Kermit: Specification and Verification*

James K. Huggins†

July 13, 1995

Kermit is a popular communication protocol. We formally specify Kermit and verify it. As far as we know, this has not been done yet, though the alternating bit and sliding window protocols, used by Kermit, have been specified and verified by many authors [Kr, SL, LM]. Our main goal is a faithful readable specification which allows one to formalize the intuitive verification proof without much overhead.

In his foreword to [DaC], Donald Knuth writes "I hope that many readers of this book will be challenged to find high-level concepts and invariant relations by which various versions of the Kermit protocol can be proved correct in a mathematical sense." We believe our specification and verification meets this challenge.

We use the evolving algebra approach. Section 1 gives a self-contained introduction to evolving algebras; a fuller definition can be found in [Gur]. We use the term "ealgebra" (read e-algebra) as an abbreviation for "evolving algebra". We begin with ealgebra specifications and verifications of two more abstract communications protocols used by various versions of Kermit: the alternating bit protocol and the sliding window protocol. The nice feature of the ealgebra approach is that the road from an intuitive proof to a precise one is very short; there is little overhead. Then we will present a series of ealgebras for the Kermit protocol, filling in the pieces where necessary to show how Kermit uses the abstract protocols.

As usual with protocols, we prove theorems dealing with properties of safety ("bad things don't happen") and liveness ("good things do happen"). Our safety theorems are of the form "Every state reachable in any relevant run satisfies property Φ" and are proved by induction on relevant runs. It is usually obvious that relevant initial states satisfy Φ; more work is required to show that the transition rules preserve Φ. The liveness theorems have the form "Every fair run has such and such property."

Acknowledgments. Yuri Gurevich directed this study; his comments throughout its development were numerous and extremely helpful. Frank da Cruz made useful comments on a later draft.

1 Evolving Algebras

An ealgebra $\mathcal{A}$ is an abstract machine. The *signature* of $\mathcal{A}$ is a (finite) collection of function names, each of a fixed arity, that are used to describe $\mathcal{A}$. A state of $\mathcal{A}$ is a set, the *superuniverse*, together with interpretations of the function names in the signature. The superuniverse does not change as $\mathcal{A}$ evolves; the interpretations of the functions may.

Formally, a function of arity r (*i.e.* the interpretation of a function name of arity r) is an r-ary operation on the superuniverse. We often use functions with $r = 0$; such functions will be called *distinguished elements*. Examples of such distingushed elements are *true*, *false*, and *undef*, which appear in every ealgebra and are known as *logical names*. Functions may be defined only on a part of the superuniverse; such partial functions are represented by total functions where $f(\overline{a}) = undef$ means f is undefined at the tuple $\overline{a}$. Relations are represented as Boolean-valued functions.

A *universe* U is a special type of function: a unary relation usually identified with the set $\{x : U(x)\}$. The universe $Bool = \{true, false\}$ is an example (in fact, *Bool* is a logical name appearing in every ealgebra).

*Originally in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995. Reprinted by permission of Oxford University Press.

†EECS Department, University of Michigan, Ann Arbor, MI 48109-2122, huggins@umich.edu. Partially supported by ONR grant N00014-91-J-1861 and NSF grant CCR-92-04742.

When we speak about a function f from a universe U to a universe V, we mean that formally f is a unary operation on the superuniverse such that $f(a) \in V$ for all $a \in U$ and $f(a) = \mathit{undef}$ otherwise. We use self-explanatory notations like $f : U \rightarrow V$, $f : U_1 \times U_2 \rightarrow V$, and $f : V$. The last means that the distinguished element f belongs to V.

An *external function* is also a special type of funciton: its values are determined outside of the scope of the algebra, and may change from state to state without being updated by the ealgebra itself.

We define *transition rules* recursively:

- An *update instruction* is a transition rule and has the form

$$f(t_1,\ \ldots,\ t_r) := t_{r+1}$$

  where each t_i is a closed term (*i.e.* a term containing no free variables) in the signature of $\mathcal{A}$. The meaning of the instruction is the obvious one: Evaluate all terms t_i in the given state, and change the value of f at tuple $(t_1, \ldots, t_r)$ to the value of t_{r+1}.

- A *guarded rule* is a transition rule and has the form

$$\begin{array}{l} \textbf{\textit{if}}\ t_0\ \textbf{\textit{then}}\ R_0 \\ \textbf{\textit{elseif}}\ t_1\ \textbf{\textit{then}}\ R_1 \\ \vdots \\ \textbf{\textit{elseif}}\ t_k\ \textbf{\textit{then}}\ R_k \\ \textbf{\textit{endif}} \end{array}$$

  where each t_i is a closed Boolean-valued term and each R_i is a rule. The meaning of the instruction is the obvious one: execute R_j where j is the smallest value for which t_j is true in the given state. If no such j exists, do nothing.

- A *block* (or sequence) of transition rules is a transition rule. To execute a block, execute all rules in the block simultaneously. If no two rules conflict (*i.e.* attempt to modify the same tuple of the same function), the effect of executing the sequence is the union of the effects of executing each rule individually. If two rules conflict, no updates are made and the algebra halts.

A distributed evolving algebra $\mathcal{A}$ consists of a set of *agents*, each of which has a corresponding *program* (*i.e.* a transition rule). We call a function *common* if all agents interpret the function name in the same way; *private* functions may be interpreted differently by different agents.

A *sequential run* is a sequence of states; each state (except for its external functions) is obtained from its predecessor by firing some collection of non-conflicting agents simultaneously.

2 The Alternating Bit Protocol (ABP)

The alternating bit protocol (ABP), first proposed in [BSW], is a simple protocol at the heart of many communication protocols currently in use. Each agent participating in the protocol (we will call them "sender" and "receiver", though both agents send and receive messages) has a private synchronization bit. The sender has data which she sends to the receiver. The receiver sends an acknowledgment message for each datum received. Messages are marked with a synchronization bit to distinguish between data which has and has not been received by the receiver.

We present a generalized version of the ABP which sends an infinite sequence of data between the two agents and prove its correctness, using sequential runs throughout. An ABP which sends a finite sequence of data and terminates can easily be developed from our more general version.

As an aid to comprehension, we use feminine pronouns to refer to the sender and masculine pronouns to refer to the receiver.

2.1 Function Descriptions

2.1.1 Common Functions

Our agents send messages, each comprising a datum and a synchronization bit, through a network. This leads to universes of *messages*, *data*, and *bits*. We represent each communication path between agents as a queue of messages; this leads to a universe of *queues*. (Note that by "queue" we mean the abstract datatype and not a particular physical device.) We also make use of the universe of *integers*.

We use the functions *Msg: data* $\times$ *bits* $\rightarrow$ *messages*, *Bit: messages* $\rightarrow$ *bits* and *Data: messages* $\rightarrow$ *data* to compose and decompose messages, respectively. (That is, if *Msg(d,b)* = m, then *Bit(m)* = b and *Data(m)* = d.) The distinguished element *Null: data* is a datum used as a placeholder for acknowledgment messages (in which the only important information is the bit, not the datum). The function *Flip: bits* $\rightarrow$ *bits* flips a bit to its opposite value. That is, *Flip(0)* = *1* and *Flip(1)* = *0*.

For *queues*, we use the functions *Append: queues* $\times$ *messages* $\rightarrow$ *queues*, *Head: queues* $\rightarrow$ *messages*, and *Tail: queues* $\rightarrow$ *queues* with the obvious meanings. The distinguished element *EmptyQueue: queues* is a queue containing no messages. We denote *Append(a,b)* by a ++ b. An external function *Shrink: queues* $\rightarrow$ *queues* returns a queue obtain by deleting zero or more messages from the input queue, essentially at random, while maintaining the relative ordering of the remaining messages.

For *integers*, we use the standard infix addition function +, as well as the constant 1.

The function *Timeout: Bool* is used to generate re-transmissions of messages.

2.1.2 Private Functions

Each agent has a distinguished element *SenderInMsg, ReceiverInMsg: messages* which holds the current message being processed by the agent, and a distinguished element *SenderBit, ReceiverBit: bits* which holds the agent's synchronization bit. Distinguished elements *SenderQueue, ReceiverQueue: queues* store messages waiting to be processed.

The data that the sender sends is stored in a function *SenderFile: integers* $\rightarrow$ *data*; the receiver stores all data received in a corresponding function *ReceiverFile: integers* $\rightarrow$ *data*. The distinguished functions *SenderNo, ReceiverNo: integers* indicate the current datum being sent or received.

2.2 Module Specifications

We use abbreviations *Defined(t), Undefined(t), Clear(t)* for $t \neq undef$, $t = undef$ and $t := undef$ respectively; here t is a term.

The sender examines each acknowledgment message sent to her by the receiver. If the sender receives an acknowledgment whose bit matches her synchronization bit, she knows that her last message arrived successfully, and she can now send a new message (with a new bit). Any messages received whose bit does not match her synchronization bit are discarded.

To insure against message loss due to an unreliable network, the sender also re-sends her last message when a timeout signal occurs. We also use this behavior to begin the communication process; during any run, no transition rules will fire until a timeout signal occurs, at which point the sender will send her first message.

The receiver acknowledges every message he receives from the sender by re-transmitting the bit from the received message. Additionally, if the bit received matches his synchronization bit, the receiver records the datum from the message in his output file, and updates his file marker and synchronization bit to be ready to accept the next datum from the sender. The sender and receiver modules are given in Fig. 1.

Agents communicate by placing messages into queues (*SenderQueue, ReceiverQueue*) and by reading messages placed into reception variables (*SenderInMsg, ReceiverInMsg*). Two communications modules, shown in Fig. 2, transfer messages between these queues and reception variables.

If the communications network were reliable, no further modules would be needed. However, messages may be lost, although not corrupted or re-ordered. As a result, we need to generate timeout signals to enable

Module: Sender
Rule: ReTransmit
if *Timeout* ***then***
    ReceiverQueue :=
        ReceiverQueue ++ Msg(SenderFile(SenderNo),SenderBit)
    Timeout := false
endif

Rule: ProcessAck
if *Defined(SenderInMsg)* ***and*** *Bit(SenderInMsg) = SenderBit* ***then***
    ReceiverQueue :=
      ReceiverQueue ++ Msg(SenderFile(SenderNo+1),Flip(SenderBit))
    SenderBit := Flip(SenderBit), SenderNo := SenderNo + 1
endif

Rule: ClearMessage
if *Defined(SenderInMsg)* ***then*** *Clear(SenderInMsg)* ***endif***

Module: Receiver
Rule: AcceptDatum
if *Defined(ReceiverInMsg)* ***and*** *Bit(ReceiverInMsg) = ReceiverBit* ***then***
    ReceiverFile(ReceiverNo) := Data(ReceiverInMsg)
    ReceiverNo := ReceiverNo + 1, ReceiverBit := Flip(ReceiverBit)
endif

Rule: AcknowledgeMessage
if *Defined(ReceiverInMsg)* ***then***
    SenderQueue := SenderQueue ++ Msg(Null,Bit(ReceiverInMsg))
    Clear(ReceiverInMsg)
endif

Figure 1: ABP sender and receiver modules.

Module: SenderCommunicate
if *Undefined(SenderInMsg)* ***and*** *SenderQueue* $\neq$*EmptyQueue* ***then***
  SenderInMsg := Head(SenderQueue)
  SenderQueue := Tail(SenderQueue)
endif

Module: ReceiverCommunicate
if *Undefined(ReceiverInMsg)* ***and*** *ReceiverQueue* $\neq$*EmptyQueue* ***then***
  ReceiverInMsg := Head(ReceiverQueue)
  ReceiverQueue := Tail(ReceiverQueue)
endif

Figure 2: ABP sender and receiver communication modules.

the sender to re-transmit messages. We present three modules which describe this behavior in Fig. 3.

Module: SenderLoseMessage
SenderQueue := Shrink(SenderQueue) ***endif***

Module: ReceiverLoseMessage
ReceiverQueue := Shrink(ReceiverQueue) ***endif***

Module: Timeout
if *SenderQueue = ReceiverQueue = EmptyQueue*
  and *SenderInMsg =ReceiverInMsg = undef* ***then***
  Timeout := true
endif

Figure 3: Network loss and timeout modules.

2.3 Run Definitions

We now wish to prove the correctness of the ABP. We cannot prove that any of the protocols to be discussed here are correct for all runs; most protocols assume, for example, that both agents satisfy some initial conditions. Consequently, we restrict our attention to certain types of runs. Call a run ρ *regular* if its initial state satisfies a specified set of initial conditions. The initial conditions for the ABP are shown in Fig. 4. Our safety properties will be proved over regular runs.

Communication may not be possible even in a regular run. The sender might never receive a timeout when one is needed, or a communications module might be too active, throwing away all messages sent by one agent. The latter corresponds to the real-world situation where the underlying communications medium breaks down; obviously, no protocol can succeed under those conditions. Thus, we must make certain minimal assumptions about the underlying medium in order to complete our proofs. Essentially, we wish to exclude the “unfair” conditions described above.

$\forall x \geq 0$ *ReceiverFile(x) = undef*	*SenderNo = ReceiverNo = 0*
SenderInMsg = ReceiverInMsg = undef	*SenderBit = ReceiverBit*
ReceiverQueue = SenderQueue = EmptyQueue	

Figure 4: Initial conditions for the ABP.

For our purposes, it seems sufficient to require only that certain agents (which we will call *positive* agents) cannot be prohibited from making a move indefinitely. We consequently define an infinite sequential run ρ to be *fair* if for every positive agent X and every tail ρ' of ρ, if X is enabled infinitely often in ρ', then X must make a move in ρ'. For the ABP, the only agents which are not positive are SenderLoseMessage and ReceiverLoseMessage. Our liveness properties will be proved for fair runs.

2.4 Proof of Correctness

We begin with a few notational definitions:

- *SQ* (respectively, *RQ*) is the sequence of messages in *SenderQueue* (*ReceiverQueue*).
- *SIM* (respectively, *RIM*) is *SenderInMsg* (*ReceiverInMsg*) except when the latter is undefined, when it is the empty sequence.
- *Bit(S)*, where S is a message sequence, is the natural extension of the *Bit* function from messages to sequences of messages.
- $+$ denotes concatenation of message sequences.

Most of our invariant conditions are proved by induction over the number of moves in a run; we will simply say "by induction" in such cases.

Lemma 1 *In any reachable state, $Bit(SIM + SQ + RIM + RQ)$ has the form $(Flip(SenderBit))^*(SenderBit)^*$.*

Proof. By induction. Initially, the specified bit sequence is empty. We consider all moves that affect functions present in the invariant.

ClearMessage empties *SenderInMsg*, eliminating the first bit in the bit sequence. This does not affect the truth of the invariant.

ReTransmit appends a copy of *SenderBit* to *ReceiverQueue*, maintaining the invariant.

ProcessAck flips *SenderBit* and appends a copy of the new value of *SenderBit* to *ReceiverQueue*. Since *Bit(SenderInMsg) = SenderBit* when ProcessAck fires, we know (by the inductive hypothesis) that all bits of the bit sequence must be copies of *SenderBit*. Thus, after ProcessAck completes, the bit sequence will have the form $((Flip(SenderBit))^x\ SenderBit)$ for some x, satisfying the invariant.

AcknowledgeMessage, SenderCommunicate, ReceiverCommunicate, SenderLoseMessage, and ReceiverLoseMessage transfer bits between or remove bits from various functions but do not alter the order of any of the bits in the bit sequence, preserving the invariant. □

Lemma 2 *In any reachable state in which SenderInMsg or SenderQueue contains a message μ such that $Bit(\mu)$ = SenderBit, SenderBit = Flip(ReceiverBit).*

Proof. By induction. Initially, both *SenderInMsg* and *SenderQueue* are empty. We consider all moves that affect functions present in the invariant.

ProcessAck flips *SenderBit*. Lemma 1 shows that all bits in *SenderQueue* will be copies of *Flip(SenderBit)* after ProcessAck fires.

AcknowledgeMessage places a copy of *Bit(ReceiverInMsg)* into *SenderQueue*. If this new bit is *Flip(SenderBit)*, then (by Lemma 1) the bit sequence of *SenderQueue* has the form $Flip(SenderBit)^*$ and the invariant is maintained.

If not, then *Bit(ReceiverInMsg) = SenderBit*. If *SenderBit = ReceiverBit* at this time, then rule AcceptDatum must also fire, flipping *ReceiverBit* and yielding *SenderBit = Flip(ReceiverBit)*. Otherwise, we have *SenderBit = Flip(ReceiverBit)* both before and after AcknowledgeMessage fires, maintaining the invariant.

ClearMessage, SenderCommunicate, and SenderLoseMessage remove messages from or transfer messages between *SenderQueue* and *SenderInMsg*, which does not affect the invariant. □

Lemma 3 *In any reachable state, if either ReceiverInMsg or ReceiverQueue contains a message μ such that $Bit(\mu)$ = Flip(SenderBit), then SenderBit = ReceiverBit.*

Proof. By induction. Initially, both *ReceiverInMsg* and *ReceiverQueue* are empty. We consider all moves that affect functions present in the invariant.

ProcessAck flips *SenderBit*. ProcessAck implies that *Bit(SenderInMsg) = SenderBit*. By Lemma 2, *SenderBit = Flip(ReceiverBit)*. Thus, flipping *SenderBit* yields *SenderBit = ReceiverBit*, maintaining the invariant.

ReTransmit appends a message with a copy of *SenderBit* to *ReceiverQueue*, which does not affect the truth of the invariant.

AcceptDatum flips *ReceiverBit*. AcceptDatum implies that *Bit (ReceiverInMsg) = ReceiverBit*. If *Bit(ReceiverInMsg) = SenderBit*, all of the messages in *ReceiverQueue* have copies of *SenderBit* (by Lemma 1), and flipping *ReceiverBit* does not affect the invariant.

Otherwise, *Bit(ReceiverInMsg) = Flip(SenderBit)*, which implies that *Flip(SenderBit) = ReceiverBit*. But this cannot occur: the induction hypothesis implies that *SenderBit = ReceiverBit*.

Modules AcknowledgeMessage, ReceiverCommunicate, and ReceiverLoseMessage remove messages from or transfer messages between *ReceiverQueue* and *ReceiverInMsg*, which does not affect the invariant. □

Lemma 4 *In any reachable state, the following are true:*

$$SenderBit = ReceiverBit \rightarrow SenderNo = ReceiverNo,$$

$$SenderBit = Flip(ReceiverBit) \rightarrow SenderNo + 1 = ReceiverNo.$$

Proof. By induction. Initially, *SenderBit = ReceiverBit* and *SenderNo = ReceiverNo=0*. We consider all moves affecting functions in the invariant.

ProcessAck flips *SenderBit* and increments *SenderNo*. By Lemma 2, ProcessAck only fires when *SenderBit = Flip(ReceiverBit)*; by the induction hypothesis, *SenderNo + 1 = ReceiverNo*. Flipping *SenderBit* and incrementing *SenderNo* yields the other condition.

AcceptDatum flips *ReceiverBit* and increments *ReceiverNo*. By Lemma 3, AcceptDatum only fires when *SenderBit = ReceiverBit*; by the induction hypothesis, *SenderNo = ReceiverNo*. Flipping *ReceiverBit* and incrementing *ReceiverNo* yields to the other condition. □

Lemma 5 *In any reachable state, for any message μ contained in either ReceiverInMsg or ReceiverQueue, $Bit(\mu)$ = SenderBit $\rightarrow$ $Data(\mu)$ = SenderFile(SenderNo), and $Bit(\mu)$ = Flip(SenderBit) $\rightarrow$ $Data(\mu)$ = SenderFile(SenderNo−1).*

Proof. By induction. Initially, no messages exist in *ReceiverInMsg* or *ReceiverQueue*. We consider all moves that affect functions present in the invariant.

ProcessAck flips *SenderBit* and increments *SenderNo*. We must have *Bit(SenderInMsg) = SenderBit* in order for ProcessAck to fire; by Lemma 1, all messages μ under consideration have *Bit*(μ) = *SenderBit*. By the induction hypothesis, we also have *Data*(μ) = *SenderFile(SenderNo)*. Incrementing *SenderNo* and flipping *SenderBit* thus results in *Bit*(μ) = *Flip(SenderBit)* and *Data*(μ) = *SenderFile(SenderNo−1)*, as desired.

Additionally, a new message μ' is appended to *ReceiverQueue*. After ProcessAck fires, we will have *Bit*(μ') = *SenderBit* and *Data*(μ') = *SenderFile(SenderNo)*, as desired.

ReTransmit appends a message containing *SenderFile(SenderNo)* and *SenderBit* to *ReceiverQueue*, maintaining the invariant.

AcknowledgeMessage, SenderCommunicate, and ReceiverLoseMessage remove messages from or transfer messages between *ReceiverInMsg* or *ReceiverQueue*, which does not affect the invariant. □

Theorem 1 *In any reachable state, Defined(ReceiverFile(x))* $\rightarrow$ *ReceiverFile(x) = SenderFile(x). That is, any data that has been accepted by the receiver is stored in the correct order.*

Proof. By induction. Fix an x. Initially, *ReceiverFile(x) = undef*.

Only AcceptDatum may change *ReceiverFile(x)*. By Lemma 3, we know that *SenderBit = ReceiverBit* in this state; Lemma 4 tells us further that *SenderNo = ReceiverNo*.

Since *Bit(ReceiverInMsg) = SenderBit*, we know that *Data(ReceiverInMsg) = SenderFile(SenderNo)* (by Lemma 5). AcceptDatum will thus assign *ReceiverFile(ReceiverNo) := SenderFile(SenderNo)*. □

Lemma 6 *In any reachable state, ReceiverNo* $> 0 \rightarrow$ *Defined(ReceiverFile(ReceiverNo−1)).*

Proof. By induction. Initially *ReceiverNo = 0*; *ReceiverNo* is incremented precisely when *ReceiverFile (ReceiverNo)* is modified by AcceptDatum. □

Lemma 7 *In the future of any state of a fair run, the first message of SIM+SQ (or RIM+RQ) will eventually be removed.*

Proof. The proof is similar for both cases; we present the case of *SenderInMsg* and *SenderQueue* here.

Suppose we have a state in a fair run where *SenderInMsg* or *SenderQueue* (or both) is not empty. If *SenderInMsg* is not empty, rule ClearMessage is enabled. ClearMessage must eventually fire (by fairness), emptying *SenderInMsg* (which satisfies the lemma).

If *SenderQueue* is not empty, two modules are enabled: SenderCommunicate and SenderLoseMessage. If SenderLoseMessage fires, it may remove the first message in *SenderQueue*, satisfying the lemma. If not, SenderCommunicate will be continuously enabled, and by fairness must eventually fire, transferring the first message of *SenderQueue* into *SenderInMsg*, where it will eventually be emptied (as shown above). □

Theorem 2 *In any fair run, any data sent is eventually received.*

Proof. For any fair run, consider a particular datum being sent by the sender; the item corresponds to a particular value of *SenderNo* (say x). Consider the states in this run where *SenderNo* = x. If at least one of those states also has *SenderBit = Flip(ReceiverBit)*, by Lemmas 4, 6, and Theorem 1, *SenderFile(x)* has already been stored in *ReceiverFile(x)*. Thus, it remains to show that from any state in a fair run where *SenderBit = ReceiverBit* and *SenderNo* = x, we eventually arrive at a state where *SenderBit = Flip(ReceiverBit)* and *SenderNo* = x.

By contradiction, assume that we have *SenderBit = ReceiverBit* for every state in which *SenderNo* = x. Since *SenderNo* is incremented (by rule ProcessAck) only when *Bit(SenderInMsg) = SenderBit*, Lemma 2 tells us that rule ProcessAck will never fire, leaving *SenderNo* and *SenderBit* unchanged for the rest of the run.

Consider *SIM+SQ*. By Lemma 7, the first message in the non-empty sequence *SIM + SQ + RIM + RQ* will be removed infinitely often. Since *SenderBit* never changes, and rule ReTransmit only sends messages with copies of *SenderBit*, eventually all messages with *Flip(SenderBit)* in the system will be removed, leaving only messages with *SenderBit*. By Lemma 2, all copies of *SenderBit* must lie in *ReceiverInMsg* or *ReceiverQueue*, so *SenderInMsg* and *SenderQueue* will be empty for the duration of the run.

The only rule which can now create new messages is ReTransmit, which requires *Timeout = true*. Since messages are always being removed from $RIM + RQ$, if no copy of the desired message reaches the receiver, eventually *ReceiverInMsg* and *ReceiverQueue* will be empty, continuously enabling module Timeout. By fairness, Timeout will eventually fire, enabling rule ReTransmit. ReTransmit will eventually fire (by fairness), placing another copy of *SenderBit* into *ReceiverQueue*. If this new copy of *SenderBit* is discarded, Timeout will be re-enabled and by fairness will re-transmit the message. A repetition of the previous argument shows that if this message is repeatedly discarded, it will be re-transmitted infinitely often.

Thus, rule SenderCommunicate is enabled infinitely often, and by fairness must fire. So, eventually *Bit(ReceiverInMsg) = SenderBit = ReceiverBit*, and rule AcceptDatum will accept the datum and flip *ReceiverBit*, yielding *SenderBit = Flip(ReceiverBit)* as desired. □

Theorem 3 *In any fair run, for any data sent, a corresponding acknowledgment arrives at the sender.*

Proof. The proof is similar to that for Theorem 2, with the focus of attention on *ReceiverQueue* instead of *SenderQueue*. The major difference is that the receiver does not re-transmit acknowledgment messages when signaled by *Timeout*, as the sender does, but must wait for another message from the sender. A simple extension of the proof in Theorem 2 shows that the sender will transmit infinitely many copies of a given message until an acknowledgment is received, and thus infinitely many copies of that message will be received by the receiver, who will thus be enabled to send the corresponding acknowledgment message infinitely often. □

Theorem 4 *In any fair run, all data is eventually sent and acknowledged.*

Proof. By induction over the number of data sent and acknowledged, represented by *SenderNo*. If *SenderNo = 0*, Theorem 3 shows that this datum is eventually sent and acknowledged.

Suppose *SenderFile(n)* has been sent. When an acknowledgment for *SenderFile(n)* arrives at the sender (assured by the induction hypothesis), ProcessAck is enabled and (by fairness) will eventually fire, incrementing *SenderNo* and sending a copy of *SenderFile(n+1)* to the receiver. Theorem 3 shows that this datum will eventually be accepted and successfully acknowledged as well. □

3 Symmetric ABP

Our version of the ABP is designed for two agents, each of which is described by a different module. In most real communication protocols, such as Kermit, the roles of sender and receiver may be interchanged by two agents during the course of a communication session. (That is, Alice may send a file to Bob, but Bob may send a file to Alice afterwards.)

We present another version of the ABP in which both agents are represented by module templates containing identical transition rules. Our rules for Kermit contain modules which are similar to this symmetric ABP; we present this version as a transition between the classic ABP presented earlier and the full Kermit descriptions to come.

3.1 Function Descriptions

As before, we use the universes of *messages*, *queues*, *data*, and *integers*. We use *integers* instead of *bits* as the second component of each message; we thus define static functions *Data: messages* $\rightarrow$ *data*, *Num: messages* $\rightarrow$ *integers*, and *Msg: data* $\times$ *integers* $\rightarrow$ *messages* in a similar manner to that shown previously. The static functions *Null*, *EmptyQueue*, *Append*, *Head*, and *Tail* are unchanged.

A universe of *ids* contains two elements corresponding to the sender and receiver. The static functions *Sender: ids* and *Receiver: ids* indicate which agent corresponds to which identifier; the private static functions *Me: ids* and *You: ids* identify to each module/agent his or her own identifier, with the obvious requirements on their values.

Each module has a private function *File: integers* $\rightarrow$ *data*, used to store data being sent or received, and a private function *MyNum: integers*, used to denote the current location within *File*. A private function *LastMsg: messages* is used to store the last message sent by each agent. A function *Timeout: ids* $\rightarrow$ *Bool* holds the timeout signal location for each agent.

Common functions *Q: ids* $\rightarrow$ *queues* and *InMsg: ids* $\rightarrow$ *message* represent the two message queues and incoming message variables, respectively.

It proves convenient for later purposes to separate the input and output activities of each agent; we thus define a universe *tags* = {*Get, Put*} and a private function *Mode: tags* to distinguish between these activities.

3.2 Module Specifications

The sender and receiver module is given in Fig. 5. The communications modules, shown in Fig. 6, are similar to those given earlier. The only unfair modules are those produced by the LoseMessage template.

The initial state for the symmetric ABP satisfies the conditions shown in Fig. 7, where *Sender.X* and *Receiver.X* refer to the values of the private function *X* as seen by the sender and receiver, respectively.

3.3 Correctness

The proof of correctness for the symmetric ABP is similar to that of the non-symmetric ABP presented earlier. Rather than repeat the proof, we instead explain the similarities between the two protocols. The reader should be able to reconstruct our proof of correctness without difficulty.

Many expressions have different names in the two protocols; we present a table of equivalent expressions below. It is easy to verify that each pair of expressions yield elements of the same universe (or are of the same "type"):

$$\begin{array}{lcl}
SenderQueue/ReceiverQueue & \equiv & \mathrm{Q}(Sender)/Q(Receiver) \\
SenderInMsg & \equiv & InMsg(Sender) \\
ReceiverInMsg & \equiv & InMsg(Receiver) \\
SenderFile/ReceiverFile & \equiv & File \\
SenderNo/ReceiverNo & \equiv & MyNum \\
SenderBit/ReceiverBit & \equiv & MyNum \ mod \ 2
\end{array}$$

We have replaced *SenderBit* and *ReceiverBit* by references to *MyNum mod 2*. In the old ABP, *SenderBit* is flipped precisely when *SenderNo* is incremented; thus, if initially we have *SenderBit* = *SenderNo* = *0*, *SenderBit* will always be equal to *SenderNo mod 2*. A similar argument holds for *ReceiverBit* and *ReceiverNo*.

LastMsg stores the last message sent by an agent to be used later in re-transmission. *LastMsg* does not appear in the old ABP; thus, we must show that whenever the sender of the symmetric ABP sends a copy of *LastMsg*, the sender of the old ABP sends an identical message.

For the sender agent, *Num(LastMsg)* = *MyNum mod 2* is an invariant; This is easily proved: *MyNum* is incremented precisely when *LastMsg* is updated, and the updates for *LastMsg* yield the invariant condition immediately. Similarly, it can be seen that *Data(LastMsg)* = *File(MyNum)* is an invariant for the sender agent. The sender of the symmetric ABP sends a copy of *LastMsg* precisely when the sender of the old ABP sends a copy of *Msg(SenderFile(SenderNo),SenderBit)*; our argument shows that these messages are identical.

The reader can easily verify that each move of the old ABP is duplicated by one or two moves of the symmetric ABP. The symmetric ABP differs in that the receiver has the capability of re-transmitting when a timeout occurs (which only the sender does in the old ABP); this difference means that the receiver may

Module: Sender/Receiver Template

```
if Mode = Put then
    if Me = Sender then
        Q(You) :=
            Q(You) ++ Msg(File(MyNum+1), (MyNum+1) mod 2)
        LastMsg := Msg(File(MyNum+1), (MyNum+1) mod 2)
    else
        Q(You) := Q(You) ++ Msg(Null, (MyNum mod 2))
        LastMsg := Msg(Null, (MyNum mod 2)
    endif
    MyNum := MyNum + 1, Mode := Get
endif

if Mode = Get then
    if Defined(InMsg(Me))
            and Num(InMsg(Me)) = (MyNum mod 2) then
        if Me = Receiver then
            File(MyNum) := Data(InMsg(Me))
        endif
        Mode := Put
    endif
    if (Defined(InMsg(Me)) and Num(InMsg(Me)) ≠(MyNum mod 2))
            or Timeout(Me) then
        Q(You) := Q(You) ++ LastMsg endif
        Timeout(Me) := false
    endif
    if Defined(InMsg(Me)) then Clear(InMsg) endif
endif
```

Figure 5: Symmetric sender/receiver module.

Module: Communicate Template
if *Undefined(InMsg(Me))* ***and*** *Q(Me) ≠ EmptyQueue* ***then***
    InMsg(Me) := Head(Q(Me)), Q(Me) := Tail(Q(Me))
endif

Module: LoseMessage Template
Q(Me) := Shrink(Q(Me)) ***endif***

Module: Timeout Template
if *Undefined(InMsg(Me))* ***and*** *Q(Me) = EmptyQueue*
        and *Undefined(InMsg(You))* ***and*** *Q(You) = EmptyQueue* ***then***
    Timeout(Me) := true
endif

Figure 6: Symmetric communications modules.

Sender.Me = Sender	*Receiver.Me = Receiver*
Sender.Mode = Put	*Receiver.Mode = Get*
Sender.MyNum = 0	*Receiver.MyNum = 0*
$\forall x$ *Receiver.File(x) = undef*	*Receiver.LastMsg = Msg(Null,-1)*
Q(Sender) = Q(Receiver) = EmptyQueue	
InMsg(Sender) = InMsg(Receiver) = undef	
Timeout(Sender) = Timeout(Receiver) = false	

Figure 7: New initial conditions.

re-transmit a lost acknowledgment more often than in the old ABP, but that does no harm to the correctness of the protocol.

4 Sliding Windows

The sliding window protocol (SWP) is an extension of the ABP. In a communications medium where two-way simultaneous communication is possible, it can be wasteful to have only one datum in transition between the agents, since the capacity of the underlying network may be grossly underutilized. In such situations, it is desirable to have a number of distinct data currently in transit between the two agents, providing for a continual stream of data rather than the sporadic activity characteristic of "stop and wait" protocols like the ABP.

In the case where the size of the window being used is 1, the behavior of the sliding window protocol is similar to that of the ABP. Other than the use of unbounded message numbers in the sliding window protocol, each agent in either protocol makes similar moves.

One should probably not speak of a single SWP; many sliding window algorithms use that window in different ways. We base our SWP upon one given in [DaC] for Kermit's implementation of sliding windows.

4.1 Function Descriptions

We use the *Msg*, *Data*, *Num*, and *Null* functions defined in the symmetric ABP to compose and decompose messages. We still represent our communication network with queues, using the *Append*, *Head*, *Tail*, and *EmptyQueue* functions as defined previously. The data storage functions *SenderFile* and *ReceiverFile* remain as in the ABP, as does *Timeout*.

Each agent uses *SenderInMsg* and *ReceiverInMsg* as in the ABP to hold the current message being processed. Private distinguished elements *SenderLo, SenderHi, ReceiverLo, ReceiverHi: integers* denote the boundaries of each agent's current window. Additionally, the sender has a function *ReceivedAck: integers* $\rightarrow$ *Bool* which notes which messages have been successfully acknowledged. The distinguished element *WinSize: integers* denotes the maximum window size for both agents.

4.2 Module Specifications

We define *SenderWindowFull* as an abbreviation for $SenderHi - SenderLo + 1 = WinSize$. Thus, *SenderWindowFull* is true exactly when there are exactly *WinSize* messages between *SenderLo* and *SenderHi*, inclusive.

As before, the sender examines each acknowledgment message sent to her by the receiver. If the acknowledgment number is within the current window, the sender marks that message as acknowledged. At any time, if the sender's window is not full, the sender sends another message to the receiver, increasing the size of her window accordingly. Also, if the oldest entry in her window has been successfully acknowledged, she slides up the lower edge of her window, thus decreasing the size of the window.

The receiver's action upon receipt of a message depends upon the message number. If the number is within his current window, he acknowledges the message and records the data in his data storage area. If the number follows his current window, the receiver slides his window up until the message falls within his window.

The sender and receiver modules are given in Fig. 8. The observant reader may note that no rule is given for the case when the receiver receives a message with a message number preceding the current window; we will prove that this situation never occurs.

We use the same communications modules (SenderCommunicate, ReceiverCommunicate, SenderLoseMessage, and ReceiverLoseMessage) used in the ABP.

We use the same definitions of regular and fair run as in the ABP. The initial state of the SWP satisfies the conditions shown in Fig. 9. The only modules which are not positive are SenderLoseMessage and ReceiverLoseMessage.

Module: Sender
Rule: SendMessage
```
if Not(SenderWindowFull) then
    ReceiverQueue :=
        ReceiverQueue ++ Msg(SenderFile(SenderHi+1),SenderHi+1)
    SenderHi := SenderHi+1
elseif Timeout then
    ReceiverQueue :=
        ReceiverQueue ++ Msg(SenderFile(SenderLo),SenderLo)
    Timeout := false
endif
```

Rule: ProcessAck
```
if Defined(SenderInMsg) then
    if SenderLo ≤ Num(SenderInMsg) ≤ SenderHi then
        ReceivedAck(Num(SenderInMsg)) := true
    endif
    Clear(SenderInMsg)
endif
```

Rule: SlideSenderWin
```
if ReceivedAck(SenderLo) then SenderLo := SenderLo + 1 endif
```

Module: Receiver
Rule: AcceptMessage
```
if Defined(ReceiverInMsg) and
        ReceiverLo ≤ Num(ReceiverInMsg) ≤ ReceiverHi then
    SenderQueue := SenderQueue ++ Msg(Null,Num(ReceiverInMsg))
    ReceiverFile(Num(ReceiverInMsg)) := Data(ReceiverInMsg)
    Clear(ReceiverInMsg)
endif
```

Rule: SlideReceiverWin
```
if Defined(ReceiverInMsg) and Num(ReceiverInMsg) > ReceiverHi then
    ReceiverHi := Num(ReceiverInMsg)
    ReceiverLo := Max(0,Num(ReceiverInMsg)−WinSize+1)
endif
```

Figure 8: Sliding window sender and receiver modules.

$\forall x \geq 0$ $ReceiverFile(x) = undef$
$\forall x \geq 0$ $ReceivedAck(x) = false$
$SenderLo = ReceiverLo = 0$
$SenderHi = ReceiverHi = -1$
$SenderInMsg = ReceiverInMsg = undef$
$ReceiverQueue = SenderQueue = EmptyQueue$

Figure 9: SWP initial conditions.

4.3 Proof of Correctness

Lemma 8 *In any reachable state, for any message μ sent by the sender and present within the state, $Num(\mu) \leq SenderHi$.*

Proof. By induction. Initially the sender has sent no messages.

SendMessage is the only rule which may affect the invariant. If the sender's window is not full, *SenderHi* is incremented (which maintains the invariant for any messages previously sent), and a new message μ is sent with $Num(\mu) = SenderHi$. If the sender's window is full, any message μ that is sent will have $Num(\mu) = SenderLo$, and *SenderWindowFull* implies that $SenderLo \leq SenderHi$. □

Lemma 9 *In any reachable state, ($SenderHi - SenderLo + 1 \leq WinSize$) and ($ReceiverHi - ReceiverLo + 1 \leq WinSize$). That is, the size of both the sender's and receiver's windows is $\leq$ WinSize.*

Proof. By induction. Initially $SenderHi - SenderLo + 1 = 0$, and $ReceiverHi - ReceiverLo + 1 = 0$. We consider all moves that affect functions present in the invariant.

SlideSenderWin increments *SenderLo*, which maintains the invariant.

SendMessage may increment *SenderHi* if $SenderHi - SenderLo + 1 < WinSize$; after incrementing *SenderHi*, the invariant still holds.

SlideReceiverWin increments *ReceiverHi* and may also increment *ReceiverLo*; the rule insures that the invariant is maintained. □

Lemma 10 *In any reachable state, ($0 \leq x < SenderLo \rightarrow ReceivedAck(x)$).*

Proof. By induction. Initially, $SenderLo = 0$. Only SlideSenderWin may change *SenderLo*; its guard ensures that the invariant is preserved. □

Lemma 11 *In any reachable state, for any message μ which exists in SenderInMsg or SenderQueue, Defined (ReceiverFile(Num(μ))) is true.*

Proof. By induction. Initially, both *SenderInMsg* and *SenderQueue* are empty. We consider all moves that affect functions present in the invariant.

AcceptMessage appends a new message to *SenderQueue*. Its guard shows that when *Msg(Null,x)* is appended to *SenderQueue*, *ReceiverFile(x)* is being defined at the same moment.

ProcessAck, SenderCommunicate, and SenderLoseMessage discard messages from or transfer messages between *SenderQueue* and *SenderInMsg*, which does not affect the invariant. □

Lemma 12 *In any reachable state, $ReceivedAck(x) \rightarrow Defined(ReceiverFile(x))$.*

Proof. By induction. Fix an x. Initially, $ReceivedAck(x) = false$. The only rule which modifies *ReceivedAck* is ProcessAck, which sets *ReceivedAck(x)* to true if $Num(SenderInMsg) = x$ and x is within the sender's current window. By Lemma 11, we know that *ReceiverFile(x)* is defined. □

Lemma 13 *In any reachable state, (ReceiverHi $\leq$ SenderHi).*

Proof. By induction. Initially, *ReceiverHi* = *SenderHi* = *−1*. We consider all moves that affect functions present in the invariant.

SendMessage increments *SenderHi*, maintaining the invariant.

SlideReceiverWin updates *ReceiverHi* to the current value of *Num(ReceiverInMsg)*, which by Lemma 8 is bounded above by *SenderHi*, maintaining the invariant. □

Lemma 14 *In any reachable state, (ReceiverLo $\leq$ SenderLo).*

Proof. By induction. The invariant is true initially, since *SenderLo* = *ReceiverLo* = *0*. We consider all moves that affect functions present in the invariant.

SlideSenderWin increments *SenderLo*, maintaining the invariant.

SlideReceiverWin updates *ReceiverLo*. If *ReceiverLo* is changed to a non-zero value, the guard for SlideReceiverWin assures us that *ReceiverLo* = *ReceiverHi*− *WinSize*+*1*. Lemmas 9 and 13 yield the result. □

Lemma 15 *In any reachable state, (0 $\leq x <$ ReceiverLo $\rightarrow$ Defined(ReceiverFile(x))).*

Proof. Immediate from Lemmas 14, 10, and 12. □

Lemma 16 *In any reachable state, consider the sequence of messages RIM+RQ. For any two messages α and β in that sequence, if α precedes β, then $(Num(\alpha) - Num(\beta))$ is at most $WinSize - 1$.*

Proof. By induction. Initially, no messages exist in *RIM+RQ*. We consider all moves that affect functions present in the invariant.

SendMessage appends a message β to *ReceiverQueue*. If *Num(β)* = *SenderHi+1* (before *SenderHi* is updated), we know from Lemma 8 that all other messages α in *ReceiverInMsg* or *ReceiverQueue* have *Num(α)* $\leq$ *SenderHi* $<$ *Num(β)*, and the invariant is preserved.

Otherwise, *Num(β)* = *SenderLo*. Lemma 8 tells us that that the largest message number present in *ReceiverInMsg* or *ReceiverQueue* is *SenderHi*; Lemma 9 shows that *SenderHi*−*SenderLo* is at most *WinSize* − 1.

ReceiverCommunicate, ReceiverLoseMessage, and AcceptMessage remove messages from or transfer messages between *ReceiverQueue* and *ReceiverInMsg*, which does not affect the invariant. □

Theorem 5 *In any reachable state, for any message μ in ReceiverInMsg or ReceiverQueue, Num(μ) $\geq$ ReceiverLo. That is, the receiver module will never receive a message whose number precedes the current window.*

Proof. By induction. Initially, no messages exist. We consider all moves that affect functions present in the invariant.

SendMessage creates a new message μ. SendMessage implies *Num(μ)* $\geq$ *SenderLo*; Lemma 14 asserts that *SenderLo* $\geq$ *ReceiverLo*.

SlideReceiverWin increments *ReceiverLo*. If *Num(ReceiverInMsg)* = *x*, SlideReceiverWin sets *ReceiverLo* to *(x*− *WinSize*+*1)*. Lemma 16 implies that all messages in *ReceiverQueue* have numbers in the desired range.

ReceiverCommunicate, ReceiverLoseMessage, and AcceptMessage remove messages from or transfer messages between *ReceiverQueue* and *ReceiverInMsg*, which does not affect the invariant. □

Theorem 6 *In any reachable state Defined(ReceiverFile(x)) $\rightarrow$ ReceiverFile(x) = SenderFile(x).*

Proof. Fix an x. Initially, *ReceiverFile(x)* = *undef*.

Only rule AcceptMessage can change *ReceiverFile(x)*. Rule SendMessage shows that every message from the sender to the receiver has the form *Message(SenderFile(n),n)* for some n; thus, the receiver's assignment to *ReceiverFile* must set *ReceiverFile(x)* to the value of *SenderFile(x)*. □

Theorem 7 *In any fair run, any datum is eventually sent, received, and acknowledged.*

Proof. The proof generally follows that of Theorems 2 and 3. Lemma 15 and Theorem 6 assure us that once the receiver's window moves past position x, message x will have been correctly received and stored. The proof that the receiver's and sender's windows continue to move forward is similar to those shown before.

An important difference involves the loss of messages within the current sender's window which are not at the bottom of the window. Since the sender only re-transmits messages at the bottom of the window, we must assure ourselves that a lost message which is not at the bottom of the window will eventually be re-transmitted. But this is easy to show; since the message at the bottom of the window is being re-transmitted, eventually the message at the bottom of the window will be received by the receiver and its acknowledgment received by the sender. The sender will then move her window forward, thus moving the lost message one position closer to the bottom of the window. A short inductive argument shows that eventually this message will reach the bottom of the window and be successfully transmitted. □

5 Bounded Sliding Windows

The SWP presented in the previous section uses arbitrary integers as message numbers. In real-world settings, one cannot use an arbitrarily increasing integer as a unique message identifier. Thus, most sliding window protocols (including the one implemented in Kermit) use a fixed set of message numbers, and restrict the size of the sliding window to one half of the total number of message numbers allowed.

We present modified rules for the SWP which use only finitely many message numbers and prove that the behavior of this protocol is identical to that of the one presented previously. Our version has different rules for the sender and receiver; a symmetric version could be produced (as with the ABP) but is unnecessary for our purposes, since Kermit's sliding window protocol description is not symmetric.

Remark. The use of finitely many message numbers, as well as the bounds which we will prove, are well known (see for example [Wal]). We do not claim that our proof of this bound is unique; rather, we intend to show that this bound can be easily proven within our framework.

5.1 Function Descriptions

All functions previously defined will be used. We will also use the infix functions $*$, $/$, and *mod* operators, representing integer multiplication, division, and remainder (or modulus).

Additionally, we define two functions *SenderNum, ReceiverNum: messages* $\rightarrow$ *integers* as follows: *SenderNum* is the largest integer less than or equal to *SenderHi* and equivalent to $Num(\mu)$ *mod* $(2*WinSize)$. *ReceiverNum* is the smallest integer greater than or equal to *ReceiverLo* and equivalent to $Num(\mu)$ *mod* $(2*WinSize)$. We will see that *SenderNum* and *ReceiverNum* represent the "true" message number; that is, the number which was used by the agent who created that message.

5.2 Module Specifications

Our communications modules remain unchanged. We need to change the sender and receiver modules to use message numbers modulo $2*WinSize$ instead of an unbounded set of numbers. The revised transition rules are given in Fig. 10, where changes to the previous rules are written in **bold**.

5.3 Proof of Correctness

Our intention here is to show that *SenderNum* and *ReceiverNum* perform the same function that *Num* did in the unbounded SWP. Having done this, the proofs presented in the previous section will still be valid for the bounded SWP. All of our proofs use the unbounded SWP.

```
Module: Sender
Rule: SendMessage
if Not(SenderWindowFull) then
    ReceiverQueue :=
        ReceiverQueue ++ Msg(SenderFile(SenderHi+1),
                (SenderHi+1) mod (2*WinSize))
    SenderHi := SenderHi+1
elseif Timeout then
    ReceiverQueue :=
        ReceiverQueue ++ Msg(SenderFile(SenderLo),
                             SenderLo mod (2*WinSize))
endif

Rule: ProcessAck
if Defined(SenderInMsg) then
    if SenderLo ≤ SenderNum(SenderInMsg) ≤ SenderHi then
        ReceivedAck(SenderNum(SenderInMsg)) := true
    endif
    Clear(SenderInMsg)
endif

Rule: SlideSenderWin
if ReceivedAck(SenderLo) then SenderLo := SenderLo + 1 endif
```

```
Module: Receiver
Rule: AcceptMessage
if Defined(ReceiverInMsg) and
        ReceiverLo ≤ ReceiverNum(ReceiverInMsg) ≤ ReceiverHi then
    SenderQueue := SenderQueue ++ Msg(Null,Num(ReceiverInMsg))
    ReceiverFile(ReceiverNum(ReceiverInMsg)) := Data(ReceiverInMsg)
    Clear(ReceiverInMsg)
endif

Rule: SlideReceiverWin
if Defined(ReceiverInMsg) and
        ReceiverNum(ReceiverInMsg) > ReceiverHi then
    ReceiverHi := ReceiverNum(ReceiverInMsg)
    ReceiverLo := Max(0,ReceiverNum(ReceiverInMsg) − WinSize + 1)
endif
```

Figure 10: Revised sliding window modules.

Lemma 17 *In any reachable state, for any message μ which exists in SenderInMsg or SenderQueue, Num(μ) $\leq$ ReceiverHi.*

Proof. By induction. Initially, *SenderInMsg* and *SenderQueue* contain no messages. We consider all moves that affect functions present in the invariant.

AcceptMessage appends a new message to *SenderQueue*; the rule assures that the number of the message is bounded above by *ReceiverHi*.

SlideReceiverWin increments *ReceiverHi*, maintaining the invariant.

ProcessAck, SenderLoseMessage, and SenderCommunicate transfer messages between or remove messages from *SenderInMsg* and *SenderQueue*, which does not affect the invariant. □

Lemma 18 *In any reachable state, ReceivedAck(x) $\rightarrow$ ReceiverHi $\geq$ x.*

Proof. By induction. Initially, *ReceivedAck(x) = false* for all x. We consider all moves that affect functions present in the invariant.

ProcessAck sets *ReceivedAck(Num(SenderInMsg)) := true*. By Lemma 17, we know that *Num(SenderInMsg)* $\leq$ *ReceiverHi*, so the invariant is maintained.

SlideReceiverWin increments *ReceiverHi*, maintaining the invariant. □

Lemma 19 *In any reachable state, SenderLo $-$ ReceiverHi $\leq$ 1.*

Proof. By induction. Initially, *SenderLo* $-$ *ReceiverHi* $= 0 - (-1) = 1$. We consider all moves that affect functions present in the invariant.

SlideSenderWin increments *SenderLo*. The rule implies that *ReceivedAck(SenderLo)* is true; by Lemma 18, we know that *SenderLo* $-$ *ReceiverHi* ≤ 0. Incrementing *SenderLo* then gives us the desired result.

SlideSenderWin increments *ReceiverHi*, maintaining the invariant. □

Lemma 20 *In any reachable state, SenderHi $-$ ReceiverLo $\leq 2 *$ WinSize $- 1$. That is, there are at most $2 * WinSize$ messages between the bottom of the receiver's window and the top of the sender's window.*

Proof. Immediate from Lemmas 9 and 19. □

Theorem 8 *In any reachable state, for any message μ, SenderNum(μ) = ReceiverNum(μ).*

Proof. By the definitions of *SenderNum* and *ReceiverNum*, we know that *SenderNum(μ)* and *ReceiverNum(μ)* are congruent modulo $2 * WinSize$. The definitions also tell us that the following are invariants:

$$ReceiverLo \leq ReceiverNum(\mu) \leq ReceiverLo + 2 * WinSize - 1,$$

$$SenderHi - 2 * WinSize + 1 \leq SenderNum(\mu) \leq SenderHi.$$

If *SenderNum(μ)* $\neq$ *ReceiverNum(μ)*, we must have *SenderNum(μ)* = *ReceiverNum(μ)* + d, where d is some non-zero multiple of $2 * WinSize$. If d is positive, we have *ReceiverLo* $\leq$ *SenderNum* + $d \leq$ *SenderHi*, which implies that *SenderHi* $-$ *ReceiverLo* $\geq 2*$ *WinSize*, contradicting Lemma 20. If d is negative, a similar argument also creates a contradiction. □

Remark. It turns out that $2 * WinSize$ message numbers are not only sufficient for the sliding window protocol, but also necessary [Wal]. Suppose that both agents are using only $2*$ *WinSize* - *1* message numbers and consider the following run. The sender sends *WinSize* messages to the receiver, numbered *0* through *WinSize* $-$ *1*. The receiver receives all of the messages and sends acknowledgments numbered *0* through *WinSize* $-$ *1* to the sender.

The next message the receiver receives has a message numbered *0*. What should he do with this message?

- If the sender received all of the receiver's acknowledgments, the sender would have moved her window forward by *WinSize*, thus enabling her to send messages numbered *WinSize* through $2 * WinSize - 1$. In that case, the message numbered $2 * WinSize - 1$ would have arrived at the receiver as message 0 (since we are counting modulo $2 * WinSize - 1$), and this message should be accepted and stored.

- If the sender didn't receive the receiver's acknowledgment for message *0*, the sender would have re-sent her message *0*. In that case, this message should be acknowledged, but not stored.

The receiver cannot determine whether this message is an old, re-transmitted message or a new message which should be stored. Thus, at least $2 * WinSize$ message numbers are necessary.

6 Alternating Bit Kermit: The Session Layer

We consider Kermit, as specified by [DaC], at three different layers of abstraction: the session, transport, and datalink layers. The session layer controls sending and receiving files; the network connection is assumed to be reliable, delivering messages intact and in the proper order. The task of the transport layer is to provide a reliable connection even though the actual connection may lose or alter messages during transmission. The datalink layer controls message representation, transforming abstract messages into strings which can be sent through typical communication networks.

Kermit also uses a presentation layer, which transforms a file (seen here as a finite string of arbitrary length) into a sequence of shorter strings which are to be sent through the network. These transformations are fairly mechanical, and we present without comment a couple of static functions which perform this transformation.

The names given to these layers are similar to those used in the ISO Open Systems Interconnection Reference Model (or OSI model) [Tan]. However, since Kermit was developed before the OSI model, the layer names do not always have the same connotations.

From the session layer, Kermit is driven by a finite state automaton. Two agents generate and accept strings of the form $S(FD^*Z)^*B$, where each letter represents a different type of message being sent:

- S represents the *start* of a communications session.

- B represents the end (or *break*) of a communications session.

- F represents the name of a *file*.

- Z represents the end of a file. (Most likely the Z is due to the widespread use of the control-Z character as an end-of-file marker in many operating systems. Alternatively, the last letter of the English alphabet may be appropriate to signal the last datum of a file.)

- D represents *data*.

A few other message types are used: Y indicates a positive acknowledgment ("*yes*"), N indicates a negative acknowledgment ("*no*"), and E indicates the occurrence of an unrecoverable *error*.

The finite state automata used by the sender and receiver agents are shown in Fig. 11. The labels on the transitions should be read in this manner: if an arc from state S to state T is labeled A/B, then in state S, if the agent receives a message labeled A, the agent may send a message labeled B and enter state T.

Most of the names of the states used in the automata are used in the original specification [DaC] and may be read as follows: *ssini* is the sender's initial state, *ssfil* is the sender's state for receiving new files, *ssdat* is the sender's state for receiving data, and *sseot* is the sender's state for ending a transaction. The states of the receiver's automaton are similar.

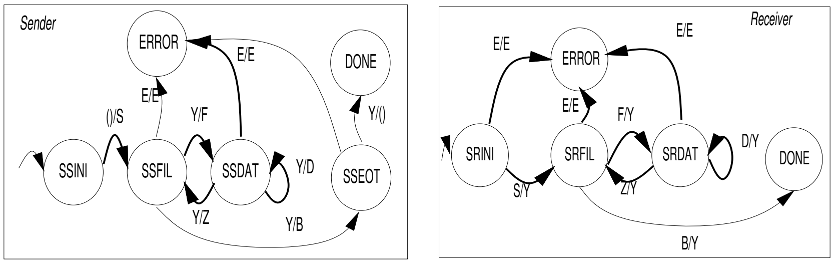

Figure 11: Kermit session layer finite automata.

6.1 Function Descriptions

As with the symmetric ABP, each agent participating in a Kermit file transfer has both the capability of sending and receiving. Our descriptions assume that the external world (*i.e.* the persons using the Kermit program) determine which agent will act as the sender and which as the receiver for a given transmission session. We describe a single module which contains the rules for any Kermit agent, as well as additional modules which describe the communications medium.

6.1.1 Common Functions

The states of the finite state machines shown above are represented by a universe of *modes*. A universe of *symbols* contains the symbols "*S*", "*F*", *etc.* that are transmitted between agents as described above. These symbols are used in messages which contain other textual data; this leads us to universes of *messages* and *strings*. We use the universe of *integers* to count files. Finally, to distinguish between the sender and receiver, we use a universe of *ids*.

Messages are composed of a symbol (the type of the message) and a string (the data content of the message). We thus use functions *Msg: symbols* $\times$ *strings* $\rightarrow$ *messages*, *Type: messages* $\rightarrow$ *symbols*, and *Data: messages* $\rightarrow$ *strings* in the usual manner: if *Msg(t,d)* $= m$, then *Type(m)* $= t$ and *Data(m)* $= d$. The function *InMsg: ids* $\rightarrow$ *messages* indicates the current incoming message for each agent.

The function *Concat: strings* $\times$ *strings* $\rightarrow$ *strings* is the usual string concatenation, which we denote with the infix operator $+$. The distinguished element ϵ*: strings* is the empty string.

For historical reasons, the data transmitted by Kermit is usually translated at the presentation layer into printable ASCII characters before transmission. (This avoids the transmission of non-printable characters which the communications medium might interpret as commands.) We represent this encoding by a pair of functions *EncodePrefix: strings* $\times$ *strings* $\rightarrow$ *strings* and *Remainder: strings* $\times$ *strings* $\rightarrow$ *strings*. The input to both of these functions is a pair *(source, params)*, where *source* is the string to be encoded, and *params* is a string containing various encoding parameters, such as the maximum length of any encoding, the prefix character to be used for non-printable characters, and so on. *EncodePrefix (source, params)* returns the encoding of an initial segment of *source*; *Remainder (source, params)* returns the segment of *source* not translated by *EncodePrefix*. A function *Decode: strings* $\times$ *strings* $\rightarrow$ *strings* decodes any input string, using a similar string of decoding parameters.

We require that applying *Decode* to an encoded datum with the same parameter string used in its encoding should yield the original datum; that is, if *Decode (Encode (source, params))* $= x$, then $x +$ *Remainder (source, params)* $=$ *source*. Further, we require that the length of *Remainder(source, params)* should be less than the length of *source* as long as *source* $\neq \epsilon$.

6.1.2 Private Functions

Each agent has a private function *Mode: modes* which indicates the current state of its finite automata. A private function *Layer:* {*session, transport*} indicates whether control is currently focused in the session or transport layer. Initially, *Layer = session* for both agents. As in the symmetric ABP, each agent has private functions *Me: ids* and *You: ids*.

The private functions *RecvdType: symbols* and *RecvdData: strings* indicate the type and datum of the last message received from the other agent. The private functions *SendType: symbols* and *SendData: strings* indicate the type and datum of the next message to be sent to the other agent. The private function *TransportCommand:* $\{send,\ receive\}$ indicates the latest command given to the transport layer.

Kermit allows for more than one file to be sent between the sender and the receiver in a single communications session; thus, we need to store the list of files being sent and received, as well as the list of corresponding file names. We do this with the private functions *FileText: integers* $\rightarrow$ *strings* and *FileName: integers* $\rightarrow$ *strings*, where the integer argument to each function denotes the position within the list of the appropriate file and file name. The private function *FileNo: integers* indicates which file is currently being sent; the private function *TextToSend: strings* indicates (for the sender) what data in the current file remains to be sent.

Finally, each agent has a private function *MyParams: strings* which provides local parameters used during the initialization of the transaction.

6.2 Transition Rules

6.2.1 Sending Rules

The sender agent begins in state *ssini* by sending an "*S*" to the receiver, along with her initialization parameters, and entering state *ssfil*. The transition rule for this state is shown in Fig. 12. Later we will present transition rules which perform the actual transmission of data implied by *TSEND*.

```
Rule: SSINI
if Layer = session and Mode = ssini then        Abbreviation: GOTO(x)
    TSEND("S",MyParams)                            Clear(RecvdType)
    GOTO(ssfil)                                    Mode := x
endif

Abbreviation: TSEND(type,datum)
    TransportCommand := send, Layer := transport
    SendType := type, SendData := datum
```

Figure 12: Transition rule for state *ssini*.

Automata state *ssini* is the only state in which an agent acts without first receiving a message. Fig. 13 presents a transition rule which performs the action of receiving input for all other automata states. As with *TSEND*, we will present transition rules which perform this action later.

In automata state *ssfil*, the sender can either begin transmission of a new file (if one remains to be sent) or end the transaction. To begin sending a file, the sender sends the file name in an "*F*" message to the receiver and moves to state *ssdat*, initializing the local *TextToSend* variable with the data contained in that file. To end the transaction, the sender sends a "*B*" message to the receiver. The transition rule for this automata state is shown in Fig. 14. (Note that here and elsewhere, "**ENDIF**" abbreviates a series of "**endif**"s).

HANDLE-INITS is an abbreviation for a set of transition rules which handles the initialization parameters sent between agents (which occurs only on the first exchange of messages). See 10 for a fuller explanation of *HANDLE-INITS*. *EINFO* is a string containing information needed to encode text properly in order to be decoded by the receiver.

Rule: GetInput
if *Layer = session* ***and*** *Undefined(RecvdType)* ***and*** *Mode ≠ssini* ***then***
    Layer := transport
    TransportCommand := receive
endif

Figure 13: Transition rule for receiving input.

Rule: SSFIL
if *Layer = session* ***and*** *Mode = ssfil* ***and*** *Defined(RecvdType)* ***then***
    if *RecvdType ≠ "Y"* ***then*** *ERROR* ***endif***
    if *RecvdType = "Y"* ***then***
        if *FileNo = 0* ***then*** *HANDLE-INITS* ***endif***
        if *Defined(FileName(FileNo))* ***then***
            TSEND("F",EncodePrefix(FileName(FileNo),EINFO))
            TextToSend := FileText(FileNo), GOTO(ssdat)
        else
            TSEND("B",ϵ), GOTO(sseot)
ENDIF

Abbreviation: ERROR
    TSEND("E", "Unexpected Message"), GOTO(error)

Figure 14: Transition rule for state *ssfil*.

In automata state *ssdat*, the sender either sends the next segment of the file being transmitted, or signals the end of transmission of this file. To send a new file segment, the sender sends the text in a "*D*" message to the receiver, and updates the local *TextToSend* variable accordingly. To end transmission of a file, the sender sends a "*Z*" message to the receiver. The transition rule for this automata state is shown in Fig. 15.

Rule: SSDAT
if *Layer = session* ***and*** *Mode = ssdat* ***and*** *Defined(RecvdType)* ***then***
    if *RecvdType* $\neq$ *"Y"* ***then*** *ERROR* ***endif***
    if *RecvdType = "Y"* ***then***
        if *TextToSend* $\neq \epsilon$ ***then***
            TSEND("D",EncodePrefix(TextToSend,EINFO))
            TextToSend := Remainder(TextToSend,EINFO)
            GOTO(ssdat)
        else
            TSEND("Z",ϵ*), FileNo := FileNo + 1, GOTO(ssfil)*
ENDIF

Figure 15: Transition rule for state *ssdat*.

In automata state *sseot*, the sender waits for an acknowledgment of its last message to the receiver, which was a "*B*" message. Upon receipt, the sender terminates. The transition rule for this state is shown in Fig. 16.

Rule: SSEOT
if *Layer = session* ***and*** *Mode = sseot* ***and*** *Defined(RecvdType)* ***then***
    if *RecvdType = "Y"* ***then*** *GOTO(done)* ***endif***
    if *RecvdType* $\neq$ *"Y"* ***then*** *ERROR* ***endif***
endif

Figure 16: Transition rule for state *sseot*.

6.2.2 Receiver Rules

The receiver agent starts in a state with *Mode = srini*, where he waits for the initial "*S*" message from the sender. After receipt, the receiver processes the sender's initialization parameters, sends his own parameters to the sender in an acknowledgment message ("*Y*"), and moves to state *srfil* to prepare to receive files. The transition rule for this state is shown in Fig. 17.

In automata state *srfil*, the receiver receives either an "*F*" message or a "*B*" message from the sender, indicating whether another file is about to be sent. The receiver moves to the appropriate state, storing any file name sent. The transition rule for this automata state is shown in Fig. 18.

In automata state *srdat*, the receiver either stores the next file segment transmitted in a "*D*" message, or ends file reception when a "*Z*" message is received. The transition rule for this automata state is shown in Fig. 19.

```
Rule: SRINI
if Layer = session and Mode = srini and Defined(RecvdType) then
    if RecvdType ≠ "S" then ERROR endif
    if RecvdType = "S" then
        HANDLE-INITS, TSEND("Y",MyParams), GOTO(srfil)
ENDIF
```

Figure 17: Transition rule for state *srini*.

```
Rule: SRFIL
if Layer = session and Mode = srfil and Defined(RecvdType) then
    if RecvdType ≠ "F" and RecvdType ≠ "B" then ERROR endif
    if RecvdType = "F" then
        FileName(FileNo) := Decode(RecvdData,DINFO)
        TSEND("Y",ε), GOTO(srdat)
    endif
    if RecvdType = "B" then
        TSEND("Y",ε), GOTO(done)
ENDIF
```

Figure 18: Transitions rule for state *srfil*.

```
Rule: SRDAT
if Layer = session and Mode = srdat and Defined(RecvdType) then
    if RecvdType ≠ "D" and RecvdType ≠ "Z" then ERROR endif
    if RecvdType = "D" then
        FileText(FileNo) :=
            FileText(FileNo) + Decode(RecvdData,DINFO)
        TSEND("Y",ε), GOTO(srdat)
    endif
    if RecvdType = "Z" then
        FileNo := FileNo + 1, TSEND("Y",ε), GOTO(srfil)
ENDIF
```

Figure 19: Transition rule for state *srdat*.

6.2.3 Interim Transport Layer Module

In our proofs of correctness, we intend to prove the correctness of each layer separately, assuming the correctness of any lower layers on which that layer depends. Through this separation of concerns, our proofs become smaller and more manageable. Consequently, at each layer, we present transition rules which act as a correct implementation of the succeeding layers. These rules will be replaced with more detailed rules later.

The session layer of Kermit calls upon the transport layer to perform the actual task of transferring messages between the two agents. We present a simple module template in Fig. 20 which describes a reliable communications medium from one agent to the other.

Module: InterimTransport
Rule: TSEND
if *Layer = transport* ***and*** *TransportCommand = send* ***then***
    InMsg(You) := Msg(SendType, SendData)
    Clear(SendType), Clear(SendData), Layer := session
endif

Rule: TGET
if *Layer = transport* ***and*** *TransportCommand = receive*
        and *Defined(InMsg(Me))* ***then***
    RecvdType := Type(InMsg(Me)), RecvdData := Data(InMsg(Me))
    Clear(InMsg(Me)), Layer := session
endif

Figure 20: Interim transport layer rules.

6.2.4 Regular Run

The initial state of our Kermit algebra satisfies the conditions shown in Fig. 21.

S.Mode = ssini	*R.Mode = srini*
S.Layer = session	*R.Layer = session*
S.RecvdType = undef	*R.RecvdType = undef*
S.FileNo = 0	*R.FileNo = 0*

Figure 21: Initial conditions for Kermit.

6.3 Proof of Correctness

Throughout our proofs, we use $S.X$ and $R.X$ to refer to the private function X of the sender and receiver, respectively.

Lemma 21 *Define*

$$R.Next = \begin{cases} R.RecvdType & \text{if } Defined(RecvdType) \\ Type(InMsg(Receiver)) & \text{if } Undefined(RecvdType) \wedge \\ & \quad Defined(InMsg(Receiver)) \\ S.SendType & \text{otherwise} \end{cases}$$

and define S.Next similarly. Then in any reachable global state, the values of S.Mode, S.Next, R.Mode, and R.Next match one of the lines of the following table:

	S.Mode	*S.Next*	*R.Mode*	*R.Next*
(a)	*ssini*	*undef*	*srini*	*undef*
(b)	*ssfil*	*undef*	*srini*	*"S"*
(c)	*ssfil*	*"Y"*	*srfil*	*undef*
(d)	*ssdat*	*undef*	*srfil*	*"F"*
(e)	*ssdat*	*"Y"*	*srdat*	*undef*
(f)	*ssdat*	*undef*	*srdat*	*"D"*
(g)	*ssfil*	*undef*	*srdat*	*"Z"*
(h)	*sseot*	*undef*	*srfil*	*"B"*
(i)	*sseot*	*"Y"*	*done*	*undef*
(j)	*done*	*undef*	*done*	*undef*

Proof. By induction. The initial state satisfies condition (a) of the table.

- GetInput, TSEND, and TGET do not affect any condition in the table.
- Rule SSINI transforms condition (a) to (b).
- Rule SRINI transforms condition (b) to (c).
- Rule SSFIL transforms condition (c) to (d) or (h).
- Rule SRFIL transforms condition (d) to (e) and transforms condition (h) to (i).
- Rule SSDAT transforms condition (e) to (f) or (g).
- Rule SRDAT transforms condition (f) to (e) and transforms condition (g) to (c).
- Rule SSEOT transforms condition (i) to (j). □

Lemma 22 *In any reachable state, the* ERROR *macro is never executed.*

Proof. It is observable from the table in Lemma 21 and the transition rules that in any state where *S.RecvdType* or *R.RecvdType* is defined, its value will not allow the *ERROR* macro to be executed. □

Lemma 23 *In any regular run, any message sent by the sender is acknowledged by the receiver.*

Proof. By Lemma 21, we know that any message sent by the sender is expected by the receiver; *i.e.* the receiver will not execute the *ERROR* macro, but execute some other transition rules. The message will not be discarded by the reliable network; thus, the receiver will accept the message. An examination of the receiver's rules shows that every message which does not generate an error (as these messages will not) is acknowledged. □

Lemma 24 *In any reachable state, if* $R.Mode = srdat$*, then* $R.FileText(x) = \epsilon$ *for every* $x > R.FileNo$*; if* $R.Mode \neq srdat$*, then* $R.FileText(x) = \epsilon$ *for every* $x \geq R.FileNo$*.*

Proof. By induction. Initially, *R.FileText(x)* $= \epsilon$ for every $x \geq 0$. We consider all moves that affect functions present in the invariant.

SRFIL may set *R.Mode* to *srdat*, but will leave *R.FileNo* and *R.FileText* unchanged; thus, the truth of the invariant is unaffected.

SRDAT has different effects on the functions named in the invariant, depending upon the value of *R.RecvdType*. Before SRDAT fires, the invariant asserts that *R.FileText(x)* $= \epsilon$ for all $x > R.FileNo$. If *R.RecvdType* = *"Z"*, SRDAT updates *R.Mode* to *srfil* and increments *R.FileNo*; afterwards, we will have *R.FileText(x)* $= \epsilon$ for all $x \geq R.FileNo$. Otherwise, *FileText(FileNo)* may be modified, but this does not affect the invariant. □

Lemma 25 *In any reachable state, the following conditions are true:*

$$S.Mode = ssfil \wedge R.Mode = srdat \rightarrow S.FileNo = R.FileNo + 1,$$

$$\neg(S.Mode = ssfil \wedge R.Mode = srdat) \rightarrow S.FileNo = R.FileNo.$$

Proof. By induction. Initially, *S.Mode = ssini*, *R.Mode = srini*, and *S.FileNo = R.FileNo = 0*.

From Lemma 21, we see that we may enter a state where *S.Mode = ssfil* and *R.Mode = srdat* (condition (g)) only when *S.Mode = ssdat* and *R.Mode = srdat* (condition (e)), by rule SSDAT. By the induction hypothesis, *S.FileNo = R.FileNo* before SSDAT fires, and we will have *S.FileNo = R.FileNo+1* after SSDAT fires, since SSDAT increments *S.FileNo*.

Similarly, we may only leave a state where *S.Mode = ssfil* and *R.Mode = srdat* (condition (g)) by rule SRDAT. SRDAT increments *R.FileNo*; thus, we will have *S.FileNo = R.FileNo* as desired after SRDAT fires. □

Lemma 26 *Define*

$$code = \begin{cases} R.RecvdData & \textit{if } \text{Defined(R.RecvdData)} \\ Data(InMsg(Receiver)) & \textit{if } \text{Undefined(R.RecvdData)} \\ & \wedge \text{ Defined(InMsg(Receiver))} \\ \epsilon & \textit{otherwise} \end{cases}$$

Then in any reachable state, if S.Mode = ssdat, *then S.FileText(S.FileNo) = R.FileText(R.FileNo) + Decode(code,DINFO) + S.TextToSend.*

Proof. By induction. Initially, *S.Mode = ssini*, making the premise of the invariant false, thus satisfying the invariant. We consider all moves that affect functions present in the invariant.

SSFIL may update *S.Mode* to *ssdat*; if it does, we will have *S.TextToSend = S.FileText(S.FileNo)* and *code* $= \epsilon$. Applying Lemmas 21 and 24 shows that *R.FileText(R.FileNo)* $= \epsilon$. If firing SSFIL does not update *S.Mode* to *ssdat*, the invariant is not affected.

SSDAT may update *S.Mode* to *ssfil*; in that case, the premise of the invariant becomes false, making the invariant true. Otherwise, text may be transferred from *S.TextToSend* into *code*, which does not affect the invariant.

TGET may move text from *InMsg(Receiver)* into *R.RecvdData*, which does not affect the invariant (since both are part of the definition of *code*).

SRDAT may move text from *code* into *R.FileText(R.FileNo)*, which maintains the truth of the invariant. Otherwise, R.FileNo may be incremented; but Lemma 21 shows that *S.Mode* $\neq$ *ssdat* in this case, violating our premise. □

Theorem 9 *In any regular run, assuming finite input both sides will terminate with* Mode = done, *and for all x,* S.FileText(x) = R.FileText(x) *and* S.FileName(x) = R.FileName(x).

Proof. We consider a regular run from the sender's perspective, since the receiver acknowledges every message sent by the sender (Lemma 23). By Lemma 22, we know the regular run will never enter an error state.

The sender starts by sending an *"S"* and enters mode *ssfil*, which is also entered any time a file has been successfully sent. If there are more files to send, a new *"F"* message is sent with the new file name and the sender enters mode *ssdat*. In a moment, we will show that the sender eventually returns to *ssfil* with *FileNo* incremented by one. Thus, if only finitely many files have been specified for sending, eventually no more files will be available and the sender proceeds to mode *sseot* where a *"B"* message is sent. After its acknowledgment, both sides move to mode *done*.

It remains to show that from mode *ssdat*, the sender eventually returns to mode *ssfil*. In mode *ssdat*, the length of *TextToSend* decreases each time a *"D"* message is sent, which is eventually acknowledged. Thus, eventually the entire file will be sent, *FileNo* will be incremented, and control will be returned back to *ssfil*.

How do we know that file names and texts are transmitted correctly? Lemma 25 shows that *S.FileNo* = *R.FileNo* when file names and file texts are transmitted; thus, file names sent from the sender to the receiver will be placed in corresponding places in the private function *FileName*. Lemma 26 shows that when the sender is about to send the end-of-file *"Z"* marker, $S.TextToSend = code = \epsilon$. Thus, *S.FileText* and *R.FileText* will match with respect to the current value of *FileNo*. □

7 Alternating Bit Kermit: The Transport Layer

The goal of the transport layer is to transform a possibly unreliable connection into a reliable one. In Alternating Bit Kermit, the transport layer is a variant on the ABP, described in section 3. This similarity allows us to rely on the proofs of correctness for the ABP for the correctness of the transport layer.

A couple of new message types are used by the transport layer. Q indicates that a message has been corrupted (altered) during transmission through the network. T indicates that a timeout signal has occurred while waiting for a message to arrive.

7.1 New Function Descriptions

7.1.1 Common Functions

Messages will now be composed of three parts: a symbol indicating the type of the message, a string indicating the data content of the message, and an integer used to maintain proper sequencing of messages. We thus re-define *Msg: symbols* × *strings* × *integers* → *messages*, and define a new function *Num: messages* → *integers*. We will only number messages with the integers $\{0, \ldots, 63\}$; consequently, we will use the modulus function over the integers to compute message numbers.

The communications paths between agents are represented by *queues*, as in the ABP. Functions *EmptyQueue: queues*, *Append: queues* × *messages* → *queues*, *Head: queues* → *messages*, *Tail: queues* → *queues*, and *Shrink: queues* → *queues* perform the usual operations. In addition, the external function *Corrupt: queues* → *queues* replaces zero or more messages in the input queue with messages of type "*Q*".

The function *Q: ids* → *queues* represents the queues between the pair of agents. For a given agent, *Q(Me)* is the queue of messages to be received by that agent, and *Q(You)* is the queue of messages sent by that agent but not yet received by the other agent.

7.1.2 Private Functions

Each agent has a private function *SeqNo: integers* which holds the message number it expects to receive in the next message from the other agent. Private functions *FetchType: symbols*, *FetchNo: integers*, and *FetchData: strings* hold the message symbol, number, and datum from the most recently retrieved message.

The private function *Retry: integers* indicates the number of times the transport layer has attempted to send the current message. The private function *RetryLimit: integers* indicates the maximum number of times re-transmission of a particular message should occur. The private function *LastMsg: messages* holds a copy of the last message sent to the other agent.

7.2 Transition Rules

We replace the interim module template presented in the last section with the rules presented in this section.

The transport layer module has two transition rules (shown in Fig. 22) which implement the sending and receiving of messages; they are essentially those of the symmetric ABP. Rule *TSEND* uses a couple of abbreviations; these are defined in Fig. 23. Finally, we need to define the *SEND* and *GET* abbreviations used throughout these rules; they are shown in Fig. 24.

```
Rule: TSEND
if Layer = transport and TransportCommand = send then
    if Sender = Me then
        SEND(SendType, SendData, (SeqNo + 1) mod 64)
    else SEND(SendType, SendData, SeqNo)
    endif
    SeqNo := (SeqNo + 1) mod 64
    Clear(SendType), Clear(SendData), Layer := session
endif

Rule: TGET
if Layer = transport and TransportCommand = receive then
    if Undefined(FetchType) then GET endif
    if Defined(FetchType) then
        if (FetchNo = SeqNo) and (FetchType ∉ {"N", "Q", "T"}) then
            RETURN(FetchType, FetchData)
        elseif FetchType = "N" and FetchNo = (SeqNo+1) mod 64 then
            RETURN("Y", FetchData)
        else RESEND
ENDIF
```

Figure 22: Transition rules for transport-level sending and receiving.

The network modules for Kermit, which allow for corruption of messages as well as discarding of messages, are a simple extension of those for the ABP and are shown in Fig. 25.

7.3 Correctness

The transition rules presented in the last section are functionally equivalent to those of the symmetric ABP. *TSEND* corresponds to the transition rules of the symmetric ABP invoked when *Mode* = *Put*, and *TGET* corresponds to the rules invoked when *Mode* = *Get*. The alternation between modes *Get* and *Put* which appears naturally in the symmetric ABP is assured by the session-layer rules, which alternate calls to *TSEND* and *TGET*.

The symmetric ABP used 0 and 1 as message numbers, while Kermit uses the entire set $\{0, \ldots, 63\}$. It can easily be shown, as in the ABP, that only two distinct message numbers exist in the various queues and message holders in the Kermit protocol at any one moment. Furthermore, these two message numbers are consecutive mod 64; thus, they can be seen to correspond to the ABP's 0 and 1 bits, where a message

Abbreviation: RESEND

  if $Retry > RetryLimit$ ***then***
    $SEND(“E”, \epsilon)$, $RETURN(“E”, \epsilon)$
  elseif $LastMsg \neq undef$ ***then*** $Q(You) := Q(You) ++ LastMsg$
  else $SEND(“N”, \epsilon, SeqNo)$
  endif
  $Clear(FetchType)$, $Clear(FetchData)$

Abbreviation: RETURN (type, datum)

  $Retry := 0$, $RecvdType := type$, $RecvdData := datum$
  $Clear(FetchType)$, $Clear(FetchData)$, $Layer := session$

Figure 23: Definitions for *RESEND* and *RETURN*.

Abbreviation: SEND (type, datum, num)

  $LastMsg := Msg(type, datum, num)$
  $Q(You) := Q(You) ++ Msg(type, datum, num)$

Abbreviation: GET

  if $Defined(InMsg(Me))$ ***then***
    $FetchType := Type(InMsg(Me))$, $FetchNo := Num(InMsg(Me))$
    $FetchData := Data(InMsg(Me))$, $Clear(InMsg(Me))$
  elseif $Timeout(Me)$ ***then***
    $FetchType :=$ “T”, $FetchNo := SeqNo$
    $Timeout(Me) := false$
  endif

Figure 24: Definitions for *SEND* and *GET*.

Module: Transfer Template

if $Undefined(InMsg(Me))$ ***and*** $Q(Me) \neq EmptyQueue$ ***then***
  $InMsg(Me) := Head(Q(Me))$, $Q(Me) := Tail(Q(Me))$
endif

Module: Shrink Template

$Q(Me) := Shrink(Q(Me))$ ***endif***

Module: Corrupt Template

$Q(Me) := Corrupt(Q(Me))$ ***endif***

Figure 25: Network module templates for Kermit.

number is mapped to 0 if it is even and 1 if it is odd. Thus, using this larger set of message numbers does not affect the correctness of the protocol.

Kermit also detects that an *"N"* message numbered x is equivalent to a *"Y"* message numbered $x - 1$, since an *"N"* message numbered x cannot be sent before a *"Y"* message numbered $x - 1$. This feature can improve efficiency slightly but has no affect on correctness.

Kermit does not re-transmit messages infinitely often, as our version of the ABP does; otherwise, Kermit would never terminate if a network failure prohibited all messages from reaching their destinations. Since such a failure may not be detectable, Kermit "gives up" after a fixed number of re-transmissions without receiving a response. This number may be changed by the user.

The network modules for Kermit may corrupt messages as well as discarding them; Kermit treats corrupted messages similarly to timeout signals by requesting re-transmission of the last message sent.

Thus, the transport layer of Alternating Bit Kermit is correct if and only if the ABP is correct. Immediately we have the following theorem:

Theorem 10 *Any message sent by the transport layer of one agent, if it is not lost or corrupted more than RetryLimit times, eventually is received by the transport layer of the other agent.*

In addition, if all rules involving *RetryLimit* were to be removed from our description of Kermit, it can be proved that the only regular Kermit runs which do not terminate are "unfair" runs; that is, runs in which the modules which discard or corrupt messages interfere do not allow the other network modules to transmit messages.

8 Alternating Bit Kermit: The Datalink Layer

The datalink layer of Kermit translates the abstract domain of "messages" into strings which may be transmitted across the network. The string corresponding to a Kermit message is the concatenation of the following:

- A fixed number of padding characters, needed by some implementations to allow an agent to detect the beginning of a message. The number of padding characters, as well as the characters themselves, are determined during the initial negotiations.
- A special "mark" character, denoting the true start of the message.
- The length of the "true" message (consisting of the length, message number, message type, and datum), encoded as a single printable character.
- The message number, encoded as a single printable character.
- The message type symbol.
- The datum being sent.
- A checksum computed on the whole string produced so far, excluding the padding and mark characters, encoded as a few printable characters (dependent on the checksum algorithm selected during the initial negotiations).
- A special end-of-line character.

Thus, all that is required to modify for the datalink layer of Kermit is to provide more specific definitions for the *Msg*, *Type*, *Data*, and *Num* functions specified earlier. We show their definitions in Fig. 26, where + denotes string concatenation and the behavior of previously unspecified functions should be clear from the above descriptions.

$$\begin{aligned}
&\mathit{Msg}(\mathit{type},\ \mathit{data},\ \mathit{num}) = \mathit{Pad}(\mathit{PadChar},\mathit{PadLen}) + \mathit{Mark} + \mathit{BODY}\\
&\qquad\qquad + \mathit{ToChar}(\mathit{ChkSum}(\mathit{BODY})) + \mathit{Eol}\\
&\quad (\mathit{where}\ \mathit{BODY} = \mathit{ToChar}(\mathit{Length}(\mathit{data}){+}3) + \mathit{ToChar}(\mathit{num})\\
&\qquad\qquad + \mathit{type} + \mathit{data})
\end{aligned}$$

$$\begin{aligned}
&\mathit{Type}(\mathit{msg}) =\\
&\quad \textbf{\textit{if}}\ \mathit{Chksum}(\mathit{Substr}(\mathit{DATA},2,\mathit{LEN})) \neq \mathit{UnChar}(\mathit{Substr}(\mathit{DATA},\mathit{LEN}{+}2,1))\\
&\quad \textbf{\textit{then}}\ \text{“}Q\text{”}\ \textbf{\textit{else}}\ \mathit{Substr}(\mathit{DATA},4,1)\\
&\mathit{Data}(\mathit{msg}) = \mathit{Substr}(\mathit{DATA},5,\mathit{LEN}{-}3)\\
&\mathit{Num}(\mathit{msg}) = \mathit{UnChar}(\mathit{Substr}(\mathit{DATA},3,1))\\
&\qquad\qquad (\mathit{where}\ \mathit{DATA} = \mathit{FindStr}(\mathit{msg},\ \mathit{Mark})\\
&\qquad\qquad\ \mathit{and}\ \mathit{LEN} = \mathit{UnChar}(\mathit{Substr}(\mathit{DATA},2,1)))
\end{aligned}$$

Figure 26: Definitions of datalink functions.

We also modify the *Corrupt* function (which corrupts messages in transit) to modify messages in such a manner that the alteration is detectable by the *Chksum* function as defined here. Certainly a message could be altered and still pass this type of test; however, there is no way in general to be certain that a given message has been unaltered by transmission through an unreliable medium. Many checksum functions have been developed which can detect most types of errors common in network transmissions; [DaC] specifies three different checksum functions which may be used in Kermit.

8.1 Correctness

Since the only changes we have made in the datalink layer are to the functions *Msg*, *Type*, *Data*, and *Msg*, the correctness of the datalink layer is directly related to the relationships between these functions:

Theorem 11 *For any symbol t, datum d, and message number n, Type (Msg (t,d,n)) = t, Data(Msg(t,d,n)) = d, and Num(Msg(T,d,n)) = n.*

Proof. Immediate from the definitions given above. □

9 Sliding Windows Kermit

Since the basic Kermit protocol was standardized, a number of optional extensions to Kermit have been developed. A Kermit agent can be instructed to act as a file server, both sending and receiving files from the other agent at her request. A protocol extension is defined for encoding 8-bit data for transmission over 7-bit networks. Most of these extensions can be modeled easily using evolving algebras, but do not provide much of a challenge. We describe here an optional extension of greater interest: the use of a sliding window protocol.

If both sides agree to the use of sliding windows, each agent may use Kermit's version of the sliding window protocol while file data is being transmitted in states *ssdat* and *srdat*. The standard ABP is used in all other states. The specification given in [DaC] is a combination of the session and transport layers, with the sliding window protocol used in place of the ABP. Consequently, we present transition rules for Kermit's use of sliding windows which are a combination of the session and transport layer rules.

9.1 New Functions

The sender agent has a private function *WinMsg: integers → messages* which is used to store messages that have been sent; the receiver agent has a corresponding function *WinData: integers → data* used to store data that has been received. Both agents have a private function *WinAck: integers → Bool* which indicates which messages have been acknowledged.

Both agents have private functions *WinLo, WinHi: integers* which indicate the current window boundaries. A global static function *WinMax: integers* indicates the largest window size permitted; as shown in section 5, at least $2*WinMax$ distinct messages are needed to use a window of size $WinMax$. Consequently, we assume that all our arithmetic with respect to message numbers is done modulo $2 * WinMax$.

A function *Windows: Bool* indicates whether or not the sliding windows protocol is to be used during data transmission. Its value is determined during the initial protocol negotiations.

9.2 Transition Rules

Rule GetInput, presented in Fig. 13, attempts to get a new input message whenever there is no message available (except in the initial state of the sender, since the sender must send the first message). In Sliding Windows Kermit, we only wish to obtain messages at specific moments; in particular, we need to disable this automatic retrieval of messages when our sliding window is in operation. Thus, we need to disable GetInput for certain values of *Mode*. The modified transition rule is shown in Fig. 27. Further, we add the Boolean condition ($Windows = false$) to the outermost guard of rules *SSDAT* and *SRDAT* presented in section 6.2.

Rule: GetInput
if *Layer = session* ***and*** *Undefined(RecvdType)* ***and*** *(Mode ≠ssini* ***or***
    (Windows = true ***and*** *Mode ∉ {ssdat, ssdatrotate, srdat,*
        srdatnak, srdatrotate})) ***then***
  Layer := transport, TransportCommand := receive
endif

Figure 27: Modified transition rule for receiving input.

Once the sender arrives in state *ssdat* during a sliding window file transfer, she repeatedly sends messages until an acknowledgment arrives, the window of unacknowledged messages fills, or the entire file has been sent. The transition rule for sending messages in state *ssdat* is shown in Fig. 28, and corresponds to rule SendMessage of the SWP shown in Fig. 8.

If acknowledgments within the sender's window are waiting to be processed, the appropriate window slots are marked as acknowledged and control passes to *ssdatrotate*, which rotates the window forward as far as possible. Otherwise, a retry mechanism is used to re-transmit certain messages a limited number of times. The transition rule for processing acknowledgments in state *ssdat* is shown in Fig. 29, and corresponds to rule ProcessAck of the SWP shown in Fig. 8.

The rules for *ssdatrotate* slide the sender's window forward as far as possible. Additionally, if all acknowledgments have been received for a particular file, we send the end-of-file *Z* indicator and resume normal Kermit operations. The transition rule for *ssdatrotate* is shown in Fig. 30, and corresponds to rule SlideSenderWin of the SWP shown in Fig. 8.

In state *srdat*, the receiver using sliding windows processes data (D) messages within the current window by storing the received data and sending an acknowledgment message. When the end-of-file Z message arrives, the receiver clears his window of all data which has been received but not yet shifted out of the window. Finally, if a timeout or corrupted message arrives, a negative acknowledgment (NAK) message is

Rule: SSDAT:Windows:1
if *Mode = ssdat* ***and*** *Windows = true* ***and*** *Undefined(FetchType)* ***then***
    if *SeqNo* $-$ *WinLo + 1* $\leq$ *WinMax* ***and*** *TextToSend* $\neq \epsilon$ ***then***
        WinMsg(SeqNo) :=
            Msg("D", EncodePrefix(TextToSend,EINFO), SeqNo)
        SEND("D", EncodePrefix(TextToSend, EINFO), SeqNo)
        TextToSend := Remainder(TextToSend, EINFO)
        SeqNo := SeqNo + 1
    else *GET*
ENDIF

Figure 28: Transition rule for sending data using sliding windows.

Rule: SSDAT:Windows:2
if *Mode = ssdat* ***and*** *Windows* ***and*** *Defined(FetchType)* ***then***
    if *WinLo* $\leq$ *FetchNo* $\leq$ *SeqNo* ***then***
        if *FetchType = "Y"* ***then***
            WinAck(FetchNo) := true, Retry(FetchNo) := 0
            Clear(FetchType)
            if *FetchNo = WinLo* ***then*** *Mode := ssdatrotate* ***endif***
        endif
        if *FetchType = "N"* ***then*** *RETRY(FetchNo)* ***endif***
        if *FetchType = "T"* ***then*** *RETRY(WinLo)* ***endif***
        if *FetchType* $\notin$ {*"Y", "N", "T", "Q"*} ***then*** *ERROR* ***endif***
    endif
endif

Abbreviation: RETRY(num)
if *Retry(num) > RetryLimit* ***then*** *ERROR*
else
    Q(You) := Q(You) ++ WinMsg(num)
    Retry(num) := Retry(num) + 1, Clear(FetchType)
endif

Figure 29: Transition rule for receiving acknowledgments using sliding windows.

Rule: SSDATROTATE
if *Mode = ssdatrotate* ***then***
  if *WinAck(WinLo) = true* ***then***
    WinAck(WinLo) := false, WinLo := WinLo + 1
  elseif *WinLo = SeqNo* ***and*** *TextToSend = ϵ* ***then***
    TSEND("Z",ϵ), GOTO(ssfil)
  else *GOTO(ssdat)*
  endif
endif

Figure 30: Transition rule for state *ssdatrotate*.

generated for the oldest message in the current window not yet received. The transition rule for state *srdat* is given in Fig. 31; it corresponds to rule AcceptMessage of the SWP shown in Fig. 8.

In state *srdatnak*, the receiver generates a NAK message for the unacknowledged message within his window with the lowest message number. If there are no such messages, the receiver generates a NAK message for the first message outside of the window (*i.e.* the next message expected by the receiver) only if this NAK was generated by a timeout condition. The transition rule for state *srdatnak* is given in Fig. 32; it has no corresponding rule in the SWP.

Messages arriving outside of the current window require that the receiver's window be shifted forward; control passes to state *srdatrotate* to accomplish this. State *srdatrotate* rotates the receiver's window forward until the just-received datum can be placed into the message. At the same time, all the data which is shifted out of the window is stored in the output file. The transition rule is shown in Fig. 33; it corresponds to rule SlideReceiverWin of the SWP shown in Fig. 8.

9.3 Correctness

The proof of correctness for this extension to Kermit is similar to that for the bounded SWP presented in section 5. We point out here the main similarities and differences between the protocols.

The principal difference between the two protocols is that Kermit runs for a finite amount of time. The Kermit sender must wait to send the Z message until every message containing data from the current file has been successfully acknowledged. In addition, the Kermit receiver must rotate all messages in the current window out of the window (and into the output file) once the end-of-file (Z) message has been received.

Agents may re-transmit lost messages more often in the Kermit protocol than in the SWP, since the receiver can generate NAKs for missing messages at certain times. This does not affect the correctness of the protocol; it may, however, improve performance as the server may not wait as long in order to re-transmit a lost or garbled message.

The way in which the agents slide their windows is different in the two protocols. In the SWP, the sender may slide her window in parallel with the sending of new messages; in Kermit, the sender must slide her window separately from sending new messages. In the SWP, the receiver may slide his window by large amounts; in Kermit, the receiver slides his window only by single messages. The net result is the same in each case.

The size of the agents' windows is determined during the initial protocol negotiation, but is at most 32 (since Kermit uses only 64 message numbers).

Immediately we have the following theorem:

Theorem 12 *Every message sent by Sliding Windows Kermit is eventually received and successfully acknowledged.*

```
Rule: SRDAT:Windows
if Mode = srdat and Windows and Defined(FetchType) then
    if FetchType = “D” then
        DSEND(“Y”, ϵ, FetchNo)
        if WinLo ≤ FetchNo ≤ WinHi then
            WinData(FetchNo) := FetchData, WinAck(FetchNo) := true
            GOTO(srdat)
        else GOTO(srdatrotate)
        endif
    endif
    if FetchType = “Z” then
        if WinLo ≤ WinHi then
            FileText(FileNo) := FileText(FileNo) +
                                Decode(WinData(WinLo),DINFO)
            WinLo := WinLo + 1, WinAck(WinLo) := false
        else
            SeqNo := FetchNo+1, FileNo := FileNo + 1
            DSEND(“Y”, ϵ, FetchNo), GOTO(srfil)
        endif
    endif
    if FetchType = “Q” or FetchType = “T” then
        Mode := srdatnak, Counter := WinLo
    endif
    if FetchType ∉ {“D”, “Z”, “Q”, “T”} then ERROR endif
endif
```

Figure 31: Transition rule for state *srdat* using sliding windows.

```
Rule: SRDATNAK
if Mode = srdatnak then
    if Counter ≤ WinHi then
        if WinAck(Counter) = false then
            DSEND(“N”,ϵ, Counter), GOTO(srdat)
        else Counter := Counter + 1
        endif
    else
        if FetchType = “T” then DSEND(“N”,ϵ, WinHi+1) endif
        GOTO(srdat)
    endif
endif
```

Figure 32: Transition rule for state *srdatnak*.

```
Rule: SRDATROTATE
if Mode = srdatrotate then
    if WinHi+1 ≤ FetchNo then
        DSEND("N",ε, WinHi)
        if WinHi − WinLo + 1 = WinMax then
            FileText(FileNo) := FileText(FileNo) +
                                  Decode(WinData(WinLo),DINFO)
            WinLo := WinLo + 1, WinAck(WinLo) := false
        endif
        WinHi := WinHi + 1
    else
        WinData(FetchNo) := FetchData, WinAck(FetchNo) := true
        GOTO(srdat)
ENDIF
```

Figure 33: Transition rule for state *srdatrotate*.

10 Kermit Initialization

The first messages sent by each agent in a Kermit file transfer (that is, the sender's "S" message and the receiver's first "Y" message) contain initialization parameters. The combination of these two sets of parameters uniquely determine information of importance to the protocol (for example, the maximum message length to be used).

The *HANDLE-INITS* transition rules handle the setting of these parameters. We present some of these rules in Fig. 34, leaving the reader to deduce the behavior of previously unspecified functions. Rules for handling other parameter initializations are similar.

11 Partially Ordered Runs of Kermit

In all our protocol proofs presented above, we used sequential runs. In distributed computations such as those of Kermit, sequential runs are too restructive; *e.g.* moves of the sender and receiver may certainly overlap in time. Fortunately, our theorems survive when we consider partial runs (defined in [Gur]).

Our safety properties are of the form "Every state reachable in a regular run satisfies property Φ" and are proved by induction over regular sequential runs. Recall that a sequential run is *regular* if it satisfies a specified set of initial conditions. Call a partial run *regular* if its initial state satisfies the same set of initial conditions. It is shown in [Gur] that a property Φ holds in every reachable state of a partially ordered run ρ if it holds in every reachable state of every linearization of ρ; each of these linearizations is a sequential run of the program in question. Thus, every safety result for regular sequential runs gives the same result for regular partial runs.

Our liveness properties have the form "Every fair run satisfies such and such property." Our definition of sequential fair runs relies implicitly on the total ordering of steps in a run; consequently, we need a different definition of fairness for partially ordered runs. Recall that a sequential run is *fair* if for every positive agent X and every tail ρ' of ρ, if X is enabled infinitely often in ρ', then X must make a move in ρ'. We call a partially ordered run ρ *fair* if some linearization of ρ is fair.

With this new definition, it can be seen that any of our fairness properties of sequential runs can be proved over partially ordered runs as well. As an example, consider Theorem 2, which asserts that in any sequential fair run of the ABP, any datum sent is eventually received.

```
if Length(RecvdData) ≥ 1 then
    if MAXL ≥ 10 then MaxMsgLen := MAXL
    else MaxMsgLen := 80 endif
endif
if Length(RecvdData) ≥ 2 then
    if TIME > 0 then TimeoutLen := TIME
    else TimeoutLen := 5 endif
endif
if Length(RecvdData) ≥ 3 then
    if NPAD > 0 then PadLen := NPAD else PadLen := 0 endif
endif
if Length(RecvdData) ≥ 4 then PadChar := PADC endif
if Length(RecvdData) ≥ 5 then
    if 2 ≤ EOL ≤ 31 then Eol := EOL
    else Eol := CarriageReturn endif
endif
if Length(RecvdData) ≥ 6 then
    if 32 ≤ QCTL ≤ 63 or 95 ≤ QCTL ≤ 127 then CtlPrefix := QCTL
    else CtlPrefix := "#" endif
endif

where MAXL = UnChar(Substr(RecvdData,1,1))
      TIME = UnChar(Substr(RecvdData,2,1))
      NPAD = UnChar(Substr(RecvdData,3,1))
      PADC = Ctl(Substr(RecvdData,4,1))
      EOL = Unchar(Substr(RecvdData,5,1))
      QCTL = Substr(RecvdData,6,1)
```

Figure 34: Some transition rules of the *HANDLE-INITS* abbreviation.

Theorem 13 *In every fair run ρ, every datum sent is eventually received. That is, if the sender sends a datum d during a move μ, there exists a move ν during which the receiver accepts d and where $\nu > \mu$. Moreover, there is a constant N such that every initial segment of ρ containing more than N moves contains ν.*

Proof. We show first that ρ contains a move ν of the receiver accepting d, where $\mu < \nu$. Since ρ is fair, let ρ' be a linearization of ρ which is fair. Theorem 2 shows that ρ' must contain the desired ν.

Let $I = \{\mu' : \mu' \not> \nu\}$. Since ρ is a partial run, I is finite. Let $N = \|I\|$. By the definition of I, if an initial segment σ of ρ contains more than N moves, σ either contains ν or a move ν' such that $\nu < \nu'$. But since σ is downward closed, σ must contain ν as well. $\square$

All other proofs of fairness can be applied to partially ordered runs in a similar manner.

References

[BSW] 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*, volume 12 (1969), no. 5, pp. 260–261, 265.

[DaC] F. DaCruz, *Kermit: A File Transfer Protocol*, Digital Press, 1987.

[Gur] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995.

[Kr] F. Kröger, *Temporal Logic of Programs*, Springer-Verlag, 1987.

[LM] K. G. Larsen and R. Milner, "A Compositional Protocol Verification Using Relativized Bisimulation," *Information and Computation*, volume 99 (1992), no. 1, pp. 80–108.

[SL] A. U. Shankar and S. S. Lam, "A Stepwise Refinement Heuristic for Protocol Construction," *ACM Transactions on Programming Languages and Systems*, volume 14 (1992), no. 3, pp. 417–461.

[Tan] A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, 1981.

[Wal] J. Walrand, *Communication Networks: A First Course*, Aksen Associates, 1991.