Bottleneck Analysis of a Gigabit Network Interface Card : Formal Verification Approach

Hyun-Wook Jin, Ki-Seok Bang, Chuck Yoo, and Jin-Young Choi

Department of Computer Science and Engineering,
Korea University
SEOUL, 136-701 KOREA
{hwjin, hxy}@os.korea.ac.kr
{kbang, choi}@formal.korea.ac.kr

Abstract. This paper addresses how formal verification can be applied to find a bottleneck in a gigabit network interface card that prevents the card from achieving the best possible performance. Finding a bottleneck in a gigabit network interface card is not an easy task because it is equipped with sophisticated hardware components, such as multiple DMA engines and separate CPU and memory. Therefore, the interactions between a network interface card and the host are very complex so that the firmware to manage the interactions is also complicated, which makes the bottleneck analysis very difficult. As an alternative approach of the bottleneck analysis, we specify the firmware in a gigabit network interface card and analyze the behavior of the specification with SPIN. As an example of gigabit network interface cards, Myrinet is used in this paper. We show that SPIN can easily verify whether the Myrinet firmware has a bottleneck once the state transitions inside the firmware are modeled properly.

1 Introduction

Gigabit network interface cards (NIC) are getting popular. A notable example is Myrinet [3]. In order to achieve the best possible performance out of Myrinet, several user-level communication primitives have been proposed [4, 6, 12], and Berkeley-VIA [4] is a well-known implementation of Virtual Interface Architecture (VIA) [5] that is an industrial standard for user-level communication primitives. VIA allows user processes to directly access NIC bypassing the kernel that has multiple communication layers. Therefore, it is generally expected that VIA can achieve near physical bandwidth of gigabit networks.

However, our research shows that Berkeley-VIA is able to achieve a slightly higher throughput than UDP on Myrinet as shown in Figure 1 (only 6% improvement at 32KB data size). Furthermore, Berkeley-VIA has much less throughput than an improved UDP named Asynchronous UDP [13]. It turns out that Berkeley-VIA utilizes only about 1/2 of bandwidth of Myrinet. On the other hand, we find that Berkeley-VIA has the shortest one-way latency as shown in Figure 2, which indicates that Berkeley-VIA has less communication overhead

than UDP and Asynchronous UDP. So a question is why Berkeley-VIA has a very low overhead but is not able to achieve the best possible throughput. Our goal is to find the performance bottleneck.

The firmware of Myrinet NIC needs to be analyzed to see where the bottleneck is. Because Myrinet NIC has three DMA engines and separate memory and CPU, the firmware itself is very complicated. Therefore, the analysis of the firmware is not an easy task. Also the interaction between the firmware and the host is very complex so that the firmware analysis becomes even more complicated.

This paper attempts an alternative approach. In order to analyze the firmware of Myrinet NIC, we first build state transition diagrams to model the firmware. Second, we translate the state transition diagrams into specifications written in PROMELA (PROcess MEta LAnguage) [7]. Third, we derive verification formulas. Then the formulas are verified with SPIN [8].

Specifically, we analyze Lanai Control Program (LCP) and Myrinet Control Program (MCP), where LCP is the firmware for Berkeley-VIA and MCP is the firmware for traditional protocols, such as UDP and TCP. Since our goal is to find a performance bottleneck, we focus on how well DMA engines of Myrinet NIC are utilized because the utilization of DMA engines determines the throughput.

This paper is organized as follows. Section 2 describes the hardware components in Myrinet NIC. Section 3 models LCP and MCP. We construct state transition diagrams and specify them with PROMELA. LCP and MCP are analyzed in Section 4 with SPIN. Finally, Section 5 concludes the paper.

2 Myrinet Network Interface Card

Myrinet is a gigabit Local Area Network (LAN). Many researches apply Myrinet to clustering systems or storage area networks [1, 2]. In this section, we describe the hardware components of Myrinet NIC based on LANai-4 [10].

Myrinet NIC consists of a RISC processor named LANai, Static Random Access Memory (SRAM), and three DMA engines (i.e. EBUS-LBUS, send-DMA, and receive-DMA engines) as shown in Figure 3. LANai executes the firmware, and SRAM stores the data for sending or receiving. Each DMA engine works as follows.

The EBUS-LBUS DMA engine is responsible for the data movement between the main memory and the SRAM. The firmware initializes the EBUS-LBUS DMA by setting the DMA direction register (DMA_DIR), main memory address register (EAR), SRAM address register (LAR), and DMA counter register (DMA_CTR). The DMA_DIR register indicates that the DMA operation moves data whether from main memory to SRAM or vice versa. The EAR and LAR registers point the start of main memory buffer and SRAM buffer, respectively. The DMA_CTR register contains the number of bytes for DMA. In the case of sending, the data in the area indicated by the EAR register is moved to the buffer indicated by the LAR register as many as the value of DMA_CTR register, which is the same in the case of receiving excepting the data moving direction.

The send-DMA engine moves the data in SRAM to the Myrinet physical network. The firmware sets the sending memory pointer register (SMP) and sending memory limit register (SML). The SMP register specifies the beginning of the SRAM buffer to send-DMA, and the SML register indicates the end of the buffer.

The receive-DMA engine receives a data from Myrinet LAN into the SRAM. The registers of receive-DMA engine are receiving memory pointer register (RMP) and receiving memory limit register (RML). The registers have the same role as the registers of send-DMA engine excepting the registers specify the receiving buffer.

The firmware initiates the DMA operations by setting the proper registers of each DMA engine and notices the completion of corresponding DMA operation via the 32-bit Interrupt Status Register (ISR) on LANai processor. Each bit of ISR indicates a specific hardware event. The bit number 4 (dma_int bit) is set when an EBUS-LBUS DMA operation is completed. The bit number 3 (send_int bit) and 1 (recv_int bit) are set when a send-DMA and receive-DMA are completed, respectively. We refer the details of Myrinet NIC to [2].

3 Modeling of Firmware

This Section performs the modeling of LCP and MCP based on their source codes. We construct the state transition diagrams for concerned modules and specify them with PROMELA.

3.1 Lanai Control Program

LCP is the firmware for Berkeley-VIA. LCP consists of four modules: hostDma, lcpTx, lcpRx, and main. Figures 4, 5, and 6 show the state transition diagrams of former three modules.

The hostDma module is responsible for EBUS-LBUS DMA. The initial sate of the hostDma module is HostDmaIdle. The lcpTx and lcpRx modules invoke the method of the hostDma module. Then, the hostDma module initializes the EBUS-LBUS DMA operation, and its state moves to HostDmaBusy. When the EBUS-LBUS DMA operation is done (i.e. dma_int bit of ISR is set), the state of the hostDma module moves from HostDmaBusy to HostDmaIdle, and the method returns to its invoker.

The lcpTx module sends a data. The initial state is LcpTxIdle and moves to the LpcTxGotASend state when there is a data to send. Then, the lcpTx module invokes the method of the hostDma module moving to LcpTxHostDma. After the return of the invoked method, the state of the lcpTx module moves to LcpTxSendDma. In this state, the lcpTx module initializes the send-DMA operation and is waiting the completion of send-DMA. If a data is received from the network during the send-DMA operation, the lcpTx module invokes the method of the lcpRx module and moves to the LcpTxInvokeRx state. When the lcpRx module has received a data completely, its method returns to the lcpTx

module, and the state of the lcpTx module moves to LcpTxSendDma again. The LcpTxSendDma state can be changed to LcpTxIdle when the send-DMA is done (i.e. send_int bit of ISR is set).

The lcpRx module is responsible for receiving a data. The initial state is LcpRxReady that initializes a receive-DMA operation beforehand because it is hard to know when a data gets in. When a data is received from the network to Myrinet NIC (i.e. recv_int bit of ISR is set), the state moves to LcpRxGotAReceive, and the lcpRx module invokes the method of the hostDma module moving its state to LcpRxHostDma. After the method of the hostDma module returns, the lcpRx module changes its sate from LcpRxHostDma to LcpRxReady and reinitializes the receive-DMA operation.

The main module invokes the methods of the lcpTx and lcpRx modules when there is a data to send or receive, respectively. Note that the entry point (gray ellipses of Figures 4, 5, and 6) of the hostDma, lcpTx, and lcpRx modules is the initial state of each module. We will discuss more about the entry point in Section 3.2.

We specify the modules as processes in PROMELA. All invocations between modules are performed in a synchronous manner. Therefore, we implement the invocation by using two rendezvous communication channel for each process. One is the channel to pass an argument, and the other returns a return value. An event is passed as an argument or return value via the rendezvous channels. In addition, we specify ISR bits that notify the completion of DMA operations. Figures 7 and 8 show the part of the specification written in PROMELA, which are the hostDma and lcpTx modules.

3.2 Myrinet Control Program

MCP is included in Myrinet Software package [11] that contains a device driver and firmware (i.e. MCP). While Berkeley-VIA supports only VIA protocol, Myrinet Software does TCP/IP protocol suite. MCP consists of five modules: hostSend, netSend, hostReceive, netReceive, and main. The hostSend and netSend modules are responsible for sending. The hostSend module moves a data from main memory to SRAM, and the netSend module sends a data in SRAM to the network. On the other hand, the receiving of data is performed by the hostReceive and netReceive modules. The netReceive module receives a data from the network to SRAM. The hostReceive module moves the received data to the main memory. The state transition diagrams of four modules are shown in Figures 9, 10, 11, and 12. The main module invokes the methods of the former four modules according to the event occurred.

The initial state of the hostSend module is HostSendIdle. The state is moves to HostSendGotASend when there is a data to send. Then, the hostSend module checks some conditions. If there is no buffer available in SRAM, the state becomes HostSendFull and returns to the main module. Otherwise, the hostSend module examines whether the EBUS-LBUS DMA engine is occupied by the hostReceive module. If the EBUS-LBUS DMA engine is idle, then the state

moves to HostSendDma initializing an EBUS-LBUS DMA operation and returns to the main module without a waiting for the completion of the DMA. If the EBUS-LBUS DMA engine is occupied by the hostReceive module, the state moves to HostSendDmaBusy and returns to the main module. In the case of reaching the HostSendFull state, the state transition is performed when the netSend module consumes a data in SRAM. The state transition from HostSendDmaBusy is occurred after the completion of EBUS-LBUS DMA performed by the hostReceive module. When the state is HostSendDma, the hostSend module moves its state to the initial state after the completion of EBUS-LBUS DMA (i.e. dma_int bit of ISR is set).

The initial state of the netSend module is NetSendIdle. If there is a data moved from main memory to SRAM by the hostSend module, the method of the netSend module is invoked moving its state to NetSendBusy. The netSend module in NetSendBusy initializes the send-DMA operation, then the method returns. When the send-DMA operation is completed (i.e. send_int bit of ISR is set), the state is changed to NetSendIdle.

The hostReceive module starts from HostReceiveIdle. If there is a data received from the network to SRAM by the netReceive module, the state moves to HostReceiveGotAReceive. Then, like the hostSend module, the hostReceive module checks whether the other party uses the EBUS-LBUS DMA engine or not. If the EBUS-LBUS DMA engine is occupied by the hostSend module, the state moves to HostSendDmaBusy. Otherwise, the state is changed to HostReceiveDma, and the module initializes EBUS-LBUS DMA. In both case, after the state transition, the method returns to the main module. The state transition is performed from HostReceiveDmaBusy to HostReceiveDma when the hostSend module releases the EBUS-LBUS DMA engine. The state moves from HostReceiveDma to HostReceiveIdle, after the completion of the EBUS-LBUS DMA operation (i.e. dma_int bit of ISR is set).

The initial state of the netReceive module is NetReceiveDma that is the same state with LcpRxReady of LCP. When the receive-DMA is done (i.e. recv_int bit of ISR is set), the state moves to NetReceiveDmaDone that checks whether the receiving buffer in SRAM is available for the next receiving. If it is available, the state is returns to the initial state; else, the state is changed to NetReceiveFull. The netReceive module can escape from the NetReceiveFull state when the hostReceive module consumes a data in SRAM.

Comparing with modules of LCP, the notable difference is that each module of MCP has plural entry points. This means that the method of each module is invoked from an entry point and returns when it reaches to another entry point without waiting for the next event. Therefore, the method invoked in the next time starts from the state in which the method returns right before. On the other hand, in the case of LCP, a method is invoked when the module is in the initial sate and returns only when it backs to the initial state.

Like LCP, we implement an invocation by using two rendezvous communication channel of PROMELA. The events are stored in the channel named Events. The main module gets an events from the Events channel and invokes the cor-

respond method. Figures 13 and 14 show the part of the specification written in PROMELA, which are the hostSend and netSend modules.

4 Comparison of Firmware

This section analyzes the behaviors of LCP and MCP from the viewpoint of throughput. The key factor that determines the throughput of NIC is how well the DMA engines are utilized. The maximum throughput can be achieved when the EBUS-LBUS DMA engine performs in parallel with the send-DMA and receive-DMA engine.

For example, let $\mathrm{DMA_{EBUS-LBUS}}$ be the throughput of the EBUS-LBUS DMA and $\mathrm{DMA_{send}}$ be the throughput of the send-DMA. $\mathrm{DMA_{EBUS-LBUS}}$ is determined by the bandwidth of the I/O bus (e.g. PCI) that connects the main memory and SRAM of NIC. On the other hand, $\mathrm{DMA_{send}}$ is determined by the network physical media. When DMA engines perform in parallel, the throughput is evaluated as follows:

$$Throughput = MIN(DMA_{EBUS-LBUS}, DMA_{send})$$

However, if DMA engines perform sequentially, the throughput is limited as follows:

$$Throughput = DMA_{EBUS-LBUS}/(1 + DMA_{EBUS-LBUS}/DMA_{send})$$

If $\mathrm{DMA_{EBUS-LBUS}}$ and $\mathrm{DMA_{send}}$ are the same, the throughput achieved is reduced to 1/2 of $\mathrm{DMA_{EBUS-LBUS}}$. The next step of the analysis is to derive verification formulas. Because the verification formulas need to reflect the utilization of DMA engines, we use the following formulas written in LTL [9]:

1. LCP
   A. $\diamond$ (LTIR && HDB && ! LTHD)
      Can the lcpTx module initiate send-DMA while the hostDma module is using EBUS-LBUS DMA that moves data from main memory to SRAM?
   B. $\diamond$ (LRR && HDB && ! LRHD)
      Can the lcpRx module initiate receive-DMA while the hostDma module is using EBUS-LBUS DMA that moves data from SRAM to main memory?
      - LTIR : The state of lcpTx is LcpTxInvokeRx.
      - LTHD : The state of lcpTx is LcpTxHostDma.
      - LRR : The state of lcpRx is LcpRxReady.
      - LRHD : The state of lcpRx is LcpRxHostDma.
      - HDB : The state of hostDma is HostDmaBusy.
2. MCP
   A. $\diamond$ (HSD && NSB)
      Can the netSend module initiate send-DMA while the hostSend module occupies the EBUS-LBUS DMA engine?

B. ◇ (HRD && NRD)
Can the netReceive module initiate receive-DMA while the hostReceive module occupies EBUS-LBUS DMA engine?
- HSD : The state of hostSend is HostSendDma.
- NSB : The state of netSend is NetSendBusy.
- HRD : The state of hostReceive is HostReceiveDma.
- NRD : The state of netReceive is NetReceiveDma.

If DMA engines perform in parallel, each verification formulas should result in "True". When we run SPIN with the above formulas, the verification formulas of MCP are "True". However, the formulas for LCP result in "False". That is, LCP cannot perform the send-DMA during the EBUS-LBUS DMA that moves data from main memory to SRAM (formula 1-A). Also LCP cannot perform the receive-DMA as well during the EBUS-LBUS DMA that moves data from SRAM to main memory (formula 1-B). This result explains why the performance of Berkeley-VIA is limited. The simulation results also show that LCP performs DMA sequentially but MCP performs DMA in parallel. We have run random and interactive simulations and confirmed the same results.

In addition, we verify the correctness that only one module should occupy the EBUS-LBUS DMA engine at a time. The EBUS-LBUS DMA engine moves data not only from main memory to SRAM for sending, but also from SRAM to main memory for receiving. Therefore, a module should wait until the DMA engine becomes idle, if the other module already occupies the DMA engine. The formulas used are as follows:

1. LCP
   □ ! (LTHD && LRHD)
   The lcpTx module cannot use the EBUS-LBUS DMA engine during the lcpRx module occupies it.
2. MCP
   □ ! (HSD && HRD)
   The hostSend module cannot use the EBUS-LBUS DMA engine during the hostReceive module occupies it.

The verification results show that both LCP and MCP satisfy the above correctness property.

5 Conclusions

This paper investigates the bottleneck of the Myrinet firmware. Specifically, we model LCP and MCP with state transition daigrams and translate them into specifications written in PROMELA. Then the verification formulas are derived, and they are verified with SPIN. The verification result shows that LCP serializes the operations of DMA engines, which leads to the low throughput of Berkeley-VIA. On the other hand, MCP fully utilizes three DMA engines. It means that the internal structure of MCP is more elaborative than that of LCP. In other

words, the modules of MCP have multiple entry points, while each module of LCP has only one entry point. The multiple entry points make MCP perform DMA engines in parallel.

In addition, we verify the correctness of the firmware that a module does not initialize the EBUS-LBUS DMA while another module occupies the EBUS-LBUS DMA engine. The verification results show that both LCP and MCP satisfy the correctness property.

In summary, this paper demonstrates that the bottleneck analysis of a gigabit network interface card can be done effectively with the formal verification approach. It also shows that SPIN is an excellent tool for modeling and analysis of the firmware running on a gigabit NIC. The firmware of gigabit NIC consists of many event handlers that perform independently with other handlers, and SPIN is suitable for the model checking of the dynamic firmware.

References

1. D. Anderson, J. Chase, S. Gadde, A. Gallatin, K. Yocum, and M. Feeley, "Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet," *Proceedings of the 1998 USENIX Technical Conference*, June 1998.
2. T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team, "A Case for Networks of Workstations: NOW," *IEEE Micro*, February 1995.
3. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. -K. Su, "Myrinet – A Gigabit-per-Second Local-Area Network," *IEEE-Micro*, Vol. 15, No. 1, pp. 29-36, February 1995.
4. P. Buonadonna, A. Geweke, and D. Culler, "An Implementation and Analysis of the Virtual Interface Architecture," *Proceedings of SC'98*, November 1998.
5. D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, A. M. Berry, E. Gronke, and C. Dodd, "The Virtual Interface Architecture,"*IEEE Micro*, Vol. 8, pp. 66-76, March-April 1998.
6. T. V. Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," *Proceedings of 15th ACM SOSP*, pp. 40-53, December 1995.
7. G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.
8. G. J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, May 1997.
9. Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.
10. Myricom Inc., *LANai 4*, http://www.myri.com, February 1999.
11. Myricom Inc., *Myrinet User's Guide*, http://www.myri.com, 1996.
12. L. Prylli and B. Tourancheau, "BIP: a new protocol designed for high performance networking on myrinet," *Proceedings of IPPS/SPDP98*, 1998.
13. C. Yoo, H. -W. Jin, and S. -C. Kwon, "Asynchronous UDP," *IEICE Transactions on Communications*, Vol.E84-B, No.12, December 2001.

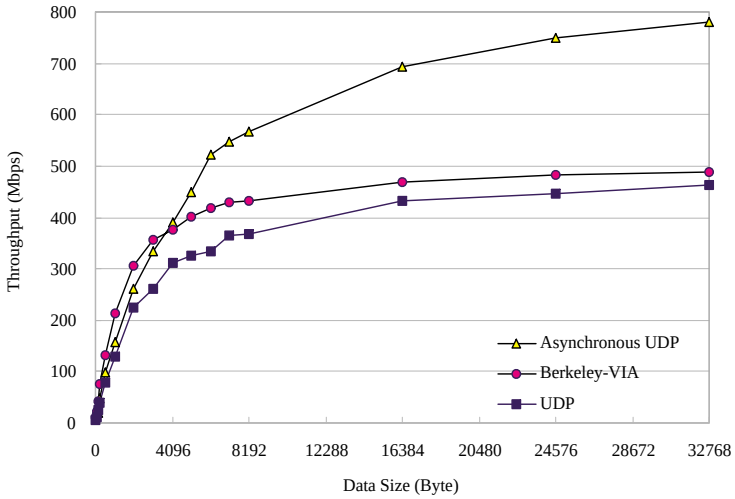

Fig. 1. Throughput comparison of Asynchronous UDP, Berkeley-VIA, and UDP

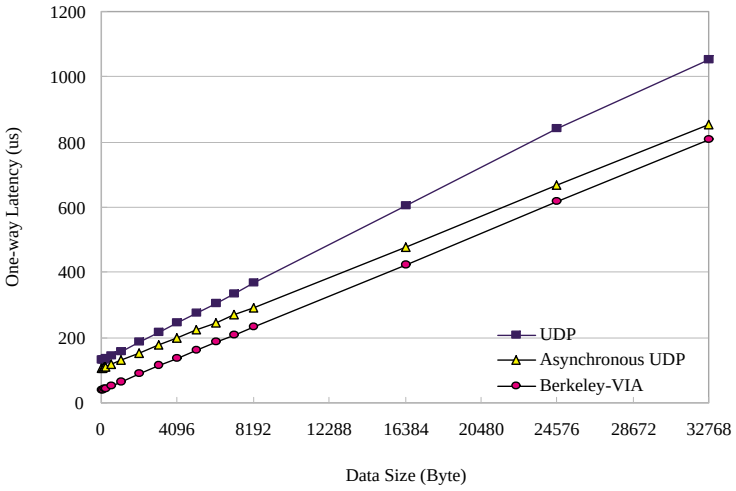

Fig. 2. One-way latency comparison of Asynchronous UDP, Berkeley-VIA, and UDP

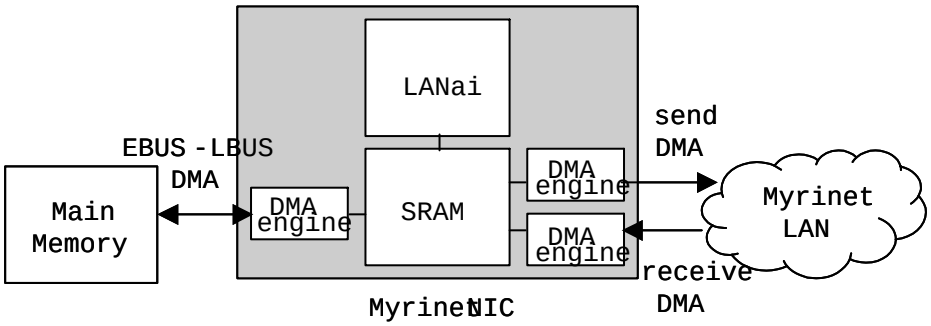

Fig. 3. Hardware feature of Myrinet NIC

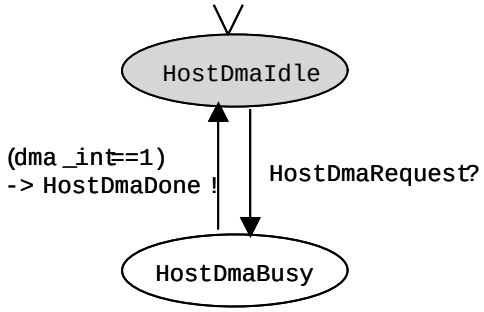

Fig. 4. State transition diagram of the hostDma module

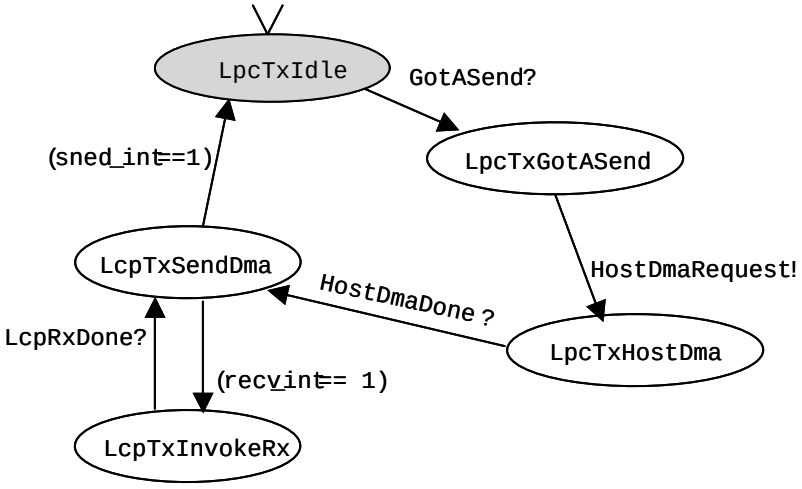

Fig. 5. State transition diagram of the lcpTx module

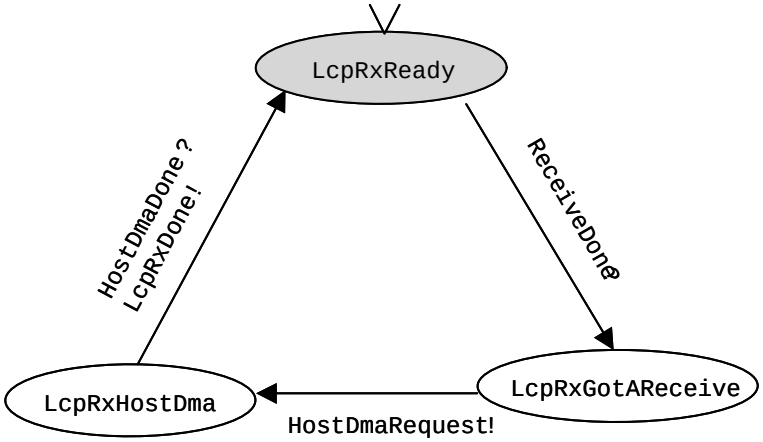

Fig. 6. State transition diagram of the lcpRx module

```
active proctype hostDma()
{ int event;

  do
  :: (hd_state == HostDmaIdle) ->
        Tohd?event;
        if
        :: (event == HostDmaRequest) ->
                hd_state = HostDmaBusy
        :: else -> skip
        fi;
        do
        :: (dma_int == 1) ->
                hd_state = HostDmaIdle;
                dma_int = 0;
                goto endofhd
        od;
  endofhd:
        ret2txrx!HostDmaDone
  od }
```

Fig. 7. Promela Specification of LCP - hostDma

```
active proctype lcpTx()
{ int event;

  do
  :: (lt_state == LcpTxIdle) ->
        Tolt?event;
        if
        :: (event == GotASend) ->
                lt_state = LcpTxGotASend;
                Tohd!HostDmaRequest;
                lt_state = LcpTxHostDma;
                ret2txrx?event;
                if
                :: (event == HostDmaDone) ->
                        lt_state = LcpTxSendDma;
                        do
                        :: if
                           :: (send_int == 1) ->
                                lt_state = LcpTxIdle;
                                send_int = 0;
                                goto endoflt
                           :: (recv_int == 1) ->
                                lt_state = LcpTxInvokeRx;
                                Tolr!ReceiveDone;
                                ret2tx?event;
                                if
                                :: (event == LcpRxDone) ->
                                        lt_state = LcpTxSendDma
                                :: else -> skip
                                fi
                           :: else -> skip
                           fi
                        od
                :: else -> skip
                fi
        :: else -> skip
        fi;
  endoflt:
        ret2lcp!0
  od }
```

Fig. 8. Promela Specification of LCP - lcpTx

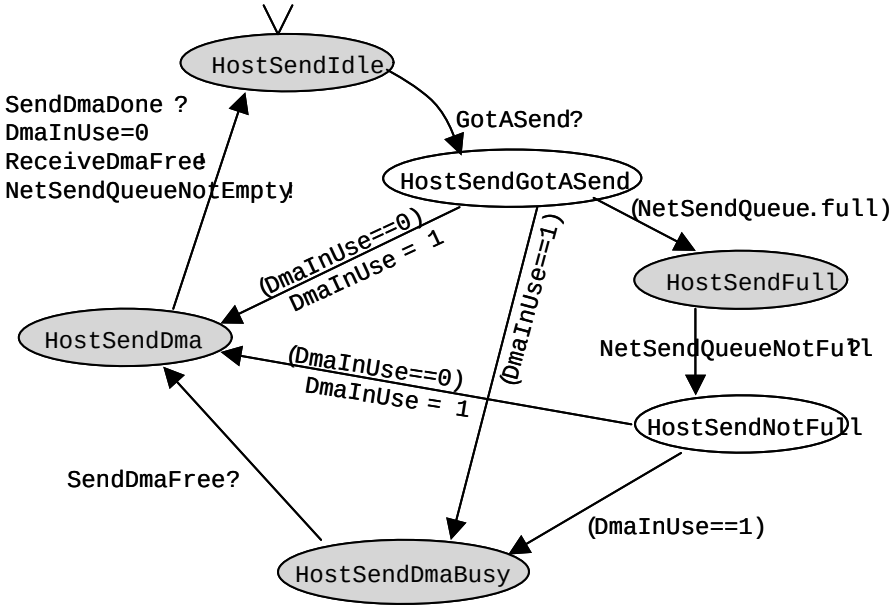

Fig. 9. State transition diagram of the hostSend module

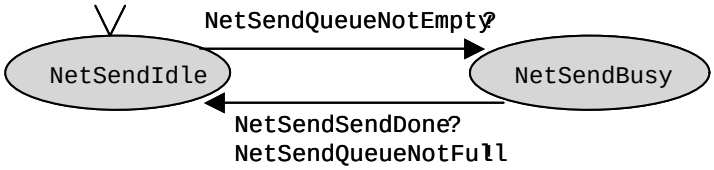

Fig. 10. State transition diagram of the netSend module

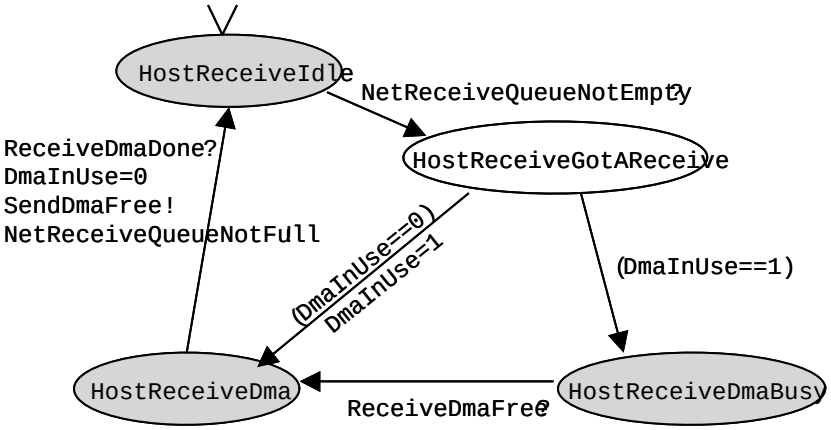

Fig. 11. State transition diagram of the hostReceive module

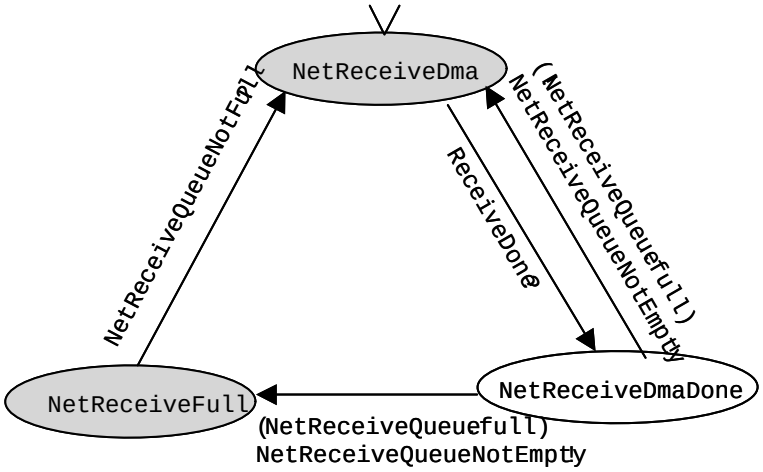

Fig. 12. State transition diagram of the netReceive module

```
active proctype hostSend()
{ int event;

  do
  :: (hs_state == HostSendIdle) ->
        Tohs?event;
        if
        :: (event == GotASend) ->
                if
                :: (nempty(hs_buffer)) ->
                        hs_buffer?hs_data;
                        hs_state = HostSendGotASend;
                        if
                        :: (full(ns_buffer)) ->
                                hs_state = HostSendFull
                        :: (nfull(ns_buffer)) ->
                                if
                                :: (DmaInUse==1) ->
                                        hs_state = HostSendDmaBusy
                                :: else ->
                                        DmaInUse = 1;
                                        hs_state = HostSendDma
                                fi
                        fi
                :: (empty(hs_buffer)) -> skip
                fi
        :: else -> skip
        fi;
        Return!0

  :: (hs_state == HostSendFull) ->
        Tohs?event;
        if
        :: (event == NetSendQueueNotFull) ->
                hs_state = HostSendNotFull;
                if
                :: (DmaInUse == 1) ->
                        hs_state = HostSendDmaBusy
                :: else ->
                        DmaInUse = 1;
                        hs_state = HostSendDma
                fi
        :: else -> skip
        fi;
        Return!0
```

```
:: (hs_state == HostSendDmaBusy) ->
      Tohs?event;
      if
      :: (event == SendDmaFree) ->
              hs_state = HostSendDma
      :: else -> skip
      fi;
      Return!0

:: (hs_state == HostSendDma) ->
      Tohs?event;
      if
      :: (event == SendDmaDone) ->
              DmaInUse = 0;
              if
              :: (hr_state == HostReceiveDmaBusy) ->
                      Return!ReceiveDmaFree;
                      DmaInUse = 1
              :: else -> skip
              fi;
              ns_buffer!hs_data;
              Return!NetSendQueueNotEmpty;
              if
              :: (nempty(hs_buffer)) ->
                      Return!GotASend
              :: (empty(hs_buffer)) -> skip
              fi;
              hs_state = HostSendIdle;
              dma_int = 0
      :: else -> skip
      fi;
      Return!0
od }
```

Fig. 13. Promela Specification of MCP - hostSend

```
active proctype netSend()
{ int event;

  do
  :: (ns_state == NetSendIdle) ->
        Tons?event;
        if
        :: (event == NetSendQueueNotEmpty) ->
                if
                :: (nempty(ns_buffer)) ->
                        ns_buffer?ns_data;
                        ns_state = NetSendBusy
                :: (empty(ns_buffer)) -> skip
                fi
        :: else -> skip
        fi;
        Return!0

  :: (ns_state == NetSendBusy) ->
        Tons?event;
        if
        :: (event == NetSendSendDone) ->
                Return!NetSendQueueNotFull;
                if
                :: (nempty(ns_buffer)) ->
                        Return!NetSendQueueNotEmpty
                :: (empty(ns_buffer)) -> skip
                fi;
                ns_state = NetSendIdle;
                send_int = 0
        :: else -> skip
        fi;
        Return!0
  od }
```

Fig. 14. Promela Specification of MCP - netSend