Cockrell School of Engineering

Verifying Distributed Algorithms
in Promela

Eric Crosson
Nhan Do
Stormy Mauldin
Daniel Officewala
EE 360P
June 9, 2016

Outline

Promela Overview

Dining Philosophers

Token Ring

Chandy and Lamport

Szymanski's

Questions

Promela Overview

- Promela is **Pr**ocess **Me**ta **La**nguage
- Spin compiles Promela into C
- C executables assert system invariants at each simulated state

Dining Philosophers

Most Philosophers

- ► Want to eat:
    - – Get left fork (or wait)
    - – Get right fork (or wait)
- ► After eating:
    - – Release both forks
    - – Contemplate life until hungry

One "Special" Philosopher

- ► Want to eat:
    - – Get left fork (or wait)
    - – Get right fork (if fail, release *both* forks)
- ► After eating:
    - – Release both forks
    - – Contemplate life until hungry

Dining Philosopher Analysis

- **Mutually Exclusive**
  - A philospher must acquire both shared forks before eating
- **Deadlock**
  - The one special philosopher will always surrender both forks, allowing someone else to start eating.
- **Starvation**
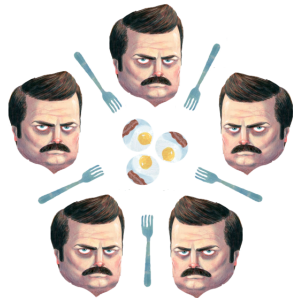 - The special philosopher always surrenders forks in case of conflict. May never get a change to eat.

Figure: Hungry, hungry philosophers

Token Ring Algorithm

- ▶ Simple and easily scalable
  - – Pass token around ring of processes
  - – Only processes with token can enter CS
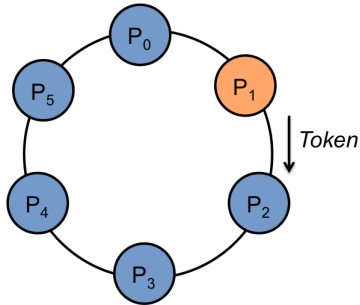 - – No Starvation

Figure: Token Ring Algorithm

Token Ring Implementation

```
bit _Permission[N];                proctype P(byte id) {                      true -> in_cs ++; }
bit _Executing[N];                 NonCritical :                       fi ;
byte in_cs ;                         _Permission[id] = true ;          Critical :
byte token ;                                                              atomic { in_cs --; }
                                   Wait :                                  _Permission[id] = false ;
init {                               _Executing[id] = true ;               _Executing[id] = false ;
  atomic {                           if                                    if
    byte i = 0;                      :: id != token ->                  :: token < N ->
       token = 0;                        _Permission[id] = false ;         token = (( token + 1) % N);
    do                                   goto Wait ;                       :: atomic{token > (N-1) ->
    :: i < N -> run P( i ); i ++;    :: id == token                        token = 0;   }
    :: else   -> break ;             fi ;                               fi ;
    od ;                             if                                     goto NonCritical ;
  }                                  :: _Permission[id] == false ->
}                                        goto NonCritical ;
                                     :: atomic { _Permission[id] ==
```

Chandy and Lamport

- ▶ Gurarantees consistent global snapshots
  - – Happened-before model
  - – Uses markers
  - – Promela model verification based on markers

Chandy Implementation

```promela
mtype { message, marker };
chan ch = [N] of
       { mtype, byte };

active proctype Sender() {
  do
  :: lastSent < NUM_MESSAGES ->
       lastSent++;
       ch ! message(lastSent)
  :: ch ! marker(o) ->
       break
  od }
```

```promela
active proctype Receiver() {
  byte received;
  do
  :: ch ? message(received) ->
       lastReceived = received
  :: ch ? marker(_) ->
       messageAtMarker =
          lastReceived;
       if
       :: !recorded ->
            messageAtRecord =
              lastReceived
```

```promela
       :: else
       fi;
       break
  :: !recorded ->
       messageAtRecord =
          lastReceived;
       recorded = true
  od
}
```

Szymanski's Algorithm

- ► Extension of Lamport's
  - – satisfies linear wait
  - – three booleans per process
- ► Extension of Lamport's

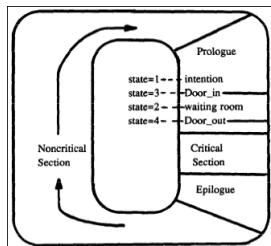
Figure: Szymanski's Algorithm

Coding of the flag values			
flag	intent	door_in	door_out
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	1	1	1

Figure: State-tracking booleans

Szymanski's Implementation

```
start:                          anteroom_check:                      /* Proceed into CS when
    /* 1. SEKCJA LOKALNA */        if                                 * it is your turn */
                                     :: (count(1,0,0) +              door_out[i] = true;
    local_section();                     count(1,0,1) > 0) —>
                                       {                             wait_forall(k, i + 1, N,
    /* 2. PROLOG */                      /* State 2 */                (!door_in[k] || door_out[k]));
    intent[i] = true;                    intent[i] = false;
                                                                     wait_forall(k, 0, i,
started_protocol:                      in_anteroom:                    (!door_in[k]));
    skip;                                ((count(0,0,1) +
                                           count(0,1,1) +            critical_section:
    /* 3. Others are trying to             count(1,0,1) +               /* SEKCJA KRYTYCZNA */
     * enter waiting room? */              count(1,1,1) > 0)
    (count(1,1,0) +                       );                            critical_section();
      count(1,1,1) == 0);
                                         /* State 3 */                /* EPILOG */
    /* 4. Enter waiting room */          intent[i] = true;            door_out[i] = false;
    door_in[i] = true;               }                               door_in[i] = false;
                                     :: else                         intent[i] = false;
                                   fi;
```

Questions

Thank you for your time.