ullman, A. D. Wyner, and M. Yannakakis. Bounds on the size and transmission rate of communication protocols. *Computers and Mathematics with Applications*, 8(3):205–214, 1982. This is a later version of [AUY79].

[AUY79] A. V. Aho, J. D. Ullman, and M. Yannakakis. Modeling communication protocols by automata. In *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pages 267–273, 1979.

[BG77] G. V. Bochmann and J. Gecsei. A unified method for the specification and verification of protocols. In B. Gilchrist, editor, *Information Processing 77*, pages 229–234. North-Holland, Amsterdam, 1977.

[BS80] G. V. Bochmann and C. A. Sunshine. Formal methods in communication protocol design. *IEEE Transactions on Communications*, COM-28:624–631, 1980.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12:260–261, 1969.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986. An early version appeared in *Proc. 10th ACM Symposium on Principles of Programming Languages*, 1983.

[CLZ87] D. D. Clark, M. L. Lambert, and L. Zhang. NETBLT: a high throughput transport protocol. In *SigComm 87 Workshop*, pages 353–359. 1987. Proceedings published in *Computer Communication Review* **17**:5.

[CM86] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[DM90] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: crash failures. *Information and Computation*, 88(2):156–186, 1990.

[FHV92] R. Fagin, J. Y. Halpern, and M. Y. Vardi. What can machines know? On the properties of knowledge in distributed systems. *Journal of the ACM*, 39(2):328–376, 1992.

[FI86] M. J. Fischer and N. Immerman. Foundations of knowledge for distributed systems. In J. Y. Halpern, editor, *Theoretical Aspects of Reasoning about Knowledge:*

Proc. 1986 Conference, pages 171–186. Morgan Kaufmann, San Francisco, Calif., 1986.

[Fit91] M. Fitting. Modal logic should say more than it does. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 113–135. MIT Press, Cambridge, Mass., 1991.

[Gaf86] E. Gafni. Perspectives on distributed network protocols: a case for building blocks. In *Proc. IEEE MILCOM '86*, 1986.

[GH91] A. J. Grove and J. Y. Halpern. Naming and identity in a multi-agent epistemic logic. In J. A. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proc. Second International Conference (KR '91)*, pages 301–312. Morgan Kaufmann, San Francisco, Calif., 1991.

[Gou85] M. Gouda. On "A simple protocol whose proof isn't". *IEEE Transactions on Communications*, COM-33(4):382–384, 1985.

[GS80] V. D. Gligor and S. H. Shattuck. On deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, SE-6(5):435–440, 1980.

[Had87] V. Hadzilacos. A knowledge-theoretic analysis of atomic commitment protocols. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pages 129–134, 1987. A revised version has been submitted for publication.

[Hai85] B. T. Hailpern. A simple protocol whose proof isn't. *IEEE Transactions on Communications*, COM-33(4):330–337, 1985.

[Hal87] J. Y. Halpern. Using reasoning about knowledge to analyze distributed systems. In J. F. Traub, B. J. Grosz, B. W. Lampson, and N. J. Nilsson, editors, *Annual Review of Computer Science, Vol. 2*, pages 37–68. Annual Reviews Inc., Palo Alto, Calif., 1987.

[HF85] J. Y. Halpern and R. Fagin. A formal model of knowledge, action, and communication in distributed systems: preliminary report. In *Proc. 4th ACM Symp. on Principles of Distributed Computing*, pages 224–236, 1985.

[HF89] J. Y. Halpern and R. Fagin. Modelling knowledge and action in distributed systems. *Distributed Computing*, 3(4):159–179, 1989. A preliminary version appeared in *Proc. 4th ACM Symposium on Principles of Distributed Computing*, 1985, with the title "A formal model of knowledge, action, and communication in distributed systems: preliminary report".

[HM90] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990. A preliminary version appeared in *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 1984.

[HMT88] J. Y. Halpern, Y. Moses, and M. R. Tuttle. A knowledge-based analysis of zero knowledge. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 132–147, 1988.

[HMW90] J. Y. Halpern, Y. Moses, and O. Waarts. A characterization of eventual Byzantine agreement. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pages 333–346, 1990.

[HO83] B. T. Hailpern and S. S. Owicki. Modular verification of communication protocols. *IEEE Transactions on Communications*, COM-31(1):56–68, 1983.

[HV86] J. Y. Halpern and M. Y. Vardi. The complexity of reasoning about knowledge and time. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 304–315, 1986.

[LMF88] N. A. Lynch, Y. Mansour, and A. Fekete. Data link layer: two impossibility results. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 149–170, 1988.

[LR86] R. E. Ladner and J. H. Reif. The logic of distributed protocols (preliminary report). In J. Y. Halpern, editor, *Theoretical Aspects of Reasoning about Knowledge: Proc. 1986 Conference*, pages 207–222. Morgan Kaufmann, San Francisco, Calif., 1986.

[LT89] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989. Also available as MIT Technical Memo MIT/LCS/TM-373.

[Maz90] M. S. Mazer. A link between knowledge and communication in faulty distributed systems. In R. Parikh, editor, *Theoretical Aspects of Reasoning about Knowledge: Proc. Third Conference*, pages 289–304. Morgan Kaufmann, San Francisco, Calif., 1990.

[Mer76] P. M. Merlin. A methodology for the design and implementation of communication protocols. *IEEE Transactions on Communications*, COM-24(4):614–621, 1976.

[ML90] M. S. Mazer and F. H. Lochovsky. Analyzing distributed commitment by reasoning about knowledge. Technical Report CRL 90/10, DEC-CRL, 1990.

[Mos86] Y. Moses. *Knowledge in a distributed environment.* PhD thesis, Stanford University, 1986.

[MT88] Y. Moses and M. R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:121–169, 1988.

[NT93] G. Neiger and S. Toueg. Simulating real-time clocks and common knowledge in distributed systems. *Journal of the ACM*, 40(2):334–367, 1993.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. on Programming Languages and Systems*, 4(3):455–495, 1982.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pages 46–57, 1977.

[PR85] R. Parikh and R. Ramanujam. Distributed processing and the logic of knowledge. In R. Parikh, editor, *Proc. Workshop on Logics of Programs*, pages 256–268, 1985.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, Lecture Notes in Computer Science, Vol. 137, pages 337–371. Springer-Verlag, Berlin/New York, 1982.

[SH86] S. R. Soloway and P. A. Humblet. On distributed network protocols for changing topologies. Technical Report LIDS-P-1564, MIT, 1986.

[SMS82] R. L. Schwartz and P. M. Melliar-Smith. From state machines to temporal logic: specification methods for protocol standards. *IEEE Transactions on Communications*, COM-30(12):2486–2496, 1982.

[Ste76] M. V. Stenning. A data transfer protocol. *Comput. Networks*, 1:99–110, 1976.

[Sun79] C. A. Sunshine. Formal techniques for protocol specification and verification. *IEEE Computer*, 12:20–27, 1979.