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

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)

```
1  chan request = [0] of { byte };
2
3  active proctype Server() {
4    byte client;
5  end:
6    do
7      ::  request ? client -> printf("Client %d\n", client);
8    od
9  }
10
11 active proctype Client0() {
12   request ! 0;
13 }
14
15 active proctype Client1() {
16   request ! 1;
17 }
```

- 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)

Syntactic Sugar

- The list of expressions `ch!e1,e2,...` can be written:
  `ch!e1(e2,...)`
- Primarily used when the first argument is an `mtype`, indicating the type of the message.
  - The type `mtype` can be used to give mnemonic names to values
- Example:

```
1  mtype { open, close, reset };
2  chan ch = [1] of { mtype, byte, byte };
3  byte id, n;
4  /* send statement can be in either of the formats: */
5  ch ! open, id, n;
6  ch ! open(id, n);
```

Example

```
1  mtype { red, yellow, green };
2  chan ch = [0] of { mtype, byte, bool };
3
4  active proctype Sender() {
5    ch ! red, 20, false;
6    printf("Sent message\n")
7  }
8
9  active proctype Receiver() {
10   mtype color;
11   byte time;
12   bool flash;
13   ch ? color, time, flash;
14   printf("Received message %e, %d, %d\n",color,time,flash)
15 }
```

Example: Rendezvous

```
1  chan request = [0] of { byte };
2  chan reply = [0] of { bool };
3
4  active proctype Server() {
5      byte client;
6  end:
7      do
8          ::  request ? client ->
9              printf("Client %d\n", client);
10             reply ! true
11     od
12 }
13
14 active proctype Client0() {
15     request ! 0;
16     reply ? _
17 }
18
19 active proctype Client1() {
20     request ! 1;
21     reply ? _
22 }
```

- _ is an anonymous variable

Another Example (eg4.pml)

```
1  chan request = [0] of { byte };
2  chan reply = [0] of { byte };
3
4  active [2] proctype Server() {
5      byte client;
6  end:
7      do
8      :: request ? client ->
9          printf("Client %d processed by server %d\n", client
               ↪ , _pid);
10         reply ! _pid
11     od
12 }
13
14 active [2] proctype Client() {
15     byte server;
16     request ! _pid;
17     reply ? server;
18     printf("Reply received from server %d by client %d\n",
           ↪ server, _pid)
19 }
```

- 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)

```
1  chan request = [0] of { byte };
2  chan reply = [0] of { byte, byte };
3
4  active [2] proctype Server() {
5    byte client;
6  end:
7    do
8      :: request ? client ->
9           printf("Client %d processed by server %d\n", client
                ↪ , _pid);
10          reply ! _pid, client
11   od
12 }
13
14 active [2] proctype Client() {
15   byte server;
16   byte whichClient;
17   request ! _pid;
18   reply ? server, whichClient;
19   printf("Reply received from server %d by client %d\n",
         ↪ server, _pid);
20   assert (whichClient == _pid);
21 }
```

Channels as Values

```
1  chan request = [0] of { byte, chan };
2  chan reply [2] = [0] of { byte, byte };
3
4  active [2] proctype Server() {
5    byte client;
6    chan replyChannel;
7  end:
8    do
9    :: request ? client, replyChannel ->
10       printf("Client %d processed by server %d\n",client,
             ↪ _pid);
11       replyChannel ! _pid, client;
12   od
13 }
14
15 active [2] proctype Client() {
16    byte server;
17    byte whichClient;
18    request ! _pid, reply[_pid-2];
19    reply[_pid-2] ? server, whichClient;
20    printf("Reply received from server %d by client %d\n",
          ↪ server, _pid);
21    assert(whichClient == _pid)
22 }
```

- `chan reply [2]` declares an array of 2 channels

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

```
1  chan request = [0] of { byte, chan };
2  chan reply [2] = [0] of { byte, byte };
3
4  proctype Server(byte me) {
5      byte client;
6      chan replyChannel;
7  end:
8      do
9      ::  request ? client, replyChannel ->
10             printf("Client %d processed by server %d\n",
                   ↪ client, me);
11             replyChannel ! me, client;
12     od
13 }
14
15 proctype Client(byte me; byte myChannel) {
16     byte server;
17     byte whichClient;
18     request ! me, reply[myChannel];
19     reply[myChannel] ? server, whichClient;
20     printf("Reply received from server %d by client %d\n",
           ↪ server, me);
21     assert (whichClient == me);
22 }
23 /* continued */
```

Previous Example Improved

```
1  init {
2      atomic {
3          run Server('s');
4          run Server('t');
5          run Client('c', 0);
6          run Client('d', 1);
7      }
8  }
```

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

```
1  proctype Client(byte me; byte myChannel) {
2    byte server;
3    request ! me, reply[myChannel];
4    numClients++;
5    assert (numClients <= 2);
6    numClients--;
7    reply[me] ? server;
8    printf("Client %d received reply on channel %d, processed
         ↪ by server %%d\n", me, reply[myChannel], server)
9  }
10
11 init {
12   atomic {
13       run Server('s');
14       run Server('t');
15       run Client('c', 0);
16       run Client('d', 1);
17       run Client('e', 2);
18   }
19 }
```

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

```
1  chan request = [2] of { byte, chan};
2  chan reply[3] = [2] of { byte };
3
4  active [2] proctype Server() {
5    byte client;
6    chan replyChannel;
7    do
8    :: empty(request) ->
9         printf("No requests for server %d\n", _pid)
10   :: request ? client, replyChannel ->
11        printf("Client %d processed by server %d\n", client,
              ↪ _pid);
12        replyChannel ! _pid
13   od
14 }
15 active [3] proctype Client() {
16   byte server;
17   do
18   :: full(request) ->
19        printf("Client %d waiting for non-full channel\n",
              ↪ _pid)
20   :: request ! _pid, reply[_pid-2] ->
21        reply[_pid-2] ? server;
22        printf("Reply received from server %d by client %d\n"
              ↪ , server, _pid)
23   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

```
1  chan request = [4] of { byte };
2  chan reply = [4] of { byte, byte };
3
4  active [2] proctype Server() {
5    byte client;
6  end:
7    do
8    :: request ? client ->
9       printf("Client %d processed by server %d\n", client,
            ↪ _pid);
10      reply ! _pid, client
11   od
12 }
13
14 active [4] proctype Client() {
15   byte server;
16   request ! _pid;
17   reply ?? server, eval(_pid);
18   printf("Reply received from server %d by client %d\n",
         ↪ server, _pid)
19 }
```

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

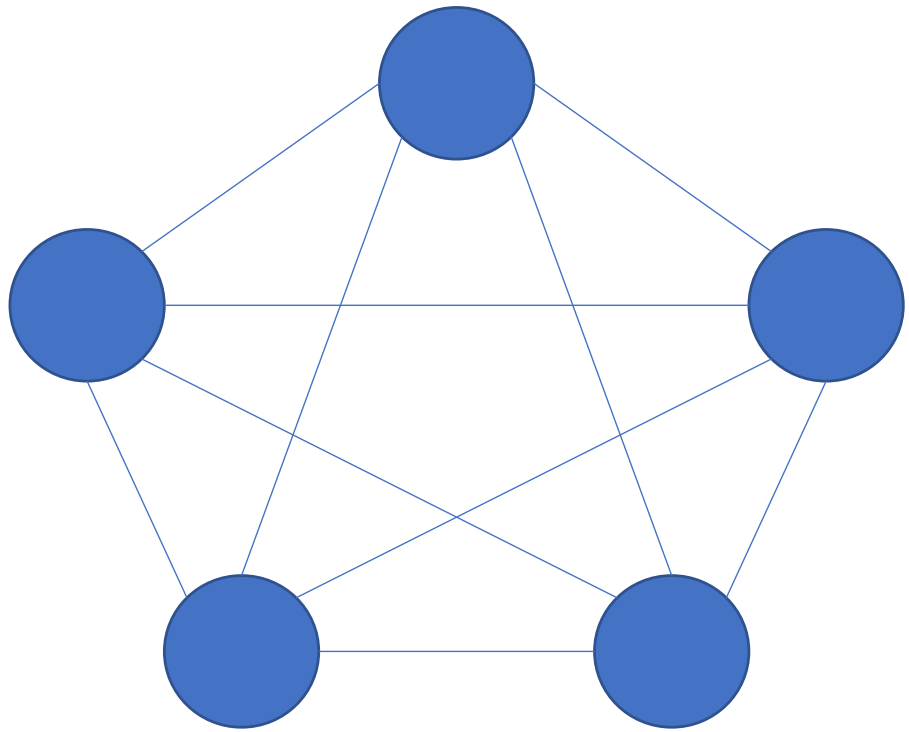
Distributed Mutual Exclusion

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] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. Comm. ACM, 24(1):9–17, 1981. Corr. ibid. 1981, p.581.

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

- main
- receive

Preliminary Version

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

Main:

```
1  /* non-critical section */
2  myNum := chooseNumber
3  for all other nodes N
4   send(request,N,myID,myNum)
5  await reply's from all
       ↪ other nodes
6  /* critical section */
7  for all nodes N in deferred
8    remove N from deferred
9    send(reply,N,myID)
```

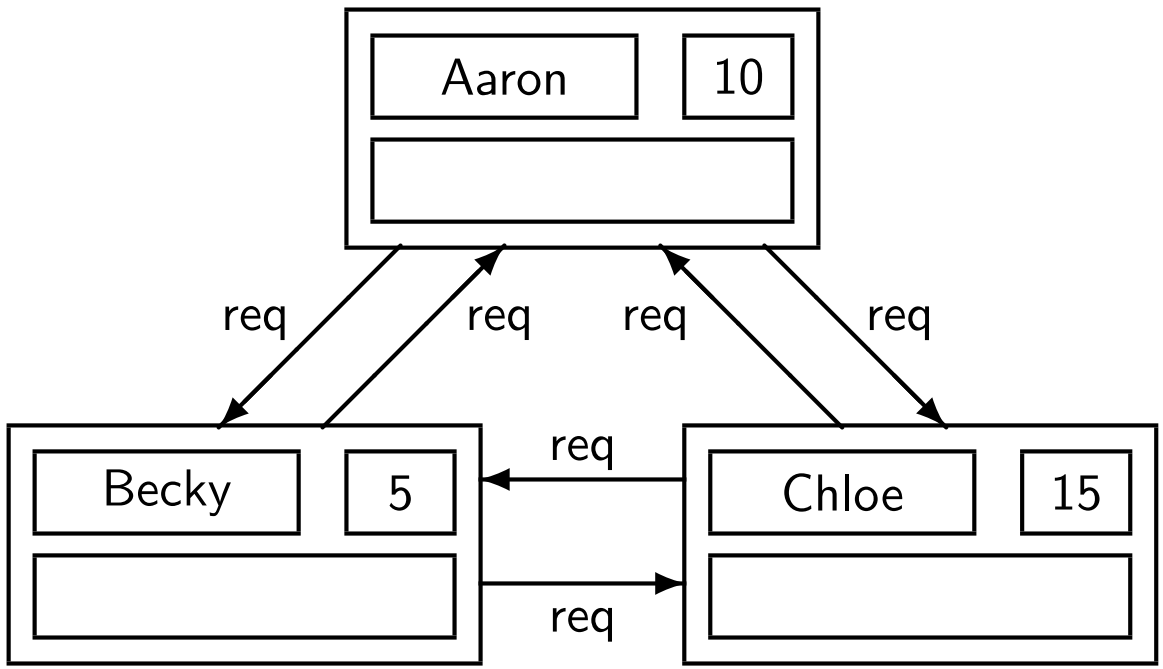
Receive

```
1  integer source, reqNum
2  receive(request,source,reqNum)
3  if reqNum < myNum
4      send(reply,source,myID)
5   else
6      add source to deferred
```

Note: Both snippets are assumed to execute inside an infinite loop (not shown)

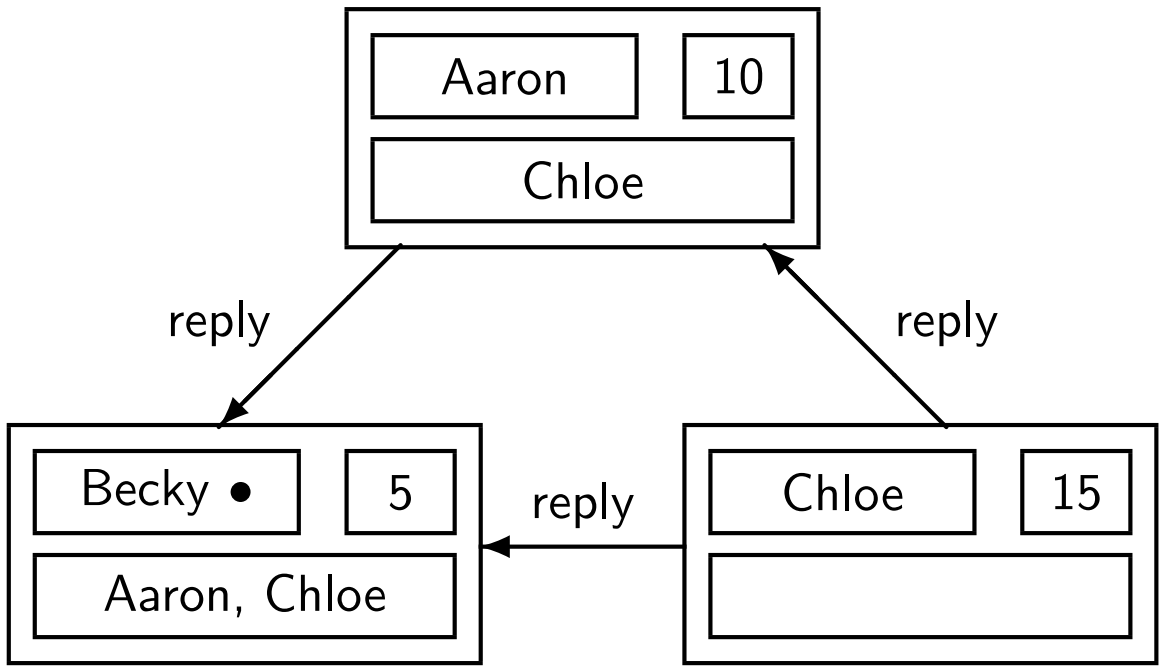
Scenario of an Example

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



Scenario of an Example

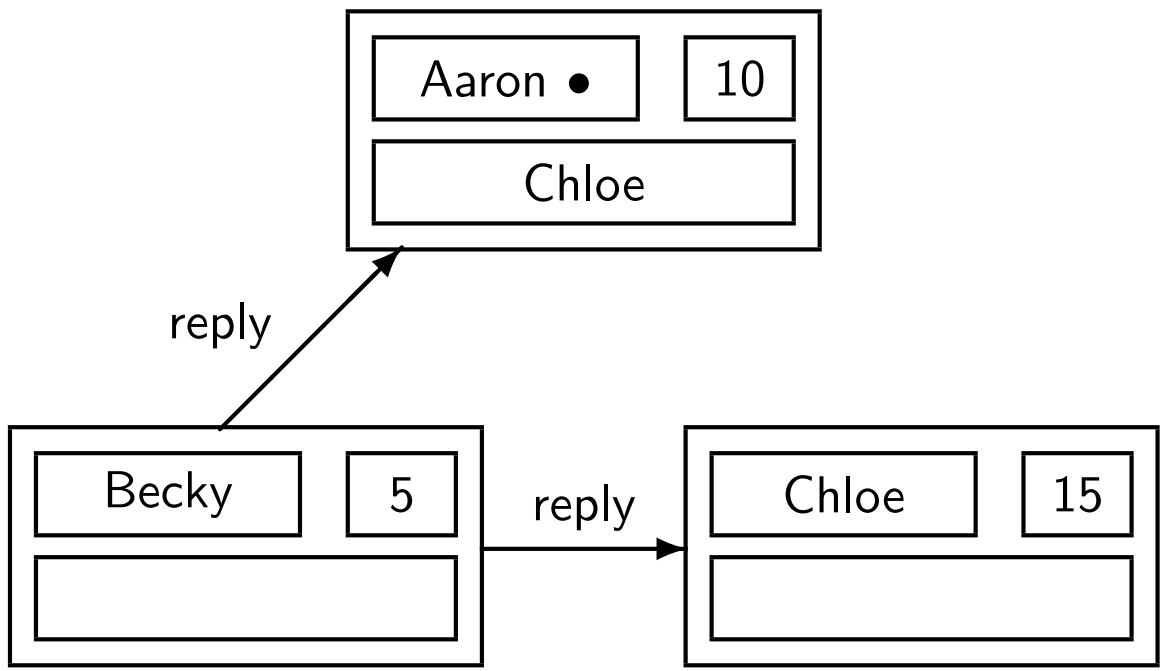
- Result of executing the **receive** process of each node



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

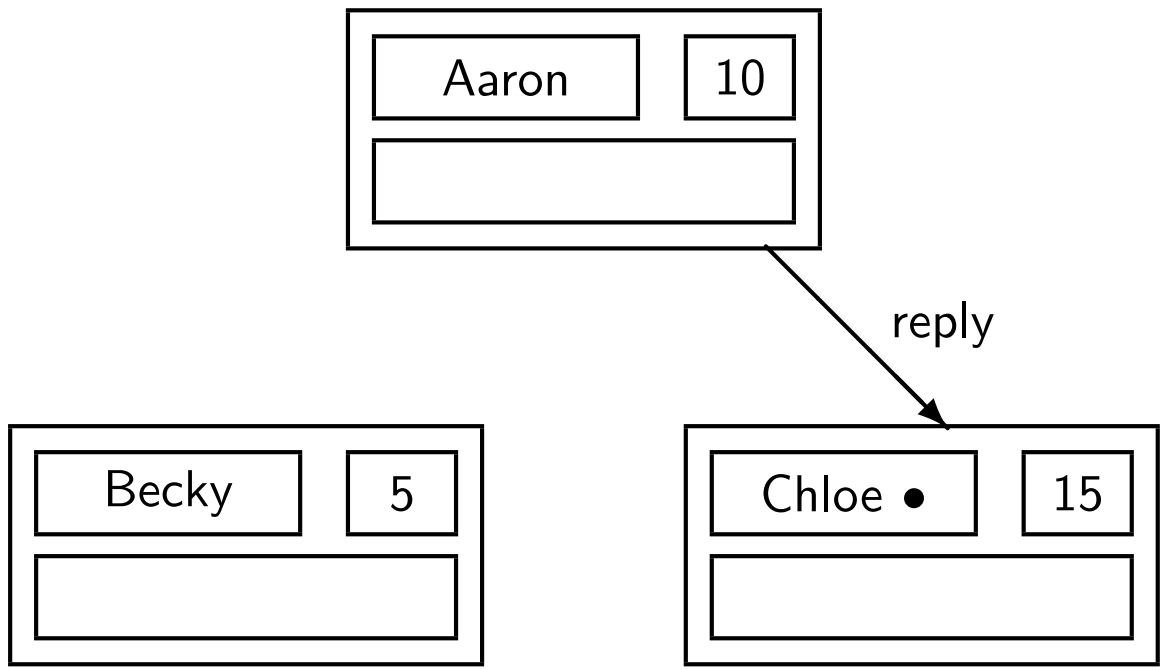
Scenario of an Example

- 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



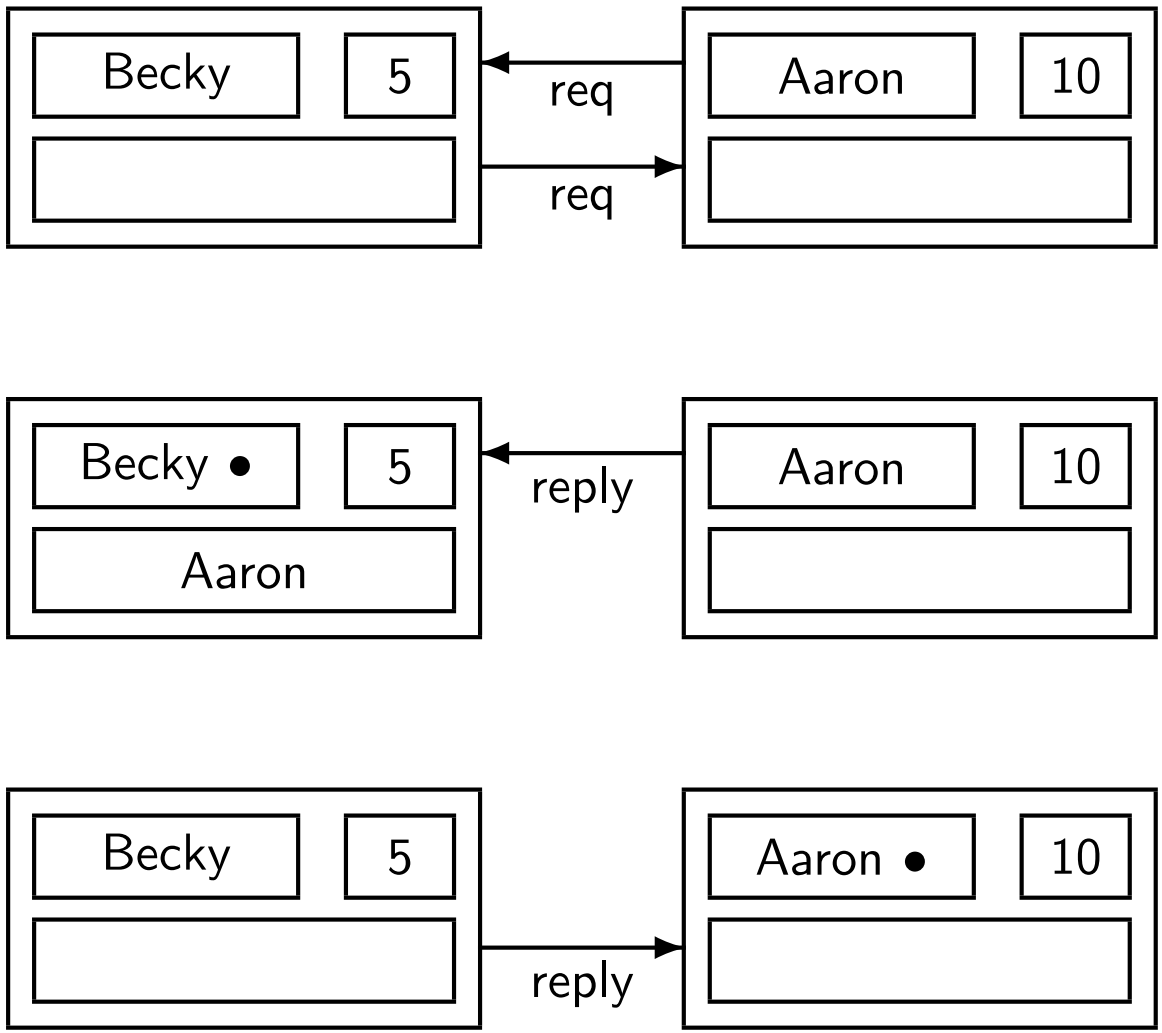
Equal Ticket Numbers

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

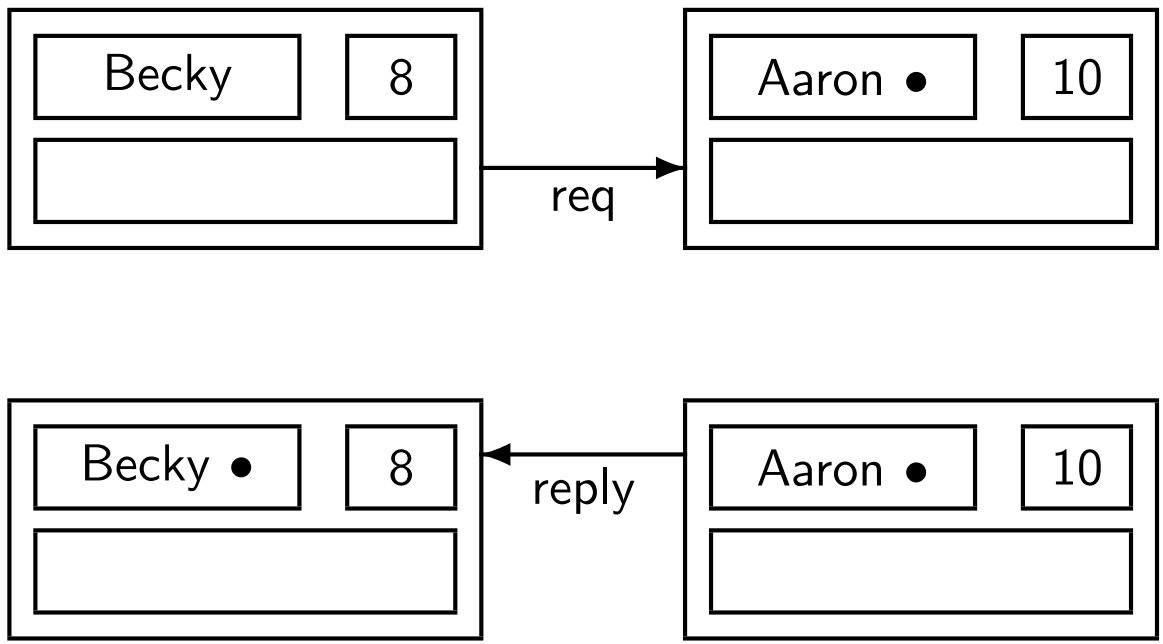
```
1  integer source, reqNum
2  receive(request, source, reqNum)
3  if (reqNum<myNum) || ((reqNum=myNum) && (source<myId))
4      send(reply,source,myID)
5   else
6      add source to deferred
```

Choosing Ticket Numbers

There is still a problem with our current algorithm



Choosing Ticket Numbers



Choosing Ticket Numbers

Main:

```
1  /* non-critical section */
2  myNum := highestNum + 1
3  for all other nodes N
4    send(request, N, myID,
         ↪ myNum)
5  await reply's from all
       ↪ other nodes
6  /* critical section */
7  for all nodes N in deferred
8    remove N from deferred
9    send(reply, N, myID)
```

Receive

```
1  integer source, reqNum
2  receive(request, source, reqNum)
3  highestNum := max(highestNum,
       ↪ reqNum)
4  if reqNum < myNum
5      send(reply,source,myID)
6   else
7      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

```
1  integer myNum := 0
2  set of node IDs deferred := empty set
3  integer highestNum := 0
4  boolean requestCS := false
```

Main:

```
1  /* non-critical section */
2  requestCS := true
3  myNum := highestNum + 1
4  for all other nodes N
5    send(request,N,myID,myNum
       ↪ )
6  await reply's from all
       ↪ other nodes
7  /* critical section */
8  requestCS := false
9  for all nodes N in deferred
10   remove N from deferred
11   send(reply, N, myID)
```

Receive

```
1  integer source, requestedNum
2  receive(request, source,
       ↪ requestedNum)
3  highestNum := max(highestNum,
       ↪ requestedNum)
4  if not requestCS or requestedNum
       ↪ << myNum
5    send(reply, source, myID)
6    else
7    add source to deferred
```

Additional Comments

- What happen 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

```
1  init {
2    atomic {
3      int i;
4          for (i : 0 .. NPROCS-1){
5              run Main(i);
6              run Receive(i);
7          }
8    }
9  }
```

```
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
            :: else ->
           ch[source] ! reply, 0, 0
      fi
        }
  od
}
```

Ricart-Agrawala and Spin

Prove, using spin, that the Ricart-Agrawala algorithm is correct in the following sense:

- It does not deadlock
- It enjoys mutual exclusion
- If a process goes into its CS, then its number is the least one

Appendix

More Details on Promela Syntax

Promela Summary

FEATURE	C	PROMELA
integers	char, short, int, long	byte, short, int
bit field	unsigned	unsigned
floats	float, double	NONE
boolean	int	bool
strings	char, char*	NONE
arrays	yes	1D & limited
operators	many	mostly same
if	as usual	similar to Erlang
loops	while, for, do	do, similar to if
output	printf	printf
input	scanf	NONE
functions	yes	NO
pointers	yes	NO
enum	enum	mtype
comments	/* */ and //	/* */
cpp	full	1-line #define, #include

If Syntax

- "Guarded commands" wrapped inside "`if ... fi`"

```
1  disc = b*b - 4*a*c;
2  if
3  :: disc > 0 ->
4     printf("two real roots\n")
5  :: disc < 0 ->
6     printf("no real roots\n")
7  :: disc == 0 ->
8     printf("duplicate real roots\n")
9  fi
```

If Semantics

- First: evaluate all guards
- Then:
  - If no guard true: statement blocks until at least one guard becomes true (which could happen due to action of some concurrent process)
  - If one guard true: execute its command(s)
  - If more than one guard true: execute command(s) of randomly chosen guard

Else

- Guard consisting of "`else`" keyword is true if all other guards are blocked
- Example:

```
1  disc = b*b - 4*a*c;
2  if
3  :: else ->
4     printf("two real roots\n")
5  :: disc < 0 ->
6     printf("no real roots\n")
7  :: disc == 0 ->
8     printf("duplicate real roots\n")
9  fi
```

Do Syntax

- Similar to `if` statement
- Example: compute GCD by repeated subtraction

```
1  /* assume x and y are initialized */
2  int a = x, b = y;
3  do :: a > b -> a = a - b
4     :: b > a -> b = b - a
5     :: a == b -> break
6  od
7  printf("GCD(%d, %d) = %d\n", x, y, a);
```

- Notes:
  - No loop test; only way out is via `break`
  - Body consists of guarded commands
  - Some true guard is chosen at random
  - Block if no true guard

Do Semantics

- Promela has no other type of loop
- Most common loop has only 2 guarded commands:

```
do
    :: [exit test] -> break
    :: else -> [body statements]
od
```

- This structure provides deterministic operation like:

```
while (not [exit test])
    { [body statements] }
```

Another Example

```
1  proctype P() {
2      int x = 15, y = 20;
3      int a = x, b = y;
4
5      do
6      :: a > b   ->   a = a - b
7      :: b > a   ->   b = b - a
8      :: a == b  ->   break
9      od
10     printf("GCD(%d, %d) = %d\n", x, y, a);
11 }
```

Note:

- `proctype P()` declares no-argument program P
- Can include arguments:

```
1      proctype P(int x, int y) {
2        int a = x, b = y;
3        etc.  }
```

Exercise: Write Dekker's Algorithm in Promela

```
1  global int turn = 1;
2  global boolean wantP = false;
3  global boolean wantQ = false;
```

```
1  thread P: {
2    while (true) {
3     // non-critical section
4     wantP = true;
5     while wantQ
6       if (turn == 2) {
7         wantP = false;
8         await (turn==1);
9         wantP = true;
10      }
11    // CRITICAL SECTION
12    turn = 2;
13    wantP = false;
14    // non-critical section
15   }
16 }
```

```
1  thread Q: {
2    while (true) {
3     // non-critical section
4     wantQ = true;
5     while wantP
6       if (turn == 1) {
7         wantQ = false;
8         await (turn==2);
9         wantQ = true;
10      }
11    // CRITICAL SECTION
12    turn = 1;
13    wantQ = false;
14    // non-critical section
15   }
16 }
```

Right to insist on entering is passed between the two processes

Exercise: Write Dekker's Algorithm in Promela

```
1  bool     wantp = false, wantq = false;
2  byte     turn = 1;
3
4  active proctype P() {
5      do
6      ::   wantp = true;
7           do
8           :: !wantq -> break;
9           :: else ->
10              if
11              :: (turn == 1)     /* no statements: leaves the
                    ↪ if */
12              :: (turn == 2) ->
13                  wantp = false;
14                  (turn == 1);
15                  wantp = true
16              fi
17          od;
18          wantp = false;
19          turn = 2
20     od
21   }
```

Spawning a Process

- Can start processes using `run` operator: `run P(15, 20)`
- Also, can declare process with `active proctype`
  - Adding "`active`" means "define and run this program"
- To start two processes executing same code, use:

  ```
  active [2] proctype P(int x, int y)
  ```
- Can create an initial process that runs before any of the "`proctype`" processes
  - This process must be named `init`

Predefined Variables

- `_pid` is process ID
- `_nr_pr` is number of active processes
- Examples:

```
1  printf("process %d: n goes from %d to %d\n", _pid, temp, n);
2
3  if
4  :: _nr_pr == 1 -> printf("at end n = %d\n", n);
5  fi
```

Blocking Statements, I

- Concurrent programs must often wait for some event
- Possible to guard any statement
- This:

```
1  _nr_pr == 1 -> printf("at end n = %d\n", n)
```

is the same as:

```
1  if
2  :: _nr_pr == 1 -> printf("at end n = %d\n", n)
3  fi
```

Blocking Statements, II

- "->" arrow is just syntactic sugar
- Can write expression by itself; if it doesn't evaluate to non-zero then program will block
- This:

```
1  _nr_pr == 1;
2  printf("at end n = %d\n", n)
```

is the same as:

```
1  _nr_pr == 1 -> printf("at end n = %d\n", n)
```

Atomicity, I

- Individual Promela statements are atomic
- Warning! In Promela, expressions are statements too (hence expressions are atomic)
- Example – here, division by zero is possible:

```
1  if
2      :: a != 0 -> c = b / a ;
3      :: else   -> c = b
4  fi
```

- In an if (and do) statement, interleaving may occur between the evaluation of the guard and the execution of the statement after the guard

Atomicity, II

- It is tempting to regard the entirety of `a != 0 -> c = b / a` as atomic
- But it consists of two atomic parts, `a != 0` and `c = b / a`
- Remember that this:

```
1       a != 0 -> c = b / a
```

  could be written as:

```
1  a != 0; /* may block */
2  c = b / a
```

- The latter more obviously contains two atomic parts

Atomicity, III

- To group statements together atomically use `atomic`

```
1  atomic {
2      a != 0;      /* may block */
3      c = b / a
4  }
```

- If any statement within the atomic sequence blocks, atomicity is lost, and other processes may start executing statements.
- When the blocked statement becomes executable again, the execution of the atomic sequence can be resumed at any time (but it has to compete with other active processes)

Atomic & Run

- `run` only starts a concurrent process
- `atomic` prevents execution of any other actions besides those in its body
- Therefore, to start a group of processes that should run concurrently:

```
1    atomic {
2        run P1(...);
3        run P2(...);
4            ...
5        run PN(...)
6    }
```

- At conclusion of `atomic` block: all processes have been started but none is yet running

Variable Size

- Use smallest integer variable that will fit the need
- E.g., for integers known to be small use "`byte`" (8 bits) instead of "`int`" (32 bits)
- Reason: "verification" simulates all possible values of variable