International Journal of Computer Applications (0975 – 8887)
Volume 179 – No.51, June 2018

Formal Modeling of Generalized Sliding Window Protocol in Promela using Spin Root Model-Checker

Arpit Gupta
Department of CSE, GL Bajaj Group of Institutions, Mathura Uttar Pradesh, India

Anil Kumar
Department of CSE, GL Bajaj Group of Institutions, Mathura Uttar Pradesh, India

Vinod Beniwal
Department of CSE, GL Bajaj Group of Institutions, Mathura Uttar Pradesh, India

Rama Kant
Department of CSE, GL Bajaj Group of Institutions, Mathura Uttar Pradesh, India

ABSTRACT

Sliding Window Protocols are an essential means of packet-form data transmission over the network. Having fixed window widths, it suffers from certain drawbacks which can be improved using concept of generalization of Sliding Window protocol. The generalized approach of sliding window protocol can have any combination of window sizes between Go-back- N and Selective-Repeat protocols. This paper presents the formal model checking of both Go-Back-N and Selective-Repeat protocols in ProMeLa using SPIN Root model-checker tool which would ultimately proceed in the verification of generalized version of sliding window protocol.

Keywords

Sliding Window Protocol, ProMeLa, SPIN Model-Checker tool, Internet, Data transmission, Formal Methods, Process Algebra, π-Calculus, Mobility Workbench

1. INTRODUCTION

Internet works through data transmission over the network between sender and one or more receivers. The data is transmitted with the help of various networking and communication protocols such as sliding window protocol. Sliding Window Protocol (SWP) [1] is a networking protocol which is used for transmission of packets over the Internet. It guarantees effective transmission of various types of messages between a sender to a recipient through a correspondence medium, in which messages may get lost because of specific factors over the medium or channel. The mean principle of SWPs is that the sender need not require waiting for an approaching acknowledgement from recipient before sending next sequence of messages, for ideal usage of system transfer speed.

This is the real motivation behind why numerous information correspondence frameworks incorporate the SWP, in significant varieties. The normal sliding window protocol suffers from limitation of fixed window size at sender and receiver end. A recent exposure to generalization of sliding window protocol helps to utilize the efficiency of this protocol. The proposed data transmission protocol is valid for any combination of sender as well as receiver window widths. The Stop-and-Wait, Go-Back-N and Selective-Repeat protocols [2] are demonstrated to be notable instances of generalized version. Studies show that the efficiency of sliding window protocol initially increases on increasing the size of the receiver window, and become saturated on further increasing the window size of receiver. Inside the process calculi group of formal demonstrating process, SWPs have pulled in much consideration, however exact formal confirmation ended up being shockingly troublesome. The primary commitment of this paper is to provide formal verification of generalized sliding window protocol in the language ProMeLa using SPIN Root model-checker tool. Since, the window size of generalized version of sliding window protocol is not fixed and can be anywhere between window widths of Go-Back-N and Selective-Repeat protocols, we will formally verify Go-Back-N and Selective-Repeat protocols which would ultimately result in the verification of generalized sliding window protocol (GSWP) [3].

For formal verification of generalized sliding window protocol, we have used concept of timeout for successful retransmission of packets, in case they exceed the maximum waiting time. Thus, ProMeLa [4], a process algebra language was a preferred choice for us [5, 11]. The major drawback of π- calculus is that it fails to implement timing concept in various networking and communication protocols. Hence, PROMELA (a Process Meta Language) [4] was developed for extending functionalities of π-calculus [5, 11].

Model checking [6] is used for verification purpose and is based on the idea of exploring and analyzing state space of a given system which is reachable. For verification of ProMeLa [4], SPIN model-checker [6] is a preferred choice. It works in two modes, either in simulation or in verification. Simulation do not need exhaustive search and thus can deal with bigger state spaces. On the other hand, verification uses exhaustive search technique. Testing is able to denote the errors only, but it cannot make a system error free. System can be guaranteed as error free only using formal verification which is essential especially for safety critical systems.

This work presents a model checking approach using the SPIN Root tool [6] to verify various properties and behavior of generalized sliding window protocol such as packet transmission, frame sequencing, acknowledgment triggering and retransmission in case of timeout. It discusses in detail how the generalized version of SWP can be modeled with the help of ProMeLa language using linear temporal logic (LTL) [7].

2. PROTOCOL OVERVIEW

In section 2.1 we present the basics of communication process. In section 2.2, we will study the efficiency issues of normal sliding window protocol. Section 2.3 describes the working of generalized sliding window protocol.

2.1 Communication Process

A sender and receiver communication process consists of one or more channels, message sequence and acknowledgment. The sender requests for sending the messages to receiver after connection establishment using 3 way handshaking. Two channels s2r and r2s are taken for message transmission. Channel s2r is used when sender is sending a message MSG to receiver while channel r2s is used by receiver to send an acknowledgment back to sender.

s2r!MSG denotes that message MSG is transmitted by sender to receiver using channel s2r, while r2s?ACK denoted that acknowledgement message ACK is received by sender through channel r2s. This is how a basic communication process works.

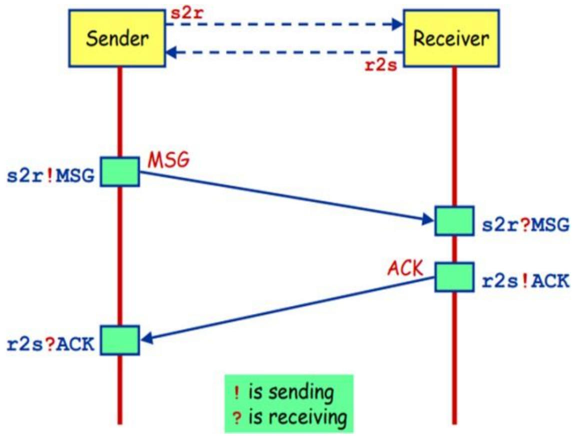


Fig 1: Sender and Receiver Communication Process

2.2 Efficiency issue in SWP

Efficiency is the critical aspect of communication protocols [8]. It can be measured by calculating the quantity of the service provided by the communication system with respect to quantity of resources used. Efficiency evenly depends on the properties of the communication channel, on communication traffic and on the communication protocol as well as its parameters. Channel loss rate [8] is the communication channel property that dramatically impacts the efficiency of the protocol. The efficiency of protocol increases with the rise in receiver window width. However, both theoretical consideration and experimental results reveal that the efficiency increases at first with the receiver window width, but may hold at some intermediate receive window widths, or even slightly drop towards the Selective-Repeat case. Selective-Repeat protocol has equal sender and receiver window width. Thus, it requires more buffer memory space (is almost twice as much as the Go-Back-N protocol) due to larger receiver window size and, due to more complex memory management at the receiver side.

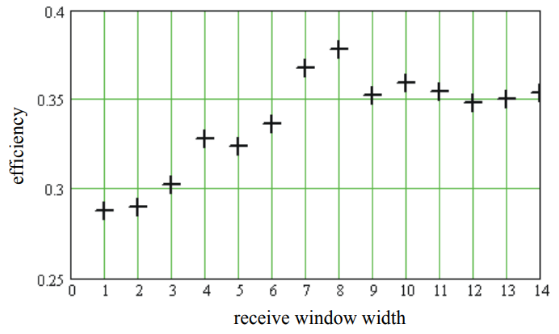


Fig 2: Dependence of protocol efficiency on receiver window width

2.3 Generalized Sliding Window Protocol

In generalized version of sliding window protocol [3], the window width of receiver is more than that of the Go-Back-N protocol and less than that of the Selective-Repeat protocol, while the sender window width is normally the same in both cases (so that the transmit window never becomes full in the lossless case). The generalization of normal sliding window protocols allows us to treat this family of communication and networking protocols more systematically. By studying the generalized version of this protocol, students can now more easily grasp the properties of the family members and the differences between them, as these members are considered as only special cases of the generalized protocol.

Logical correctness is essentially required for any communication protocol. It is therefore necessary to teach students how to determine the logical correctness of protocols and how to verify it, either by reasoning, analysis, verification or simulation. Of course, we need certain examples to verify that. Without any doubt, the generalized sliding window protocol is much suitable for this purpose, firstly because of its moderate complexity and secondly, it can represent all forms of sliding window protocol named as Stop- and-Wait, Go-Back-N and Selective-Repeat ARQ [2] as its special conditions.

3. PROMELA MODEL

Formal methods utilize scientific verification as a supplement to test the systems keeping in mind the end goal to guarantee correct behavior. As systems turn out to be more complex, and security turns into a more vital issue, the formal way to deal with system configuration offers another level of protection. Process algebra, as a formal method, is the study of the operational response of a parallel or distributed system. Leading examples of process algebra (also called process calculi) are π–calculus [5, 11], Algebra of Computing Processes (ACP), PEPA and ProMeLa [4].

We have already described the reason of implementing sliding window protocols in ProMeLa over π–calculus in above sections. ProMeLa can support almost all functionalities that π–calculus [5, 11] features, except for bi-simulation equivalence. However, for verifying sliding window protocols formally, ProMeLa is the best choice for implementation of these protocols because ProMeLa also allows implementing linear temporal logic (LTL) in code which π–calculus [5, 11] fails to deliver. ProMeLa stands for Process Meta Language which means this language has been created for scheduling process in communication and networking protocols along with support of time related

concept. Developed by Gerard J. Holzmann, ProMeLa has syntax close to C language making it easier to understand for newbies and supports more functionalities than π–calculus making it a desirable choice of users.

ProMeLa model consists of:

- variable declarations along with their types
- channel declarations
- type declarations
- process declarations
- init process (optional)

Basic example of ProMeLa model:

```
Bool  flag;
```

```
chan PtoQ;

mtype = \{msg, ack\}; proctype P()\{ proctype Q() \{

\} \}

Init \{

}
```

4. SPIN ROOT MODEL-CHECKER

SPIN Root is a open source tool developed by Gerad J. Holzmann in order to verify the implementation of code written in ProMeLa [4]. It is used for analyzing the reasonable consistency of various concurrent systems, specifically of data communication protocols. SPIN Root is a model checking tool that systematically verifies that, given for a logical property and finite state, whether a property holds for a given model.

Inputs: M, a finite state model of the system and ϕ, a requirement.

Output: Yes or No + a system run violating the requirement (Counter example).

SPIN model checking verification [6] is focused mainly on proving the correctness of interactions of processes; not much importance is given to the internal computations of different processes. In SPIN tool, communication is done through rendezvous primitives (synchronous), with non- concurrent message going through various cradled channels, through access to shared factors or with any blend of these..

SPIN model-checker [6] contributes:

- C-language like notation for specifying various system designs or finite-state abstraction unambiguously.
- Notation for representing the general correctness using LTL.
- Helps in establishing the logical consistency of system design specified in ProMeLa and matching the correctness requirements written as LTL formulae.

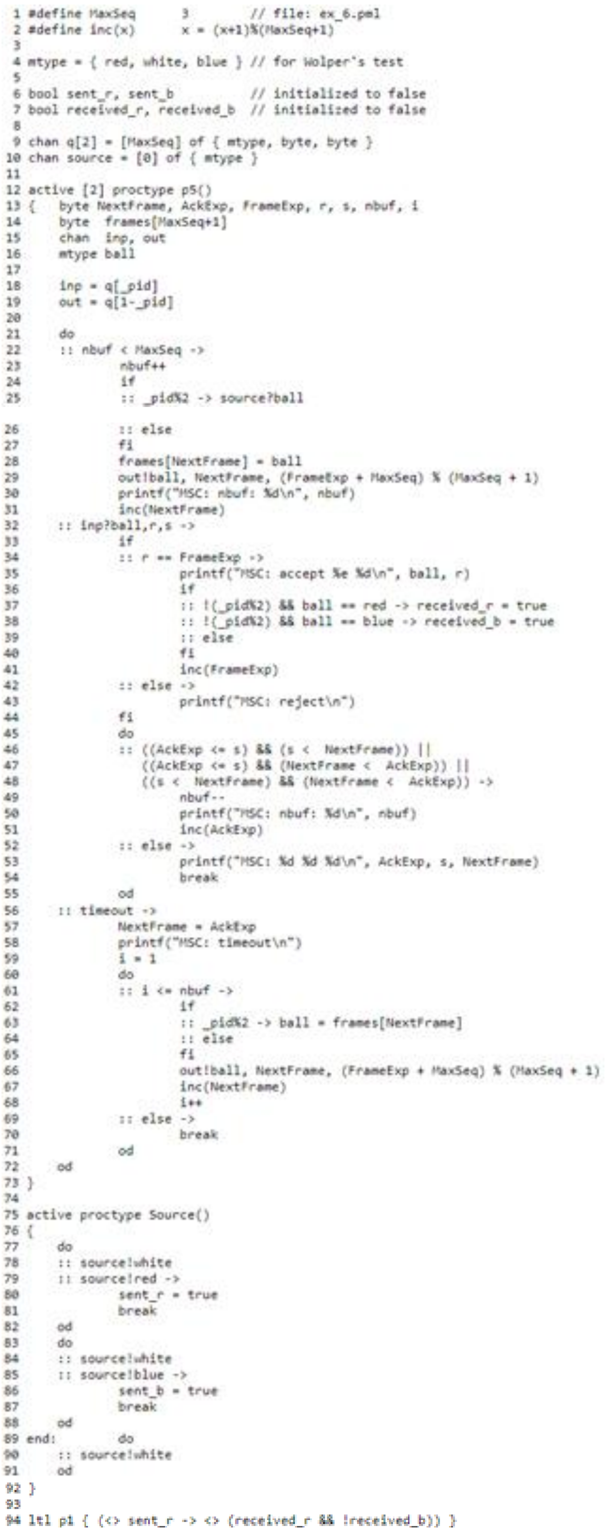
SPIN [6] being a model-checker tool has won various awards on account of its performance for formal modeling. Some of the precious awards including Thomas Alva Edison patent award in the Information Technology Category, for the patent on software verification with SPIN in 2003 and the popular ACM software system award for 2001.

5. CODE IMPLEMENTATION

The generalized version of sliding window protocol is verified using the code execution of Go-Back-N and Selective-Repeat ARQ protocol [2] in ProMeLa. The reason is that the window size of generalized sliding window protocol is not fixed and can be of any arbitrary size between Go- Back-N and Selective-Repeat ARQ. Since code implementation of Go-Back-N ARQ and Selective-Repeat ARQ is verifiable in ProMeLa using SPIN root model-checker [6], hence generalization of sliding window protocol is automatically verified.

5.1 Go-Back-N ARQ Code

The following code has been referred from Tanenbaum [10] in 1989. MaxSeq has been defined to size 3. Two processes Source and p5 are taken. According to Wolper's law [9], if we can transmit three different messages, rest of messages can be transmitted successfully. Sequence number is represented using colors namely: red, white and blue. mtype keyword is used as an enumeration to select one of the colors red, white and blue. Timeout is used for retransmission of message.



```
#define MaxSeq      3        // file: ex_6.pml
#define inc(x)      x = (x+1)%(MaxSeq+1)

mtype = { red, white, blue } // for Wolper's test

bool sent_r, sent_b          // initialized to false
bool received_r, received_b  // initialized to false

chan q[2] = [MaxSeq] of { mtype, byte, byte }
chan source = [0] of { mtype }

active [2] proctype p5()
{   byte NextFrame, AckExp, FrameExp, r, s, nbuf, i
    byte  frames[MaxSeq+1]
    chan  inp, out
    mtype ball

    inp = q[_pid]
    out = q[1-_pid]

    do
    :: nbuf < MaxSeq ->
            nbuf++
            if
            :: _pid%2 -> source?ball

            :: else
            fi
            frames[NextFrame] = ball
            out!ball, NextFrame, (FrameExp + MaxSeq) % (MaxSeq + 1)
            printf("MSC: nbuf: %d\n", nbuf)
            inc(NextFrame)
    :: inp?ball,r,s ->
            if
            :: r == FrameExp ->
                    printf("MSC: accept %e %d\n", ball, r)
                    if
                    :: !(_pid%2) && ball == red -> received_r = true
                    :: !(_pid%2) && ball == blue -> received_b = true
                    :: else
                    fi
                    inc(FrameExp)
            :: else ->
                    printf("MSC: reject\n")
            fi
            do
            :: ((AckExp <= s) && (s <  NextFrame)) ||
               ((AckExp <= s) && (NextFrame <  AckExp)) ||
               ((s <  NextFrame) && (NextFrame <  AckExp)) ->
                    nbuf--
                    printf("MSC: nbuf: %d\n", nbuf)
                    inc(AckExp)
            :: else ->
                    printf("MSC: %d %d %d\n", AckExp, s, NextFrame)
                    break
            od
    :: timeout ->
            NextFrame = AckExp
            printf("MSC: timeout\n")
            i = 1
            do
            :: i <= nbuf ->
                    if
                    :: _pid%2 -> ball = frames[NextFrame]
                    :: else
                    fi
                    out!ball, NextFrame, (FrameExp + MaxSeq) % (MaxSeq + 1)
                    inc(NextFrame)
                    i++
            :: else ->
                    break
            od
    od
}

active proctype Source()
{
    do
    :: source!white
    :: source!red ->
            sent_r = true
            break
    od
    do
    :: source!white
    :: source!blue ->
            sent_b = true
            break
    od
end:        do
    :: source!white
    od
}

ltl p1 { (<> sent_r -> <> (received_r && !received_b)) }
```

5.2 Selective Repeat ARQ Code

We have developed a code for Selective-Repeat ARQ which is verified in SPIN model-checker tool [6] and written in ProMeLa [4]. The following code has been referred from

Tanenbaum in 1989 [10]. MaxSeq has been defined to size 3. According to Wolper's law [9], if we can transmit three different messages, rest of messages can be transmitted successfully. Sequence number is represented using colors namely: red, white and blue. mtype keyword is used as an enumeration to select one of the colors red, white and blue. Timeout is used for retransmission of message.

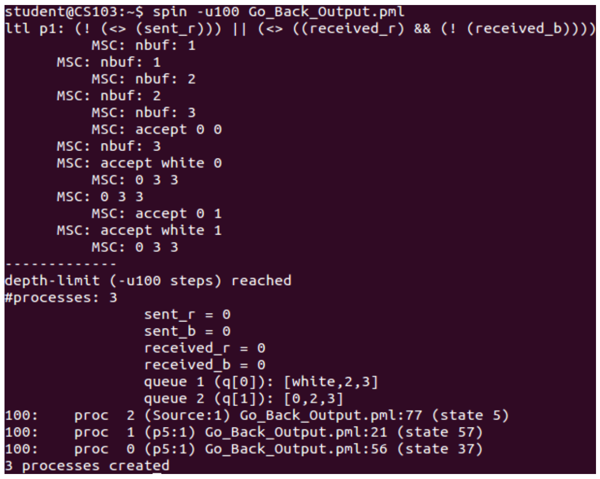
```
#define MaxSeq        3        // file: ex_6.pml
#define inc(x)        x = (x+1)%((MaxSeq+1)/2)
#define NR_Bufs (MaxSeq+1)/2
mtype = { red, white, blue } // for Wolper's test
bool sent_r, sent_b            // initialized to false
bool received_r, received_b    // initialized to false
bool no_nak = true, nak
byte oldestframe = MaxSeq+1
chan q[2] = [MaxSeq] of { mtype, byte, byte }
chan source = [0] of { mtype }
active [2] proctype p5()
{     byte NextFrame, AckExp, arr[5], inbuf[5],  r, s, nbuf, rkind
      byte  frames[MaxSeq+1],FrameExp, toofar=NR_Bufs, i, data
      chan  inp, out
      mtype ball
      inp = q[_pid]
      out = q[1-_pid]
      do
      :: nbuf < MaxSeq ->
              nbuf++
              if
              :: _pid%2 -> source?ball
              :: else
              fi
              frames[NextFrame] = ball

             out!data, ball, NextFrame, (FrameExp + MaxSeq) % (MaxSeq + 1)
             printf("MSC: nbuf: %d\n", nbuf)
             inc(NextFrame)
     :: inp?ball,r,s ->
             if
             :: rkind == data ->
                  printf("MSC: accept %e %d\n", ball, r)
                  if
                  :: ((r != FrameExp) && (no_nak)) ->
                  :: out!nak,0,ball,(FrameExp + MaxSeq) % (MaxSeq + 1)
                  fi
                  if
                  :: ((((FrameExp <= r) && (r <  toofar)) ||
                      ((toofar <= FrameExp) && (FrameExp <  r)) ||
                        ((r <  toofar) && (toofar <  FrameExp))) &&
                      (arr[r%NR_Bufs]==false)) ->
                  :: arr[r%NR_Bufs]=true
                  :: inbuf[r%NR Bufs]=ball
                    do
                      :: arr[FrameExp%NR_Bufs]
                         !inbuf[FrameExp%NR_Bufs]
                       no nak = true
                       arr[FrameExp%NR_Bufs] = false
                       inc(FrameExp)
                       inc(toofar)
                    od
                 fi
                    :: else ->
                    printf("MSC: reject\n")
            fi
                    if
                    :: (rkind==nak) && (AckExp <= (((s+1)%(MaxSeq+1)) &&
                       ((s+1)%(MaxSeq+1)) <  NextFrame)) ||
                       ((NextFrame <= AckExp && (AckExp <  ((s+1)%(MaxSeq+1)))) ||
                       ((((s+1)%(MaxSeq+1)) <  NextFrame) && (NextFrame <  AckExp))) ->
                       out!data, ball, ((s+1)%(MaxSeq+1)), FrameExp
                    fi
            do
            :: ((AckExp <= s) && (s <  NextFrame)) ||
               ((AckExp <= s) && (NextFrame <  AckExp)) ||
               ((s <  NextFrame) && (NextFrame <  AckExp)) ->
                   nbuf--
                   printf("MSC: nbuf: %d\n", nbuf)
                   inc(AckExp)
            :: else ->
                   printf("MSC: %d %d %d\n", AckExp, s, NextFrame)
                   break
            od
     :: timeout ->
          out!data, ball, oldestframe, FrameExp
        break
        od
}
active proctype Source()
{
    do
    :: source!white
    :: source!red ->
            sent_r = true
            break
    od
    do
    :: source!white
    :: source!blue ->
            sent_b = true
            break
    od
end:        do
    :: source!white
    od
}
ltl p1 { (<> sent_r -> <> (received_r && !received_b)) }
#define MaxSeq       3         // file: ex_6.pml
#define inc(x)       x = (x+1)%((MaxSeq+1)/2)
 #define NR_Bufs (MaxSeq+1)/2
 mtype = { red, white, blue } // for Wolper's test
bool sent_r, sent_b          // initialized to false
bool received_r, received_b  // initialized to false
bool no nak = true. nak
byte oldestframe = MaxSeq+1
chan q[2] = [MaxSeq] of { mtype, byte, byte }
chan source = [0] of { mtype }
active [2] proctype p5()
{    byte NextFrame, AckExp, arr[5], inbuf[5], r, s, nbuf
     byte  frames[MaxSeq+1], FrameExp, toofar=NR_Bufs, rkind, i, data
     chan  inp, out
     mtype ball
     inp = q[_pid]
     out = q[1-_pid]
     do
     :: nbuf < MaxSeq ->
             nbuf++
             if
             :: _pid%2 -> source?ball
             :: else
             fi
             frames[NextFrame] = ball
             out!data, ball, NextFrame, (FrameExp + MaxSeq) % (MaxSeq + 1)
             printf("MSC: nbuf: %d\n", nbuf)
             inc(NextFrame)
     :: inp?ball,r,s ->
             if
             :: rkind == data ->
                  printf("MSC: accept %e %d\n", ball, r)
                 if
                    :: ((r != FrameExp) && (no_nak)) ->
                 :: out!nak,0,ball,(FrameExp + MaxSeq) % (MaxSeq + 1)
                 fi
                 if
                 :: ((((FrameExp <= r) && (r <  toofar)) ||
                     ((toofar <= FrameExp) && (FrameExp <  r)) ||
                       ((r <  toofar) && (toofar <  FrameExp))) &&
                         (arr[r%NR_Bufs]==false)) ->
                 :: arr[r%NR_Bufs]=true
                 :: inbuf[r%NR_Bufs]=ball
                    do
                        :: arr[FrameExp%NR_Bufs]
                           !inbuf[FrameExp%NR_Bufs]
                        no_nak = true
                        arr[FrameExp%NR_Bufs] = false
                        inc(FrameExp)
                        inc(toofar)
                     od
                  fi
                     :: else ->
                     printf("MSC: reject\n")
            fi
                     if
                     :: (rkind==nak) && (AckExp <= (((s+1)%(MaxSeq+1)) &&
                        ((s+1)%(MaxSeq+1)) <  NextFrame)) ||
                        ((NextFrame <= AckExp && (AckExp <  ((s+1)%(MaxSeq+1)))) ||
                        ((((s+1)%(MaxSeq+1)) <  NextFrame) && (NextFrame <  AckExp))) ->
                        out!data, ball, ((s+1)%(MaxSeq+1)), FrameExp
                     fi
            do
            :: ((AckExp <= s) && (s <  NextFrame)) ||
               ((AckExp <= s) && (NextFrame <  AckExp)) ||
               ((s <  NextFrame) && (NextFrame <  AckExp)) ->
                     nbuf--
                     printf("MSC: nbuf: %d\n", nbuf)
                     inc(AckExp)

                     inc(AckExp)
             :: else ->
                     printf("MSC: %d %d %d\n", AckExp, s, NextFrame)
                     break
             od
     :: timeout ->
             out!data, ball, oldestframe, FrameExp
          break
         od
 }
active proctype Source()
 {
    do
    :: source!white
    :: source!red ->
            sent_r = true
            break
    od
    do
    :: source!white
    :: source!blue ->
            sent_b = true
            break
    od
 end:       do
    :: source!white
    od
 }
 ltl p1 { (<> sent_r -> <> (received_r && !received_b)) }
```

6. CONCLUSION

We have successfully executed the above codes of Go-Back-N ARQ and Selective-Repeat ARQ [2] written in ProMeLa [4] using SPIN Root Model-checker tool [6]. Since, Generalized sliding window protocol has no fixed window size and can have any arbitrary window width between Go-Back-N ARQ and Selective-Repeat ARQ. By verifying both Go-Back-N ARQ and Selective-Repeat ARQ protocols in

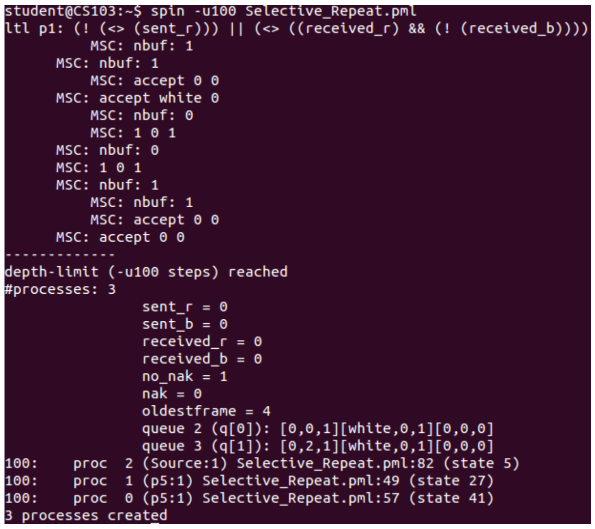
ProMeLa using SPIN Model-checker tool, the generalization phenomenon of sliding window protocol is automatically verified.

In future prospects, instead of using a fixed size window for data transmission over the network, we could use the concept of generalized version of sliding window protocol which would not only reduce the congestion and jitter delay, but will also increase the predictability of the data transmission over the network.



```
student@CS103:~$ spin -u100 Go_Back_Output.pml
ltl p1: (! (<> (sent_r))) || (<> ((received_r) && (! (received_b))))
          MSC: nbuf: 1
      MSC: nbuf: 1
          MSC: nbuf: 2
      MSC: nbuf: 2
          MSC: nbuf: 3
          MSC: accept 0 0
      MSC: nbuf: 3
      MSC: accept white 0
          MSC: 0 3 3
      MSC: 0 3 3
          MSC: accept 0 1
      MSC: accept white 1
          MSC: 0 3 3
-------------
depth-limit (-u100 steps) reached
#processes: 3
                sent_r = 0
                sent_b = 0
                received_r = 0
                received_b = 0
                queue 1 (q[0]): [white,2,3]
                queue 2 (q[1]): [0,2,3]
100:    proc  2 (Source:1) Go_Back_Output.pml:77 (state 5)
100:    proc  1 (p5:1) Go_Back_Output.pml:21 (state 57)
100:    proc  0 (p5:1) Go_Back_Output.pml:56 (state 37)
3 processes created
```

Fig 3: Executed code of Go-Back-N ARQ protocol



```
student@CS103:~$ spin -u100 Selective_Repeat.pml
ltl p1: (! (<> (sent_r))) || (<> ((received_r) && (! (received_b))))
          MSC: nbuf: 1
      MSC: nbuf: 1
          MSC: accept 0 0
      MSC: accept white 0
          MSC: nbuf: 0
          MSC: 1 0 1
      MSC: nbuf: 0
      MSC: 1 0 1
      MSC: nbuf: 1
          MSC: nbuf: 1
          MSC: accept 0 0
      MSC: accept 0 0
-------------
depth-limit (-u100 steps) reached
#processes: 3
                sent_r = 0
                sent_b = 0
                received_r = 0
                received_b = 0
                no_nak = 1
                nak = 0
                oldestframe = 4
                queue 2 (q[0]): [0,0,1][white,0,1][0,0,0]
                queue 3 (q[1]): [0,2,1][white,0,1][0,0,0]
100:    proc  2 (Source:1) Selective_Repeat.pml:82 (state 5)
100:    proc  1 (p5:1) Selective_Repeat.pml:49 (state 27)
100:    proc  0 (p5:1) Selective_Repeat.pml:57 (state 41)
3 processes created
```

Fig 4: Executed code of Selective Repeat ARQ protocol

7. REFERENCES

[1] Peterson, Larry L. & Davie, Bruce S. "Computer Networks: A Systems Approach", Morgan Kaufmann, 2000. ISBN 1-55860-577-0Ding, W. and Marchionini, G. 1997 A Study on Video Browsing Strategies. Technical Report. University of Maryland at College Park.

[2] Ikegawa, Takashi & Takahashi, Yukio. (2007). Sliding window protocol with Selective-Repeat ARQ: Performance modeling and analysis. Telecommunication Systems. 34. 167-180. 10.1007/s11235-007-9032-6.

[3] Hercog, Drago. (2018). The Importance of Sliding Window Protocol Generalisation in a Communication Protocols Course.

[4] Sharma, Asankhaya. "A Refinement Calculus for ProMeLa." Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on. IEEE, 2013.

[5] Abadi, Martín & Blanchet, Bruno & Fournet, Cédric. (2016). The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. Journal of the ACM. 65. 10.1145/3127586.

[6] Peter C. Dillinger , Panagiotis (Pete) Manolios, Fast, All-Purpose State Storage, Proceedings of the 16th International SPIN Workshop on Model Checking Software, June 26-28, 2009, Grenoble, France

[7] Andreas Bauer , Martin Leucker , Christian Schallhart, Runtime Verification for LTL and TLTL, ACM Transactions on Software Engineering and Methodology (TOSEM), v.20 n.4, p.1-64, September 2011

[8] BANSAL, KAPIL. (2012). Analysis of Sliding Window Protocol for Connected Node. International Journal of Soft Computing and Engineering (IJSCE). 2. 292-294.

[9] R. Gerth, D. Peled, M.Y. Vardi and P. Wolper, "Simple On-The-Fly Automatic Verification of Linear Temporal Logic," Proc. IFIP/WG6.1 Symp. Protocol Specification, Testing, and Verification (PSTV95), pp. 3-18,Warsaw, Poland, Chapman & Hall, June 1995.

[10] David Wetherall; Tanenbaum, Andrew S. (2011). Computer networks. Upper Saddle River, NJ: Pearson Prentice Hall. ISBN 0-13-212695-8.

[11] Kant, Rama & Gaur, Manish. (2015). A Stochastic Extension of the Routing Calculi.