Automatic Verification of Asynchronous Retransmission Go-Back-N ARQ Protocols Using the Concurrency Workbench

Hai Wang and Hwei Sheng Teoh

Department of Computer Science

University of Toronto

Toronto, Ontario, Canada M5S 3G4

Abstract

The Go-Back-N and Selective-Repeat automatic repeat request (ARQ) protocols are widely used for error control in computer-communication networks. They are nearly optimal for channels characterized by low error rates and small propagation delays, and thus suitable for many classical data transmission applications. However, these protocols are not efficient for modern non-conventional channels with high error rates and/or large propagation delays, such as mobile links and satellite links. In this paper we present a family of new protocols, termed Asynchronous Retransmission Go-Back-N ARQ protocols, which offer the potential for better performance in environments characterized by high error rates and/or large propagation delays. We verify that these protocols are deadlock free using the Concurrency Workbench. We also verify that all these new protocols provide the same services to applications as the basic Go-Back-N protocol, and all of them are equivalent to the basic Go-Back-N protocol from the application point view.

1 Introduction

We consider computer-communication systems in which data transmission channels are not ideally error-free. A major concern in such systems is how to control transmission errors caused by the unreliable channels so that virtually error-free data transmission can be attained. Approaches to this problem can be classified into three categories, namely, automatic repeat request (ARQ), forward error control (FEC), and hybrid ARQ-FEC techniques [3].

In ARQ techniques, an error-detecting code is used to detect whether a received packet contains errors. If so, the receiver discards the erroneous packet, and a retransmission request is sent to the sender through a feedback channel. Retransmission continues until an error-free packet or a packet containing an undetectable error pattern is received.

In FEC techniques, an error-correcting code is used for combating error patterns that are more frequently introduced by the channel. When the receiver detects the presence of errors in a received packet, it attempts to locate and correct the errors. In this case, no retransmission is required.

In hybrid ARQ-FEC techniques, appropriate ARQ and FEC techniques are incorporated. Generally, hybrid ARQ-FEC techniques lead to better performance if the two basic error-control techniques are properly combined.

There are three basic types of ARQ techniques: Stop-and-Wait, Go-Back-N and Selective-Repeat protocols [3]. In the Stop-and-Wait protocol, the sender sends a packet to the receiver and waits for an acknowledgment. Once the packet has been successfully received or it has been detected in error, the receiver returns a positive acknowledgment (ACK) or a negative acknowledgment (NAK) respectively. A certain time period after the transmission of a packet, either the sender has received the corresponding ACK/NAK or a time-out occurs. If the sender receives an ACK, it then transmits the next packet. If the sender receives a NAK or a time-out occurs, it then retransmits the packet and again waits for an acknowledgment. Retransmissions continue until the sender receives an ACK. This protocol is simple but inherently inefficient due to the idle time spent in waiting for the receiver acknowledgment after each transmission [3].

In Go-Back-N protocols, the sender continuously transmits packets in order and then stores them pending receipt of an acknowledgment for each until $N+1$ unacknowledged packets have been transmitted or a NAK has been received. Each packet contains a unique identifier. The receiver returns an ACK together with the identifier of the corresponding packet for each correctly received packet until it detects an erroneous packet. Once the receiver detects an erroneous packet, it returns a NAK together with the identifier of last correctly received packet, and discards the erroneous packet and all subsequently received packets regardless whether or not they are error-free. A certain time period after the transmission of a packet, either the sender has received the corresponding ACK/NAK or a time-out occurs. If the sender receives an ACK, it removes the packet from the retransmission list. If the sender receives a NAK or a time-out occurs, it then retransmits the packet and all successive packets. Retransmission continues until the first erroneous packet is positively acknowledged. As soon as the erroneous packet is positively acknowledged, the sender proceeds to transmit new packets. Go-Back-N protocols are more efficient than the Stop-and-Wait protocol, and present a satisfactory performance when the error rate is not too high and the propagation delay is small, but they become rapidly inefficient for high error rates and/or large propagation delays [3, 5].

In the Selective-Repeat protocol, the sender also transmits packets continuously, but only retransmits those packets that are negatively acknowledged. This protocol is more efficient than Go-Back-N protocols, particularly when the propagation delay is large. However, this protocol also requires extensive system resources (theoretically, an infinite buffer at receiver) and more complex logic at both sender and receiver [3].

Both the Go-Back-N and Selective-Repeat ARQ protocols are nearly optimal for channels characterized by low error rates and small propagation delays, and thus they are widely used for error control in many classical computer-communication networks. However, these protocols are not efficient for modern non-conventional channels with high error rates and/or large propagation delays, such as mobile and satellite links. Many ARQ protocols have been developed to achieve better performance in this kind of environment. Towsley's Stutter Go-Back-N protocol is a well-known example [5]. In the Stutter Go-Back-N protocol, the sender repeatly transmits the last unacknowledged packet during the period of time the channel would normally be idle under the basic Go-Back-N protocol.

In this paper we present a family of new protocols, termed Asynchronous Retransmission Go-Back-N ARQ protocols, which offer the potential for better performance in environments characterized by high error rates and/or large propagation delays. The Stutter Go-Back-N protocol is just one instance of this family. We verify that these protocols are deadlock free using the Concurrency Workbench. We also verify that all these new protocols provide the same services to applications as the basic Go-Back-N protocol, and all of them are equivalent to the basic Go-Back-N protocol from the application point view.

The remainder of the paper is organized as follows. Section 2 presents the new protocols. Section 3 illustrates the verification of the new protocols using the Concurrency Workbench. Finally, a summary and conclusions are provided in Section 4.

2 Asynchronous Retransmission Go-Back-N Protocols

In the basic Go-Back-N protocol, there are many time intervals that the channel is idle while the sender has at least one packet waiting of an acknowledgment. This is the reason that the basic Go-Back-N protocol leads to much degraded performance when error rates are high and/or propagation delays are large [3, 5]. Like the Stutter Go-Back-N protocol, all Asynchronous Retransmission Go-Back-N Protocols try to improve the performance by utilizing these idle periods.

Let p be the mean error rate and let d be the mean propagation delay during a given time interval. Also, let r be the mean data rate during that time interval. We define

$$K = F(p, d, r) \tag{1}$$

where F is a mapping from $\Re^3$ to a set of all integers between 1 and $N+1$ inclusive.

Generally, an Asynchronous Retransmission Go-Back-N protocol is the same as the basic Go-Back-N protocol except that it repeatly retransmits the last K unacknowledged packets, if any, during the time the channel is idle. Therefore, depending on how F is defined, we have a family of protocols. In fact, the basic Go-Back-N protocol can be treated as an Asynchronous Retransmission Go-Back-N protocol when K is always 0. When K is always 1, the Asynchronous Retransmission Go-Back-N protocol is the Stutter Go-Back-N protocol. If K is a constant, then we obtain a deterministic non-adaptive protocol. If K is a variable depending on F which is deterministic, then we have a deterministic adaptive protocol. If K is a variable depending on F which is random, then we obtain a random protocol.

These Asynchronous Retransmission Go-Back-N protocols offer the potential for better performance in environments characterized by high error rates and/or large propagation delays. It has been proved that the Stutter Go-Back-N protocol achieves better performance than the basic Go-Back-N protocol when error rates are high and/or propagation delays are large [5].

3 Verification of Protocols Using the Concurrency Workbench

The Concurrency Workbench is an automatic tool that supports the verification of finite-state processes. It is particularly suitable for automatic verification of communication protocols [1, 4]. In this section, we present the verification of the correctness of the basic Go-Back-N, the Stutter Go-Back-N, and all Asynchronous Retransmission Go-Back-N protocols for $N = 3$.

The specifications of the basic Go-Back-N, the Stutter Go-Back-N, and all Asynchronous Retransmission Go-Back-N protocols can be formalized as a parallel composition of three processes: one for the sender, one for the receiver, and one for the channel.

3.1 Specification of a Realistic Channel

In a realistic channel, packets may be delivered correctly, or may be corrupted by noise. In addition, a realistic channel also permits packet losses. The following is the state transition diagram of a realistic channel.

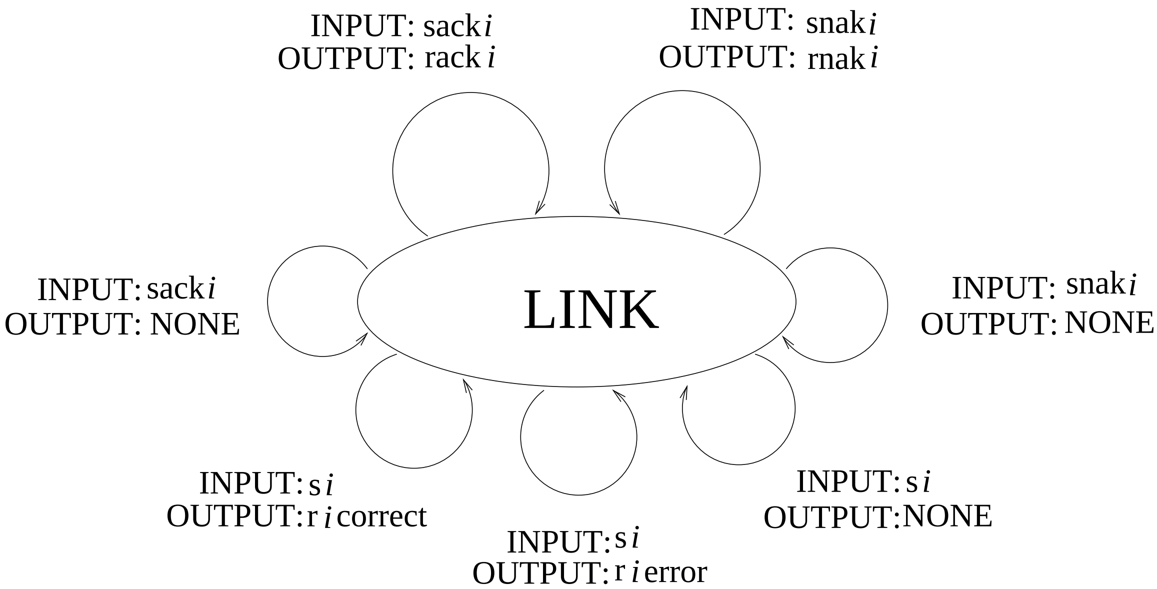

Figure 1: The state transition diagram of a realistic channel

Here si represents the event that the sender sends a packet with identifier i; r$i$correct denotes the event that the receiver receives packet i correctly; r$i$error represents the event that the receiver receives packet i which has been corrupted by noise; sack$i$ denotes the event that the receiver sends a positive acknowledgment for packet i; and snak$i$ denotes the event that the receiver sends a negative acknowledgment for packet i, where i is an integer between 0 and N inclusive. Note that all packets may be lost in the channel according to this specification, and the effect when an acknowledgment is corrupted by noise is the same as when that acknowledgment is lost in the channel. The Concurrency Workbench specification for $N = 3$ is presented in Appendix A.

3.2 Specification of the Basic Go-Back-N Protocol

According to the definition of the basic Go-Back-N protocol [2], the state transition diagram of the receiver of the basic Go-Back-N protocol is quite simple. The receiver has $2(N+1)$ states, namely, Receive$i$WhenNoNakSent and Receive$i$WhenNakSent for $i = 0, 1, ..., N$.

When the receiver is in state Receive$i$WhenNoNakSent, it is expecting to receive packet i correctly. If it receives packet i correctly, then it returns a positive acknowledgment (ACK), delivers that packet to the application, and changes to state Receive$j$WhenNoNakSent, where $j = (i+1)\mathrm{mod}(N+1)$. If it receives an erroneous packet or a packet with an outstanding identifier, then it returns a negative acknowledgment (NAK) and changes to state Receive$i$WhenNakSent.

When the receiver is in state Receive$i$WhenNakSent, if it receives packet i correctly, then it returns an positive acknowledgment (ACK), delivers that packet to the application, and changes to state Receive$j$WhenNoNakSent, where $j = (i+1)\mathrm{mod}(N+1)$. If it receives an erroneous packet or a packet with an outstanding identifier, then it returns a negative acknowledgment (NAK) and remains in the same state. If a time-out occurs, it again sends a negative acknowledgment to the sender. The following is the state transition diagram of the receiver.

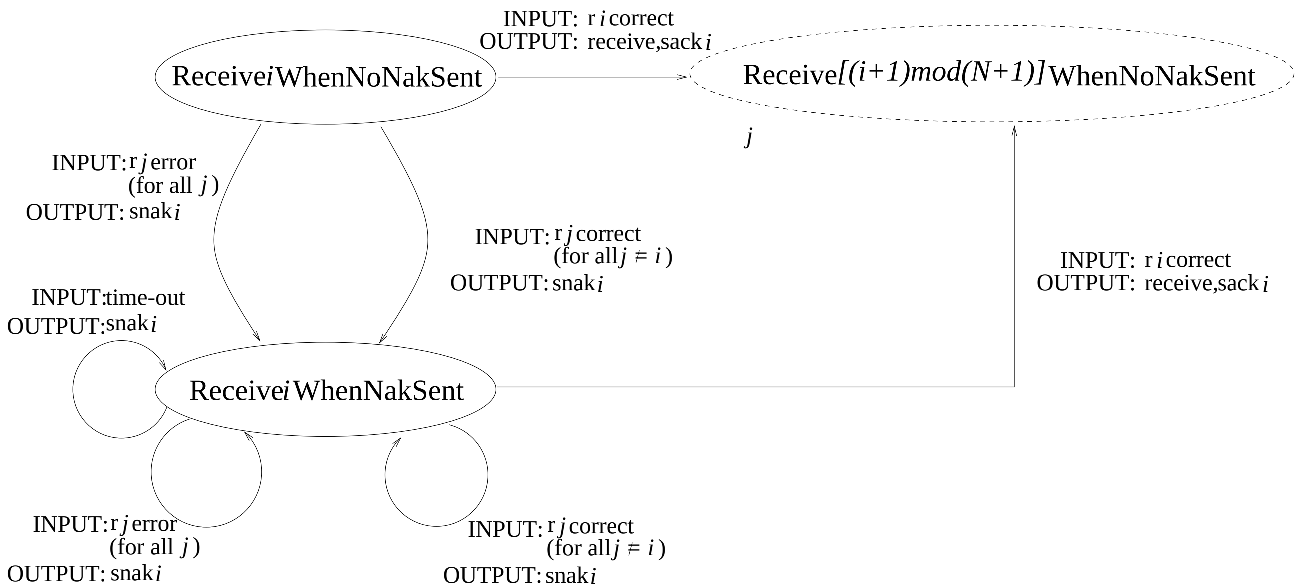

Figure 2: The state transition diagram of the receiver

Here receive means delivering a correctly received packet to the application, and time-out represents the occurrence of a time-out. The Concurrency Workbench specification of the receiver for $N = 3$ is presented in Appendix A.

According to the definition of the basic Go-Back-N protocol [2], the state transition diagram of the sender of the basic Go-Back-N protocol is complicated. It has $(N+1)^2$ states for sending packets and $\frac{(N+1)^2(N+2)}{2}$ states for retransmitting packets. The states for sending packets have a general form Send$i$WhenThereAre$j$UnackMsgs for $i = 0, 1, ..., N$, and $j = 0, 1, ..., N$, which means sending packet i while there are j unacknowledged packets. The states for retransmitting packets have a general form Retx$i$WhenThereAre$j$UnackMsgsAnd$k$Behind for $i = 0, 1, ..., N$, $j = 1, 2, ..., N+1$, and $k = 0, ..., j$, which means retransmitting packet i while there are j unacknowledged packets and k of them are transmitted after packet i has been transmitted for the first time.

When the sender is in state Send$i$WhenThereAre$j$UnackMsgs, the sender accepts a packet from the application and sends the packet through the channel. Then it is ready to send the next packet. If it receives any acknowledgment for any previous packets, then it reduces the number of unacknowledged packets correspondingly. If it receives a negative acknowledgment, then it starts to retransmit packets. A time-out occurs when there are N unacknowledged packets and no acknowledgment has been received. In this care, the sender starts to retransmit packets.

When the sender is in state Retx$i$WhenThereAre$j$UnackMsgsAnd$k$Behind, the sender retransmits packet i. If it receives any acknowledgment for any previous packets, then it reduces the number of unacknowledged packets correspondingly. The state transition diagrams of the sender are shown in Figures 3 and 4, in which send means accepting a packet from the application and NONE means no event taking place during the state transition. The Concurrency Workbench specification of the sender for $N = 3$ is presented in Appendix A.

The basic Go-Back-N protocol is formalized as a parallel composition of three processes: one for the sender, one for the receiver, and one for the channel. The following is the Concurrency Workbench specification of the basic Go-Back-N specification when $N = 3$.

```
set InternalActions =
     {s0,s1,s2,s3,
      r0correct,r1correct,r2correct,r3correct,
      r0error,r1error,r2error,r3error,
      sack0,sack1,sack2,sack3,
      rack0,rack1,rack2,rack3,
      snak0,snak1,snak2,snak3,
      rnak0,rnak1,rnak2,rnak3};
```

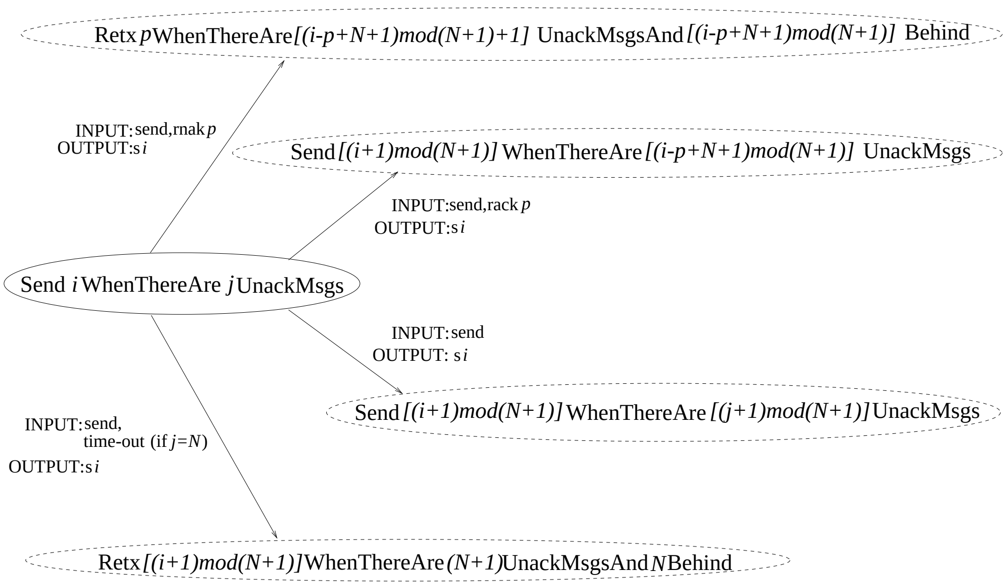

Figure 3: The state transition diagram of the sender of the basic Go-Back-N protocol, where $p = (i - h + N + 1)mod(N + 1)$ for all integers $h \in [0, j]$.

```
agent GBN = (Send0WhenThereAre0UnackMsgs |
             Receive0WhenNoNakSent |
             Link) \ InternalActions;
```

Note that the use of the restriction operator $\backslash$ InternalActions ensures that the channel is not accessible to processes other than the sender and receiver.

3.3 Specification of the Stutter Go-Back-N Protocol

The specifications of the channel and receiver for the Stutter Go-Back-N protocol are the same as those for the basic Go-Back-N protocol. Actually, the specification of the Stutter Go-Back-N protocol is the same as that of the basic Go-Back-N protocol except that the specifications of the senders are different. The only difference between the specification of the sender for the Stutter Go-Back-N protocol and that for the basic Go-Back-N protocol is that the sender retransmits last unacknowledged packet in the Stutter Go-Back-N protocol while it waits for a time-out to occur in the basic Go-Back-N protocol. Figures 5 and 6 show the state transition diagrams of the sender for the Stutter Go-Back-N protocol which are different from those for the basic Go-Back-N protocol. The Concurrency Workbench specification of the Stutter Go-Back-N protocol for $N = 3$ is presented in Appendix A.

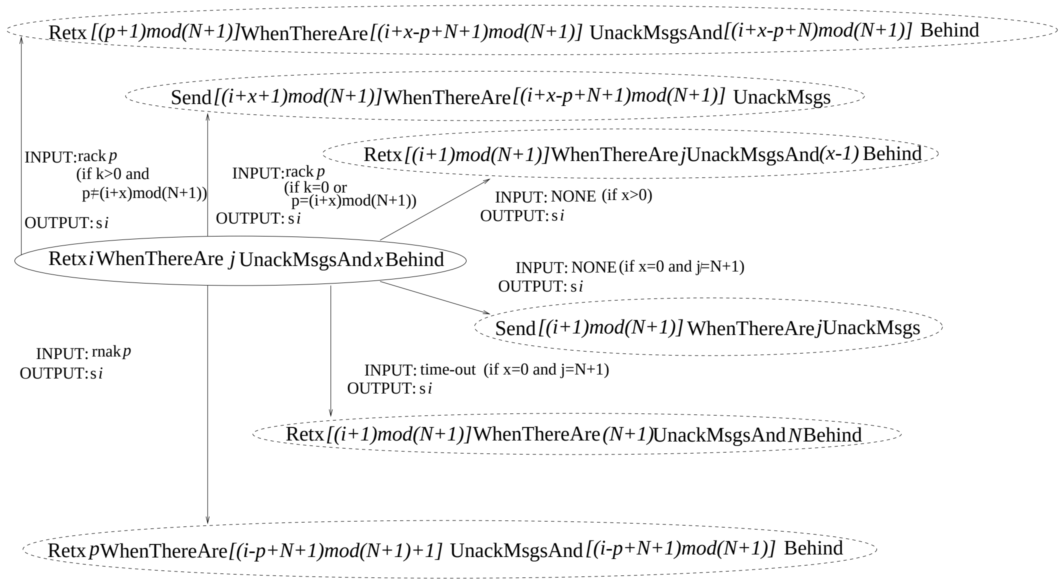

Figure 4: The state transition diagram of the sender of the basic Go-Back-N protocol (Cont.), where $p = (i + x - h + N + 1)mod(N + 1)$ for all integers $h \in [0, j - 1]$.

3.4 Specification of the Asynchronous Retransmission Go-Back-N Protocols

In the Concurrency Workbench, we can enumerate all possible states of a process [1, 4]. As mentioned in Section 2, the only difference between two different Asynchronous Retransmission Go-Back-N protocols is that they use different K which is an integer between 1 and $N+1$ inclusive. Therefore, the specifications of all possible Asynchronous Retransmission Go-Back-N protocols can be represented by one process in the Concurrency Workbench.

The specifications of the channel and receiver for the Asynchronous Retransmission Go-Back-N protocols are the same as those for the basic Go-Back-N protocol. Actually, the specification of the Asynchronous Retransmission Go-Back-N protocols is the same as that of the basic Go-Back-N except that the specifications of the senders are different. The only difference between the specification of the sender for the Asynchronous Retransmission Go-Back-N protocols and that for the basic Go-Back-N protocol is that the sender retransmits last K unacknowledged packet in the Asynchronous Retransmission Go-Back-N protocols for any possible $K = 1, ..., N + 1$ while it waits a time-out to occur in the basic Go-Back-N protocol. Figures 7 and 8 show the state transition diagrams of the sender for the Asynchronous Retransmission Go-Back-N protocol which are different from those for the basic

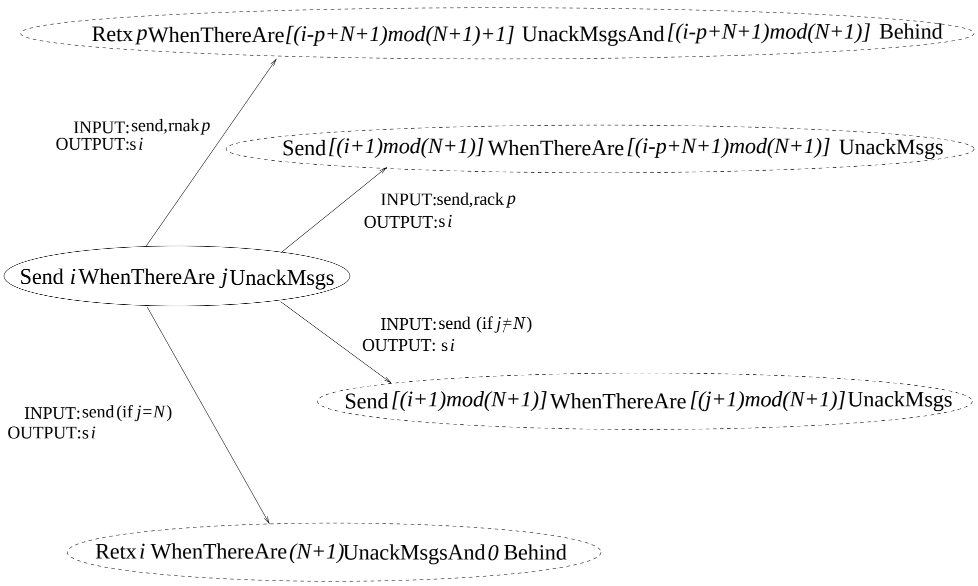

Figure 5: The state transition diagram of the sender of the Stutter Go-Back-N protocol, where $p = (i - h + N + 1)mod(N + 1)$ for all integers $h \in [0, j]$.

Go-Back-N protocol. The Concurrency Workbench specification of the Asynchronous Retransmission Go-Back-N protocols for $N = 3$ is presented in Appendix A.

3.5 Model Checking

We use the model checking facility of the Concurrency Workbench to illustrate that all of the basic Go-Back-N, the Stutter Go-Back-N, and the Asynchronous Retransmission Go-Back-N protocols provide correct services to applications.

We first define some propositional macros.

```
prop Can(a) = <<a>>T;
prop Bx(P) = max(Z.P & [-]Z);
```

Intuitively, a state satisfies Bx(P) if every state reachable from the argument state satisfies P, while a state satisfies Can(a) if a may occurs as the next visible action [1].

The following formulas are used to verify whether or not the basic Go-Back-N, the Stutter Go-Back-N, and the Asynchronous Retransmission Go-Back-N protocols can deliver packets for applications.

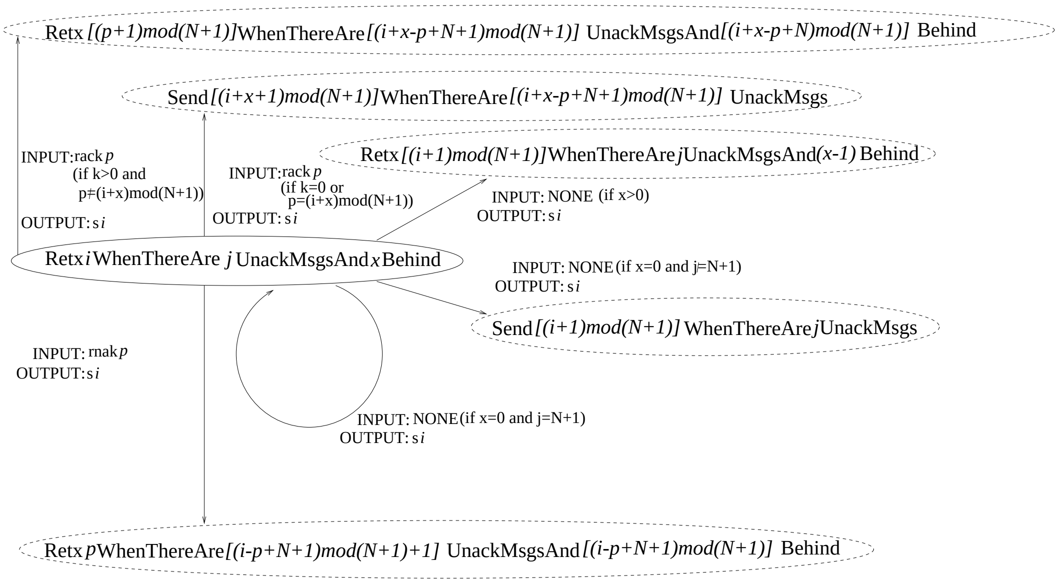

Figure 6: The state transition diagram of the sender of the Stutter Go-Back-N protocol (Cont.), where $p = (i + x - h + N + 1)mod(N + 1)$ for all integers $h \in [0, j - 1]$.

```
cp (GBN, Bx ((Can (send)) | (Can ('receive))));
cp (StutterGBN, Bx ((Can (send)) | (Can ('receive))));
cp (AsynRetransGBN, Bx ((Can (send)) | (Can ('receive))));
```

The following formulas are used to verify whether the protocols are deadlock free.

```
fd GBN;
fd StutterGBN;
fd AsynRetransGBN;
```

The Concurrency Workbench results show that all of these protocols can deliver packets for applications and are deadlock free. The Concurrency Workbench formulas and results are also shown in Appendix A and Appendix B respectively.

3.6 Equivalence Checking

In this section we test the equivalences among the basic Go-Back-N, the Stutter Go-Back-N, and the Asynchronous Retransmission Go-Back-N protocols. In the Concurrency Workbench, command **cong**, **eq**, **mayeq**, **musteq**, **testeq** are used to test whether two protocols are observationally congruent (i.e., observationally equal), observationally equivalent (i.e., weakly bisimilar), may equivalent (i.e.,

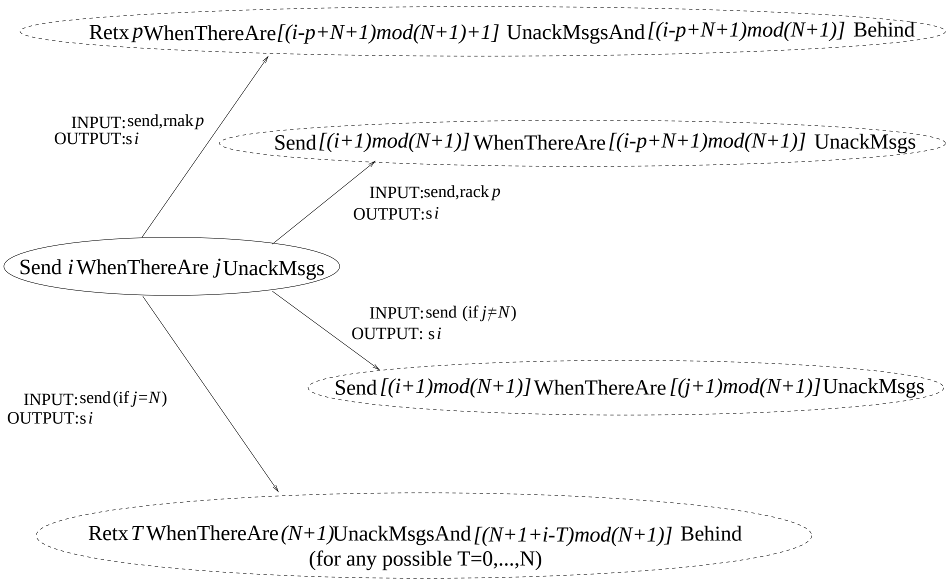

Figure 7: The state transition diagram of the sender of the Asynchronous Retransmission Go-Back-N protocol, where $p = (i - h + N + 1) mod(N + 1)$ for all integers $h \in [0, j]$.

trace equivalent), must equivalent, testing equivalent (i.e., failures equivalent) respectively [1, 4]. The following formulas are used to verify the equivalences of the protocols.

```
cong (GBN, StutterGBN);
cong (GBN, AsynRetransGBN);
cong (AsynRetransGBN, StutterGBN);

eq (GBN, StutterGBN);
eq (GBN, AsynRetransGBN);
eq (AsynRetransGBN, StutterGBN);

mayeq (GBN, StutterGBN);
mayeq (GBN, AsynRetransGBN);
mayeq (AsynRetransGBN, StutterGBN);

musteq (GBN, StutterGBN);
musteq (GBN, AsynRetransGBN);
musteq (AsynRetransGBN, StutterGBN);

testeq (GBN, StutterGBN);
testeq (GBN, AsynRetransGBN);
testeq (AsynRetransGBN, StutterGBN);
```

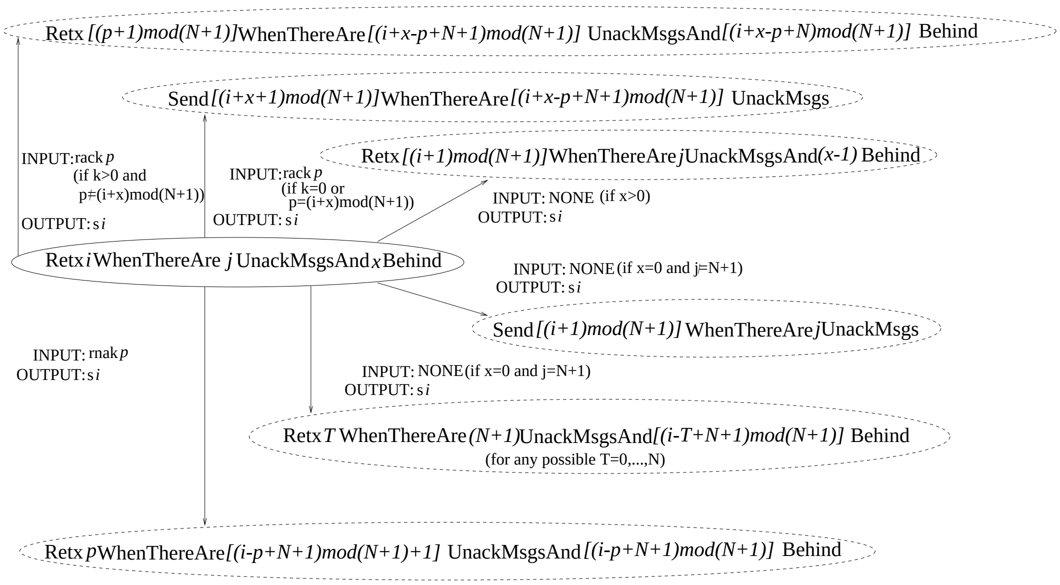

Figure 8: The state transition diagram of the sender of the Asynchronous Retransmission Go-Back-N protocol (Cont.), where $p = (i + x - h + N + 1)mod(N + 1)$ for all integers $h \in [0, j - 1]$.

The Concurrency Workbench results show that all of these protocols are observationally congruent, observationally equivalent, may equivalent, must equivalent, testing equivalent. Therefore, all Asynchronous Retransmission Go-Back-N protocols (including the Stutter Go-Back-N protocol) provide the same services to applications as the basic Go-Back-N protocol, and all of them are equivalent to the basic Go-Back-N protocol from the application point view. The Concurrency Workbench formulas and results of the verification are shown in Appendices A and B, respectively.

4 Summary and Conclusions

In this paper we present a family of new protocols, termed Asynchronous Retransmission Go-Back-N ARQ protocols, which offer the potential for better performance in environments characterized by high error rates and/or large propagation delays. Using the Concurrency Workbench, we verify that these protocols can deliver packets for applications, and they are deadlock free. We also verify that these protocols provide the same services to applications as the basic Go-Back-N protocol, and all of them are equivalent to the basic Go-Back-N protocol from the application point view.

References

[1] R. Cleaveland, J. Parrow and B. Steffen. The Concurrency Workbench: A Semantics Based Tool for Verification of Concurrent Systems. *ACM Transaction on Programming Languages and Systems*, 15(1):36-72, January 1993.

[2] F. Halsall. *Data Communications, Computer networks and Open Systems*, Addison-Wesley, Reading, MA, 1992.

[3] S. Lin, D. J. Costello, Jr. and M. J. Miller. Automatic-repeat-request error-control schemes. *IEEE Communication Magazine*, 22(12):5-17, December 1984.

[4] F. Moller. *The Edinburgh Concurrency Workbench (Version 7).* Laboratory for Foundations of Computer Science, University of Edinburgh, UK, September 1996.

[5] D. Towsley. The stutter go-back-N ARQ protocol. *IEEE Transactions on Communications*, COM-27(6):869-875, June 1979.