Verification of Randomized Dining Philosophers

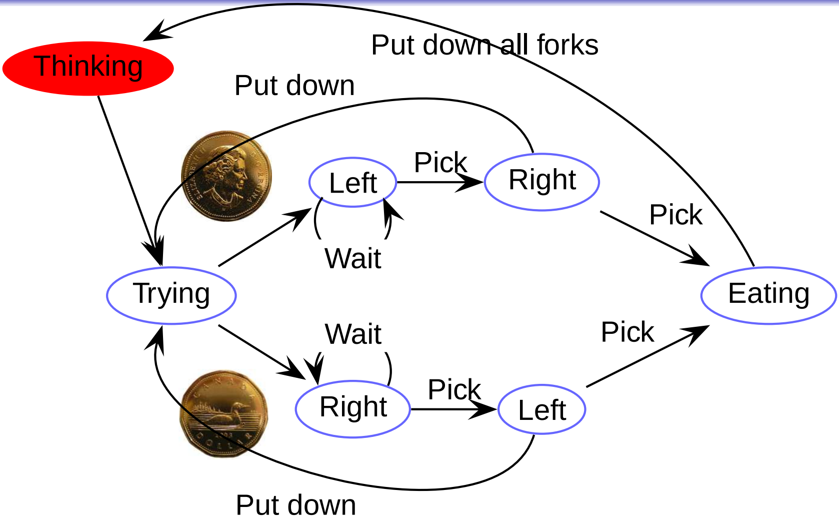
Presented by Xin Zhang

YORK U UNIVERSITÉ UNIVERSITY

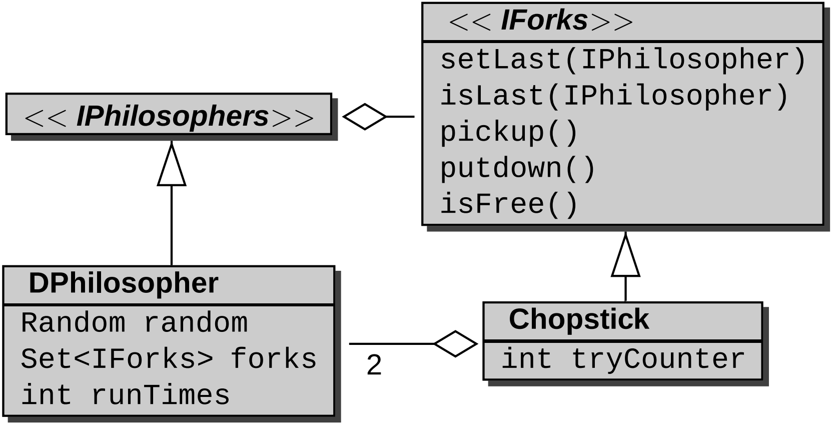
Recall

- We use randomized algorithm to break the initial symmetry
- Use different mechanism to guarantee mutual exclusion and deadlock free
- With extra information, we can achieve liveness property-starvation free

Control Flow Graph



Structure


<< IPhilosophers>>
<< IForks>>
setLast(IPhilosopher)
isLast(IPhilosopher)
pickup()
putdown()
isFree()
DPhilosopher
Random random
Set<IForks> forks
int runTimes
2
Chopstick
int tryCounter

Extra Setting in JPF

- We have randomized algorithms, and let JPF enumerate all randomized choice
  cg.enumerate_random=true
- For asynchronized algorithm, we do not need to verify
  vm.por.field_boundaries.never =
  ,messages.MessageQueue,messages.Message

Chopstick Assertion of Mutual Exclusion

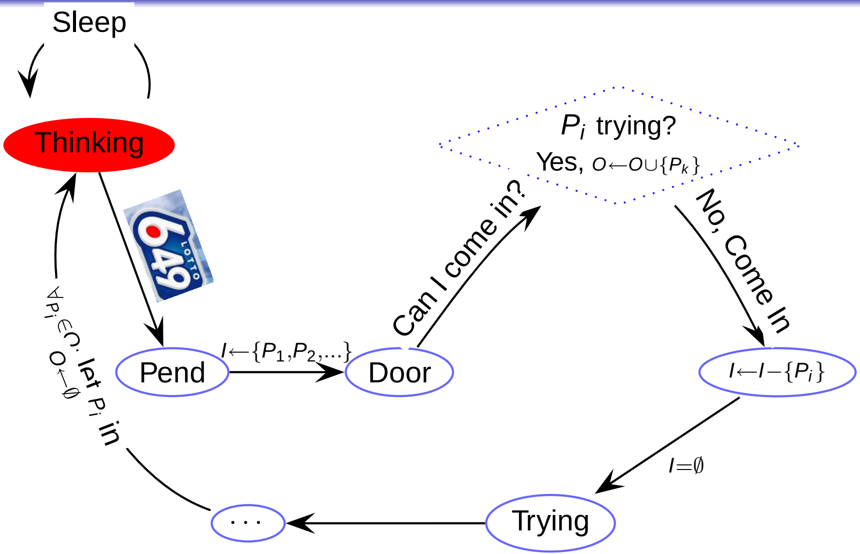
```
public void pickup()
    {
        state = INUSE;
        continousTries = 0;
        usageCount ++;
        assert usageCount == 1;
    }

public void putdown()
    {
        state = FREE;
        usageCount --;
        aassert usageCount == 0;
    }
```

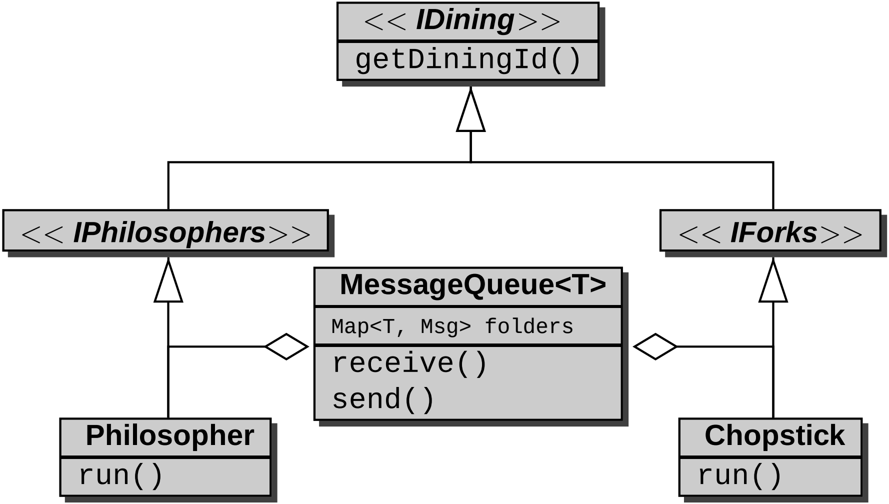
JPF: No Error

Type	Threads	Cycles	Time (Min)	Memory (MB)	States (backtrack)
DeadLock	2	5	1.5	236	548,760
DataRace	2	5	1.5	156	548,760
DataRace	2	7	4.5	642	1,347,985
DataRace	2	10	33	900	13,278,200
DataRace	3	1	160	1026	66,104,170
DeadLock	3	5	881	2482	288,508,129

Control Flow


Sleep
Thinking
LOTTO 649
Pend
$I \leftarrow \{P_1, P_2, \ldots\}$
Door
Can I come in?
P_i trying?
Yes, $O \leftarrow O \cup \{P_k\}$
No, Come In
$I \leftarrow I - \{P_i\}$
$I = \emptyset$
Trying
$\cdots$
$\forall P_i \in O$. let P_i in
$O \leftarrow \emptyset$

Structure


<< IDining>>
getDiningId()
<< IPhilosophers>>
<< IForks>>
MessageQueue<T>
Map<T, Msg> folders
receive()
send()
Philosopher
run()
Chopstick
run()

Assertion of mutual exclusion

```
public void Chopstick.run(){
    while(true){
        IMessage<IDining> msg = mQueue.
            receiveBlocked(this);
        ......
        else if (type == TYPE_REQUEST){
            if (state){
                state = INUSE;
                usageCounter++;
                assert usageCount == 1;} }
        else if(type == TYPE_FAIL || type ==
            TYPE_RELEASE){
            state = FREE;
            usageCounter--;
            assert usageCount == 0;

        }
```

JPF Result

Threads	Life Cycles	Time (Hours)	Memory (MB)	States (backtrack)
2	1	21:40	2497	331,691,872
2	3	20:45	2499	335,012,426

Not finished, no error

Improvement of the Program

- Store all used message in a hashmap
- When needed, we retrieve message from the hashmap, instead of creating new message every time
- Hope to reduce the state space and interleaving

Results

Not success, though No error so far

Threads	Life Cycles	Time (Min)	Memory (MB)	States (backtrack)
2	1	2112	2498	336,460,438
2	3	2145	2497	368,723,606

Conclusion

- It is difficult to verify a concurrent program
- The number of threads cause the state space exploited
- On the other hand, the number of loops increases the state space in a relatively slow speed