Propriété de sûreté et vivacité
Safety and liveness
Part 1 - Deadlock

riveill@unice.fr
http://www.i3s.unice.fr/~riveill

Evitement et autres approches

Evitement

- Les processus sont numérotés selon un ordre croissant à l'aide d'une estampille
- Plusieurs types d'algorithmes existent. En voici un.

```
if (Stamp(P1)< Stamp(P2) {
    /* Le processus demandeur est plus ancien
       que celui qui possède la ressource.
       Il est mis en attente */
    Wait(P1);
} else {
    /* le processus demandeur est plus jeune
       que celui qui possède la ressouce.
       Il est arrêté. */
    Stop(P1);
}
```

→ Un processus n'attendra jamais sur un processus plus vieux : pas de cycle, pas d'interblocage

Autres solutions : Time Out

- A chaque processus est associé un temps d'exécution
- Le dépassement de celui-ci est interprété comme une situation d'interblocage
- Le processus est alors arrêté, ses ressources réquisitionnées
- Intérêt : supprime la manipulation du graphe et les surcoûts induits
- Inconvénients : un processus peut être arrêté même s'il n'y a pas interblocage.
  - Un timeout trop court peut conduire à une privation
  - Un timeout trop long dégrade les performances du système
  - Et surtout, il faut savoir arrêter un processus

deadlock analysis - avoidance

```
// Ajout d'un 'time out'
RESOURCE = (get-                                    =
(printer.get-
GETSCANNE                                        nter.put
                                                 ner.put->P
                                                 >P

Q
GETPRINTER                                       put
                                                 ut->Q


||SYS = (p:P||
      ||{p,q}::p                                 RCE
     ||{p,q}::sca                                ESOURCE
     ).
```

```
Compiled: P
Compiled: Q
Compiled: RESOURCE
Composition:
SYS = p:P || q:Q ||
{p,q}::printer:RESOURCE ||
{p,q}::scanner:RESOURCE
State Space:
 6 * 6 * 2 * 2 = 2 ** 8
Composing...
-- States: 17 Transitions: 31
Memory used: 3030K
Composed in 32ms

No deadlocks/errors
```

Summary

- Concepts
  - deadlock: no futher progress
  - four necessary and sufficient conditions:
    - serially reusable resources
    - incremental acquisition
    - no preemption
    - wait-for cycle

Aim: deadlock avoidance - to design systems where deadlock cannot occur.

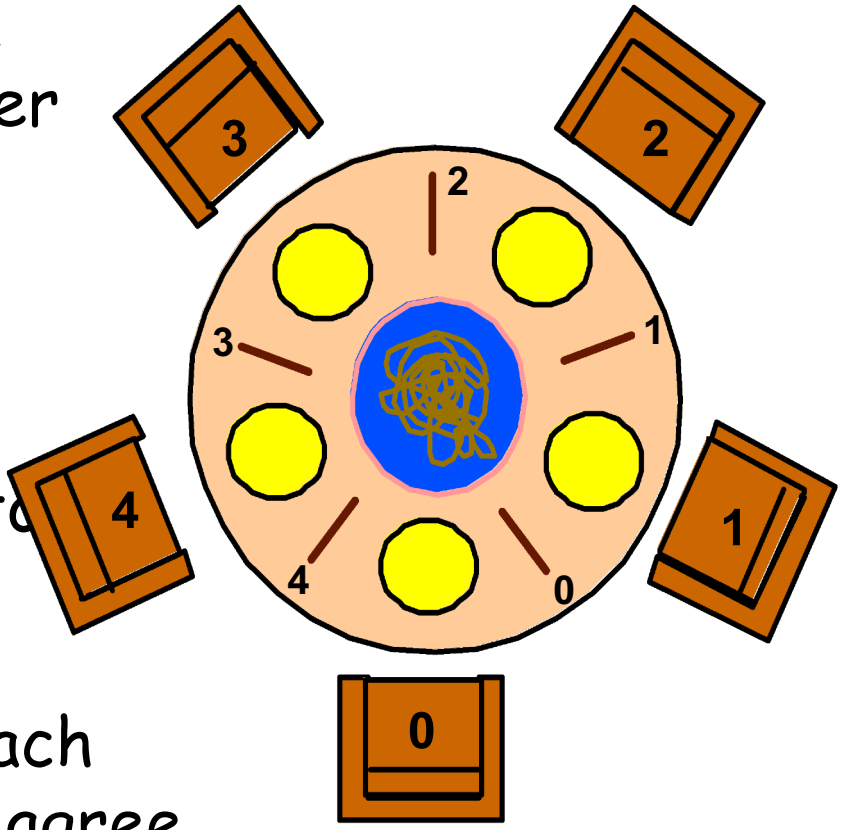
- Models
  - no eligable actions (analysis gives shortest path trace)
- Practice
  - blocked threads

Q&A

http://www.i3s.unice.fr/~riveill

6.2 Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.

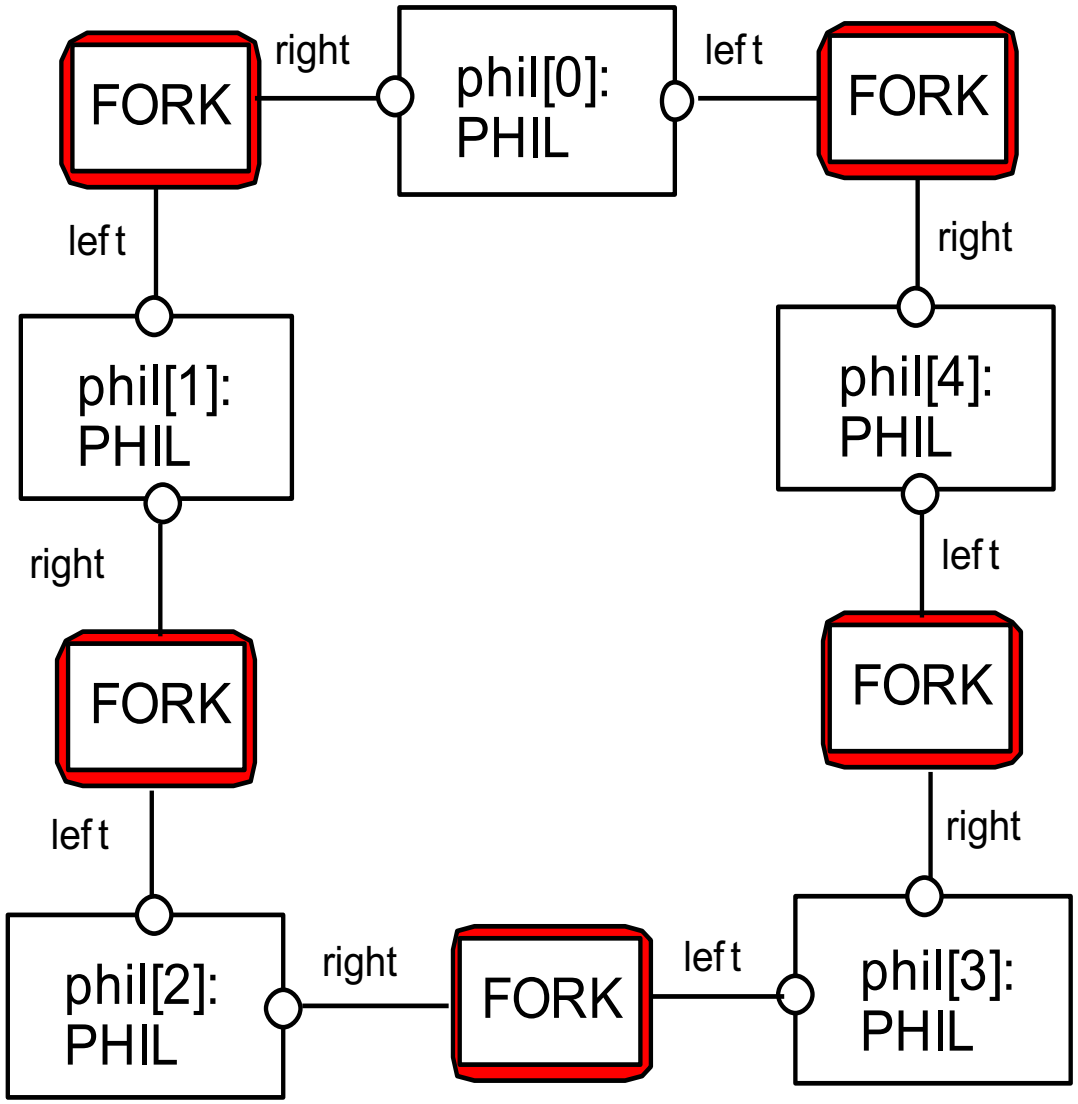


One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

Dining Philosophers - model structure diagram

Each FORK is a shared resource with actions **get** and **put**.

When hungry, each PHIL must first get his right and left forks before he can start eating.



Dining Philosophers - model

```
FORK = (get -> put -> FORK).
PHIL = (sitdown      ->right.get->left.get
                ->eat        ->right.put->left.put
                ->arise->PHIL).
```

Table of philosophers:

```
||DINERS(N=5)= forall [i:0..N-1]
       (phil[i]:PHIL ||
       {phil[i].left,phil[((i-1)+N)%N].right}::FORK
       ).
```

Can this system deadlock?

Dining Philosophers - model analysis

```
Trace to DEADLOCK:
    phil.0.sitdown
    phil.0.right.get
    phil.1.sitdown
    phil.1.right.get
    phil.2.sitdown
    phil.2.right.get
    phil.3.sitdown
    phil.3.right.get
    phil.4.sitdown
    phil.4.right.get
```

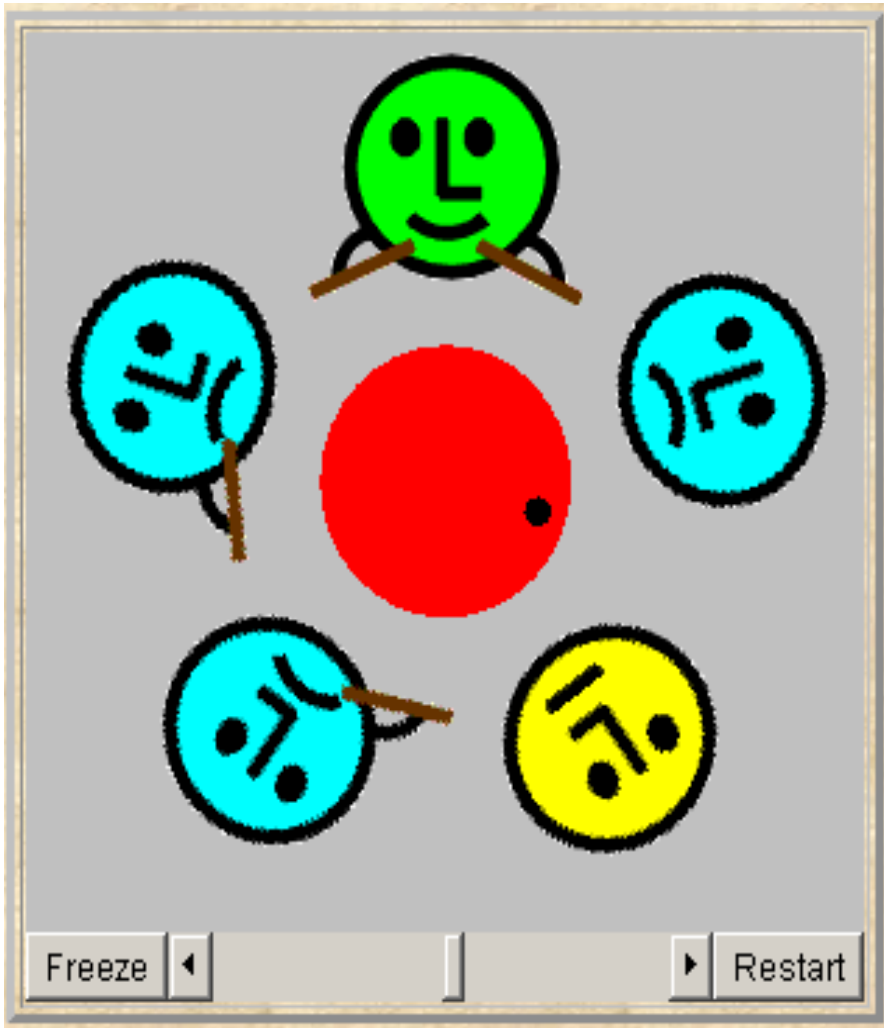
This is the situation where all the philosophers become hungry at the same time, sit down at the table and each philosopher picks up the fork to his right.

The system can make no further progress since each philosopher is waiting for a fork held by his neighbor i.e. a ***wait-for cycle*** exists!

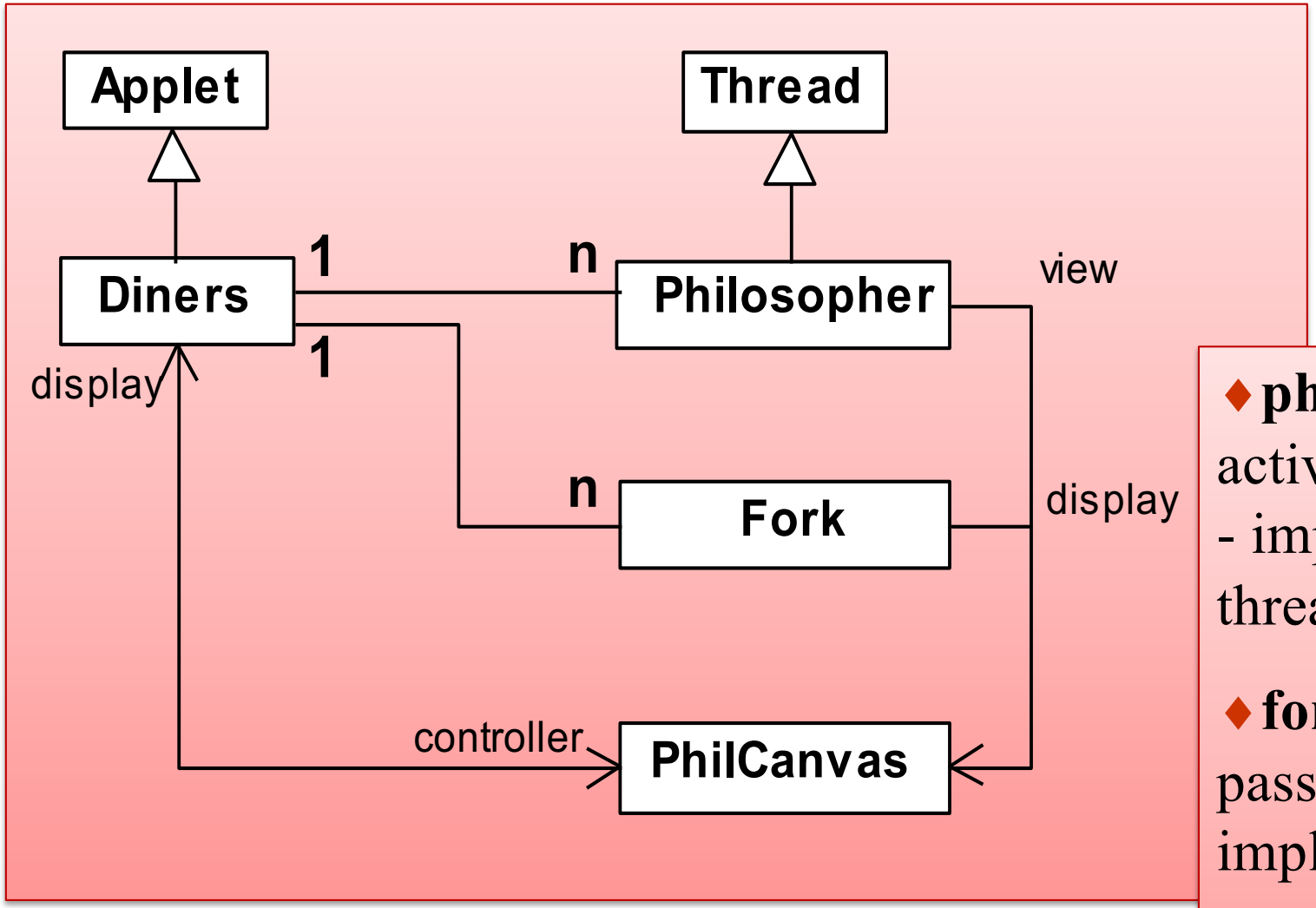
Dining Philosophers

Deadlock is easily detected in our model.

How easy is it to detect a potential deadlock in an implementation?



Dining Philosophers - implementation in Java



- **philosophers**: active entities - implement as threads
- **forks**: shared passive entities - implement as monitors
- **display**

Dining Philosophers - Fork monitor

```
class Fork {
  private boolean taken=false;
  private PhilCanvas display;
  private int identity;

  Fork(PhilCanvas disp, int id)
    { display = disp; identity = id;}

  synchronized void put() {
    taken=false;
    display.setFork(identity,taken);
    notify();
  }

  synchronized void get()
      throws java.lang.InterruptedException {
    while (taken) wait();
    taken=true;
    display.setFork(identity,taken);
  }
}
```

taken encodes the state of the fork

Dining Philosophers - Philosopher implementation

```
class Philosopher extends Thread {
  ...
  public void run() {
    try {
      while (true) {                              // thinking
        view.setPhil(identity,view.THINKING);
        sleep(controller.sleepTime());    // hungry
        view.setPhil(identity,view.HUNGRY);
        right.get();                                // gotright chopstick
        view.setPhil(identity,view.GOTRIGHT);
        sleep(500);
        left.get();                                           // eating
        view.setPhil(identity,view.EATING);
        sleep(controller.eatTime());
        right.put();
        left.put();
      }
    } catch (java.lang.InterruptedException e){}
  }
}
```

Follows from the model (sitting down and leaving the table have been omitted).

Dining Philosophers - implementation in Java

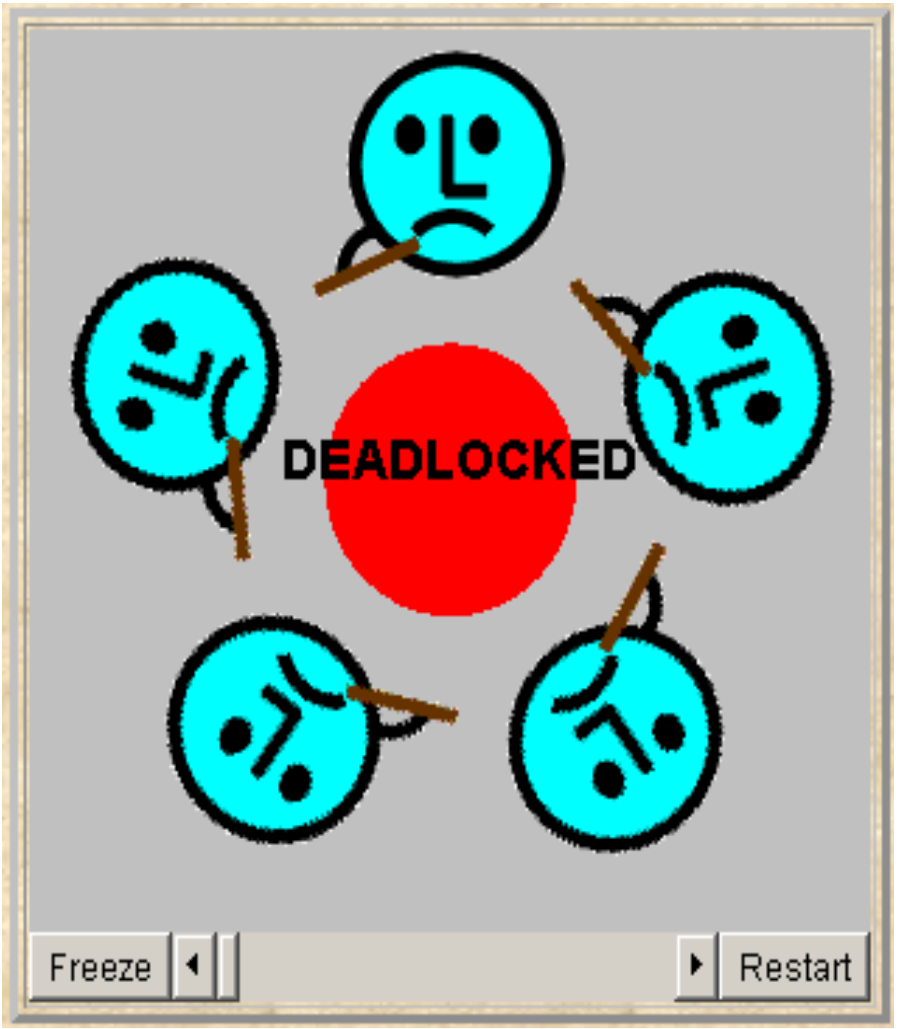
Code to create the philosopher threads and fork monitors:

```
for (int i =0; i<N; ++i)
  fork[i] = new Fork(display,i);
for (int i =0; i<N; ++i){
  phil[i] =
    new Philosopher
            (this,i,fork[(i-1+N)%N],fork[i]);
  phil[i].start();
}
```

Dining Philosophers

To ensure deadlock occurs eventually, the slider control may be moved to the left. This reduces the time each philosopher spends thinking and eating.

This "speedup" increases the probability of deadlock occurring.



Deadlock-free Philosophers

Deadlock can be avoided by ensuring that a wait-for cycle cannot exist. *How?*

Introduce an *asymmetry* (take ressources in same order) into our definition of philosophers.

Use the identity I of a philosopher to make even numbered philosophers get their left forks first, odd their right first.

Other strategies?

```
PHIL(I=0)
  = (when (I%2==0) sitdown
            ->left.get->right.get
            ->eat
            ->left.put->right.put
            ->arise->PHIL
    |when (I%2==1) sitdown
       ->right.get->left.get
       ->eat
            ->left.put->right.put
            ->arise->PHIL
    ).
```