Model Checking

CS511

Summary

- We've seen some examples of Promela models
- We've verified some Promela models that involved the `assert` statement
- This lecture:
  - Channels in Promela
  - An example from distributed programming: distributed mutual exclusion

Channels in Promela

Synchronous Channels

Asynchronous Channels

Distributed Mutual Exclusion

Channels

- Inspired in CSP (1978, Tony Hoare)
- A channel is a data type with two operations
  - send
  - receive
- Channels are global: any process can send on a channel and receive from a channel
- Every channel has a message type

```
chan ch = [capacity] of { typename,...,typename }
```

- 0 capacity: rendezvous channels (=synchronous)
- >0 capacity: buffered channels

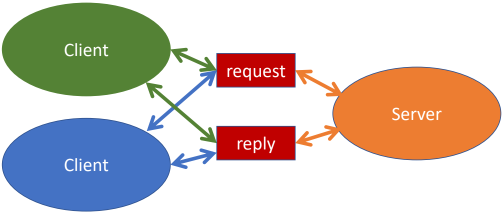
Example 1 (eg1.pml)

```
chan request=[0] of {byte};

active proctype Server() {
  byte client;
end:
  do
   ::request ? client -> pri
  od
}

active proctype Client0() {
  request ! 0;
}

active proctype Client1() {
  request ! 1;
}
```



- label `end` ensures that an end state with the server blocked on a receive statement is not considered invalid

Running the Example

```
> spin eg1.pml
      Client 0
      Client 1
      timeout
#processes: 1
 6: proc 0 (Server:1) eg1.pml:8 (state 3) <valid end state>
 3 processes created
```

- Computation in which execution terminates with the output `timeout` means that no statements are executable (deadlock)

Symbolic Names as Values

- Example:

```
mtype { open, close, reset }
init {
        mtype x = reset;
        printm(x);
}
```

- Shorthand for send and receive commands
  - `ch!e1,e2,...` can be written: `ch!e1(e2,...)`
  - `ch?e1,e2,...` can be written: `ch?e1(e2,...)`

  when `e1` is an `mtype`

```
mtype { open, close, reset };
chan ch = [0] of { mtype, byte, byte };
byte id, n;
/* send statement can be in either of the formats: */
ch ! open, id, n;
ch ! open(id, n);
```

Example

```
mtype { red, yellow, green };
chan ch = [0] of { mtype, byte, bool };

active proctype Sender() {
  ch ! red, 20, false;
  printf("Sent message\n")
}

active proctype Receiver() {
  mtype color;
  byte time;
  bool flash;
  ch ? color, time, flash;
  printf("Received message %e, %d, %d\n",color,time,flash)
}
```

Example: Rendezvous

```
chan request = [0] of { byte };
chan reply = [0] of { bool };

active proctype Server() {
        byte client;
end:
        do
        ::  request ? client ->
                        printf("Client %d\n", client);
                        reply ! true
        od
}

active proctype Client0() {
        request ! 0;
        reply ? _
}

active proctype Client1() {
        request ! 1;
        reply ? _
}
```

- _ is an anonymous variable

Another Example (eg4.pml)

```
chan request = [0] of { byte };
chan reply = [0] of { byte };

active [2] proctype Server() {
    byte client;
end:
    do
    :: request ? client ->
        printf("Client %d processed by server %d\n", client, _pid
        reply ! _pid
    od
}

active [2] proctype Client() {
    byte server;
    request ! _pid;
    reply ? server;
    printf("Reply received from server %d by client %d\n", server,
}
```

- Note quite right, why?

Verifying Previous Example

- Use Spin to verify that this program is not correct
  - In Server: Send back `client`
  - In Client: insert an `assert` to compare the received pid with its own

Verifying Previous Example (eg5.pml)

```
chan request = [0] of { byte };
chan reply = [0] of { byte, byte };

active [2] proctype Server() {
  byte client;
end:
  do
    :: request ? client ->
         printf("Client %d processed by server %d\n", client, _pid
         reply ! _pid, client
  od
}

active [2] proctype Client() {
  byte server;
  byte whichClient;
  request ! _pid;
  reply ? server, whichClient;
  printf("Reply received from server %d by client %d\n", server, _
  assert (whichClient == _pid);
}
```

Channels as Values

```
chan request = [0] of { byte, chan };
chan reply [2] = [0] of { byte, byte }; /* array of 2 channels */

active [2] proctype Server() {
  byte client;
  chan replyChannel;
end:
  do
  :: request ? client, replyChannel ->
       printf("Client %d processed by server %d\n",client,_pid);
       replyChannel ! _pid, client;
  od
}

active [2] proctype Client() {
   byte server;
   byte whichClient;
   request ! _pid, reply[_pid-2];
   reply[_pid-2] ? server, whichClient;
   printf("Reply received from server %d by client %d\n", server,
   assert(whichClient == _pid)
}
```

Slight Improvement on the Previous Example

- Problem: depends on `pid` to access array of channels
- We can correct this by passing arguments on to Client and Server with these indices

Previous Example Improved

```
chan request = [0] of { byte, chan };
chan reply [2] = [0] of { byte, byte };

proctype Server(byte me) {
    byte client;
    chan replyChannel;
end:
    do
    ::  request ? client, replyChannel ->
            printf("Client %d processed by server %d\n", client,
            replyChannel ! me, client;
    od
}

proctype Client(byte me; byte myChannel) {
    byte server;
    byte whichClient;
    request ! me, reply[myChannel];
    reply[myChannel] ? server, whichClient;
    printf("Reply received from server %d by client %d\n", server,
    assert (whichClient == me);
}
/* continued */
```

Previous Example Improved

```
init {
    atomic {
        run Server('s');
        run Server('t');
        run Client('c', 0);
        run Client('d', 1);
    }
}
```

Exercise

- Spawn an additional client so that there are three of them
- Prove that at most two clients can be served at any given time
- For that
  - add a global variable `numClients`
  - count the clients that have sent a request but not yet received a reply
  - insert an appropriate `assert`

Exercise: Solution

Replace `Client` and modify `init` as indicated below

```
proctype Client(byte me; byte myChannel) {
  byte server;
  request ! me, reply[myChannel];
  numClients++;
  assert (numClients <= 2);
  numClients--;
  reply[me] ? server;
  printf("Client %d received reply on channel %d, processed by ser
}

init {
  atomic {
      run Server('s');
      run Server('t');
      run Client('c', 0);
      run Client('d', 1);
      run Client('e', 2);
  }
}
```

Channels in Promela

- Synchronous Channels
- Asynchronous Channels

Distributed Mutual Exclusion

Buffered Channels

```
chan ch = [3] of mtype, byte, bool ;
```

- Send and receive statements treat the channel as a FIFO (first in-first out) queue.
- Send statement is executable if there is room in the channel for another message (the channel is not full)
  - The `-m` option in spin causes send not to block when channel is full but message is discarded
- Receive statement is executable if there are messages in the channel (the channel is not empty)
- The channel is part of the states of the computation.

Operations on Buffered Channels

- `full` and `empty`, and their negations `nfull` and `nempty`.
  - The negations `!full` and `!empty` are not allowed in Promela
- `len(ch)` returns the number of messages in channel `ch`

Example

```
chan request = [2] of { byte, chan};
chan reply[4] = [2] of { byte };

active [2] proctype Server() {
  byte client;
  chan replyChannel;
  do
  :: empty(request) ->
      printf("No requests for server %d\n", _pid)
  :: request ? client, replyChannel ->
      printf("Client %d processed by server %d\n", client, _pid);
      replyChannel ! _pid
  od
}
active [4] proctype Client() {
  byte server;
  do
  :: full(request) ->
      printf("Client %d waiting for non-full channel\n", _pid)
  :: request ! _pid, reply[_pid-2] ->
      reply[_pid-2] ? server;
      printf("Reply received from server %d by client %d\n", serv
  od
}
```

Example: Random Receive

- Suppose we want to replace the array of four reply channels with just one channel
- The message sent on the reply channel contains the server ID, as well as the ID of the client that was received from the request channel
- We must ensure that it is possible for a client to receive only messages meant for it
- A random receive statement (denoted `??`) will remove the first message that matches the variables and values in the statement

Example: Random Receive

```
chan request = [4] of { byte };
chan reply = [4] of { byte, byte };

active [2] proctype Server() {
  byte client;
end:
  do
  :: request ? client ->
     printf("Client %d processed by server %d\n", client, _pid);
     reply ! _pid, client
  od
}

active [4] proctype Client() {
  byte server;
  request ! _pid;
  reply ?? server, eval(_pid);
  printf("Reply received from server %d by client %d\n", server, _
}
```

Note the use of `eval(_pid)` given that the value of `_pid` is only known at runtime

Channels in Promela

Synchronous Channels

Asynchronous Channels

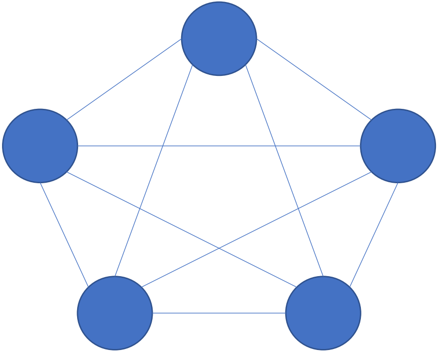
Distributed Mutual Exclusion

Assumptions on our Distributed Model

- Nodes communicate through message passing with every other node
- Nodes execute one or more processes
- Nodes do not fail
- Channels deliver messages without error, though not necessarily in the order in which they were sent
- Transit time of messages is arbitrary but finite

Ricart-Agrawala[1]

Problem: given a set of distributed nodes connected as indicated below, devise an algorithm for mutual exclusion



[1] Ricart, Glenn; Agrawala, Ashok K. (1 January 1981). "An optimal algorithm for mutual exclusion in computer networks". Communications of the ACM. 24 (1): 9–17.

Ricart-Agrawala (Outline)

1. Each node picks a ticket number
2. Sends a request with the number to all other nodes
3. Waits to receive a reply
4. Once it gets a reply from all other nodes, it enters the CS
5. A node only sends a reply to a request if the number is smaller than its own

This is best implemented with two processes per node

- main
- receive

Preliminary Version

```
integer myNum := 0
set of node IDs deferred := empty set
```

Main:

```
// non-CS
myNum = chooseNumber
for all other nodes N
 send(request,N,myID,myNum)
await replies from all other nodes
// CS
for all nodes N in deferred
  remove N from deferred
  send(reply,N,myID)
```

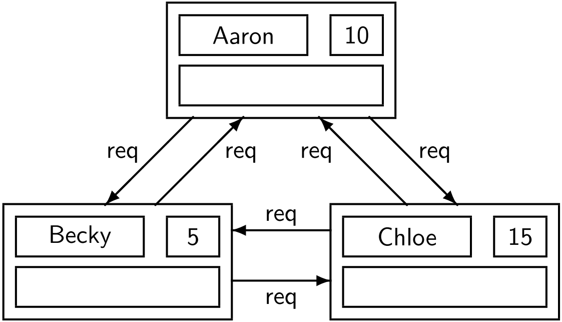
Receive

```
int source, reqNum
receive(request,source,reqNum)
if reqNum < myNum
    send(reply,source,myID)
else
    add source to deferred
```

NB: Code in each process above runs inside an infinite loop

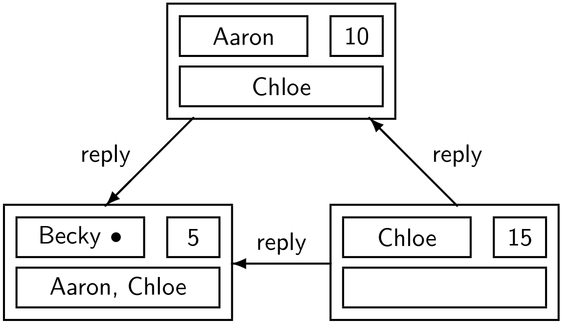
Scenario of an Example

- All nodes have chosen ticket numbers and sent `request` messages to all other nodes



Scenario of an Example

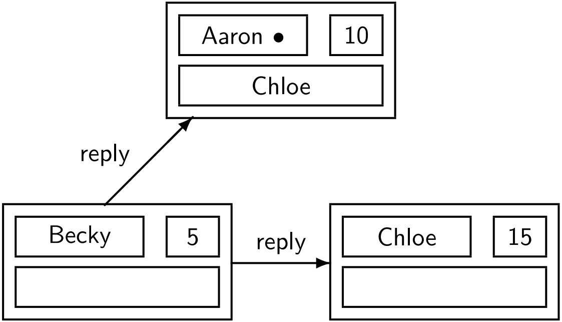
- Result of executing the **receive** process of each node



- Note the virtual queue: Becky $\leftarrow$ Aaron $\leftarrow$ Chloe
- Becky executes her CS (indicated with the bullet)

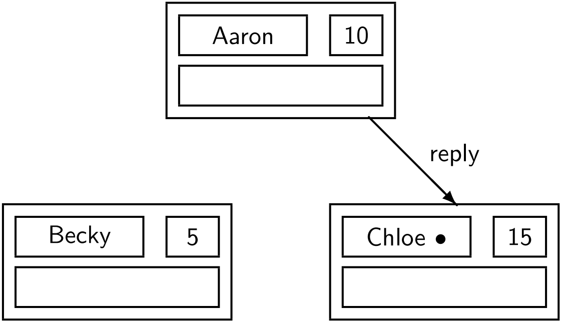
Scenario of an Example

- After completing her CS, Becky sends the two deferred reply messages to Aaron and Chloe
- Aaron now has both replies and can execute his CS



Scenario of an Example

- Upon completion Aaron sends a `reply` message to Chloe who can now enter her CS



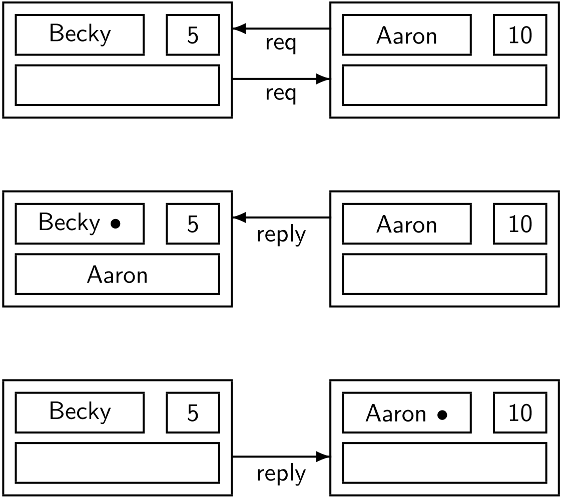
Equal Ticket Numbers

- Several nodes could choose the same number
- We break symmetry using the ID
- The **Receive** process is modified as follows:

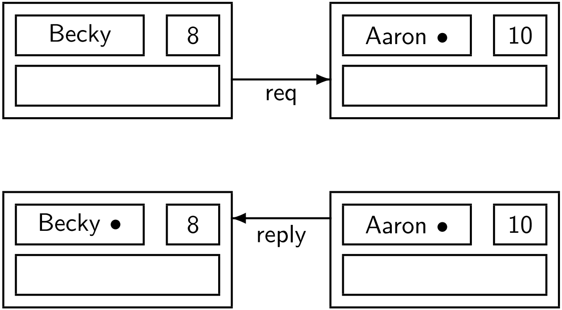
```
int source, reqNum
receive(request, source, reqNum)
if (reqNum<myNum) || ((reqNum=myNum) && (source<myId))
    send(reply,source,myID)
 else
    add source to deferred
```

Choosing Ticket Numbers

There is still a problem with our current algorithm



Choosing Ticket Numbers



Choosing Ticket Numbers

Main:

```
// non-CS
myNum = highestNum + 1
for all other nodes N
  send(request, N, myID, myNum)
await replies from all other no
// CS
for all nodes N in deferred
  remove N from deferred
  send(reply, N, myID)
```

Receive

```
int source, reqNum
receive(request, source, reqNum)
highestNum = max(highestNum,reqNum)
if reqNum < myNum
    send(reply,source,myID)
 else
    add source to deferred
```

Quiescent Nodes

- What if a node N does not want to enter its CS?
- N will never send a reply since its initial value of `myNum` is 0
- Solution:
  - We add a flag `requestCS`
  - **Main** process sets before choosing a ticket number and then resets it after exiting the CS
  - We update the **receive** accordingly

Quiescent Nodes

```
int myNum = 0
set of node IDs deferred = empty set
int highestNum = 0
boolean requestCS = false
```

Main:

```
while (true) {
  // non-CS
  requestCS := true
  myNum := highestNum + 1
  for all other nodes N
    send(request,N,myID,myNum)
  await replies from all other nod
  // CS
  requestCS := false
  for all nodes N in deferred
   remove N from deferred
   send(reply, N, myID)
}
```

Receive

```
int source, requestedNum
while (true) {
  receive(request, source, request
  highestNum := max(highestNum, re
  if not requestCS or requestedNum
    send(reply, source, myID)
   else
    add source to deferred
}
```

Additional Comments

- What happens if a node fails?
- Does performance improve if there is no contention?
- If all nodes enter their CSs once, how many total number of messages were sent?

R-A in Promela

```
init {
  atomic {
    int i;
    for (i : 0 .. NPROCS-1){
       run Main(i);
       run Receive(i);
    }
  }
}
```

Channel declaration:

```
mtype={request, reply};
chan ch[NPROCS]=[NPROCS] of {mtype, byte, byte};
```

```
proctype Main(byte myID) {
  do ::
        atomic {
          requestCS[myID] = true ;
          myNum[myID] = highestNum[myID] + 1;
        }
    int J;
        for (J : 0 .. NPROCS-1){
          if
                :: J != myID -> ch[J] ! request, myID, myNum[myID]
                :: else
          fi;
        }
    int K;
        for (K : 0 .. NPROCS-2){
          ch[myID] ?? reply, _, _;
        }
        /*  critical_section */
        requestCS[myID] = false;
        byte N;
        do
          :: empty(deferred[myID]) -> break;
          :: deferred[myID] ? N -> ch[N] ! reply, 0, 0
        od
  od
}
```

```
proctype Receive( byte myID ) {
  byte reqNum;
  byte  source;

end2:
  do ::
     ch[myID] ?? request, source, reqNum;
     highestNum[myID] = ((reqNum > highestNum[myID]) ->
            reqNum : highestNum[myID]);
     atomic {
          if
        :: requestCS[myID] &&
           ( (myNum[myID] < reqNum) ||
             ( (myNum[myID] == reqNum) &&
               (myID < source) ) )
              ->
           deferred[myID] ! source /* I have priority */
        :: else ->
           ch[source] ! reply,0,0 /* source has priority */
          fi
     }
  od
}
```