Combining System Properties: A Cautionary Example and Formal Examination[1]

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

June 27, 1995

[1]This research was supported by the US Naval Research Laboratory under contract N00014-92-C-2177.

Abstract

This report presents a simple example to demonstrate that the composition of a "fault-tolerant" service and a "secure" service does not necessarily provide a secure and fault-tolerant service.

The example is originally due to Peleska, who argues that to achieve the combined service, it is necessary to strengthen one of the individual services to address both concerns. In contrast, I argue that the individual services should be "deconstructed" into smaller and weaker components that can be reassembled in different ways, and I show that the combined service can be achieved by composing the fault-tolerant service with a weaker version of the secure service.

The report also provides an introduction to the use of mechanized formal state exploration methods (specifically, the Murϕ system from Stanford) for the purpose of examining and debugging protocols.

Contents

List of Figures

Chapter 1

Introduction

In an earlier report [11], I described some of the characteristics of systems that are designed to satisfy critical properties such as dependability, safety, security, and real-time operation, and I discussed some of the techniques used in developing and providing assurance for these systems. I also examined some of the issues in building systems to satisfy two or more of these critical properties simultaneously.

In this report, I focus on a very small tutorial example. Suppose we know how to satisfy each of two requirements separately, how might we satisfy them jointly? The example I use is based on one by Jan Peleska [10], who presents one communications protocol that can tolerate message loss and another that can defeat message corruption, and considers how to develop a protocol that can deal with both problems simultaneously. Peleska suggests that the first protocol may be considered "fault tolerant," and the other "secure," so that the exercise is paradigmatic of the construction of fault-tolerant and secure systems. It might be hoped that the combined requirement could be satisfied by stacking the fault tolerant protocol on top of the secure one or vice versa, but it turns out that neither of these approaches is successful; it is necessary either to synthesize a single protocol that addresses the combined requirements, or to "deconstruct" the two protocols to provide weaker services whose combination is, paradoxically, stronger than the combination of the originals.

A second objective of this report is to exemplify the use of formal methods in examining the behavior of systems under different assumptions. This topic was also examined by Peleska, but whereas he used CSP [6] and the FDR model checker [5], I use a notation based on Unity [3] and the Murϕ state exploration tool [8].

Chapter 2

Informal Description and Analysis

A protocol specification must describe the *service* to be provided, the *assumptions* on the environment in which it executes, and the *procedural rules* governing the behavior of the participants to the protocol and the messages exchanged between them. When protocols are "stacked" one on top of another, the assumptions of the upper protocol must match the service provided by the lower one.

The service provided by the protocols considered here is to move messages reliably from a sender to a receiver, despite various faults that may afflict the underlying communications medium. This medium is assumed to provide separate *transmit* (i.e., sender to receiver) and *reply* (i.e., receiver to sender) channels. The particular kinds of faults that may afflict these channels constitute the major assumptions on the protocols. In this chapter, I present the protocols informally and explain the difficulties in combining them.

2.1 Fault Tolerance: The Alternating Bit Protocol

The alternating bit protocol (ABP) is one of the simplest and earliest protocols designed to overcome message loss in the underlying communications medium [1]. It operates as follows.

Sender: The sender attaches a single *control* bit to each message, alternating the value of the bit on successive messages. The sender transmits the message and its attached control bit and waits for a reply message carrying the same control bit. If no reply is received within some interval, or a reply is received that carries the wrong control bit, then the sender repeats the transmission until a satisfactory reply is received.

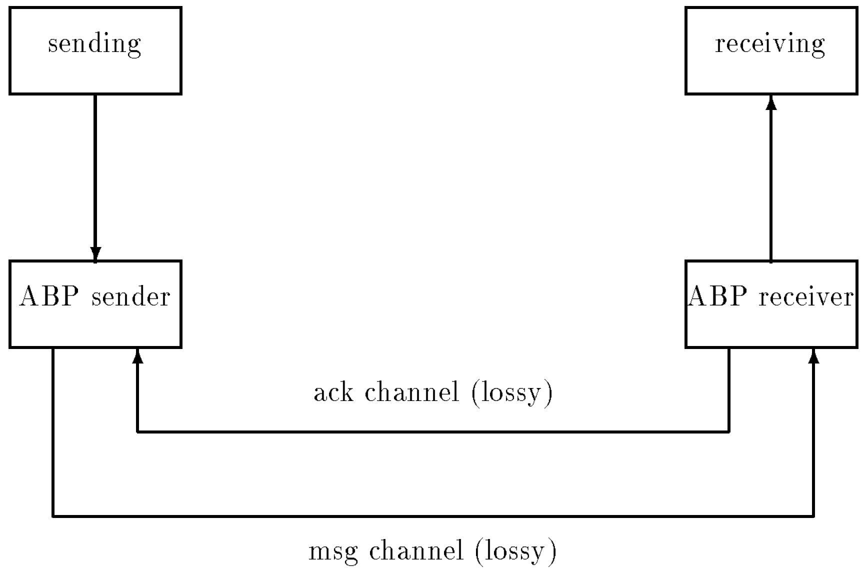


Figure 2.1: Alternating Bit Protocol

Receiver: The receiver replies to every message received with a message carrying the same control bit as the incoming message. After replying, the receiver discards messages carrying the same control bit as the previous one received, and retains those whose control bit differs from the previous one.

The ABP has only one unacknowledged message outstanding at a time. More efficient "sliding window" protocols allow several unacknowledged messages to be outstanding at a time.

The claim for ABP is that it works correctly even when messages and their replies can be lost arbitrarily. By "works correctly," I mean that the sequence of messages retained by the receiver is the same as that sent by the sender, with no messages lost, duplicated, or reordered. The assumptions are that the underlying transmit and reply channels may have several messages and replies in transit simultaneously, and can lose messages and replies but cannot change or reorder them. Despite its simplicity, the ABP is not completely straightforward to analyze: the sender may perform actions based on the absence of a reply when the reply (or several replies) is actually in transit.

2.2 Security: The Checksum Protocol

A simple way to detect message corruption is to attach a checksum to each message. The receiver can recalculate the checksum of each message received and verify that it matches the checksum that was sent with the message. Suitably sophisticated checksums can detect all corruptions within a given class (e.g., any m-bit error). The checksum protocol (CP) operates as follows.

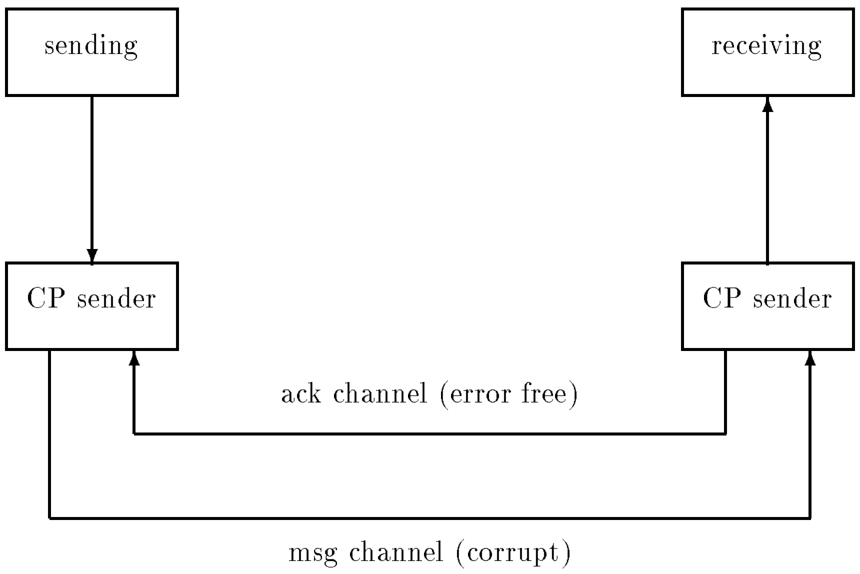


Figure 2.2: Checksum Protocol

Sender: The sender attaches a checksum to each message transmitted and awaits a reply carrying a positive or negative acknowledgment. If a positive acknowledgment is received, the sender moves on to the next message; if a negative acknowledgment is received, it retransmits the message and its attached checksum.

Receiver: The receiver checks the checksum of each message received. If it is correct, the receiver sends a reply containing a positive acknowledgment; otherwise, it sends one containing a negative acknowledgment. The receiver removes the checksum and retains those messages that it acknowledges positively, and discards those messages that it acknowledges negatively.

The claim for this protocol is that it works correctly (in the same sense as ABP) under the assumptions that the transmit channel can corrupt, but not lose messages, and that the reply channel is perfectly reliable. It is also assumed that the checksum allows message corruption to be detected with perfect reliability. These are strong (and unrealistic) assumptions, but they will serve our purpose here.

2.3 Combining Fault Tolerance and Security

We have one protocol that tolerates message loss, and another that withstands message corruption. Suppose we require a reliable message service that works in the face of both threats: it might seem that we should be able to achieve this by "stacking" one protocol on top of the other. We can do this in two ways. I will consider each separately.

2.3.1 The Checksum Protocol Above the Alternating Bit Protocol

The idea here is to use the ABP to provide the transmit channel for the CP; the ABP and CP will each have its own reply channel (see Figure 2.3). The underlying transmit channel used by the ABP is assumed to both lose and corrupt messages. Although the ABP (when used alone) can tolerate a lossy reply channel, I will assume here that both reply channels are perfectly reliable.

It seems plausible that the ABP will overcome message losses in the underlying transmit channel and thereby present the CP with a transmit service that matches its assumptions. Unfortunately, this is not so. The underlying transmit service can change, as well as lose, messages, and in particular it can change the control bit attached to messages by the ABP. This bit is not protected by the CP, since it is provided at a lower level of the protocol hierarchy. Corruption of the control bit violates the assumptions of the ABP, and it is not hard to see that it can lose or duplicate messages under these circumstances. This behavior, in turn, violates the assumptions of the CP, causing it to fail also.

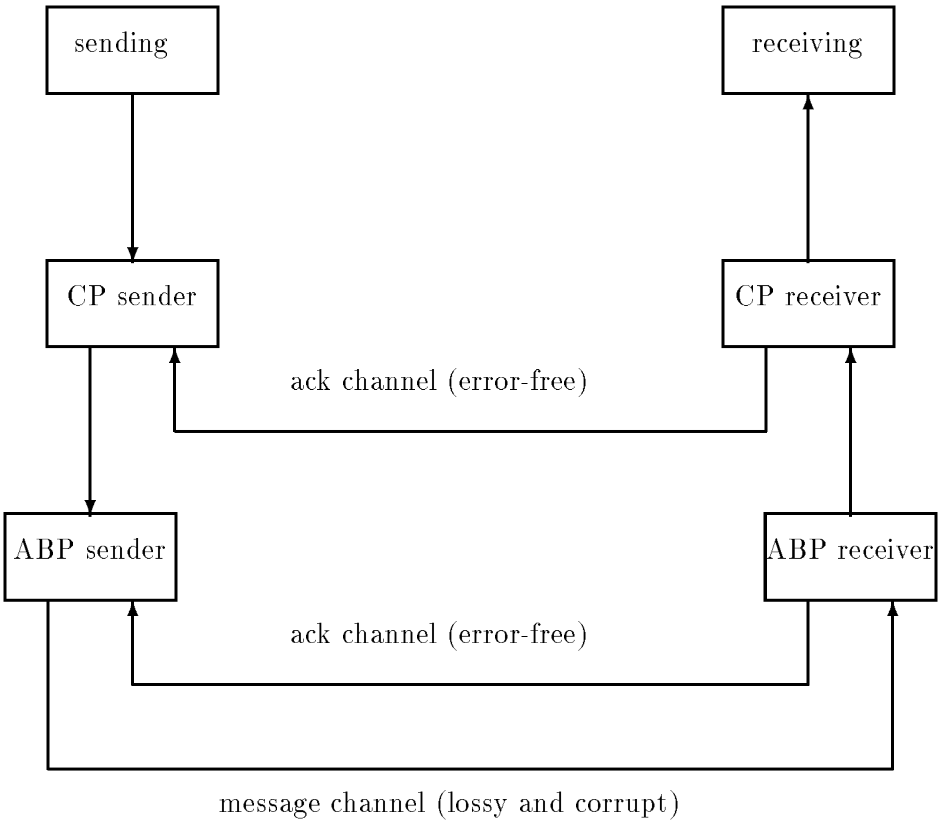


Figure 2.3: Checksum Protocol Above Alternating Bit Protocol

2.3.2 The Alternating Bit Protocol Above the Checksum Protocol

This arrangement reverses the order of the protocols: the CP provides the transmit channel for the ABP. The underlying transmit channel used by the CP is assumed to both lose and corrupt messages. As before, the CP and ABP will each have its own reply channel, which is assumed to be perfectly reliable.

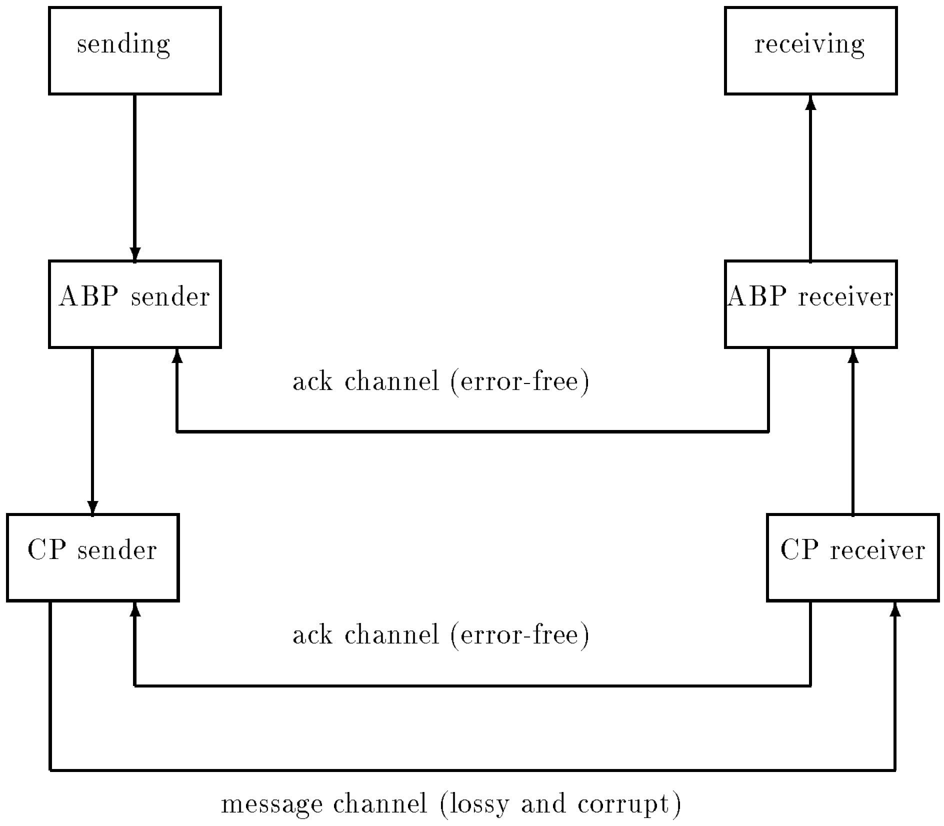


Figure 2.4: Alternating Bit Protocol Above Checksum Protocol

The problem with this arrangement is that the CP sender expects to receive a reply for every message sent. This expectation is violated when the underlying transmit channel can lose messages. In this case, the CP protocol deadlocks, with its sender waiting for a reply that will never arrive.

2.3.3 A Correct Solution

In retrospect, it is obvious that neither arrangement of one protocol stacked on top of the other can be expected to work correctly, since the underlying transmit channel does not satisfy the assumptions of either protocol. Peleska [10] argues that this means we cannot expect to solve the overall problem by stacking two individually inadequate protocols, but must design a protocol to deal specifically with the combined threat.

This is certainly one approach, but I believe there is a better one. Rather than consider that the individual protocols are too weak, or that they do too little, an alternative approach is to consider the possibility that they do too *much*. Both the individual protocols provide reliable transmission, subject to their particular assumptions. When we stack the protocols, it is unnecessary for the lower one to provide reliable transmission: all that is needed is that it should provide a service that satisfies the assumptions of the upper protocol. In particular, if we stack the ABP above the CP, then all that is required of the CP is that it provides uncorrupted, though possibly lossy, transmissions.

We can accomplish this by simply deleting the reply channel and its associated procedural rules from CP to yield the following modified protocol CP′ .

Sender: The sender attaches a checksum to each message transmitted.

Receiver: The receiver checks the checksum of each message received. If it is correct, the receiver removes the checksum and retains the remainder of the message. Otherwise, it discards the message.

The service specification for CP′ is that the sequence of messages retained is a subsequence of those transmitted (i.e., the protocol provides a lossy, but noncorrupting channel). The assumptions are that the transmit channel may lose or corrupt messages, but not reorder them (there is no reply channel).

Since the assumptions of CP′ match those of the lossy, corrupting transmit channel that we wish to deal with, and its service specification matches the assumptions on the transmit channel for ABP, it follows that stacking ABP above CP′ satisfies our requirements.

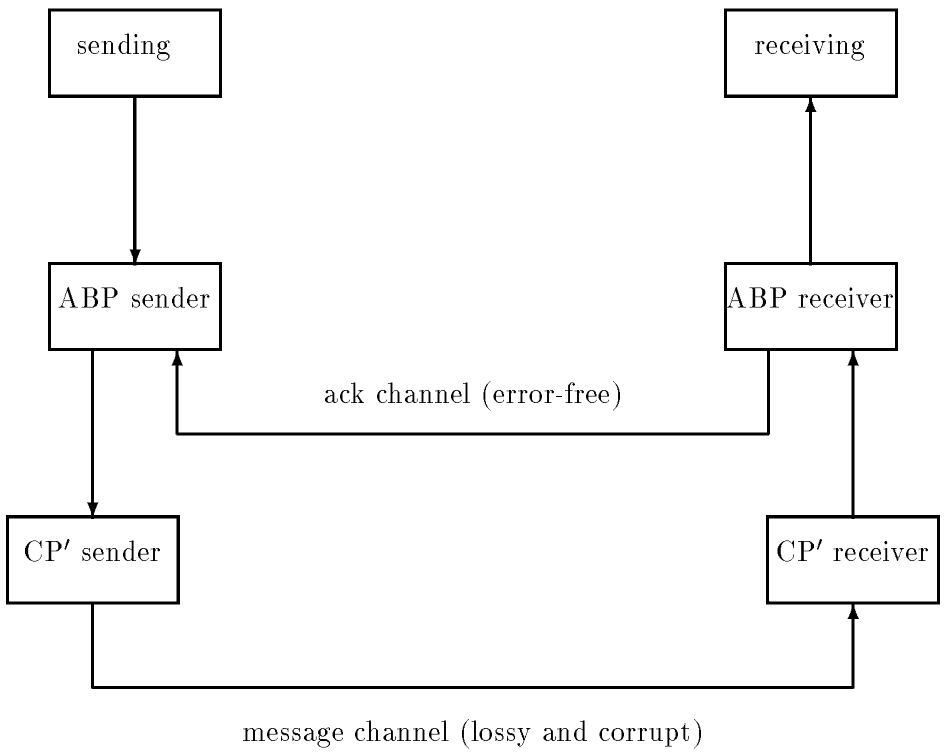


Figure 2.5: Alternating Bit Protocol Above Weakened Checksum Protocol

2.4 Discussion

Increasingly, systems are required to satisfy several critical system properties simultaneously, or to withstand multiple types of threat. The most promising way to maintain intellectual control in the face of such complexity is to "divide and conquer," that is, to construct the system from components that individually address only a single property or threat, but whose composition satisfies all the requirements. The difficulty with this approach, exemplified by the simple case study presented here, is that the composition of two components, each adequate to its individual purpose, does not necessarily yield a synthesis that achieves the combined purpose: the whole is less than the sum of its parts.

To overcome this difficulty, it is necessary to reexamine the service specifications and assumptions of the components and to adjust them so that they compose more effectively. As demonstrated by the example considered here, the most effective adjustment may sometimes be to *weaken* a component's service specification. Indeed, it seems to me likely that the demands of versatile composition will require that some familiar building blocks are "deconstructed" into smaller and weaker components that can be reassembled in many different ways. This approach is already being explored in the fields of protocols and fault tolerance, where Schlichting, and others are developing "RISC-like" building blocks for tailoring fault-tolerant systems to particular constraints and assumptions [7, 9]. My belief is that it will be necessary to extend this approach to include small building blocks for security, real-time, and safety properties, to enlarge the range of attributes (assumptions and service properties) considered for each component (e.g., real-time as well as fault-tolerance properties), and to make them more realistic (e.g., to consider average-case as well as worst-case behavior).

Chapter 3

Formal State Exploration of Protocol Properties

This chapter presents a tutorial description of how the protocols and their compositions considered in the previous chapter can be explored, systematically and with automated assistance, using formal methods.

Our ultimate goal is to develop a protocol that works in the presence of message corruption as well as loss, so at this stage in the development of the example I am less interested in formulating and verifying classical correctness of ABP, than in exploring its behavior under different assumptions and in discovering the circumstances under which it fails. For these purposes, explicit state exploration is a very effective tool. The idea in state exploration is to model the protocol as a finite-state transition system, and then to explore the reachable states of the model systematically and completely. We can mark certain states as errors and can also specify invariants that should hold for all reachable states, and can then run the state exploration to see if the invariants really do hold, and whether any error states can be reached. State exploration is similar to model checking: in both cases we completely explore a finite state model to determine whether it satisfies certain properties. The difference is that for model checking the properties are specified as logical formulas (usually in a modal logic such as "Computation Tree Logic" [CTL]) independently of the model, whereas for state exploration the properties are described through direct annotation of the model.

State exploration, model checking, and related techniques such as language inclusion differ from simulation in that they explore *all* the states of a system description; they differ from formal verification in that they can consider only finite-state systems. It is the ability to explore very large numbers of states in a reasonable time that makes state exploration techniques useful: brute-force methods can often explore millions of states in a few hours, while symbolic methods based on Binary Decision Diagrams (BDDs) can explore even larger systems (numbers such as 10^{20} [2]

and 10^{1300} [4] appear in the literature). Nonetheless, it is often necessary to consciously "downscale" a system description in order to reduce it to a size that can be explored effectively. The evidence seems to be that exploring *all* the states of a downscaled system model is often more efficient for debugging and learning about a system than visiting *some* of the states of the full system through simulation or testing. In the following section, I will describe exploration of ABP using the Murϕ state exploration system.

3.1 Exploring ABP with Murϕ

Murϕ is a state exploration system developed by David Dill's group at Stanford University [8]. Murϕ "programs" are written in a language based on Unity [3], with a C-like syntax. The Murϕ compiler generates a C++ program, which is then compiled and run to perform brute-force state exploration on the Murϕ program presented to the compiler.

The states of a Murϕ program consist of all possible assignments to the variables appearing in the program. To avoid the state explosion problem, we may need consciously to downscale the range of some of these variables. In the case of the ABP, we will need several variables that record the values of messages stored by the sender and receiver, and in transmission over the underlying communications medium. Now a real message might be, say, 8 bytes in length, giving rise to 2^{64} possible different values. With several (say n) variables recording the states of various messages, these variables alone will give rise to a state space having $n \times 2^{64}$ states—an excessively large number to explore. Now the protocol is mainly concerned with managing the control bit that it attaches to each message and is largely indifferent to the actual content of the messages transmitted, so we lose little by downscaling the size of messages to, say, one or two bits. We must be careful not to downscale too far, however, by making the messages constant (i.e., zero bits), because all messages will then look alike and we will be unable to test for delivery of wrong messages. Because the control bit alternates in value, we should probably arrange for the messages to follow a different pattern. In the program developed below, I will cause the messages produced by the sender to cycle through the values $1, 2, \ldots, M$, where M is a small number relatively prime to 2 (e.g., 3).

Another aspect of the problem that needs to be downscaled is the number of messages that can be in transit in the communications medium at any one time. The simple way to model the movement of messages is as a queue of some length N—this in turn can be programmed as an array where the sender places messages in at one end, the receiver removes them from the other, and the "network" moves all the messages along whenever there is space at the receiving end. The state space of this array will be on the order of M^N. A real communications medium, such as a wide-area network, might have a very large number N of messages in transit simultaneously, but for state exploration purposes, a small number such as two is probably sufficient. Note that eliminating the queuing altogether (i.e., setting $N = 1$) is probably not a good idea, as some classes of behaviors (and with them, the chance to detect some potential bugs) will thereby be eliminated.

The decisions described so far can be cast into Murϕ as follows.

```
const M: 3;         -- Number of different messages.  Must be >=2
type msg: 0..M+1;   -- 0 is used for empty, M+1 for bad value
const empty_msg: 0;
const bad_msg: M+1;
const ack_msg: 1;

type bit: 0..1;
const dont_care: 0;
type packet: record
  control:  bit;
  data:     msg;
end;
var empty_packet: packet;    -- initialized to [# 0, 0 #]

const N: 2; -- Max number of messages buffered in the network
type index: 1..N; -- 1 is sender's end, N is receiver's end
type channel: array [ index ] of packet;
var msg_channel, ack_channel: channel;
```

The `msg_channel` and `ack_channel` are modeled as arrays of packets, where a packet (modeled as a record) consists of a message with a control bit attached. We will need a way to indicate that a particular slot in the array is empty, and I will use a `packet` with `data` field equal to the `empty_msg` (i.e., 0) to do this. "Real" packets will have `data` fields with values in the range `1,...,M`. Murϕ does not provide a way to specify a record constant, so the `empty_packet` must be specified as a variable and initialized later in the `startstate` declaration.

Next, we need to describe how the "network" moves messages along the channels, and the kinds of faults that can arise. These are described by Murϕ "rules," which are constructions of the form *condition* $\Longrightarrow$ *action*. A Murϕ program "executes" in simulation mode by selecting some rule whose condition is satisfied in the current state, then executing the corresponding action to create a new current state, and so on, repeatedly. In simulation mode, Murϕ selects nondeterministically from among the enabled rules at each step; in state exploration mode, it examines all possible execution sequences. Functions and procedures are structuring constructs that can be called by rules.

```
function is_empty_packet(p:packet): boolean;
begin
  return p.data = empty_msg;
end;

procedure move(var c:channel);
-- Moves the messages in c along by one
-- If  efficiency matters, would be better
-- to use pointers in a circular queue
  begin
    for i := N to 2 by -1 do c[i]:=c[i-1]; endfor;
    c[1] := empty_packet;
  end;

rule "move msg channel"
  is_empty_packet(msg_channel[N]) ==>
         begin move(msg_channel); end;

rule "move ack channel"
  is_empty_packet(ack_channel[N]) ==>
         begin move(ack_channel); end;
```

Notice that the procedure `move` overwrites the packet at position N and does not check that it is an empty packet. In this program, we can see that `move` is called only by rules whose conditions ensure this property; in general, we might want to cause Murϕ to check it at run time by adding an `assert` statement to the beginning of the `move` procedure.

The ABP is required to work in the presence of arbitrary packet loss on both the message and acknowledgment channels; messages cannot be reordered or altered, however. We can specify this behavior of lossy channels with rules that simply replace the packet at some arbitrary position in each channel with the empty packet—a Murϕ *ruleset* could be used to nondeterministically select the position. To downscale this action, however, I have chosen to delete the message at the final position in the channel concerned. This can be specified as follows.

```
rule "lose msg" begin msg_channel[N] := empty_packet; end;

rule "lose ack" begin ack_channel[N] := empty_packet; end;
```

Since these rules have no conditions, they are always enabled for execution and Murϕ can choose, at each stage, whether or not to execute them. In state exploration mode, Murϕ will consider all possible combinations of these choices.

In later protocols, and also so that we can explore the behavior of ABP outside its intended domain, we will wish to allow messages to be corrupted as well as lost.

These simple rules therefore need to be extended somewhat. We begin by introducing the type phys_char that records the physical characteristics of a channel (either lossy, corrupt, lossy_corrupt, or good) and the variables phys_char_m, phys_char_a that record the characteristics of the message and ack channels, respectively.

```
type phys_char: enum {lossy, corrupt, lossy_corrupt, good};
var phys_char_m, phys_char_a: phys_char;
```

Then we can adjust the lose msg and lose ack rules to apply only when the corresponding channels are lossy or lossy_corrupt.

```
rule "lose msg"
  phys_char_m = lossy | phys_char_m = lossy_corrupt ==>
    begin msg_channel[N] := empty_packet; end;

rule "lose ack"
  phys_char_a = lossy | phys_char_a = lossy_corrupt ==>
    begin ack_channel[N] := empty_packet; end;
```

We can also add rules for corrupting messages in both the msg and ack channels—we use two rules for each channel: one that flips the control bit, and one that changes the data field to an identifiable bad message. As before, we corrupt messages only in the final position of the channel concerned.

```
rule "corrupt msg data"
  (phys_char_m = corrupt | phys_char_m = lossy_corrupt)
    & !is_empty_packet(msg_channel[N]) ==>
        begin
          msg_channel[N].data := bad_msg;
        end;

rule "corrupt msg control"
  (phys_char_m = corrupt | phys_char_m = lossy_corrupt)
    & !is_empty_packet(msg_channel[N]) ==>
        begin
          msg_channel[N].control := 1 - msg_channel[N].control;
        end;

rule "corrupt ack data"
  (phys_char_a = corrupt | phys_char_a = lossy_corrupt)
    & !is_empty_packet(ack_channel[N]) ==>
        begin
          ack_channel[N].data := bad_msg;
        end;

rule "corrupt ack control"
  (phys_char_a = corrupt | phys_char_a = lossy_corrupt)
    & !is_empty_packet(ack_channel[N]) ==>
        begin
          ack_channel[N].control := 1 - ack_channel[N].control;
        end;
```

Next, we need to provide interface procedures for sending and receiving packets (`send` and `receive`, respectively) and functions for testing whether it is permissible to perform those operations (ready-to-send and ready-to-receive, abbreviated `rtr` and `rts`, respectively). In order to be able to stack protocols on top of each other, it will be useful to parameterize these operations in terms of the service providing them. To start, we will have just three services: `abp` (the alternating-bit protocol—the service we are in the process of constructing), and `physical_m` and `physical_a` (the underlying physical message and acknowledgment channels, respectively). To start, I will specify these operations for the physical channels only.

```
type service: enum {abp, physical_m, physical_a};

function rtr(svc: service): boolean;
begin
  switch svc
    case physical_m: return !is_empty_packet(msg_channel[N]);
    case physical_a: return !is_empty_packet(ack_channel[N]);
  end;
end;

function rts(svc: service): boolean;
begin
  switch svc
    case physical_m: return is_empty_packet(msg_channel[1]);
    case physical_a: return is_empty_packet(ack_channel[1]);
  end;
end;

procedure send(svc: service; p: packet);
begin
 assert rts(svc);
  switch svc
    case physical_m: msg_channel[1] := p;
    case physical_a: ack_channel[1] := p;
  end;
end;

procedure receive(svc: service; var p: packet);
begin
  assert rtr(svc);
  switch svc
    case physical_m: p :=  msg_channel[N]; msg_channel[N] := empty_packet;
    case physical_a: p :=  ack_channel[N]; ack_channel[N] := empty_packet;
  end;
end;
```

Our task now is to use these low-level primitives to construct higher-level operations that provide reliable transmission using the ABP. The sending operation will work as follows. After accepting a new message to transmit, it generates the "next" control bit value b, attaches it to the message, and transmits the resulting packet. The sender then waits for an acknowledgment packet. There are three possibilities.

- An acknowledgment packet arrives carrying bit value b—in which case the sender marks the message as received, and is ready to accept the next message.
- An acknowledgment packet arrives carrying the "wrong" bit value b'—in which case the sender simply throws it away.

- The sender decides (probably because a timeout has expired) to send the packet again.

The receiving operation works as follows. If an incoming packet is available, the operation removes it, and sends back an acknowledgment carrying the same control bit as the incoming packet. If the control bit is different from the last one received, the message is saved; otherwise, it is discarded.

This informal presentation reveals that certain state variables must be maintained by the sender and receiver: the sender must record whether it is currently in the process of sending a message, and if so it must remember the value of that message and the current control bit; the receiver must remember the last control bit received. We can specify these variables by means of packets `sval_a` and `rval_a`, which record the packets sent and received, respectively, by the ABP: `sval_a` will be empty if there is no sending operation in progress, otherwise it will record the packet that is being sent; `rval_a` will likewise record the packet (if any) that has been received. The bits `sbit_a` and `rbit_a` will record the last control bit used by the sender and received by the receiver, respectively. In addition, `a_msg` is used to remember the last message sent. These state variables are specified in Murϕ as follows.

```
var a_msg: msg;
var sval_a, rval_a: packet;
var sbit_a, rbit_a: bit;
```

We can now extend the `rts`, `rtr`, `send`, and `receive` functions to the case of the ABP service.

```
function rtr(svc: service): boolean;
begin
  switch svc
    case physical_m: return !is_empty_packet(msg_channel[N]);
    case physical_a: return !is_empty_packet(ack_channel[N]);
    case abp:        return !is_empty_packet(rval_a);
  end;
end;

function rts(svc: service): boolean;
begin
  switch svc
    case physical_m: return is_empty_packet(msg_channel[1]);
    case physical_a: return is_empty_packet(ack_channel[1]);
    case abp:        return is_empty_packet(sval_a);
  end;
end;

procedure send(svc: service; p: packet);
begin
 assert rts(svc);
  switch svc
    case physical_m: msg_channel[1] := p;
    case physical_a: ack_channel[1] := p;
    case abp:        sval_a := p; sbit_a := next_bit(sbit_a);
                        sval_a.control := sbit_a;
  end;
end;

procedure receive(svc: service; var p: packet);
begin
  assert rtr(svc);
  switch svc
    case physical_m: p :=  msg_channel[N]; msg_channel[N] := empty_packet;
    case physical_a: p :=  ack_channel[N]; ack_channel[N] := empty_packet;
    case abp:        p:= rval_a; rval_a := empty_packet;
  end;
end;
```

The function `next_bit` computes the next control bit to use.

```
function next_bit(b: bit): bit;
begin
  return 1 - b; -- flip the bit
end;
```

Now we can program the high-level rules. The sending rule represents the user of the protocol. It generates and sends a new message whenever the previous one has been sent. It generates the new message by adding 1 modulo M to the previous message, and saves it in sval_a. It then creates a packet containing the message just generated and a clear control bit and calls the send function (which will set the correct control bit).

```
function next_msg(m: msg): msg;
begin
  return (m = M) ? 1 : m+1;
end;

rule "sending"
  rts(abp) ==>
  var p: packet;
    begin
      a_msg := next_msg(a_msg);
      clear p.control;
      p.data := a_msg;
      send(abp, p);
    end;
```

Notice that the rule "sending" just initiates the sending of a new message. We need another rule to actually send its packet out over the message channel and to handle the retransmission after a timeout. Both these actions are performed by the rule sender_a, which can fire whenever sval_a is nonempty (i.e., when !rts(abp)) and the message channel used by ABP is ready to send.

```
rule "sender_a"
  !rts(abp) & rts(physical_m) ==>
    begin send(physical_m, sval_a); end;
```

The receiving user of the ABP is able to fire whenever a message is available.

```
rule "receiving"
  rtr(abp) ==>
  var p: packet;
    begin
      receive(abp, p);
    end;
```

The lower-level message reception is enabled whenever the previous message has been removed (i.e., !rtr(abp)), the physical message channel is ready to receive, and the physical ack channel is ready to send. It returns a packet with the same control bit as the one just received (the message in the packet is irrelevant, but I will set it to ack_msg); if that bit is different than the control bit of the previous message, it places in rval_a and sets rbit_a to the value of that bit.

```
rule "receiver_a"
  !rtr(abp) & rtr(physical_m) & rts(physical_a) ==>
    var p, q: packet;
    begin
      receive(physical_m, p);
      q := p;
      q.data := ack_msg;
      send(physical_a, q);
      if p.control != rbit_a then
        rval_a := p;
        rbit_a := p.control;
      end;
    end;
```

Back at the sending end, we need a rule to deal with the returned acknowledgment packet. This rule is enabled whenever a packet is available on the physical acknowledgment channel and the ABP is in the process of sending a message (i.e., `!rts(abp)`). (If a packet becomes available when the ABP is not in the process of sending a message, it will just sit there until a new message transmission begins.)

```
rule "check abp ack"
  !rts(abp) & rtr(physical_a) ==>
    var p: packet;
    begin
      receive(physical_a, p);
      if p.control = sval_a.control then
         sval_a := empty_packet;
      end;
    end;
```

To start things off, we need to initialize the various state variables.

```
startstate
begin
  empty_packet.data := empty_msg;
  clear empty_packet.control;
  for i: index do msg_channel[i] := empty_packet;
                  ack_channel[i] := empty_packet;
  endfor;
  clear rbit_a;
  clear sbit_a;
  clear a_msg;
  sval_a := empty_packet;
  rval_a := empty_packet;
  phys_char_m := lossy;
  phys_char_a := lossy;
end;
```

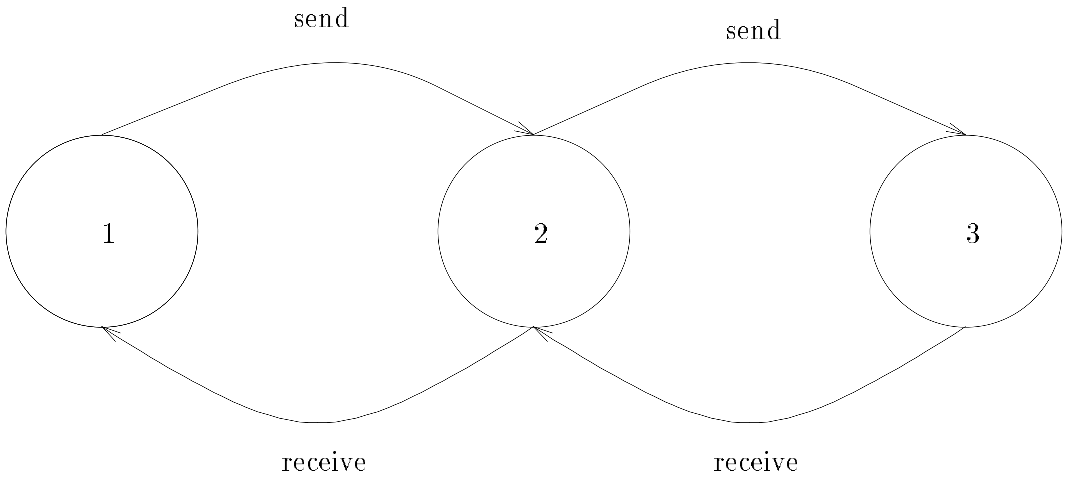
At this stage we have enough specified to run Murϕ in simulation mode and can check from its trace that things seem to be working properly. To do significant debugging, however, we must provide some invariants and assertions that can be tested under full state exploration. One key property required of the protocol is that no messages should be lost or reordered. This can be partially ensured by checking that each message received is indeed the message that was sent—and this check can be performed by adding the assertion

```
assert rval_a.data = a_msg;
```

to the rule `receiver_a`.

In state exploration mode, Murϕ verifies this assertion in 4.93 seconds after visiting 2113 states and firing 9,305 rules. However, it also verifies the assertion when we modify the program so that the control bit does not alternate (i.e., when the body of `next_bit` is changed from `return 1-b` to `return b`)! It is clear that we need some additional and stronger assertions if we are to gain confidence in the correctness of our model of ABP.

Now the intuitive correctness requirement for ABP is that it masks message loss and presents a service that is equivalent to a perfect buffer. Because it is possible for the transmitter to start sending a new message before the receiver has collected the previous one, the buffer may contain as many as two messages. Therefore, one way to check correctness of ABP is to compare its top-level behavior against that of a two-place buffer. Klaus Havelund of the University of Paris VI first suggested this approach to me, and he also suggested how it can be implemented in Murϕ. The idea is to describe the required behavior of the protocol by the following three-state automaton.



Any attempt to receive in state 1, or to send in state 3 is an error; furthermore a receive in state 3 should return the message sent in the transition from state 2, and a receive in state 2 should return the message sent in the transition from state 1. This can be programmed in Murϕ as follows.

```
procedure automaton(x: action; m: msg);
begin
  switch state
   case 1:
     if x=snd then the_msg := m; state := 2;
       else error "*** receive in state 1";
     end;
   case 2:
     if x=snd then another_msg := m; state := 3;
       else
         if m != the_msg then error "*** wrong message received(1)"; end;
         state := 1;
     end;
   case 3:
     if x=snd then error "*** send in state 3";
       else
         if m != the_msg then error "*** wrong message received(2)"; end;
         the_msg := another_msg; state := 2;
     end;
  end;
end;
```

We then place the call `automaton(snd, a_msg)` in the rule `sending`, and the call `automaton(rcv, p.data)` in the rule `receiving`.

In this way we establish a simulation, or abstraction, relation between the modeled implementation of ABP and its requirement specification. With this more sophisticated correctness check, Murϕ detects the error in the protocol that does not alternate the bit (in 0.56 second after exploring 48 states and firing 176 rules). Murϕ reports the error "send in state 3" and its backtrace indicates that one circumstance that triggers the incorrect behavior is when the sender retransmits a message that has already been received—resulting in duplicate reception. When the alternation of the control bit is restored, Murϕ verifies correctness as before in 4.93 seconds after visiting 2113 states and firing 9,305 rules. If we change the characteristics of the channels by, for example, changing `phys_char_m := lossy` to `phys_char_m := corrupt` in the `startstate`, Murϕ again detects an error (wrong message received in state 1; 0.52 second, 26 states explored, and 72 rules fired), thereby demonstrating that ABP cannot cope with corrupt channels.

3.2 Exploring CP with Murϕ

To explore the CP protocol, we must add some additional state variables to our Murϕ program. First, we need to extend the packet type to include a checksum bit.

```
type packet: record
  checksum: bit;
  control:  bit;
  data:     msg;
end;
```

The checksum is set to 1 to model a good packet, and to 0 to model one that has been (detectably) corrupted. The rules for corrupting messages are then changed to the following.

```
rule "corrupt msg data"
  (phys_char_m = corrupt | phys_char_m = lossy_corrupt)
    & !is_empty_packet(msg_channel[N]) ==>
         begin
           msg_channel[N].data := bad_msg;
           msg_channel[N].checksum := 0;
         end;

rule "corrupt msg control"
  (phys_char_m = corrupt | phys_char_m = lossy_corrupt)
    & !is_empty_packet(msg_channel[N]) ==>
         begin
           msg_channel[N].control := 1 - msg_channel[N].control;
           msg_channel[N].checksum := 0;
         end;
```

Because there is no checksum protection on the acknowledgment channel, the rules `corrupt ack data` and `corrupt ack control` are not changed from their previous definitions.

As with the ABP, we need state variables to remember the packets being sent and received, and a bit to record whether the protocol is busy.

```
var sval_c, rval_c: packet;
var busy_c: boolean;
```

These are initialized in the `startstate`, together with the characteristics of the physical channels.

```
startstate
  ...
  busy_c := false;
  sval_c := empty_packet;
  rval_c := empty_packet;
  phys_char_m := corrupt;  -- can be corrupt
  phys_char_a := good;     -- must be good
```

Then we extend `rtr` and `rts`.

```
function rtr(svc: service): boolean;
begin
  switch svc
    case physical_m:  return !is_empty_packet(msg_channel[N]);
    case physical_a:  return !is_empty_packet(ack_channel[N]);
    case abp:         return !is_empty_packet(rval_a);
    case cp:          return !is_empty_packet(rval_c);
  end;
end;

function rts(svc: service): boolean;
begin
  switch svc
    case physical_m:  return is_empty_packet(msg_channel[1]);
    case physical_a:  return is_empty_packet(ack_channel[1]);
    case abp:         return is_empty_packet(sval_a);
    case cp:          return is_empty_packet(sval_c);
  end;
end;
```

And, similarly, we extend `send` and `receive`.

```
procedure send(svc: service; p: packet);
begin
 assert rts(svc);
  switch svc
    case physical_m:  msg_channel[1] := p;
    case physical_a:  ack_channel[1] := p;
    case abp:         sval_a := p; sbit_a := next_bit(sbit_a);
                        sval_a.control := sbit_a;
    case cp:          sval_c := p; sval_c.checksum := 1;
  end;
end;

procedure receive(svc: service; var p: packet);
begin
  assert rtr(svc);
  switch svc
    case physical_m:  p :=  msg_channel[N];  msg_channel[N] := empty_packet;
    case physical_a:  p :=  ack_channel[N];  ack_channel[N] := empty_packet;
    case abp:         p:= rval_a; rval_a := empty_packet;
    case cp:          p:= rval_c; rval_c := empty_packet;
  end;
end;
```

The CP sender rule is enabled when the protocol is not busy. It marks the protocol as busy and calls the send procedure.

```
rule "sender_c"
  !is_empty_packet(sval_c) & rts(physical_m) & !busy_c ==>
    begin
      busy_c := true;
      send(physical_m, sval_c);
    end;
```

The receiver examines the checksum on any packet received and returns either a positive or negative acknowledgment packet according to whether or not the checksum is good.

```
const nack_msg: 2;

rule "receiver_c"
  is_empty_packet(rval_c) & rtr(physical_m) & rts(physical_a) ==>
    var p, q: packet;
    begin
      receive(physical_m, p);
      q := p;
      if p.checksum = 0 then
        q.data := nack_msg;
      else
        rval_c := p;
        q.data := ack_msg;
      end;
      send(physical_a, q);
    end;
```

When the sender receives an acknowledgment packet, it turns off its busy flag if the acknowledgment is positive; otherwise, it retransmits the packet.

```
rule "check cp ack"
  busy_c & rtr(physical_a) & rts(physical_m) ==>
    var p: packet;
    begin
      receive(physical_a, p);
      if p.data = ack_msg then
        sval_c := empty_packet;
        busy_c := false;
      else
        send(physical_m, sval_c);
      end;
    end;
```

To complete the protocol, we must modify the `sending` and `receiving` rules to use the CP rather than ABP protocol. We parameterize this by means of the `top` state variable, which is initialized to `cp` in the startstate.

```
var top: service;

rule "sending"
  rts(top) ==>
  var p: packet;
    begin
      a_msg := next_msg(a_msg);
      automaton(snd, a_msg);
      clear p.control;
      p.data := a_msg;
      send(top, p);
    end;

rule "receiving"
  rtr(top) ==>
  var p: packet;
    begin
      receive(top, p);
      automaton(rcv, p.data);
    end;
```

We also need to make sure that the ABP `receiver_a` rule does not grab a packet off the `physical_m` channel. We can do this by making sure `rval_a` is initialized to a nonempty packet. (The ABP `sending_a` and `check abp ack` rules will not get in the way because `sval_a` is initialized to an empty packet.)

```
var empty_packet, nonempty_packet: packet;

startstate
  ...
  nonempty_packet.data := ack_msg;
  clear nonempty_packet.control;
  clear nonempty_packet.checksum;

  rval_a := nonempty_packet;

  top := cp;
```

Exploration with Murϕ quickly establishes that the CP protocol is correct as described (exploration visits 226 states and fires 684 rules in 1.12 seconds). It also reveals that the protocol cannot cope with a lossy as well as corrupt message channel (deadlock found after exploring 14 states and firing 33 rules in 0.6 second), nor with a corrupt acknowledgment channel (an erroneous "receive in state 1" is detected after exploring 53 states and firing 151 rules in 0.73 second).

3.3 Exploring the Combined Protocols with Murϕ

We now have sufficient machinery in place to explore the behavior of ABP above CP, and vice versa, in the face of a lossy and corrupt message channel. In order to allow one protocol to use the other as its message channel, we introduce `abp_m`, `abp_a`, `cp_m`, `cp_a` as names for the services used by ABP for messages and acknowledgments, and by CP for messages and acknowledgments, respectively. Then, all references to `physical_a` and `physical_m` in the rules for ABP and CP are changed to use these names, which are initialized appropriately in the start state: for example, the initialization for ABP above the CP protocol reads, in part, as follows (where `physical_a2` is a second acknowledgment channel, defined identically to the first).

```
  phys_char_m := lossy_corrupt;
  phys_char_a := good;
  phys_char_a2 := good;
  top := abp;
  abp_m := cp;
  abp_a := physical_a2;
  cp_m := physical_m;
  cp_a := physical_a;;
```

An important change must be made to the rule for corrupting the ABP control bit on the message channel: this is detected (i.e., the checksum bit is set to zero) only if CP provides the message channel for ABP (i.e., if `abp_m = cp`.

```
rule "corrupt msg control"
  (phys_char_m = corrupt | phys_char_m = lossy_corrupt)
    & !is_empty_packet(msg_channel[N]) ==>
         begin
           msg_channel[N].control := 1 - msg_channel[N].control;
           if abp_m = cp then msg_channel[N].checksum := 0; end;
         end;
```

The Murϕ program for the combinations of the two protocols is shown in Appendix A.

Murϕ quickly establishes that neither combination of ABP and CP works in the presence of a lossy and corrupt message channel and good acknowledgment channels. When ABP is above CP, Murϕ reports a deadlocked state after exploring only 15 states and firing 46 rules in 0.65 second. The backtrace provides the following scenario that manifests the problem: the very first message sent is lost by the message channel, causing the CP sender to deadlock waiting for an acknowledgment that will never arrive.

When CP is above ABP, Murϕ explores 595 states and fires 2419 rules in 2.81 seconds before reporting "receive in state 1." The backtrace provides the following scenario that manifests the fault: a message is sent and correctly received; before its acknowledgment is received, the ABP sender times out and sends another copy; the control bit of this message is corrupted (since this is below the protection of the CP, this is not detected), causing the receiver to mistake it for a new message.

3.3.1 Exploring a Correct Protocol with Murϕ

By simply deleting the ack channel and its associated checks and retransmissions from CP, we obtain the protocol CP$'$. Murϕ is able to establish that ABP above CP$'$ correctly solves the problem of tolerating a lossy corrupt message channel, although the number of states it must explore for this purpose is quite large:

- 28,273 states explored and 180,053 rules fired in 149.08 seconds when the acknowledgment channel is good, and
- 30,577 states explored and 226,182 rules fired in 179.27 seconds when the acknowledgment channel is lossy.

When the acknowledgment channel is corrupt, Murϕ detects a fault ("send in state 3") after exploring 4,826 states and firing 30,714 rules in 24.87 seconds. The scenario that manifests the fault is 27 transitions long. If CP$'$ is used on the ack channel as well as the message channel, then the combined protocol can tolerate loss and corruption on both channels.

Chapter 4

Conclusion

We have seen, by means of simple examples, that it is not straightforward to combine mechanisms for separate problems to achieve a mechanism for the combined problem, and that formal state exploration can be an effective tool for debugging and exploring the behavior of systems that involve complex interactions.

The most interesting result of our experiment involving the ABP and CP protocols is the discovery that one of these protocols needs to be weakened, rather than strengthened, to provide the most suitable building block for larger compositions. Interesting work for the future lies in characterizing those system specifications and assumptions that are suitable for composition and distinguishing them from those that are not.

Our examination of these protocols with Murϕ demonstrates the utility of the simple technology of explicit state exploration. More sophisticated forms of analysis for finite systems include model checking with respect to temporal logic specifications and testing language containment on ω-automata. Neither of these techniques permits as simple or complete a specification of required behavior as the method—due to Klaus Havelund—that we employed in Murϕ: namely, direct comparison against an automaton that implements the requirements specification. Furthermore, the explicit state enumeration employed by Murϕ (as opposed to the BDD representations used in more sophisticated systems) is fully adequate for the simple protocols considered here.

Bibliography

[1] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260, 261, May 1969.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[3] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[4] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *19th ACM Symposium on Principles of Programming Languages*, pages 343–354, Albuquerque, NM, January 1992. Association for Computing Machinery.

[5] *Failures Divergence Refinement: User Manual and Tutorial*. Formal Systems (Europe) Ltd, Oxford, UK, 1.2β edition, October 1992.

[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1985.

[7] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[8] Ralph Melton and David L. Dill. *Murϕ Annotated Reference Manual*. Computer Science Department, Stanford University, Stanford, CA, March 1993.

[9] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal*, 1(2):87–103, December 1993.

[10] Jan Peleska. On a unified formal approach for the development of fault-tolerant and secure systems. In Hans Rischel, editor, *Nordic Seminar on Dependable Computing Systems*, pages 69–80, Lyngby, Denmark, August 1994. Technical University of Denmark.

[11] John Rushby. Critical system properties: Survey and taxonomy. Technical Report SRI-CSL-93-1, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1993. Also available in *Reliability Engineering and System Safety*, Vol. 43, No. 2, pp. 189–219, 1994.

Appendix A

Listing of the full Murphi Program

```
const M: 3;          -- Number of different messages.  Must be >=2
type msg: 0..M+1;    -- 0 is used for empty, M+1 for bad value
const empty_msg: 0;
const ack_msg: 1;
const nack_msg: 2;
const bad_msg: M+1;

type bit: 0..1;
const dont_care: 0;
type packet: record
  checksum: bit;
  control:  bit;
  data:     msg;
end;
var empty_packet, nonempty_packet: packet;

const N: 2; -- Max number of messages buffered in the network xx
type index: 1..N; -- 1 is sender's end, N is receiver's end
type channel: array [ index ] of packet;
var msg_channel, ack_channel, ack2_channel: channel;

var sval_c, rval_c: packet;
var busy_c: boolean;

type phys_char: enum lossy, corrupt, lossy_corrupt, good;
var phys_char_m, phys_char_a, phys_char_a2: phys_char;

type service: enum not_used, abp, cp, physical_m, physical_a, physical_a2;
```

```
var top, abp_m, abp_a, cp_m, cp_a: service;

var a_msg: msg;
var sval_a, rval_a: packet;
var sbit_a, rbit_a: bit;

type states: 1..3;
type action: enumsnd, rcv;
var state: states; -- initially 1
var the_msg, another_msg: msg;  -- initially clear
type err: 1..4;

procedure squawk(n:err);
begin
  switch n
   case 1: error "****** receive in state 1";
   case 2: error "****** wrong message received(1)";
   case 3: error "****** send in state 3"; -- xx
   case 4: error "****** wrong message received(2)";
  end;
end;

procedure automaton(x: action; m: msg);
begin
  switch state
   case 1:
     if x=snd then the_msg := m; state := 2;
       else squawk(1);
     end;
   case 2:
     if x=snd then another_msg := m; state := 3;
       else
         if m != the_msg then squawk(2); end;
--         assert m = the_msg;
         state := 1;
     end;
   case 3:
     if x=snd then squawk(3);
       else
         if m != the_msg then squawk(4); end;
--         assert m = the_msg;
         the_msg := another_msg; state := 2;
     end;
  end;
end;
```

```
function is_empty_packet(p:packet): boolean;
begin
  return p.data = empty_msg;
end;

procedure move(var c:channel);
-- Moves the messages in c along by one
-- If  efficiency matters, would be better
-- to use pointers in a circular queue
  begin
    for i := N to 2 by -1 do c[i]:=c[i-1]; endfor;
    c[1] := empty_packet;
  end;

function next_bit(b: bit): bit;
begin
--  return b;
  return 1 - b; -- flip the bit
end;

function next_msg(m: msg): msg;
begin
  return (m = M) ? 1 : m+1;
end;

function rtr(svc: service): boolean;
begin
  switch svc
    case physical_m:  return !is_empty_packet(msg_channel[N]);
    case physical_a:  return !is_empty_packet(ack_channel[N]);
    case physical_a2: return !is_empty_packet(ack2_channel[N]);
    case abp:         return !is_empty_packet(rval_a);
    case cp:         return !is_empty_packet(rval_c);
  end;
end;

function rts(svc: service): boolean;
begin
  switch svc
    case physical_m:  return is_empty_packet(msg_channel[1]);
    case physical_a:  return is_empty_packet(ack_channel[1]);
    case physical_a2: return is_empty_packet(ack2_channel[1]);
    case abp:         return is_empty_packet(sval_a);
    case cp:         return is_empty_packet(sval_c);
  end;
end;
```

```
procedure send(svc: service; p: packet);
begin
 assert rts(svc);
  switch svc
    case physical_m:  msg_channel[1] := p;
    case physical_a:  ack_channel[1] := p;
    case physical_a2: ack2_channel[1] := p;
    case abp:         sval_a := p; sbit_a := next_bit(sbit_a);
                        sval_a.control := sbit_a;
    case cp:         sval_c := p; sval_c.checksum := 1;
  end;
end;

procedure receive(svc: service; var p: packet);
begin
  assert rtr(svc);
  switch svc
    case physical_m:  p :=  msg_channel[N];  msg_channel[N] := empty_packet;
    case physical_a:  p :=  ack_channel[N];  ack_channel[N] := empty_packet;
    case physical_a2: p :=  ack2_channel[N]; ack2_channel[N] := empty_packet;
    case abp:         p:= rval_a; rval_a := empty_packet;
    case cp:         p:= rval_c; rval_c := empty_packet;
  end;
end;

procedure init_abp();
begin
  phys_char_m := lossy;
  phys_char_a := lossy;
  top := abp;
  abp_m := physical_m;
  abp_a := physical_a;
  cp_a := not_used;
  cp_m := not_used;
  rval_c := nonempty_packet;
  put "Alternating Bit Protocol";
end;

procedure init_cp();
begin
  phys_char_m := corrupt;
  phys_char_a := good;
  top := abp;
  abp_m := not_used;
  abp_a := not_used;
```

```
  cp_a := physical_m;;
  cp_m := physical_a;
  rval_a := nonempty_packet;
  put "Checksum Protocol";
end;


procedure init_abp_cp();
begin
  phys_char_m := lossy_corrupt;
  phys_char_a := good;
  phys_char_a2 := good;
  top := abp;
  abp_m := cp;
  abp_a := physical_a2;
  cp_m := physical_m;
  cp_a := physical_a;;
  put "Alternating Bit above Checksum Protocol";
end;

procedure init_cp_abp();
begin
  phys_char_m := lossy_corrupt;
  phys_char_a := good;
  phys_char_a2 := good;
  top := cp;
  abp_m := physical_m;
  abp_a := physical_a;
  cp_m :=  abp;
  cp_a :=  physical_a2;;
  put "Checksum above Alternating Bit Protocol";
end;


rule "move msg channel"
  is_empty_packet(msg_channel[N]) ==>
       begin move(msg_channel); end;

rule "move ack channel"
  is_empty_packet(ack_channel[N]) ==>
       begin move(ack_channel); end;

rule "move ack2 channel"
  is_empty_packet(ack2_channel[N]) ==>
       begin move(ack2_channel); end;
```

```
rule "lose msg"
  phys_char_m = lossy | phys_char_m = lossy_corrupt ==>
    begin msg_channel[N] := empty_packet; end;

rule "lose ack"
  phys_char_a = lossy | phys_char_a = lossy_corrupt ==>
    begin ack_channel[N] := empty_packet; end;

rule "lose ack2"
  phys_char_a2 = lossy | phys_char_a2 = lossy_corrupt ==>
    begin ack2_channel[N] := empty_packet; end;

rule "corrupt msg data"
  (phys_char_m = corrupt | phys_char_m = lossy_corrupt)
    & !is_empty_packet(msg_channel[N]) ==>
         begin
           msg_channel[N].data := bad_msg;
           msg_channel[N].checksum := 0;
         end;

rule "corrupt msg control"
  (phys_char_m = corrupt | phys_char_m = lossy_corrupt)
    & !is_empty_packet(msg_channel[N]) ==>
         begin
           msg_channel[N].control := 1 - msg_channel[N].control;
           if abp_m = cp then msg_channel[N].checksum := 0; end;
         end;

rule "corrupt ack data"
  (phys_char_a = corrupt | phys_char_a = lossy_corrupt)
    & !is_empty_packet(ack_channel[N]) ==>
         begin
           ack_channel[N].data := bad_msg;
         end;

rule "corrupt ack control"
  (phys_char_a = corrupt | phys_char_a = lossy_corrupt)
    & !is_empty_packet(ack_channel[N]) ==>
         begin
           ack_channel[N].control := 1 - ack_channel[N].control;
         end;

rule "corrupt ack2 data"
  (phys_char_a2 = corrupt | phys_char_a2 = lossy_corrupt)
    & !is_empty_packet(ack2_channel[N]) ==>
         begin
```

```
          ack2_channel[N].data := bad_msg;
        end;

rule "corrupt ack2 control"
  (phys_char_a2 = corrupt | phys_char_a2 = lossy_corrupt)
    & !is_empty_packet(ack2_channel[N]) ==>
        begin
          ack2_channel[N].control := 1 - ack2_channel[N].control;
        end;

rule "sending"
  rts(top) ==>
  var p: packet;
    begin
      a_msg := next_msg(a_msg);
      automaton(snd, a_msg);
      clear p.control;
      p.data := a_msg;
      send(top, p);
    end;

rule "sender_a"
  !rts(abp) & rts(abp_m) ==>
    begin send(abp_m, sval_a); end;

rule "receiving"
  rtr(top) ==>
  var p: packet;
    begin
      receive(top, p);
      automaton(rcv, p.data);
    end;

rule "receiver_a"
  !rtr(abp) & rtr(abp_m) & rts(abp_a) ==>
    var p, q: packet;
    begin
      receive(abp_m, p);
      q := p;
      q.data := ack_msg;
      send(abp_a, q);
      if p.control != rbit_a then
 rval_a := p;
 rbit_a := p.control;
      end;
    end;
```

```
rule "check abp ack"
  !rts(abp) & rtr(abp_a) ==>
    var p: packet;
    begin
      receive(abp_a, p);
      if p.control = sval_a.control then
         sval_a := empty_packet;
      end;
    end;

rule "sender_c"
  !is_empty_packet(sval_c) & rts(cp_m) & !busy_c ==>
    begin
      busy_c := true;
      send(cp_m, sval_c);
    end;

rule "receiver_c"
  is_empty_packet(rval_c) & rtr(cp_m) & rts(cp_a) ==>
    var p, q: packet;
    begin
      receive(cp_m, p);
      q := p;
      if p.checksum = 0 then
         q.data := nack_msg;
      else
        rval_c := p;
        q.data := ack_msg;
      end;
      send(cp_a, q);
    end;

rule "check cp ack"
  busy_c & rtr(cp_a) & rts(cp_m) ==>
    var p: packet;
    begin
      receive(cp_a, p);
      if p.data = ack_msg then
        sval_c := empty_packet;
        busy_c := false;
      else
        send(cp_m, sval_c);
      end;
    end;
```

```
startstate
begin
  empty_packet.data := empty_msg;
  clear empty_packet.control;
  clear empty_packet.checksum;

  nonempty_packet.data := ack_msg;
  clear nonempty_packet.control;
  clear nonempty_packet.checksum;

  for i: index do msg_channel[i] := empty_packet;
                  ack_channel[i] := empty_packet;
                  ack2_channel[i] := empty_packet;
  endfor;
  clear rbit_a;
  clear sbit_a;
  clear a_msg;
  sval_a := empty_packet;
  rval_a := empty_packet;
  busy_c := false;
  sval_c := empty_packet;
  rval_c := empty_packet;

  state := 1;
  clear the_msg;
  clear another_msg;
  init_abp_cp();

end;
```

Appendix B

Error Trace for ABP Above CP' With Corrupt Acknowledgment Channel

```
==========================================================================
Murphi Beta Release 2.73S (With Symmetry)
Finite-state Concurrent System Verifier.

Copyright (C) 1992, 1993
by the Board of Trustees of Leland Stanford Junior University.

==========================================================================
This program should be regarded as a DEBUGGING aid, not as a
certifier of correctness.
Call with the -l flag or read the license file for terms
and conditions of use.
Run this program with "-h" for the list of options.

Bugs, questions, and comments should be directed to
"murphi@snooze.stanford.edu".

Murphi compiler last modified date: Apr 14 1994
Include files   last modified date: Apr 14 1994
==========================================================================

Algorithm:
Verification by breadth first search.
with symmetry algorithm 1 -- fast canonicalization.

Memory usage:
```

```
* The size of each state is 125 bits (rounded up to 16 bytes).
* The memory allocated for the hash table is 2 Mbytes.
  With two words of overhead per state, the maximum size of
  the state space is 80021 states.
   * Use option "-k" or "-m" to increase this, if necessary.
* Capacity in queue for breadth-first search: 20005 states.
   * Change the constant gPercentActiveStates in mu_verifier.h
     to increase this, if necessary.
Alternating Bit above Modified Checksum Protocol

Progress Report:

1000 states explored in 5.33s, with 6046 rules fired and 183 states in the queue.
2000 states explored in 10.11s, with 12130 rules fired and 352 states in the queue.
3000 states explored in 15.06s, with 18438 rules fired and 529 states in the queue.
4000 states explored in 20.44s, with 25275 rules fired and 645 states in the queue.

The following is the error trace:

Startstate Startstate 0 fired.
empty_packet.checksum : 0
empty_packet.control : 0
empty_packet.data : 0
nonempty_packet.checksum : 0
nonempty_packet.control : 0
nonempty_packet.data : 1
msg_channel[1].checksum : 0
msg_channel[1].control : 0
msg_channel[1].data : 0
msg_channel[2].checksum : 0
msg_channel[2].control : 0
msg_channel[2].data : 0
ack_channel[1].checksum : 0
ack_channel[1].control : 0
ack_channel[1].data : 0
ack_channel[2].checksum : 0
ack_channel[2].control : 0
ack_channel[2].data : 0
ack2_channel[1].checksum : 0
ack2_channel[1].control : 0
ack2_channel[1].data : 0
ack2_channel[2].checksum : 0
ack2_channel[2].control : 0
ack2_channel[2].data : 0
sval_c.checksum : 0
```

```
sval_c.control : 0
sval_c.data : 0
rval_c.checksum : 0
rval_c.control : 0
rval_c.data : 0
busy_c : false
phys_char_m:lossy_corrupt
phys_char_a:corrupt
phys_char_a2:good
top:abp
abp_m:cp
abp_a:physical_a
cp_m:physical_m
cp_a:absent
a_msg : 0
sval_a.checksum : 0
sval_a.control : 0
sval_a.data : 0
rval_a.checksum : 0
rval_a.control : 0
rval_a.data : 0
sbit_a : 0
rbit_a : 0
state : 1
the_msg : 0
another_msg : 0
----------

Rule sending fired.
a_msg : 1
sval_a.checksum:Undefined
sval_a.control : 1
sval_a.data : 1
sbit_a : 1
state : 2
the_msg : 1
----------

Rule sender_a fired.
sval_c.checksum : 1
sval_c.control : 1
sval_c.data : 1
----------

Rule sender_c fired.
msg_channel[1].checksum : 1
```

```
msg_channel[1].control : 1
msg_channel[1].data : 1
sval_c.checksum : 0
sval_c.control : 0
sval_c.data : 0
----------

Rule sender_a fired.
sval_c.checksum : 1
sval_c.control : 1
sval_c.data : 1
----------

Rule move msg channel fired.
msg_channel[1].checksum : 0
msg_channel[1].control : 0
msg_channel[1].data : 0
msg_channel[2].checksum : 1
msg_channel[2].control : 1
msg_channel[2].data : 1
----------

Rule receiver_c fired.
msg_channel[2].checksum : 0
msg_channel[2].control : 0
msg_channel[2].data : 0
rval_c.checksum : 1
rval_c.control : 1
rval_c.data : 1
----------

Rule sender_c fired.
msg_channel[1].checksum : 1
msg_channel[1].control : 1
msg_channel[1].data : 1
sval_c.checksum : 0
sval_c.control : 0
sval_c.data : 0
----------

Rule receiver_a fired.
ack_channel[1].checksum : 1
ack_channel[1].control : 1
ack_channel[1].data : 1
rval_c.checksum : 0
rval_c.control : 0
```

```
rval_c.data : 0
rval_a.checksum : 1
rval_a.control : 1
rval_a.data : 1
rbit_a : 1
----------

Rule receiving fired.
rval_a.checksum : 0
rval_a.control : 0
rval_a.data : 0
state : 1
----------

Rule sender_a fired.
sval_c.checksum : 1
sval_c.control : 1
sval_c.data : 1
----------

Rule move ack channel fired.
ack_channel[1].checksum : 0
ack_channel[1].control : 0
ack_channel[1].data : 0
ack_channel[2].checksum : 1
ack_channel[2].control : 1
ack_channel[2].data : 1
----------

Rule check abp ack fired.
ack_channel[2].checksum : 0
ack_channel[2].control : 0
ack_channel[2].data : 0
sval_a.checksum : 0
sval_a.control : 0
sval_a.data : 0
----------

Rule sending fired.
a_msg : 2
sval_a.checksum:Undefined
sval_a.data : 2
sbit_a : 0
state : 2
the_msg : 2
----------
```

```
Rule move msg channel fired.
msg_channel[1].checksum : 0
msg_channel[1].control : 0
msg_channel[1].data : 0
msg_channel[2].checksum : 1
msg_channel[2].control : 1
msg_channel[2].data : 1
----------

Rule receiver_c fired.
msg_channel[2].checksum : 0
msg_channel[2].control : 0
msg_channel[2].data : 0
rval_c.checksum : 1
rval_c.control : 1
rval_c.data : 1
----------

Rule sender_c fired.
msg_channel[1].checksum : 1
msg_channel[1].control : 1
msg_channel[1].data : 1
sval_c.checksum : 0
sval_c.control : 0
sval_c.data : 0
----------

Rule receiver_a fired.
ack_channel[1].checksum : 1
ack_channel[1].control : 1
ack_channel[1].data : 1
rval_c.checksum : 0
rval_c.control : 0
rval_c.data : 0
----------

Rule move ack channel fired.
ack_channel[1].checksum : 0
ack_channel[1].control : 0
ack_channel[1].data : 0
ack_channel[2].checksum : 1
ack_channel[2].control : 1
ack_channel[2].data : 1
----------
```

```
Rule corrupt ack control fired.
ack_channel[2].control : 0
----------

Rule check abp ack fired.
ack_channel[2].checksum : 0
ack_channel[2].data : 0
sval_a.checksum : 0
sval_a.data : 0
----------

Rule sending fired.
a_msg : 3
sval_a.checksum:Undefined
sval_a.control : 1
sval_a.data : 3
sbit_a : 1
state : 3
another_msg : 3
----------

Rule move msg channel fired.
msg_channel[1].checksum : 0
msg_channel[1].control : 0
msg_channel[1].data : 0
msg_channel[2].checksum : 1
msg_channel[2].control : 1
msg_channel[2].data : 1
----------

Rule receiver_c fired.
msg_channel[2].checksum : 0
msg_channel[2].control : 0
msg_channel[2].data : 0
rval_c.checksum : 1
rval_c.control : 1
rval_c.data : 1
----------

Rule receiver_a fired.
ack_channel[1].checksum : 1
ack_channel[1].control : 1
ack_channel[1].data : 1
rval_c.checksum : 0
rval_c.control : 0
rval_c.data : 0
```

```
----------

Rule move ack channel fired.
ack_channel[1].checksum : 0
ack_channel[1].control : 0
ack_channel[1].data : 0
ack_channel[2].checksum : 1
ack_channel[2].control : 1
ack_channel[2].data : 1
----------

Rule check abp ack fired.
ack_channel[2].checksum : 0
ack_channel[2].control : 0
ack_channel[2].data : 0
sval_a.checksum : 0
sval_a.control : 0
sval_a.data : 0
----------

Rule sending fired.
empty_packet.checksum : 0
empty_packet.control : 0
empty_packet.data : 0
nonempty_packet.checksum : 0
nonempty_packet.control : 0
nonempty_packet.data : 1
msg_channel[1].checksum : 0
msg_channel[1].control : 0
msg_channel[1].data : 0
msg_channel[2].checksum : 0
msg_channel[2].control : 0
msg_channel[2].data : 0
ack_channel[1].checksum : 0
ack_channel[1].control : 0
ack_channel[1].data : 0
ack_channel[2].checksum : 0
ack_channel[2].control : 0
ack_channel[2].data : 0
ack2_channel[1].checksum : 0
ack2_channel[1].control : 0
ack2_channel[1].data : 0
ack2_channel[2].checksum : 0
ack2_channel[2].control : 0
ack2_channel[2].data : 0
sval_c.checksum : 0
```

```
sval_c.control : 0
sval_c.data : 0
rval_c.checksum : 0
rval_c.control : 0
rval_c.data : 0
busy_c : false
phys_char_m:lossy_corrupt
phys_char_a:corrupt
phys_char_a2:good
top:abp
abp_m:cp
abp_a:physical_a
cp_m:physical_m
cp_a:absent
a_msg : 1
sval_a.checksum : 0
sval_a.control : 0
sval_a.data : 0
rval_a.checksum : 0
rval_a.control : 0
rval_a.data : 0
sbit_a : 1
rbit_a : 1
state : 3
the_msg : 2
another_msg : 3
----------

End of the error trace.

==========================================================================

Result:

Error: ****** send in state 3

when firing rule:
sending

State Space Explored:

4826 states, 30714 rules fired in 24.99s.
```