

# Deadlocks and Safety Properties

Jens Chr. Godskesen  
IT University of Copenhagen

# Overview

- Deadlocks
  - Avoiding deadlocks
  - Deadlock conditions
  - The Dining Philosophers
- Safety properties
  - What are properties?
  - Hennessy-Milner Logic
  - Recursive formulas and fixed points
- Exercises

# What is a deadlock?

Let  $P$  be defined by

$$\text{agent } P(a,b) = a.(b.P\langle a,b\rangle + a.0)$$

Issuing in MWB the command `deadlocks P<a,b>` results in

```
Deadlock found in 0  
reachable by 2 commitments
```

In *CCS* a *deadlocked state* is a state with no outgoing transitions.

# Deadlock due to concurrency

[techspeak] A situation wherein two or more processes are unable to proceed because each is waiting for one of the others to do something. ...

A common example is a program communicating to a server, which may find itself waiting for output from the server before sending anything more to it, while the server is similarly waiting for more input from the controlling program before outputting anything.

# Deadlock due to concurrency

Will this system deadlock?

```
agent Printer(acq,rel) = acq.rel.Printer<acq,rel>
agent Scanner(acq,rel) = acq.rel.Scanner<acq,rel>
agent PS(acqP,relP,acqS,relS) = Printer<acqP,relP> | Scanner<acqS,relS>

agent P(acqP,relP,acqS,relS,copy) =
    'acqP.'acqS.'copy.'relP.'relS.P<acqP,relP,acqS,relS,copy>
agent Q(acqP,relP,acqS,relS,copy) =
    'acqS.'acqP.'copy.'relP.'relS.Q<acqP,relP,acqS,relS,copy>
agent PQ(acqP,relP,acqS,relS,copyP,copyQ) =
    P<acqP,relP,acqS,relS,copyP> | Q<acqP,relP,acqS,relS,copyQ>

agent Sys(cP,cQ) = (^ aP,rP,aS,rS)(PQ<aP,rP,aS,rS,cP,cQ> | PS<aP,rP,aS,rS>)
```

# Deadlock due to concurrency

Asking

```
deadlocks Sys<copyP, copyQ>
```

tells that the system deadlocks in two steps.

The deadlock occur if first P gets Printer and then Q gets Scanner before P does.

**Exercise** Explain in detail from the transitions of the model above the deadlock reported by running `deadlocks Sys<copyP, copyQ>`.

# Deadlock due to concurrency

Recall the model for the bounded buffer from last lecture

```
agent Semaphore0(u,d) = u.Semaphore1<u,d>
agent Semaphore1(u,d) = u.Semaphore2<u,d> + d.Semaphore0<u,d>
agent Semaphore2(u,d) = u.Semaphore3<u,d> + d.Semaphore1<u,d>
agent Semaphore3(u,d) = d.Semaphore2<u,d>

agent Spaces(u,d) = Semaphore3(u,d)
agent Items(u,d) = Semaphore0(u,d)

agent Buf(p,g,s_d,s_u,i_d,i_u) = p.'s_d.'i_u.Buf<p,g,s_d,s_u,i_d,i_u>
                                + g.'i_d.'s_u.Buf<p,g,s_d,s_u,i_d,i_u>

agent BB(put,get) =
  (^ s_d,s_u,i_d,i_u)( Buf<put,get,s_d,s_u,i_d,i_u>
                       | Spaces<s_u,s_d> | Items<i_u,i_d> )
```

**Exercise** Load `ErrBoundBuf.ag` and run `deadlocks BB<put,get>` Explain the reported deadlocks.

## Avoiding Deadlocks, the printer/scanner system

Deadlock can be avoided if we *order* resources, e.g.

```
agent P(acqP,relP,acqS,relS,copy) =  
    'acqP.'acqS.'copy.'relP.'relS.P<acqP,relP,acqS,relS,copy>  
agent Q(acqP,relP,acqS,relS,copy) =  
    'acqP.'acqS.'copy.'relP.'relS.Q<acqP,relP,acqS,relS,copy>
```

or if we allow them to be released (using e.g. a *time-out*)

```
agent P(acqP,relP,acqS,relS,copy) = 'acqP.GetS<acqP,relP,acqS,relS,copy>  
agent GetS(acqP,relP,acqS,relS,copy) =  
    'acqS.'copy.'relP.'relS.P<acqP,relP,acqS,relS,copy>  
    + t.'relP.P<acqP,relP,acqS,relS,copy>
```

**Exercise** Explain why the two techniques guarantee absence of deadlock. Check using the techniques that there is no deadlock running deadlocks `Sys<copyP,copyQ>`.

# Deadlock Conditions

There are four necessary and sufficient conditions for a deadlock to occur. To avoid deadlocks, make sure that at least one of the conditions is not satisfied.

- Resources are *serially reusable*
- Processes allow *incremental acquisition*
- *No pre-emption* of resources
- There exists a *wait-for cycle* of processes

# Serially Reusable Resources

Serially reusable resources are resources shared and repetitively used under mutual exclusion.

A *database table* is a shared resources used repetitively under mutual exclusion as far as writing is concerned.

If for instance a database table could be accessed by several writing processes at the same time it would not cause deadlock (but how would the database contents then look like?).

*Monitors* and *semaphores* are serially reusable resources.

# Incremental Acquisition

Incremental acquisition means that processes hold on to their granted resources while they are waiting to obtain the additional resources.

Incremental acquisition was encountered in the nested monitor problem where the consumer holds the lock of the monitor while it is waiting for the producer to deliver a character.

The *two-phase commit* principle in database systems avoids incremental acquisition.

Incremental acquisition is denied by the time-out in the printer-scanner system.

## No pre-emption

No pre-emption means that resources acquired by a process are only voluntarily released by it, i.e. they cannot be forcibly pre-empted.

Suppose a process is waiting for yet another resource, if it then happens (say due to the operating systems) that all its allocated resources are pre-empted then deadlock cannot occur.

## A Wait-for-Cycle

A Wait-for-Cycle is a *cycle* of processes where each process holds a resource its successor in the cycle is waiting for.

**Exercise** The printer-scanner system has a wait-for-cycle. Letting a circle represent a process and letting a box represent a resource how will you draw the wait-for-cycle for the printer-scanner system? To ease the reading of the cycle you may choose to write 'has a' and 'awaits' on the arcs in the cycle.

# A Wait-for-Cycle

The wait-for cycle condition cannot be fulfilled if resources are always taken in some specific order.

Suppose  $n$  resources ordered  $r_1 < r_2 < \dots < r_n$

Assume a wait-for-cycle among the processes  $p_1, \dots, p_k$  such that:

$p_1$  has  $r'_1$  awaits  $r'_2$       implies       $r'_1 < r'_2$

$p_2$  has  $r'_2$  awaits  $r'_3$       implies       $r'_2 < r'_3$

...

$p_k$  has  $r'_k$  awaits  $r'_1$       implies       $r'_k < r'_1$

It follows that  $r'_1 < r'_2 < \dots < r'_k < r'_1$  and we have a contradiction.

# The Dining Philosophers

*Five* philosophers are sitting at a circular table. Each philosopher alternates between: *thinking* and *eating*. There is a plate in front of each philosopher. Five forks are placed such that there is one to the right and one to the left of each plate. A philosopher may pick up the forks to his left and right, but only one at the time. A philosopher needs two forks to eat.

**Exercise** Make a drawing of the system consisting of the five philosophers, their forks, and the plates.

# The Dining Philosophers

Make sure a philosopher eats only if she has two forks, assuming no two philosophers may hold the same fork simultaneously.

```
agent Fork(get,put) = get.put.Fork<get,put>
agent Forks(g1,g2,g3,g4,g5,p1,p2,p3,p4,p5) =
  Fork(g1,p1) | Fork(g2,p2) | Fork(g3,p3) | Fork(g4,p4) | Fork(g5,p5)

agent Phil(getL,putL,getR,putR,eat) =
  'getL.'getR.eat.'putL.'putR.Phil<getL,putL,getR,putR,eat>
agent Phils(g1,g2,g3,g4,g5,p1,p2,p3,p4,p5,e1,e2,e3,e4,e5) =
  Phil(g1,p1,g5,p5,e1) | Phil(g2,p2,g1,p1,e2) | Phil(g3,p3,g2,p2,e3)
  | Phil(g4,p4,g3,p3,e4) | Phil(g5,p5,g4,p4,e5)

agent Table(e1,e2,e3,e4,e5) = (^ g1,g2,g3,g4,g5,p1,p2,p3,p4,p5)
  (Forks(g1,g2,g3,g4,g5,p1,p2,p3,p4,p5)
  | Phils(g1,g2,g3,g4,g5,p1,p2,p3,p4,p5,e1,e2,e3,e4,e5))
```

# The Dining Philosophers

**Exercise** Give an example of five steps in `Table(e1, e2, e3, e4, e5)` that leads to a deadlock.

**Exercise** According to the deadlock conditions there must be a wait-for-cycle among the processes in `Table(e1, e2, e3, e4, e5)`. What is the wait-for-cycle in your solution to the previous exercise.

# The Dining Philosophers

We break the wait-for-cycle by enforcing an *asymmetry* in a philosophers behaviour.

```
agent AsymPhil(getL,putL,getR,putR,eat) =  
  'getR.'getL.eat.'putL.'putR.AsymPhil<getL,putL,getR,putR,eat>  
agent Phil(getL,putL,getR,putR,eat) =  
  'getL.'getR.eat.'putL.'putR.Phil<getL,putL,getR,putR,eat>  
agent Phils(g1,g2,g3,g4,g5,p1,p2,p3,p4,p5,e1,e2,e3,e4,e5) =  
  AsymPhil(g1,p1,g5,p5,e1) | Phil(g2,p2,g1,p1,e2) |  
  Phil(g3,p3,g2,p2,e3) | Phil(g4,p4,g3,p3,e4) | Phil(g5,p5,g4,p4,e5)
```

**Exercise** Why will this asymmetric model not deadlock? Or, why is the wait-for-cycle broken?

**Exercise** Why is it sufficient to let only one of the philosophers have asymmetric behaviour?

# The Dining Philosophers

We may also avoid having a deadlock by breaking incremental acquisition using time-out, e.g.

```
agent Phil(getL,putL,getR,putR,eat) =  
    'getL.GetRPhil(getL,putL,getR,putR,eat)  
agent GetRPhil(getL,putL,getR,putR,eat) =  
    t.'putL.Phil<getL,putL,getR,putR,eat>  
    + 'getR.eat.'putL.'putR.Phil<getL,putL,getR,putR,eat>
```

Another solution would be to let only four philosophers sit at the table simultaneously.

**Exercise** Why is that a solution?

# Starvation and Livelock

Any of the deadlock free solutions to the Dining Philosophers suffer from individual *starvation*, i.e. no philosopher is ever guaranteed to be eating.

**Exercise** Why are even the asymmetric solution subject of starvation?

The solution where a time-out is used suffers from *livelock*, i.e. there exists executions in which no philosopher ever gets two forks.

**Exercise** Why does the time-out solution suffer from livelock? Why does the asymmetric solution not suffer from livelock?

# What are Properties?

A *property* for a process  $P$  is an assertion about the behaviour of  $P$ .

The *behaviour* of a process is the *execution tree* of a process.

**Exercise** What is the execution tree of the process

$$P(a,b) = a.(b.P\langle a,b\rangle + a.0)$$

**Exercise** What is the execution tree of the (deadlock free) system with the two users of a scanner and a printer?

# What are Properties?

The most common properties for *sequential programs* are

- the program terminates
- the right result is computed by the program (eg. inserting, deleting and searching correctly, sorting correctly, ...)

# What are Properties?

Concurrent programs are not always supposed to terminate, so for these we may like to express other properties:

- The buffer is not always full (is not always empty).
- After the printer is acquired it is released later on.
- The Dining Philosophers don't deadlock.
- When a philosopher is thinking he eventually gets the opportunity to eat.

# Validity

A property is *valid* or *invalid* wrt. a process's execution tree.

- The assertion “P(a,b) deadlocks” is valid because P(a,b) has a finite trace in its execution tree.
- “Sys(cP, cQ) doesn't deadlock” is invalid (in the deadlock potential model) because it has a finite path in its execution tree of length two (when P has the printer and Q the scanner)
- “The printer is eventually released after it is taken” is valid in the deadlock free model, because in any state where the printer has just been taken there is a finite sequence of steps to a state where the printer will be returned.

# Hennesy-Milner-Logic

Hennesy-Milner-logic (HML) is the set of formulas  $\Phi$  ranged over by  $\phi$  where

$$\phi ::= tt \mid ff \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi \mid [a] \phi$$

where  $a \in \mathcal{A}$  is an action.

The last two operators are referred to as *modalities*. HML is a *modal logic*.

A process  $p$  satisfies  $\langle a \rangle \phi$  if there exists a transition  $p \xrightarrow{a} p'$  such that  $p'$  satisfy  $\phi$ .

$p$  satisfies  $[a] \phi$  if for all  $p \xrightarrow{a} p'$  it holds that  $p'$  satisfy  $\phi$ .

# Hennessey-Milner-Logic Semantics

The semantics of HML is formally defined relative to a labeled transition system  $\mathcal{L} = (\mathcal{P}, \mathcal{A}, \rightarrow)$  by a relation  $\models \subseteq \mathcal{P} \times \Phi$

$p \models tt$  for any  $p \in \mathcal{P}$

$p \models ff$  for no  $p \in \mathcal{P}$

$p \models \phi \wedge \phi'$  iff  $p \models \phi$  and  $p \models \phi'$

$p \models \phi \vee \phi'$  iff  $p \models \phi$  or  $p \models \phi'$

$p \models \langle a \rangle \phi$  iff there exists  $p \xrightarrow{a} p'$  such that  $p' \models \phi$

$p \models [a] \phi$  iff for all  $p \xrightarrow{a} p'$  it holds that  $p' \models \phi$

## Hennesy-Milner-Logic, examples

**Exercise** Explain why  $\langle a \rangle tt$  intuitively means “can do  $a$ ” whereas  $[a]ff$  intuitively means “can’t do  $a$ ”.

**Exercise** Why is it that  $p$  trivially satisfy  $[a]\phi$  if there exists no  $p \xrightarrow{a} p'$ ?

**Exercise** Explain why  $\langle a_1 \rangle \dots \langle a_m \rangle tt$  characterizes processes that can perform the trace  $a_1 \dots a_m$ .

**Exercise** Explain why  $\langle a_1 \rangle \dots \langle a_m \rangle ([b_1]ff \wedge \dots \wedge [b_n]ff)$  is satisfied by a process that can perform the trace  $a_1 \dots a_m$  but then cannot engage in any of  $b_1, \dots, b_n$ .

## Hennessy-Milner-Logic, examples

**Exercise** Explain why  $a.(b.0 + c.0)$  satisfy  $\langle a \rangle \langle b \rangle tt$

**Exercise** Explain why  $a.(b.0 + c.0)$  satisfy  $[a] \langle b \rangle tt$  but  $a.b.0 + a.c.0$  doesn't.

**Exercise** Give a property satisfied by  $a.b.0 + a.c.0$  but not by  $a.(b.0 + c.0)$ .

**Exercise** Why are  $tt$  and  $[a]tt$  the same? Why are  $ff$  and  $\langle a \rangle ff$  the same?

# Hennessy-Milner-Logic, Tool Support

```
agent P(a,b,c) = a.b.0 + a.c.0  
agent Q(a,b,c) = a.(b.0 + c.0)
```

```
MWB>prove P<a,b,c> <a><b>TT  
Model Prover says: YES!
```

```
MWB>prove P<a,b,c> [a]<b>TT  
Model Prover says: NO.
```

```
MWB>prove Q<a,b,c> [a]<b>TT  
Model Prover says: YES!
```

```
MWB>prove P<a,b,c> <a>[b]FF  
Model Prover says: YES!
```

```
MWB>prove Q<a,b,c> <a>[b]FF  
Model Prover says: NO.
```

```
MWB>prove P<a,b,c> <a>TT & [a](<b>TT | <c>TT)  
Model Prover says: YES!
```

# Recursive Hennessy-Milner-Logic

How to define “absence of deadlock” when only formulas describing finite behaviour are allowed?

How to specify that  $P(a) = a.P(a)$  can do an infinite number of  $a$ 's

We allow HML to be defined with respect to a set of variables  $\mathcal{X}$  extending the syntax, where  $X \in \mathcal{X}$ , with

$$\phi ::= \dots \mid X \mid \nu X.\phi$$

Then  $\nu X.\langle a \rangle X$  is satisfied by all processes that can do an infinite number of  $a$ -transitions.

# Maximal Fixed Points

Let  $\langle a \rangle^k$  denote a sequence of  $k$   $\langle a \rangle$ -modalities and consider  $\phi$  as the subset of processes in  $\mathcal{P}$  that satisfy  $\phi$ . Then  $\nu X.\langle a \rangle X$  is a *maximal fixed point* defined as the limit of

$$tt \supseteq \langle a \rangle tt \supseteq \langle a \rangle \langle a \rangle tt \supseteq \dots \langle a \rangle^k tt \supseteq \dots$$

The set of processes satisfying  $\nu X.\langle a \rangle X$  is the *largest* subset of  $\mathcal{P}$  that is a solution to

$$X = \langle a \rangle X$$

The *smallest* set satisfying the equation is  $ff$  (the empty set) because  $ff$  semantically is the same as  $\langle a \rangle ff$ .

# Safety Properties

Safety properties are properties stating that “something bad never happens”.

A safety property is *always* true, i.e. it holds true in every state in the computation tree.

If  $\phi$  is the property that must hold in every state then we may define the corresponding safety property by

$$\nu X.(\phi \wedge \bigwedge_{a \in \mathcal{A}} [a]X) \quad \textit{Safety}$$

Intuitively it means that  $\phi$  must hold in the current state and no matter what the next state is  $\phi$  must hold again.

# Safety Properties, Examples

P and Q copy under mutual exclusion:

```
MWB>prove Sys<cP,cQ>
```

```
    nu X.(((['cP]FF | ['cQ]FF) & ([t]X & ['cP]X & ['cQ]X))
```

Model Prover says: YES!

Absence of deadlock, i.e. “in any state at least one transition is possible”, is a safety property

```
MWB>prove Sys<cP,cQ> nu X.((<'cP>TT | <'cQ>TT | <t>TT)
```

```
    & ([t]X & ['cP]X & ['cQ]X))
```

Model Prover says: YES!

# Exercises

1. Go through the exercises we didn't make during the lecture.
2. Make a deadlock free model of the Dining Philosophers where only four philosophers may sit at the table at any one time.
3. Do the programming lab about the Dining Philosophers.
4. Write a safety property for a deadlock free model of the Dining Philosophers stating that two neighbouring philosophers eat under mutual exclusion. Prove the safety property in MWB.

5. We may choose a weaker variant of the general format of the safety property presented above; instead of requiring that any state in the computation tree satisfy  $\phi$  it may suffice only a single path in the tree doing so. Give a HML formula stating that  $\phi$  holds in every state along some path in the computation tree.