

## Module CSC3324: Understanding Concurrency



**Lecturer:** Dr Victor Khomenko  
**E-mail:** [Victor.Khomenko@ncl.ac.uk](mailto:Victor.Khomenko@ncl.ac.uk)  
**Office:** USB.5.026 (but mostly WFH)

1

## What is covered in this course

- Theory, modelling, reasoning, links to other areas
- **Not** covered: implementation details, APIs, programming – read the manuals!

## Warnings:

- **Maths content**
- **Research-led teaching**
- **Home-brewed software**



2

## Contact hours & assessment

- Lectures (pre-recorded): ~2 hours per week
- Practicals (online): ~1 hour per week
  - Q&A via Zoom, see the details in Canvas
  - You will need to install **Workcraft** from [workcraft.org](http://workcraft.org), see the installation instructions in Canvas
- Assessment (coursework-only this year):
  - 4 online tests (approximately fortnightly): 4\*10%
  - Modelling assignment: 60%

3

## Introduction: Why concurrency?

4

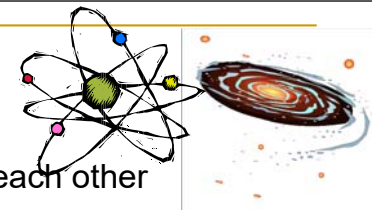
## Why concurrency?

- **Sequential** systems belong to the last millennium!
- Nowadays a CS professional must be able to understand, design and reason about **concurrent** systems:
  - multicore PCs, GPUs
  - networks of workstations, Cloud
  - digital circuits
  - reactive systems, including device drivers and GUIs
  - mobile, reconfigurable and multi-agent systems
  - biological systems
  - organisations
  - etc.

5

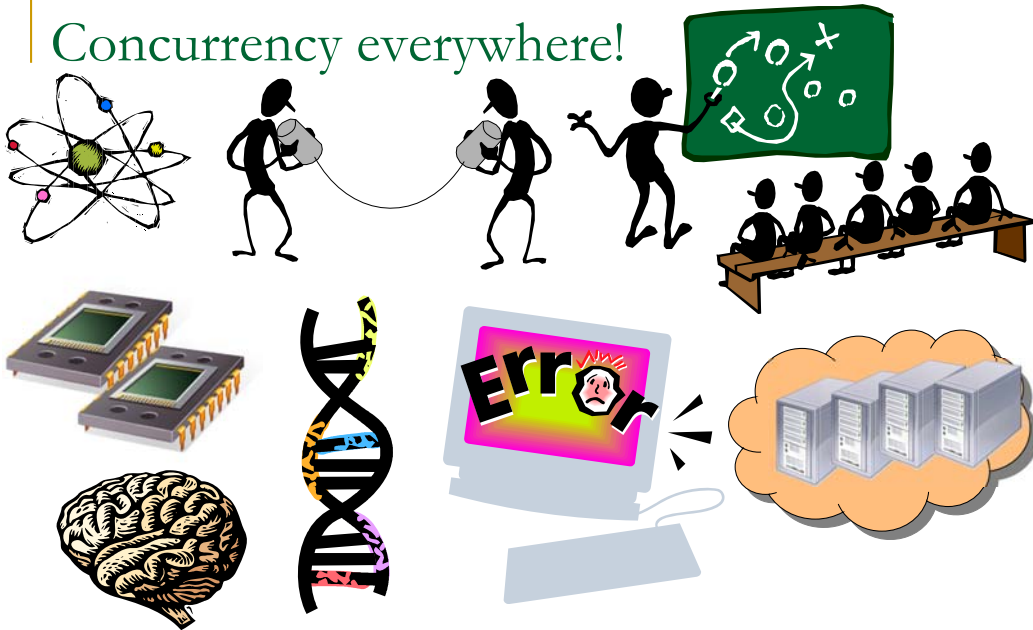
## Concurrency everywhere!

- The Universe is highly concurrent!
  - Elementary particles are concurrent to each other
  - So are atoms, molecules, ..., cells in a body, people in an organisation, ..., planets, galaxies, ...
- Concurrent objects can occasionally interact / react / synchronise with other objects
  - $2\text{H}_2 + \text{O}_2 \rightarrow 2\text{H}_2\text{O}$
  - Java thread reading a variable that another thread is writing
  - $N \text{ Students} + 1 \text{ Lecturer} \rightarrow N \text{ Yawning Students} + 1 \text{ Tired Lecturer}$
- **Concurrency control:** prevent 'bad' interactions and minimise unnecessary ones



6

## Concurrency everywhere!



7

## Considerations: Performance

- To do things faster, do them concurrently! E.g. for a microprocessor:
  - Pipeline the execution, e.g. basic RISC pipeline:  
Fetch→Decode→Execute→MemAccess→RegWriteBack
  - Data parallelism: handle more bits per step, e.g. wider registers, handle 3D or 4D vectors in a single instruction,...
  - Floating-point unit concurrent to the (rest of) CPU
  - Memory and instruction caches, instruction and data pre-fetch
  - Superscalar architecture and out-of-order execution: execute multiple instructions by dispatching them to different functional units on the CPU (provided there are no dependencies)
  - Speculative execution

8

## Considerations: Energy consumption

- From ultra-low-power devices: no battery, energy harvesting, very tight energy budget...
- ... to mobile battery-powered devices: battery life...
- ... to PCs: have to pay that electricity bill! Consume  % of global electricity production, and this proportion is rising...
- ... to datacentres: consume  % of global electricity production (mainly running the servers and cooling), and this proportion is rising

9

## Considerations: Energy consumption

- Consumed power:  $P \sim C \cdot V^2 \cdot F$ , where  $C$  is switching capacitance,  $V$  is the voltage, and  $F$  is the frequency
- $F \sim V$  so  $P \sim F^3$ , i.e. halving  $F$  allows to decrease  $P$  by a factor of 8! Performance/power trade-off...
- What if a circuit can be replaced by its two copies running at half the frequency, achieving similar performance?
  - $F$  is halved and  $C$  is doubled, i.e. the changes in  $C$  and  $F$  cancel each other
  - $V$  is halved, i.e.  $P$  is decreased by a factor of 4
- Hence much more energy-efficient to run two chips at half the frequency! In practice some overhead is introduced by such a transformation, but the gains are still high

10

## Considerations: Energy consumption

### ■ Example: Human brain

- ~30 billion neurons
- neurons work at frequencies 10s-100s Hz
- so very high concurrency, very low frequency
- consumes  % of body energy budget
- digital hardware simulation would require a dedicated power plant!



<http://apt.cs.manchester.ac.uk/projects/SpiNNaker/>

11

## Considerations: Heat dissipation

- Frequency → energy consumption → **HEAT**
- CPU max operating temperature:  °C
- Cooling is a HUGE problem: from mobile devices to PCs to datacentres
- End of **frequency scaling**: In 2004 Intel cancelled its Tejas and Jayhawk processors, thought to be due to heat problems caused by extreme power consumption; focused on dual-core chips for the Itanium platform instead

12

## Moore's law: frequency & technology scaling

- Graph: collapse of **frequency scaling** – plots clock frequencies of real microprocessors over time, with a clear flattening ~2005

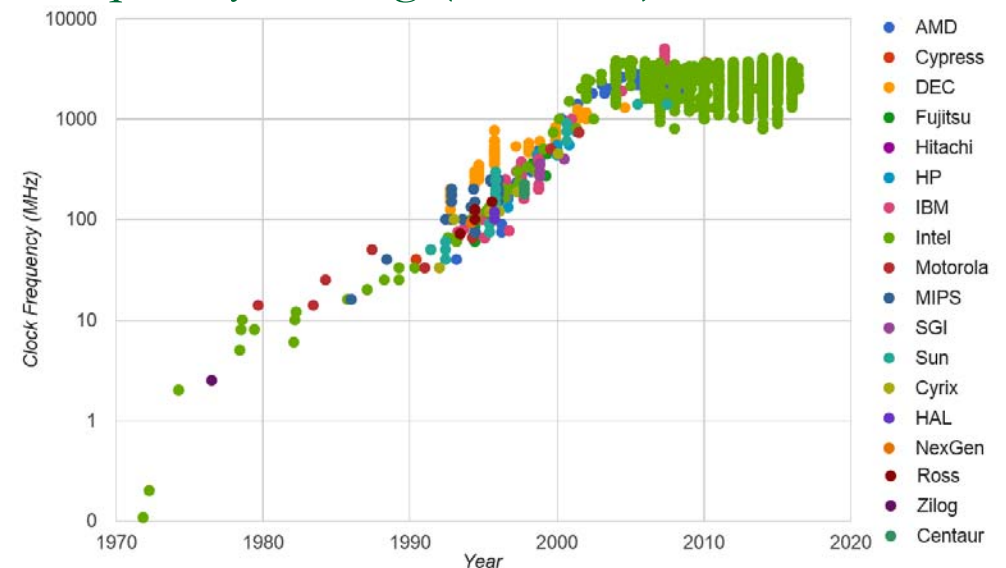
[http://cpudb.stanford.edu/visualize/clock\\_frequency](http://cpudb.stanford.edu/visualize/clock_frequency)

- Graph: **technology scaling** – plots transistor feature sizes over time – still continues

[http://cpudb.stanford.edu/visualize/technology\\_scaling\\_by\\_manufacturer](http://cpudb.stanford.edu/visualize/technology_scaling_by_manufacturer)

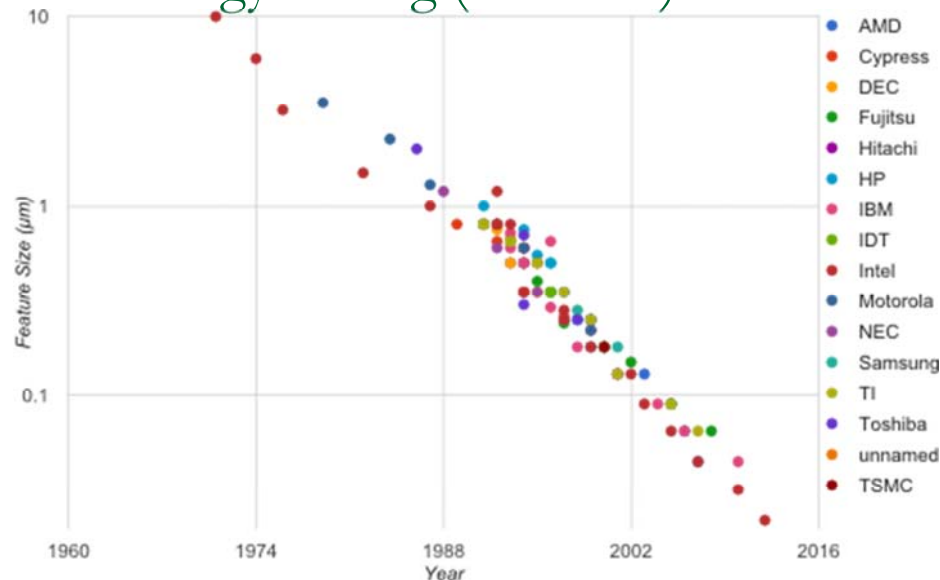
13

## Frequency scaling (CPUDB)



14

## Technology scaling (CPUDB)



15

## Considerations: Parallel scaling / multicore

- As a consequence, **frequency scaling** has been replaced by **parallel scaling**, causing an industry-wide shift to parallel computing in the form of multicore processors; predicted to continue for some time:
  - 5nm semiconductor manufacturing process: mass production started in 2020
  - 3nm process: production planned in 2022
- Another development: Those unable to afford powerful computing equipment can now rent it on **Cloud**
- So the H/W engineers passed the buck to S/W engineers, who now have to write concurrent code

16

## Extra reading (won't be assessed)

- **The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software** by Herb Sutter:

<http://www.gotw.ca/publications/concurrency-ddj.htm>

*"The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency."*

- **Welcome to the Jungle** by Herb Sutter:

<http://herbsutter.com/welcome-to-the-jungle/>

Analyses the effects of exiting Moore's law, and makes predictions about future trends in h/w and s/w. Highly relevant for future programmers! Concurrency features prominently in the article.

17

## Understanding concurrency

- Humans / programmers find it difficult to think about concurrent processes
  - **Challenge:** Draw simultaneously a square and a triangle with your two hands! Try to improve the performance over sequential drawing!
- New kinds of bugs and phenomena, like race conditions, deadlocks, livelocks, resource contention, etc.
- Bugs are often subtle, difficult to find, and unrepeatable, and so the usual testing / debugging is of limited use



18

## Example: Concurrent money transfers

The following procedure is to be executed concurrently, by multiple threads. Can you spot a bug?

```
procedure Transfer(accFrom, accTo, amount):  
    // assume accFrom!=accTo  
    acquire accFrom.lock;  
    acquire accTo.lock;  
    // assume canTransfer cannot throw an exception  
    if canTransfer(accFrom, accTo, amount) then  
        accFrom.amount-=amount;  
        accTo.amount+=amount;  
    release accFrom.lock;  
    release accTo.lock;
```

19

## Formal Models of Concurrency: Languages and Traces

20

## Formalisms covered in the Course

- **Finite state machines (FSM):** modelling and analysis of sequential reactive systems; often used for analysis of concurrent reactive systems by substituting 'true concurrency' with **interleaving**
- **Petri nets (PN):** modelling and analysis of concurrent reactive systems

21

## Reactive vs. computational systems

- **Computational systems:** given some input, produce (compute) some result upon termination
  - **Example:** given a natural number  $n$ , compute the smallest prime number  $p \geq n$
- **Reactive systems:** maintain ongoing interaction with their **environment**
  - specified in terms of their **behaviours**, which are built upon **(atomic) actions**
  - often are not supposed to terminate
  - **Examples:** control system for air traffic, thermostat, nuclear reactor, vending machine, etc.; digital circuits; GUIs; various network protocols; biological systems

22

## Atomic actions

- An **atomic** action is one that happens "all at once":
  - cannot stop "in the middle": it either happens completely or does not happen at all
  - no side effects of an atomic action are visible until the action is complete
- Examples:
  - **Asynchronous digital circuits:** signal (voltage) on a wire going **monotonically** from 0 to 1 (or from 1 to 0)
  - **Synchronous digital circuits:** voltage can fluctuate, but must be stable by the end of the clock cycle, when it is latched
  - **Databases:** transactions are either successfully committed, at which point the changes are published, or rolled back, having no visible effect
  - **Multithreaded programmes:** critical section protected by a mutex

23

## Alphabets

An **alphabet** is a finite set of (indivisible or atomic) **symbols** (e.g. actions). Sometimes it is partitioned into **input** and **output** actions.

### Examples:

- $\{a, b, c, \dots, x, y, z\}$  (writing)
- $\{0, 1, \dots, 9\}$  (decimal numbers)
- $\{0, 1\}$  (binary numbers)
- $\{A, C, G, T\}$  (DNA)
- $\{i^+, i^-, o^+, o^-\}$  (events on wires of an asynchronous digital circuit with one input and one output)
- $\{\text{deposit}\pounds 1, \text{withdraw}\pounds 1\}$  (bank account)

24



## Strings (words)

A **string** (or **word**) over an alphabet  $\Sigma$  is a finite sequence of symbols from  $\Sigma$

### Notation:

- $\varepsilon$  denotes the **empty string** (no symbols)
- $\Sigma^*$  denotes the set of all strings over  $\Sigma$  (including  $\varepsilon$ )

### Examples:

- If  $\Sigma = \{a, b, c, \dots, z\}$  then  $\text{hello} \in \Sigma^*$  but  $\text{csc3324} \notin \Sigma^*$
- $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$

25

## Concatenation and Prefix

If  $w = a_1 a_2 \dots a_k$  and  $u = b_1 b_2 \dots b_m$  are strings then:

- $w \circ u = a_1 a_2 \dots a_k b_1 b_2 \dots b_m$  is the **concatenation** of  $w$  and  $u$
- $w$  is a **prefix** of  $u$ , denoted  $w \leq u$ , if  $u = w \circ w'$  for some  $w'$

### Examples:

- $\text{Hello} \circ \text{World} = \text{HelloWorld}$ ,  $\text{He} \leq \text{Hello}$  and  $\text{Hello} \leq \text{Hello}$
- $\text{ab} \circ \text{ba} = \text{abba}$  and  $\text{abba} \leq \text{abbawasgreat}$

**Note:**  $\varepsilon \circ w = w \circ \varepsilon = w$  and  $\varepsilon \leq w$ , for any word  $w$

26

## Languages

A **language** over an alphabet  $\Sigma$  is a set of strings  $L \subseteq \Sigma^*$ .  
 $L$  is **prefix-closed** if for any  $w \leq u$ ,  $u \in L$  implies  $w \in L$ .

**Examples / Exercises:** which of the languages below are prefix-closed?

- $L_1 = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$  is the language of all binary strings of length  $\leq 2$ 
  - is it prefix-closed?
- $L_2 = \emptyset$  is the **empty language** (note that  $\varepsilon \notin L_2$ )
  - is it prefix-closed?
- $L_3 = \{\varepsilon\}$ 
  - is it prefix-closed?

27

## Languages

These two languages model the allowed sequences of deposits and withdrawals for a bank account:

- $L_4 = \{w \mid \text{the number of occurrences of withdraw}\pounds 1 \text{ in } w \text{ does not exceed those of deposit}\pounds 1\}$
- $L_5 = \{w \mid \forall u \leq w: \text{the number of occurrences of withdraw}\pounds 1 \text{ in } u \text{ does not exceed those of deposit}\pounds 1\}$

Intuitively, what is the difference between them?

Are these languages prefix-closed?

28

## Operations on languages: set operations

Languages are **sets** of strings, so we can apply the usual set operations (union  $\cup$ , intersection  $\cap$ , difference  $\setminus$  and complement  $\bar{\phantom{x}}$ )

**Note:**

$$\bar{L} = \Sigma^* \setminus L \text{ and } L \setminus L' = L \cap \bar{L}'$$

**Examples:**

■ If  $\Sigma = \{a, b\}$ ,  $L = \{\epsilon, a, b, abba\}$  and  $L' = \{a, abba, ba\}$  then

$$L \cup L' = \quad L \cap L' =$$

$$L \setminus L' = \quad \bar{L} =$$

29

## Operations on languages: other operations

- $L \circ L' = \{w \circ u \mid w \in L \text{ and } u \in L'\}$  **concatenation**
- $L^* = \{w_1 \circ w_2 \circ \dots \circ w_k \mid k \geq 0 \text{ and each } w_i \in L\}$  **Kleene star**
  - if  $k=0$  then  $w_1 \circ w_2 \circ \dots \circ w_k = \epsilon$

**Derived operations:**

- $L^2 = L \circ L, L^3 = L \circ L \circ L, \dots$
- $L^+ = L \circ L^* = L \cup L^2 \cup L^3 \cup \dots$

**Notes and examples:**

- $L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \dots = \{\epsilon\} \cup L^+$
- What is the result of applying the  $*$  operation to  $\Sigma$ ?
- $L_{inv} = \{i^+ \circ i \circ i^+\}^* \circ \{\epsilon, i^+, i^+ \circ, i^+ \circ i\}$  is the correct **behaviour** of an inverter starting from  $i=0$  and  $o=1$

30

## Formal Models of Concurrency: Finite State Machines (FSMs)

31

## Finite State Machines (FSM)

A **finite state machine (FSM)** (a.k.a. **finite automaton**) is a tuple  $(Q, \Sigma, \Delta, q_0)$  where:

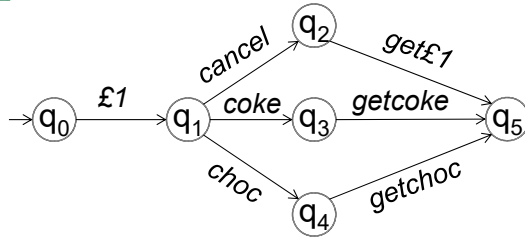
- $Q$  is a finite set of **states**
- $\Sigma$  is a finite alphabet of **actions**
- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$  is the **transition relation**
- $q_0 \in Q$  is the **initial state**

**Graphical representation:** states are circles or ovals (may optionally be labelled by a state id), transition are arcs labelled by actions from  $\Sigma \cup \{\epsilon\}$ , the initial state is pointed with a short arrow with no source

32



## Example



$FSM_{VM}$  represents a single attempt to use a vending machine selling chocolates and coke:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
- $\Sigma = \{\text{£1}, \text{cancel}, \text{coke}, \text{choc}, \text{get£1}, \text{getcok}, \text{getchoc}\}$
- $q_0$  is the initial state

33

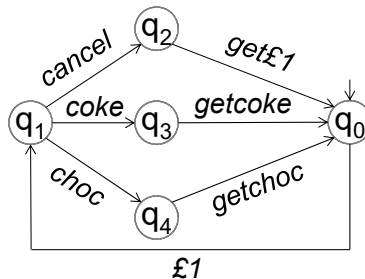
## Intuition

- Alphabet (actions performed by a vending machine to be designed):  
 $\{\text{£1}, \text{choc}, \text{coke}, \text{getcok}, \text{getchoc}, \text{get£1}, \text{cancel}\}$
- Language (the desired behaviour of vending machine):
  - a single usage of the machine:  
 $L = \{\varepsilon, \text{£1}, \text{£1 choc}, \text{£1 cancel}, \text{£1 coke}, \text{£1 choc getchoc}, \text{£1 cancel get£1}, \text{£1 coke getcok}\} =$   
 $L_{incomplete} \cup L_{complete}$
  - $L$  is prefix-closed

34

## Example: Multiple usage

Glue together  $q_0$  and  $q_5$  in the single usage FSM into one state  $q_0$  that is initial:



Language:  $L_{complete}^* \circ L_{incomplete}$

35

## Finite State Machines: notation

We write:

$$q \xrightarrow{a} q' \text{ if } (q, a, q') \in \Delta$$

$$q \xrightarrow{a_1 a_2 a_3 \dots a_k} q' \text{ if there are states } q_1, \dots, q_{k-1} \in Q \text{ such that}$$

$$q \xrightarrow{a_1} q_1, q_1 \xrightarrow{a_2} q_2, q_2 \xrightarrow{a_3} q_3, \dots, q_{k-2} \xrightarrow{a_{k-1}} q_{k-1}, q_{k-1} \xrightarrow{a_k} q'$$

Note that  $q \xrightarrow{\varepsilon} q$  for every  $q \in Q$

**Examples:** The following hold for  $FSM_{VM}$ :

$$q_1 \xrightarrow{\text{coke}} q_3 \quad q_4 \xrightarrow{\varepsilon} q_4 \quad q_0 \xrightarrow{\text{£1 choc}} q_4$$

36

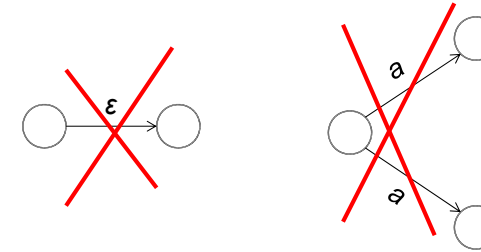
## Exercise: Build an FSM modelling the correct behaviour of an inverter circuit

37

## Deterministic FSMs

An FSM is **deterministic** if

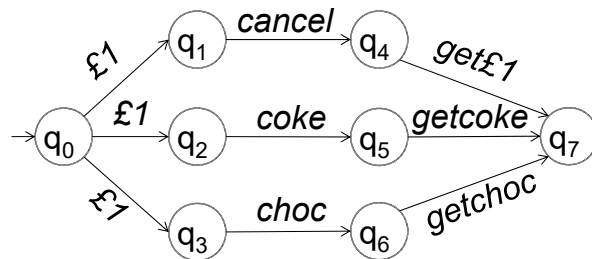
- no transition is labelled by  $\epsilon$ , and
- no two transitions  $(q, a, q')$  and  $(q, a, q'')$  are such that  $q' \neq q''$



The FSMs on previous slides were deterministic

38

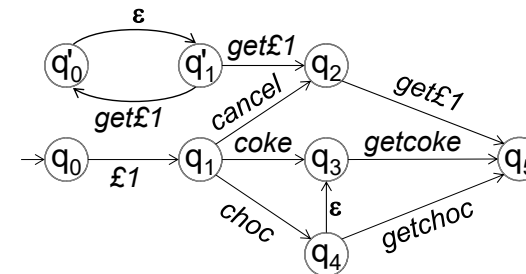
## Example (non-deterministic FSM)



FSM'<sub>VM</sub> can 'decide' how to respond

39

## Example (non-deterministic FSM)



FSM''<sub>VM</sub> can serve coke instead of chocolate and (perhaps!) return several coins

40

## Language and reachable states

To formally capture the behaviour of the three vending machines, look at their languages and reachable states:

- A string  $w$  over  $\Sigma$  is **accepted** or **generated** by an FSM if  $q_0 \xrightarrow{w} q$  for some state  $q \in Q$
- The **language of an FSM** (or **accepted** or **generated by an FSM**) is the set  $\mathcal{L}(\text{FSM})$  of all strings accepted by it; note that  $\mathcal{L}(\text{FSM})$  is prefix-closed
- A state  $q \in Q$  is **reachable** if  $q_0 \xrightarrow{w} q$  for some string  $w$

**Intuition:**

- $\mathcal{L}(\text{FSM})$  represents the **behaviour** (in some sense)
- Unreachable states can be ignored

41

## Analysing vending machines

**Examples:**

- All states of  $\text{FSM}_{\text{VM}}$  and  $\text{FSM}'_{\text{VM}}$  are reachable, but in  $\text{FSM}''_{\text{VM}}$  two states are unreachable ( $q'_0$  and  $q'_1$ ) – thus  $\text{FSM}''_{\text{VM}}$  never dispenses several coins
- $\mathcal{L}(\text{FSM}_{\text{VM}}) = \mathcal{L}(\text{FSM}'_{\text{VM}}) \neq \mathcal{L}(\text{FSM}''_{\text{VM}})$ 
  - £1 choc getcoke belongs only to  $\mathcal{L}(\text{FSM}''_{\text{VM}})$

42

## Behavioural properties: Examples

- A state  $q \in Q$  is called a **deadlock** if it has no exit arcs
- An FSM is called **deadlock-free** if none of its reachable states is a deadlock
- An FSM is called **reversible** if  $q_0$  can be reached from any reachable state  $q$
- If an FSM has no unreachable states:
  - it is reversible iff its graph is **strongly connected**, i.e. there is a directed path between any pair of its states
  - it is enough to check that there is a directed path from any state  $q$  to  $q_0$

43

## Relationship between deterministic and non-deterministic FSMs

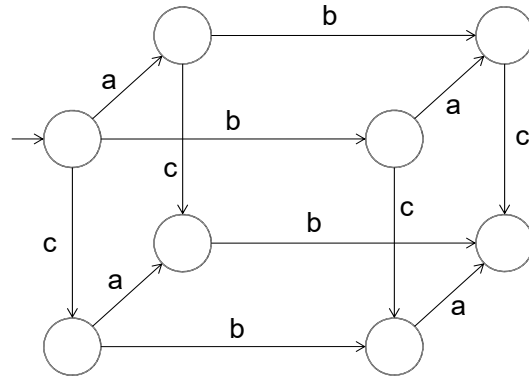
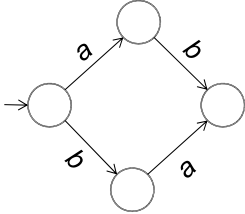
- Given a non-deterministic FSM, one always can construct a deterministic  $\text{FSM}'$  with  $\mathcal{L}(\text{FSM}) = \mathcal{L}(\text{FSM}')$
- Hence non-determinism does not add expressive power from the language point of view, but this construction may exponentially increase the number of FSM states
- For non-deterministic FSMs, the language is a rather blunt representation of the behaviour: e.g.  $\mathcal{L}(\text{FSM}_{\text{VM}}) = \mathcal{L}(\text{FSM}'_{\text{VM}})$  but the FSMs have important differences from the user's point of view!
- Sharper behavioural equivalences exist, e.g. **bisimulation**

44

## Interleaving: 'false concurrency'

- FSMs can represent concurrency between (atomic!) actions by allowing them to happen *in any order*, i.e. by **interleaving** them

- Examples:**



Note the diamond shapes characteristic of interleaving

45

## Example: Concurrent vending machine

- Design an FSM model of a vending machine allowing one to insert £1 and make an order concurrently (multiple usage, no *cancel* button); make sure that the interleaving 'diamonds' are drawn as such

46

## Is interleaving semantics of any use?

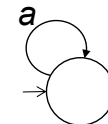
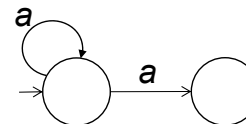
- YES!!!**
- Semantics of almost all concurrency formalisms can be defined in terms of FSMs (or infinite SMs)
- Used for formal verification – many tools build and analyse FSMs (graphs algorithms come in handy)
- Not so good for performance analysis though
- State Space Explosion:** small systems may generate huge FSMs! E.g. Rubik's cube has 43,252,003,274,489,856,000 reachable states



47

## Bisimulation

- Language equivalence is too blunt for some applications:
  - e.g. an FSM with a deadlock may be language-equivalent to a deadlock-free one:



- The reason is that languages describe sequences of actions, but *cannot capture non-deterministic choices*
- Hence, FSMs are a richer behaviour representation than languages, and sharper equivalences can be defined for them

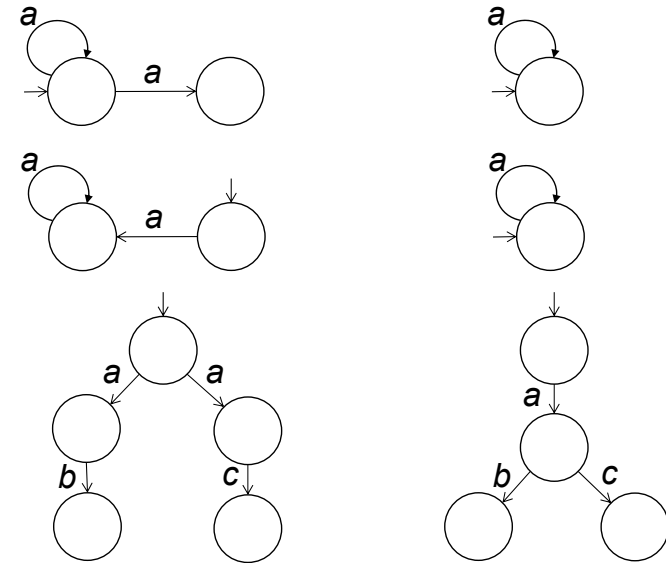
48

## Strong bisimulation

- FSM and FSM' are **bisimilar** if one can establish a correspondence between their states, i.e. build a **bisimulation relation**  $\sim$  such that:
  - the initial states of FSM and FSM' are related:  $q_0 \sim q'_0$
  - if two states are related,  $q \sim q'$ , then:
    - if FSM can make a step  $q \xrightarrow{a} r$  then FSM' can match it,  $q' \xrightarrow{a} r'$  and  $r \sim r'$
    - and vice versa, if FSM' can make a step then FSM can match it and end up in a related state
- If FSM and FSM' are bisimilar then  $\mathcal{L}(\text{FSM}) = \mathcal{L}(\text{FSM}')$ , but not vice versa in general. For deterministic FSMs bisimilarity coincides with language equivalence.

49

## Strong bisimulation: examples



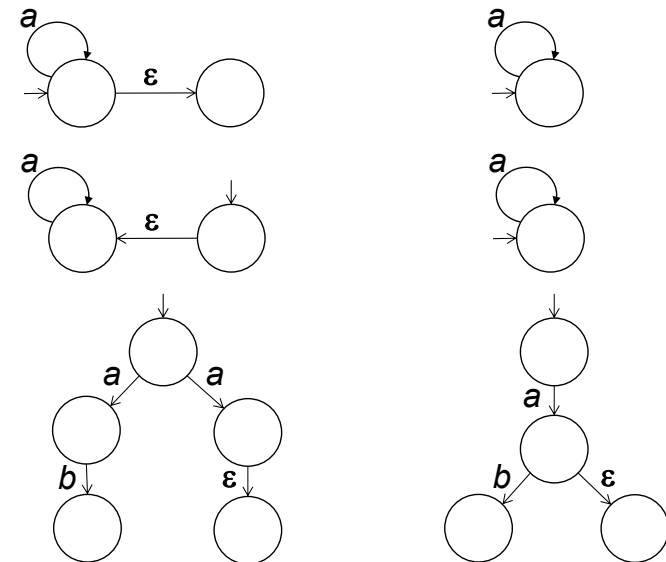
50

## Weak bisimulation

- Strong bisimulation is often too sharp an equivalence, in particular when FSMs have **internal** actions (modelled by  $\epsilon$ ) – such actions are invisible to the external observer and so do not have to be matched
- One can relax the notion of bisimulation to take this into account – just replace all  $\xrightarrow{a}$  by  $\xrightarrow{w}$  where string  $w$  is either  $\epsilon$  or  $a \in \Sigma$ , i.e.
  - each  $\epsilon$ -step of one of the FSM can be matched by 0 or more  $\epsilon$ -steps of the other FSM ( $w = \epsilon \circ \epsilon \circ \dots \circ \epsilon = \epsilon$  in this case)
  - each  $a$ -step of one of the FSM can be matched by 0 or more  $\epsilon$ -steps followed by an  $a$ -step followed by 0 or more  $\epsilon$ -steps of the other FSM ( $w = \epsilon \circ \epsilon \circ \dots \circ \epsilon \circ a \circ \epsilon \circ \epsilon \circ \dots \circ \epsilon = a$  in this case)

51

## Weak bisimulation: examples



52

# Formal Models of Concurrency:

## Petri Nets

53

## Petri nets – general remarks

- A simple but powerful formalism for modelling concurrent, asynchronous, distributed, parallel, non-deterministic, stochastic, etc., systems
- Introduced by Carl Adam Petri (1962)
- Come with a graphical representation (visualisation)
- Formal analytical framework (various techniques and implemented software tools)
- Petri Nets World:  
<http://www.informatik.uni-hamburg.de/TGI/PetriNets/>

54

## Application areas

- concurrent and parallel programs
- communication protocols
- distributed software systems
- manufacturing control systems
- embedded systems
- hardware systems, circuits
- fault tolerant systems
- network security
- bioinformatics
- etc.

55

## From FSMs to PNs

- Represent a concurrent system using several FSMs, each with its own **local state**, communicating by jointly executing common actions
- The overall **system state** is represented in a distributed manner, as a collection of the local states of the individual FSMs (in contrast to a single currently active state of an FSM)
- The system's structure and potential changes of states can be visualised using a directed graph

56



## Example: Producer/Buffer/Consumer

**begin parallel**

Producer:

```
while true do
  produce item
  deposit item
```

Buffer:

```
while true do
  deposit item
  remove item
```

Consumer:

```
while true do
  remove item
  consume item
```

**end parallel**

common action

common action

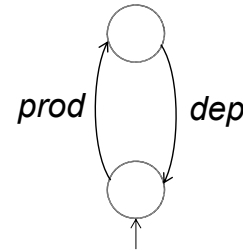
57

## Modelling Producer

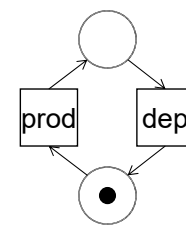
Producer:

```
while true do
  produce item
  deposit item
```

FSM model:



PN model:



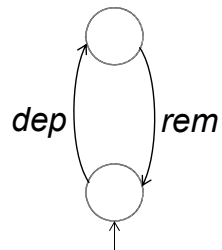
58

## Modelling Buffer

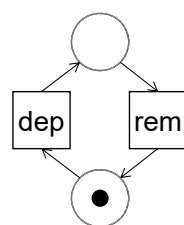
Buffer:

```
while true do
  deposit item
  remove item
```

FSM model:



PN model:



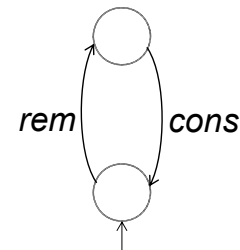
59

## Modelling Consumer

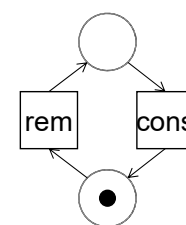
Consumer:

```
while true do
  remove item
  consume item
```

FSM model:



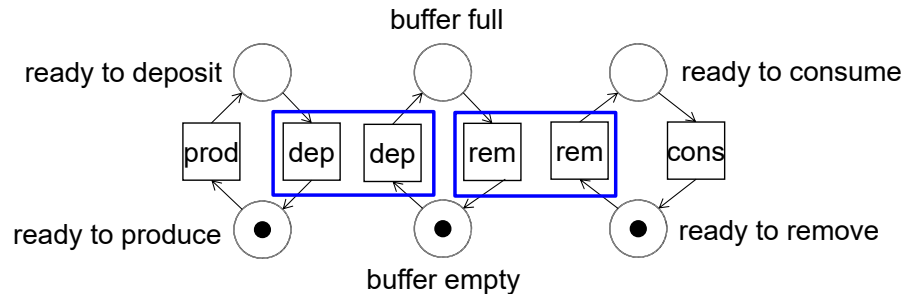
PN model:



60

## Composing Petri net models

Draw the three Petri nets next to each other and glue together boxes with the same label (synchronous communication = joint execution of common actions); some (informal) annotation can optionally be added:



61

## Structure of Petri nets

Interpretation of different components:

- circles are called **places** and represent local states
- boxes are called **transitions** and represent actions changing local states
- black dots are called **tokens** and represent the current holding of local states (in general, a place may contain several tokens – e.g. to model a counter)
- **arcs** indicate how executing a transition modifies the state of a Petri net

62

## Formal definition of a Petri net

A **Petri net** is a tuple

$$PN = (P, T, F, M_0, \ell)$$

where:

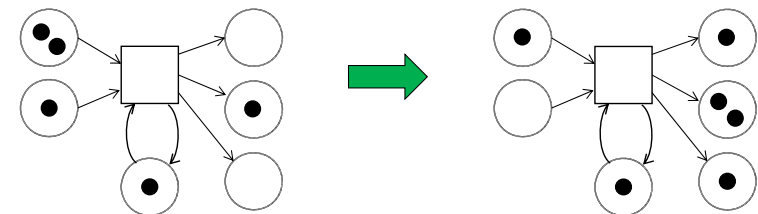
- $P$  is a finite set of **places**
- $T$  is a finite set of **transitions**
- $F \subseteq (P \times T) \cup (T \times P)$  is the **flow relation** (we draw an arc from  $x$  to  $y$  whenever  $(x, y) \in F$ )
- $M_0: P \rightarrow \{0, 1, 2, \dots\}$  is the **initial marking**
- $\ell: T \rightarrow \Sigma$  is a **labelling** of transitions using some alphabet  $\Sigma$  of symbols (optional)

63

## Transition firing rule

Consider a PN with some marking  $M: P \rightarrow \{0, 1, 2, \dots\}$

A transition  $t$  is **enabled** at  $M$  if for every place  $p$  such that  $(p, t) \in F$  we have  $M(p) > 0$  (i.e. all its preceding places are marked). **Firing** an enabled transition  $t$  removes a token from each place  $p$  such that  $(p, t) \in F$  (i.e. from each preceding place) and then adds a token to each place  $r$  such that  $(t, r) \in F$  (i.e. to each succeeding place).



64

## Firing sequences and reachable markings

**Notation:**  $M[t]M'$  means that transition  $t$  is enabled at marking  $M$ , and firing it leads to marking  $M'$

Starting from the initial marking  $M_0$  we may fire a sequence of transitions  $t_1 t_2 \dots t_n$  if there are markings  $M_1, M_2, \dots, M_n$  such that  $M_0[t_1]M_1[t_2]M_2 \dots M_{n-1}[t_n]M_n$

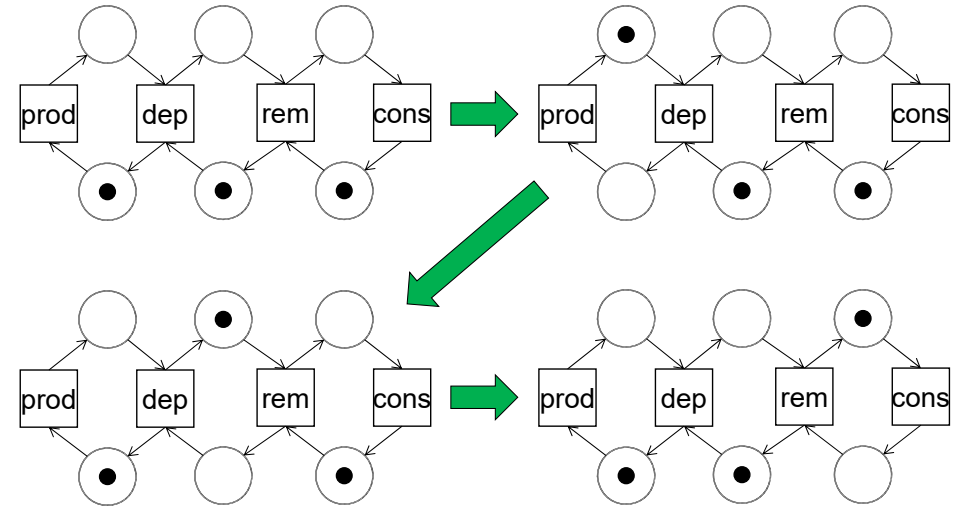
Then  $fs = \ell(t_1)\ell(t_2)\dots\ell(t_n)$  is a **firing sequence** or **trace**, and  $M_n$  is a **reachable marking**

Firing sequences represent what an **observer** of the system would see, and reachable markings are the ones the system can find itself in. In the FSM terminology, firing sequences are strings generated/accepted by a PN.

65

## Example

Firing sequence: prod dep rem



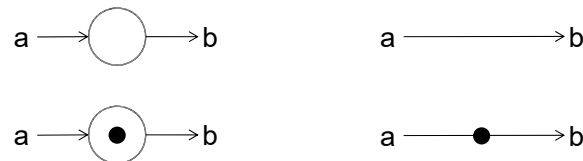
66

## Short-hand drawing conventions

- Omit drawing boxes for transitions – draw just labels:



- If a place has one incoming and one outgoing arc, just draw an arc through it:

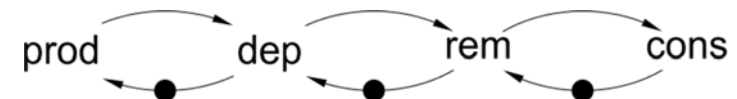


- Colours of the labels: **input**, **output**, **internal**; the latter behave like outputs but are ignored by the environment

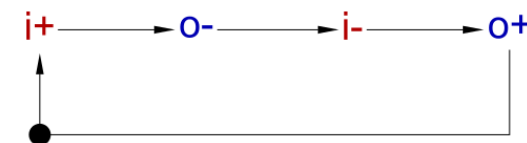
67

## Short-hand drawing conventions: examples

- Producer/Buffer/Consumer system:



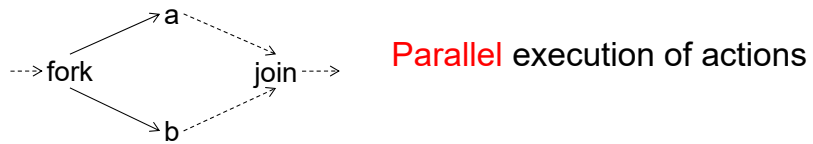
- Inverter circuit:



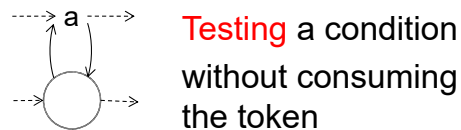
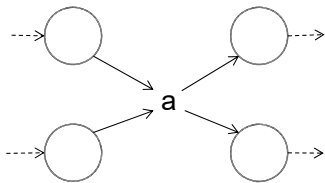
68

## Modelling techniques: basic

-----> a -----> b -----> **Sequential** execution of actions



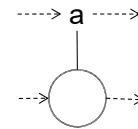
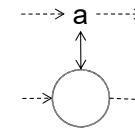
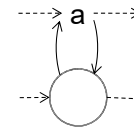
**Synchronisation**  
(joint execution of an action)



69

## Drawing conventions: Read arcs

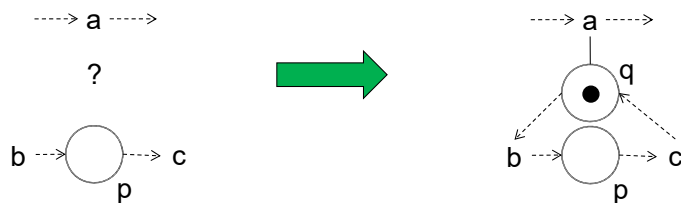
Testing for the presence of a token without consuming it is common. Instead of drawing two arcs going in opposite directions one can either draw a single arc with two arrowheads, or just a line without arrowheads, so the following three drawings have the same meaning. Such arcs are called **read arcs**.



70

## Modelling techniques: complementary places

- Places p and q in are **complementary** iff any transition removing a token from one of them puts a token to the other
- If p can contain at most one token then creating a new place q complementary to p (q is initially marked iff p isn't) does not change the behaviour of the PN
- Can use a complementary place to test the negation of a condition

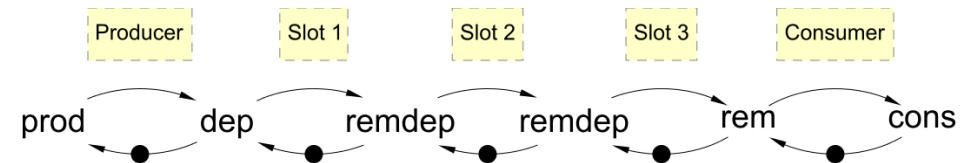


71

## Modelling techniques: synchronisation

**Example:** producer / 3-slot buffer / consumer:

**Idea:** synchronise the **rem** action of a slot with the **dep** action of the next slot:

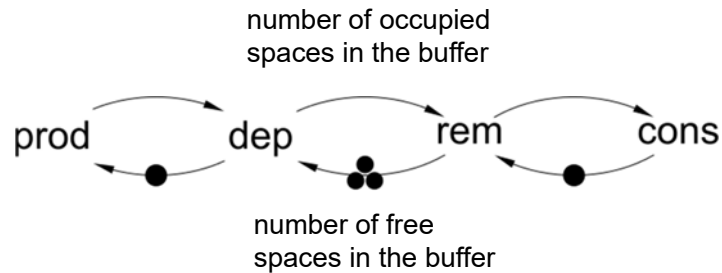


72

## Modelling techniques: counters

**Idea:** use multiple tokens on a place to represent a counter

**Example:** producer / buffer of capacity 3 / consumer:



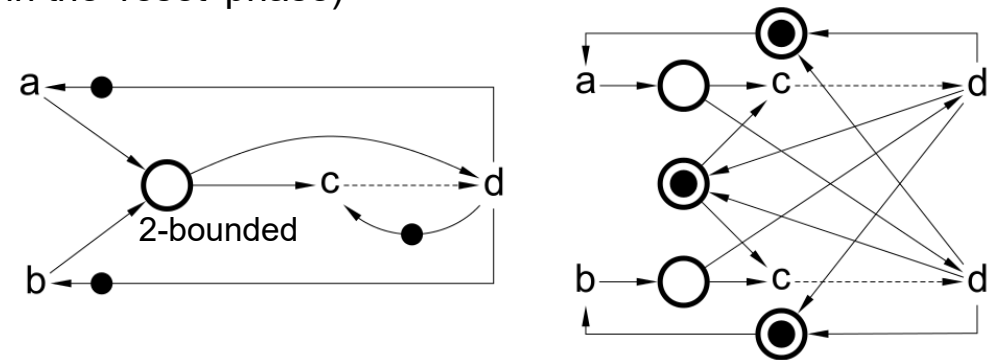
**Note:** the 'identities' of items in the buffer are lost!

73

## Modelling techniques: OR-causality

**Motivation:** quickly react to any of several stimuli

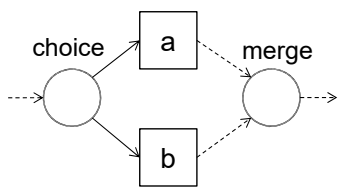
**Assumption:** all stimuli are eventually provided (if not, still can use OR-causality, but **arbitration** will be required in the 'reset' phase)



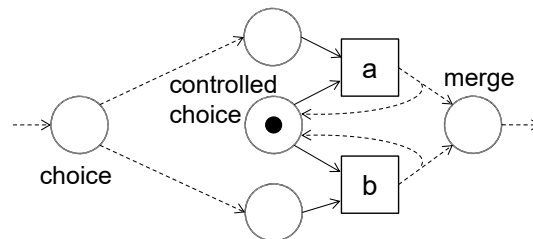
74

## Modelling techniques: choices

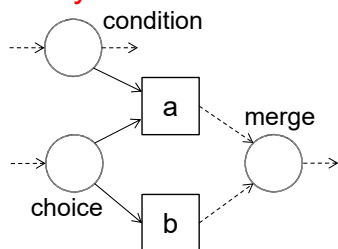
**Free choice**



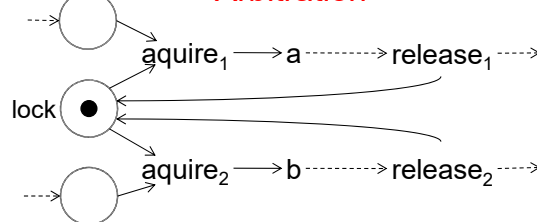
**Controlled Choice**



**Asymmetric choice**



**Arbitration**

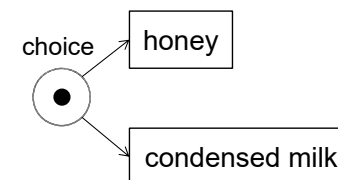


75

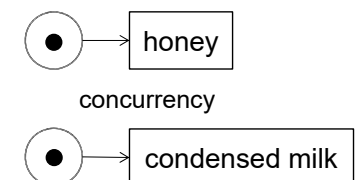
## Modelling techniques: choices

*Pooh always liked a little something at eleven o'clock in the morning, and he was very glad to see Rabbit getting out the plates and mugs; and when Rabbit said, "**Honey or condensed milk with your bread?**" he was so excited that he said, "**Both,**" and then, so as not to seem greedy, he added, "But don't bother about the bread, please."*

**A. A. Milne, "Winnie-the-Pooh"**



Is a choice really necessary? Can it be eliminated?



76

## Modelling techniques: choices

- Why are there any choices at all?
  - Abstraction of the environment (do not want a detailed model of the rest of the Universe!) – the ‘implementation’ of the environment may well be without choices
  - Resource contention: have to arbitrate between several clients trying to use the same resource
  - Structural choices: not ‘semantical’, e.g. due to modelling concurrency by interleaving; can be removed by restructuring the specification
  - Non-deterministic choices: either not ‘real’ and can be removed (by determinising the specification or making OR-causality explicit) or indicate that the system does not have enough information to perform its function (e.g. more inputs from the environment are required)

77

## Modelling techniques: inputs and outputs

- Often actions are partitioned into **inputs** that are performed by the **environment** and **outputs** that the system has to produce (and perhaps **internal** actions)
- Then the PN can be viewed as a **contract** between the system and its environment:
  - if an **input** is enabled then the environment is allowed (but not obliged) to produce it; the environment must not send any **inputs** which are not enabled
  - if an **output** is enabled then the system is obliged to produce it eventually (or it is eventually disabled – such scenarios can be tricky though); the system must not produce any **outputs** which are not enabled

78

## Modelling techniques: inputs and outputs

- **Example:** inverter circuit



- Initial state: **i**=0, **o**=1
- Initially **i+** is enabled, i.e. the environment may (but does not have to) change the value of input to 1
- After **i+** fires, **o-** becomes enabled, i.e. the inverter must **eventually** change the value of its output to 0
- Meanwhile, the environment is obliged not to change the value of the input (no input transition is enabled) – it must wait for **o-** before doing that

79

## Modelling techniques: i/o and choices

- **Input / input** choices: usually appear due to abstraction of the environment
- **Example:** Vending machine models the user's behaviour as a *free choice* between chocolate and coke. The real user might well have no choice (e.g. wants a drink, not a snack). However, a detailed model of the user would be infeasible, so this free choice **overapproximates** the relevant (from the vending machine's point of view) part of user's behaviour.
- **Example:** Memory circuit handles read and write requests from the CPU. This can be modelled / overapproximated by a free choice, to avoid detailed modelling of the CPU, which might not have a choice what request to send.

80



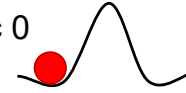
## Modelling techniques: i/o and choices

- **Output / output** choices: usually due to *arbitration* between clients contending for a shared resource, e.g.:
  - which of two threads gets hold of the mutex first?
  - which of the two requests for a device gets granted first?
  - should a signal from the environment be processed in the current clock cycle or in the next one?
- If clients' requests arrive too close in time, the system has to make an arbitrary decision (cf. *Buridan's ass*)
- In circuits, this leads to **metastability**, which cannot be resolved in bounded time

81

## Metastability

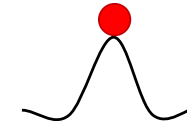
Stable:  
logic 0



Stable:  
logic 1



Metastable



82

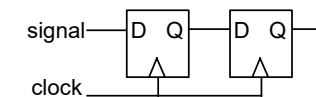
## Metastability

- Occasionally the system has to make an **arbitrary** decision, i.e. **either** alternative is acceptable, but a choice has to be made
- Metastability can persist for an indefinitely long time!
  - there is a non-zero probability that Buridan's ass will starve to death
- Issues:
  - though the probability of a long delay is small, when repeated sufficiently many times, a nasty scenario **will** happen, and **will** cause malfunction in some kinds of systems, in particular synchronous (clocked) circuits – e.g. when this delay gets longer than a clock cycle (MTBF can be calculated for such systems, and there are ways to trade off performance for MTBF)
  - need to contain metastability (which is analogue by nature) – it must not propagate to the digital part of the system!

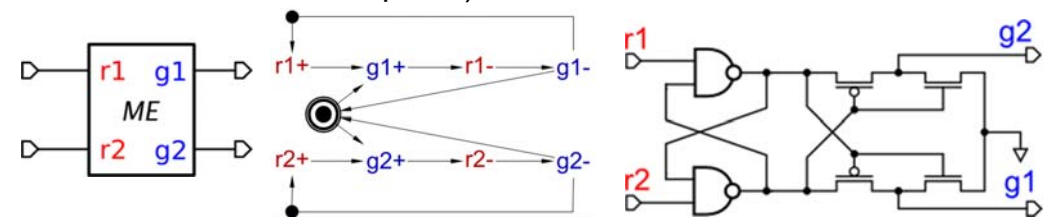
83

## Metastability in circuits

- Synchronous (clocked) circuits: need to arbitrate between a clock edge and an input from the environment; a **synchroniser** is used (may fail occasionally – MTBF):



- Asynchronous circuits: **Mutex Element** [Seitz, 1979] is used (never fails but may take indefinitely long to resolve, so try to remove from critical paths):



84

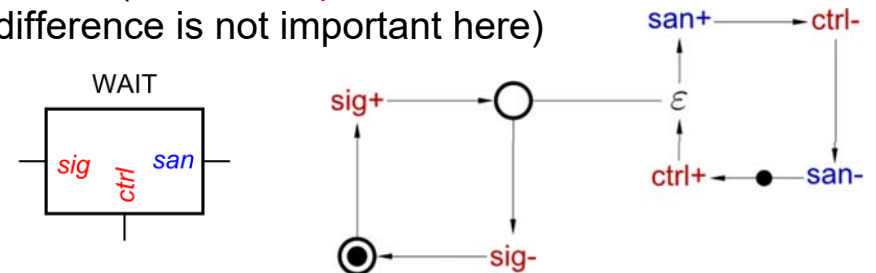
## Modelling techniques: i/o and choices

- **Input / output** choices: very problematic, usually indicate a mistake in the design or lack of relevant information / behaviour
- Intuitively, the system and the environment have to make a consistent decision in a distributed manner; this cannot be implemented without allowing for the possibility that both actions are performed!
- If such a decision is really necessary and the system collaborates with the environment then it can be delegated to one of the parties (or to a 3<sup>rd</sup> party), which will make a local decision (**output/output** choice) and notify the other party (which will see it as an **input/input** choice)

85

## Exception (?): WAIT element

- WAIT element has a read-consume **input / output** choice (or rather **input / internal** choice – but the difference is not important here)

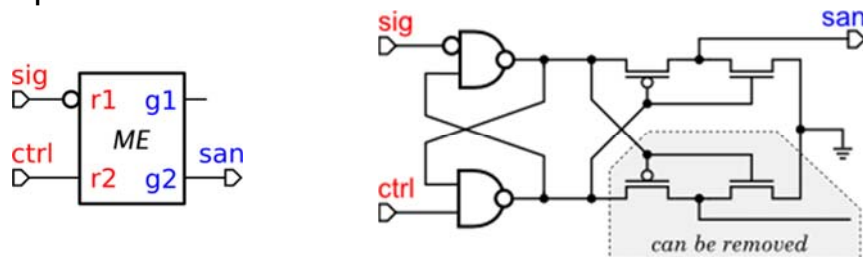


- Upon activation by **ctrl+**, waits for **sig**=1 (may ignore short spikes) and latches it as 'clean' **san**
- 'Clean' **ctrl** / **san** handshake controlled by 'dirty' **sig**

86

## Exception: WAIT element (cont'd)

- Implementation

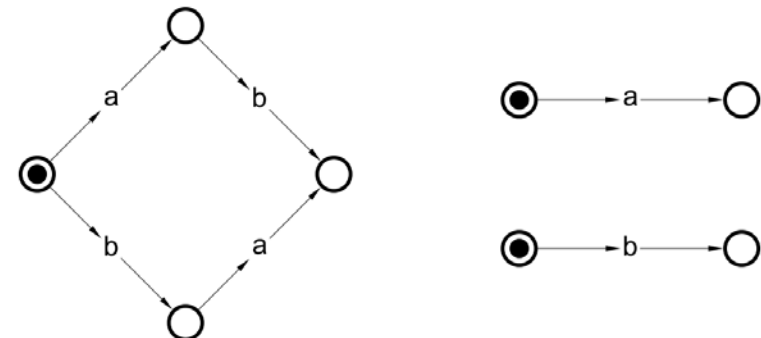


- The bubble on **sig** input can be detached as an inverter
- Early version with a NOR4 gate instead of 'contemporary' metastability filter: Fig 6b in J. Kessels and P. Marston: *Designing Asynchronous Standby Circuits for a Low-Power Pager*. Proceedings of the IEEE, Vol. 87, No. 2, 1999

87

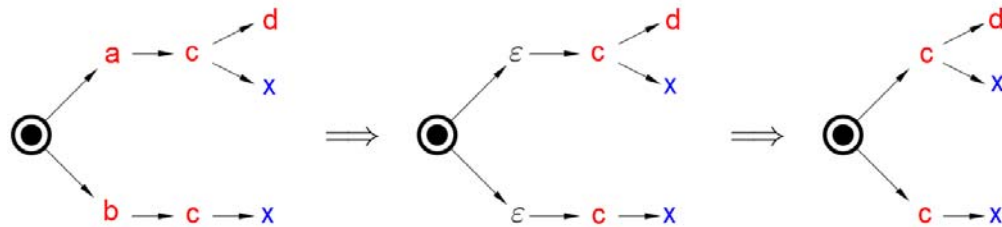
## Modelling techniques: structural choices

- Some structural choices are not semantical, e.g. due to concurrency being modelled by interleaving; such choices usually can (and should!) be eliminated:

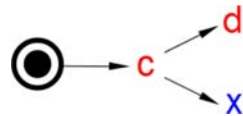


88

## Modelling techniques: non-deterministic choices (benign case)



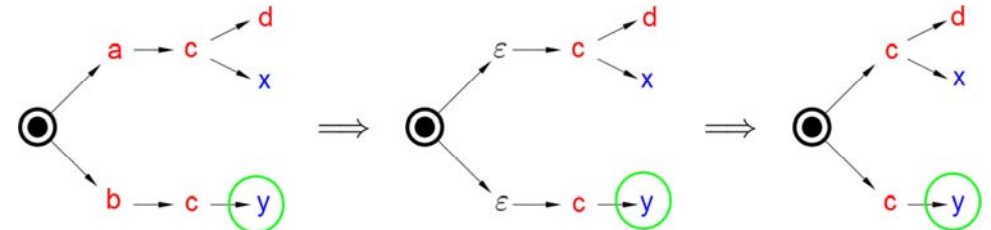
The lower branch is 'subsumed' by the upper one and so can be removed:



89

## Modelling techniques: non-deterministic choices (malignant case)

- If too much is hidden, cannot determinise / implement:



- In this case can hide either **a** or **b**, but not both, as these signals decide whether **x** or **y** is eventually output

90

## Modelling techniques: parallel composition

- **Complex systems are not monolithic:** they are designed by **composing** smaller blocks, which in turn are composed from even smaller blocks, etc.
- **Compositionality:** Semantics / behaviour of a composed system must be easily predictable from the semantics / behaviour of its constituent blocks
- **Examples:**
  - Producer/Buffer/Consumer system
  - Constructing an algebraic expression by e.g. adding two simpler expressions
  - Constructing a digital circuit by connecting smaller circuits with wires

91

## Modelling techniques: parallel composition

- **Problem statement:** Given several PNs that share actions, construct their **parallel composition** – a PN that models the overall behaviour, with the shared actions being executed jointly:

$$PN_1 \mid PN_2 \mid \dots \mid PN_k$$

- **Desired properties:** The order of composition should not matter, i.e. ' $\mid$ ' is commutative and associative:
  - $PN_1 \mid PN_2 = PN_2 \mid PN_1$
  - $(PN_1 \mid PN_2) \mid PN_3 = PN_1 \mid (PN_2 \mid PN_3)$
- Hence composing several PNs reduces to a number of binary compositions

92

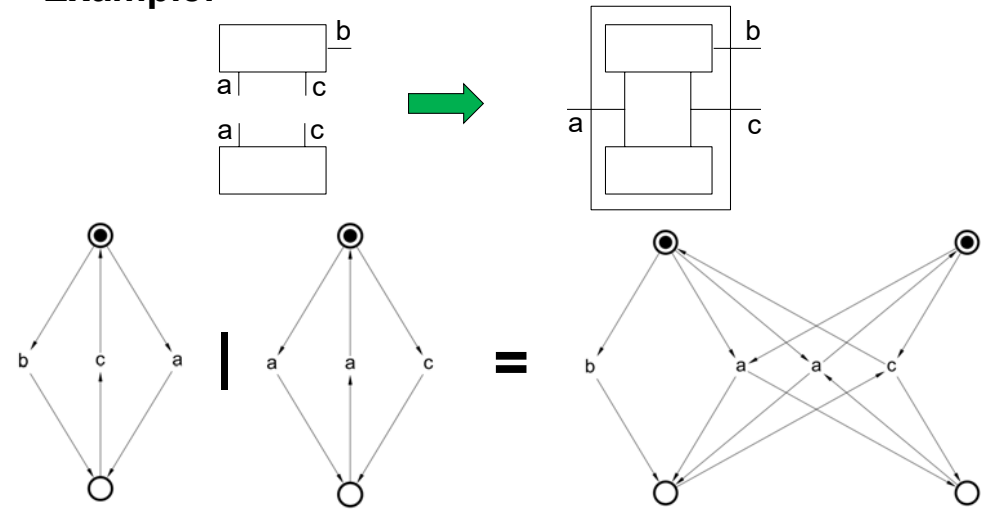
## Modelling techniques: parallel composition

- **Idea:** Glue transitions from different PNs that have the same label (if PNs have several transitions labelled with e.g. **a**, glue each such transition in  $PN_1$  with each such transition in  $PN_2$ )

93

## Modelling techniques: parallel composition

- **Example:**



94

## Modelling techniques: parallel composition

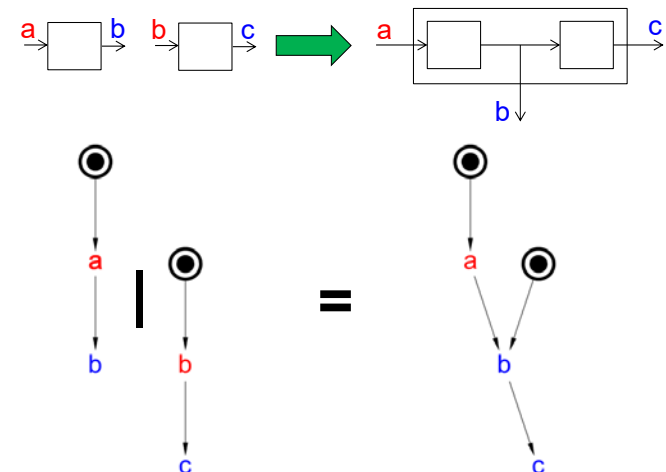
When actions are divided into **outputs** / **inputs** / **internal**:

- An action can be an **output** of at most one PN!
- The **outputs** of the composition are all the **outputs** of all the components
- An action can be an **input** of 0 or more PNs; gluing two **input** transitions produces an **input** transition
- An **output** of a PN can be an **input** of 0 or more other PNs; gluing an **input** transition with an **output** transition produces an **output** transition
- **Internal** actions cannot be shared (rename them if necessary to avoid name clashes), and so their transitions are never glued

95

## Modelling techniques: parallel composition

- **Example:**



96

## Reachability graph

All the reachable markings and firing sequences of a PN can be represented using its **reachability graph (RG)** – a labelled directed graph  $RG(PN) = (V, Arcs, v_0)$  such that:

- $V$  is the set of nodes: all reachable markings
- $Arcs$  is the set of labelled arcs: there is an arc labelled  $a$  from  $M$  to  $M'$  iff for some transition  $t$ ,  $M[t]M'$  and  $\ell(t)=a$
- $v_0$  is the initial node:  $M_0$

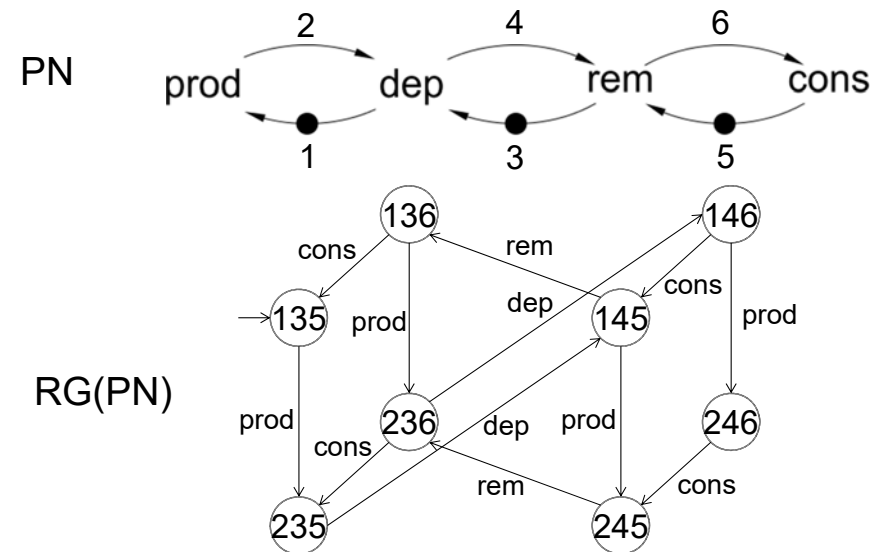
**Theorem:**  $fs$  is a firing sequence of PN iff there is a directed path from the initial node of  $RG(PN)$  such that  $fs$  is the sequence of arc labels encountered along this path

### Notes:

- RGs can be infinite. **Exercise:** give an example of a PN with infinite  $RG(PN)$
- if a RG is finite then it is an FSM
- RGs can be used to reason about behavioural properties of PNs

97

## Reachability graphs: Example



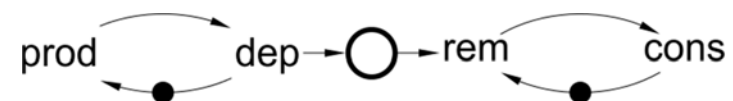
98

## Behavioural properties: Examples

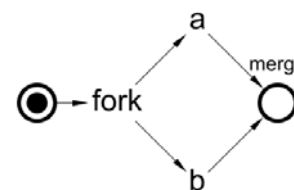
- A PN is called ***k*-bounded** (for a natural number  $k$ ) if it never puts more than  $k$  tokens on any of its places; a PN is called **bounded** if it is  $k$ -bounded for some  $k$
- PN is bounded iff its  $RG(PN)$  is finite
- Unboundedness indicates unlimited resources (e.g. a buffer of infinite capacity), or the possibility to create an arbitrary large backlog
- A PN is called **safe** if it is 1-bounded
- Many (but not all) models require the PN to be safe, i.e. unsafeness often indicates a bug in the design
- These properties can be checked during the construction of  $RG(PN)$

99

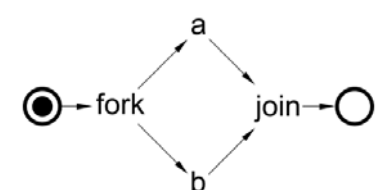
## Example: Producer/Buffer/Consumer



**Unbounded** – the buffer has infinite capacity and so cannot be implemented!



**Unsafe** – a merge after a fork is wrong!



**Safe** – a join after a fork is ok

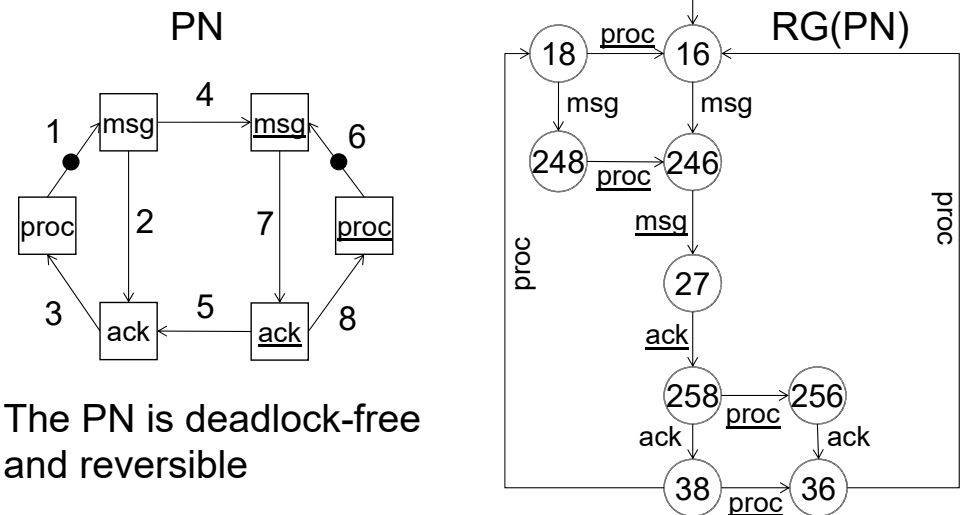
100

## Behavioural properties: Examples

- Clearly,  $M$  is a reachable marking of a PN iff it appears as a node in  $RG(PN)$
- A marking  $M$  of a PN is called a **deadlock** if it enables no transitions; PN is called **deadlock-free** if none of its reachable markings is a deadlock
  - PN is deadlock-free iff  $RG(PN)$  is deadlock-free, i.e. from each node in  $RG(PN)$  there is at least one exit arc
- PN is called **reversible** if  $M_0$  can be reached from any reachable marking  $M$ 
  - PN is reversible iff  $RG(PN)$  reversible, i.e. there is a directed path from any node of  $RG(PN)$  to  $M_0$

101

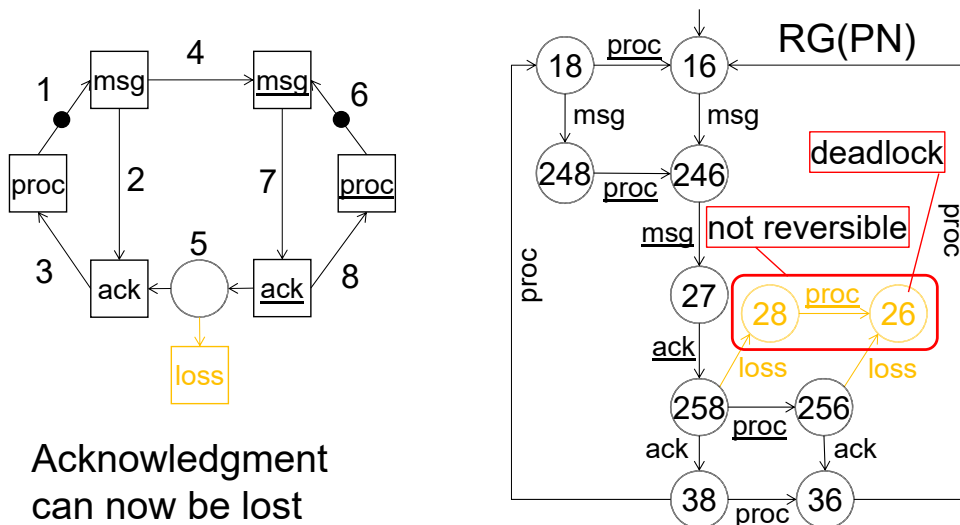
## Example: ping-pong protocol



The PN is deadlock-free and reversible

102

## Example: ping-pong protocol



Acknowledgment can now be lost

103

## Example: Money transfer

Recall the Transfer procedure that can deadlock. Can one automate deadlock checking?

**procedure** Transfer(accFrom, accTo, amount):

**acquire** accFrom.lock;

**acquire** accTo.lock;

**if** transfer\_is\_possible(accFrom, accTo, amount) **then**

accFrom.decrease(amount);

accTo.increase(amount);

**release** accFrom.lock;

**release** accTo.lock;

104



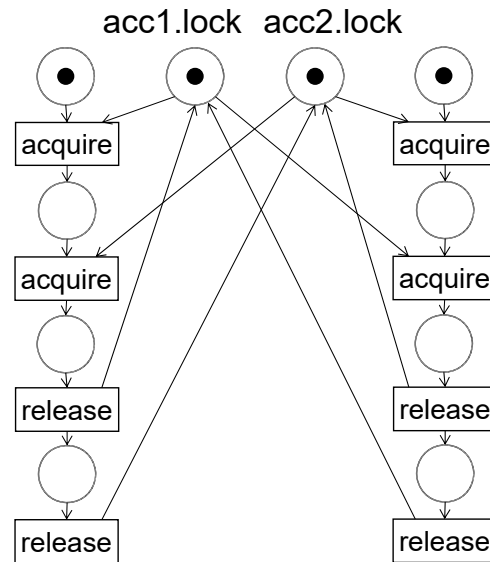
## Example: Money transfer

A Petri net model of concurrent execution of

Transfer(acc1,acc2,£x)

and

Transfer(acc2,acc1,£y)



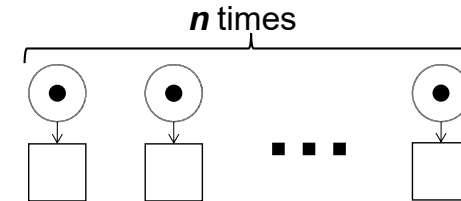
105

## Verifying concurrent systems

**Concurrent systems are difficult to analyse!**

The number of reachable states of a concurrent system can be **much** larger than the size of its specification, even if we ignore the data; **thus manual analysis is often infeasible**

**Example:**



- The system has  $2^n$  reachable states!

This phenomenon is called **state space explosion**; coping with it is the main challenge in formal verification

106

## Automated verification (model checking)

**Model checking** is an approach to automated formal verification; in 2008, its pioneers (E. Clarke, E.A. Emerson and J. Sifakis) won the ACM Turing award

The idea of MC is to inspect the reachable states (markings) of a system to check if some state fails to satisfy a given correctness criterion (e.g. it is a deadlock)

For a Petri net PN one can use the depth-first search (DFS) to generate all the markings reachable from the initial marking  $M_0$

107

## Building reachability graph

Explore( $M_0$ ) // top-level recursive call

**procedure** Explore(M): // Depth-First Search

**if** M is not in the storage **then**

    store M

**for all** firings  $M[t]M'$  **do**

      // there is a t-labelled arc from M to  $M'$

      Explore( $M'$ )

- We do not distinguish between a PN marking M and the FSM state corresponding to M
- Can use a hash table (or a specially designed data structure) to efficiently implement the storage for visited markings
- No need to store the arcs explicitly – the arcs originating from any marking M can easily be computed from the PN

108

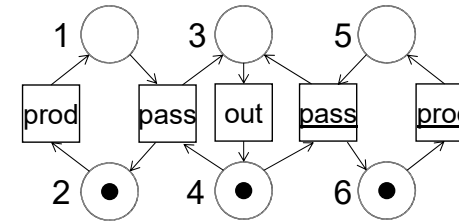
## Notes about reachability graph algorithm

- Can easily add error checking – if a newly generated M is erroneous (e.g. a deadlock) then stop immediately
  - the trace from  $M_0$  to M can be retrieved from the stack of recursive calls – useful for debugging
  - replace Depth-First Search by Breadth-First Search for a shortest trace, or by some kind of Guided Search for quickly finding an error
- The algorithm will work only on small examples due to State Space Explosion
- Improving the efficiency (hot research topic), e.g.:
  - condensed representations of RG
  - exploiting the structure of the PN, e.g. symmetries, invariants, modular representation, ...
  - generating reduced state graph which is still sufficient to check the property
  - approximations of RG

109

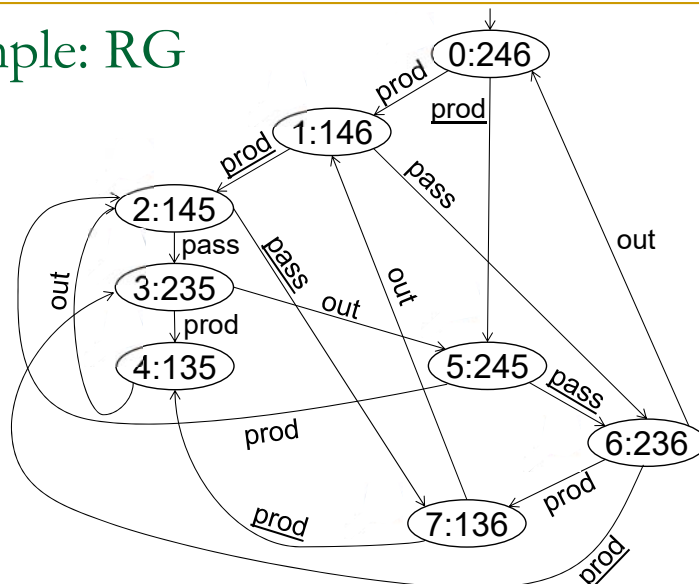
## Example

The PN below consists of three sub-systems: two producers and a merging process:



110

## Example: RG

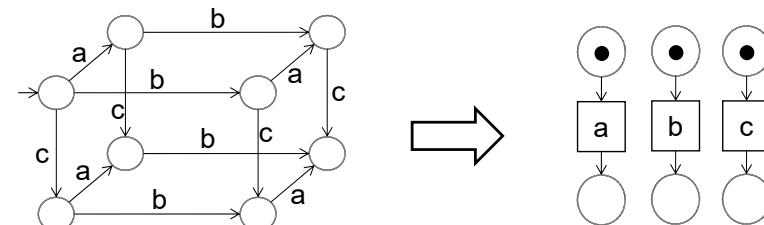


states are numbered according to their creation order

111

## PN Synthesis

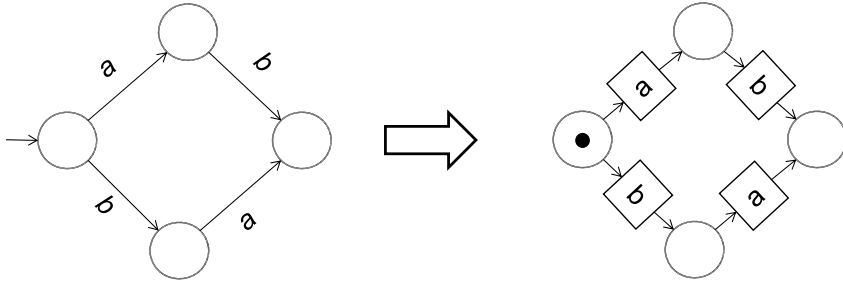
- Problem statement:** Construct a PN whose reachability graph coincides with a given FSM; the PN is usually required to be from some class, e.g. safe
- Motivation:** PNs are often more compact than FSMs, and can represent concurrency naturally rather than by interleaving, so easier for humans to comprehend:



112

## PN Synthesis: naïve approach

- FSMs can be considered as PNs where each transition has one incoming and one outgoing arcs and the initial marking consists of one token; hence can replace states with places (with a token on the initial one) and arcs with transitions:

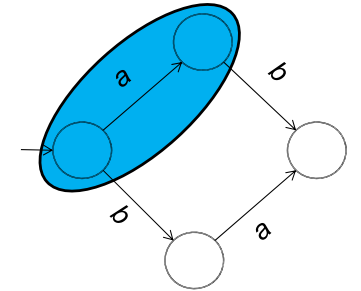


- This works but defeats the original motivation ☹

113

## PN Synthesis: approach based on regions

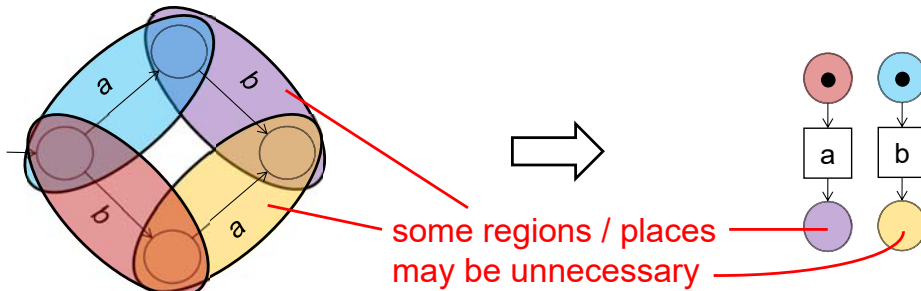
- Idea:** identify **regions** – sets of states with which all occurrences of each event have the same **crossing relationship** (enter, exit, non-crossing, ...); e.g. the following set of states is a region that is **non-crossing** wrt. **a** and **exit** wrt. **b** (the precise definition of a region depends on the PN class and on the behavioural equivalence one would like to achieve)



114

## PN Synthesis: approach based on regions

There are 4 non-trivial regions in this example ( $\emptyset$  and the set of all states are the **trivial regions**); for each of them one can create a place whose entering and exiting transitions are determined by the crossing relationships; the place is initially marked if the region contains the initial state:



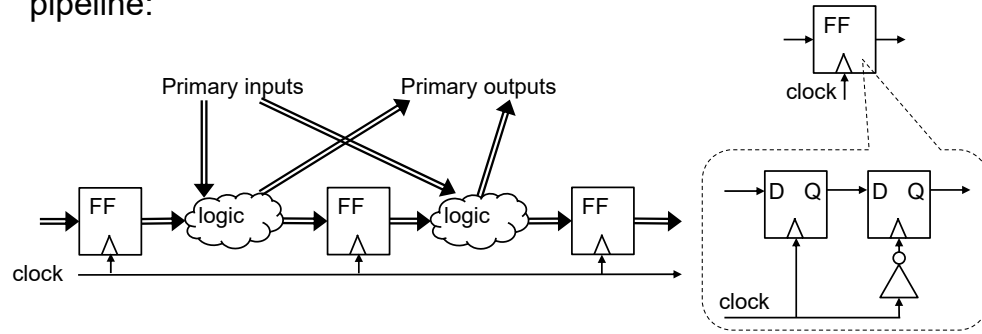
115

## Concurrency in Electronic Circuits

116

## Synchronous (clocked) circuits

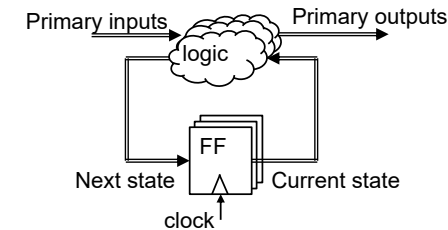
- Usually the **Register-Transfer Level (RTL)** abstraction is used in design – supported by **Hardware Description Languages** like Verilog, VHDL, etc.
- Two main components: flip-flops (memory) and combinational (i.e. acyclic & memoryless) Boolean logic, together forming a pipeline:



117

## Synchronous (clocked) circuits

- Folded view:



- Operation:

- Rising clock edge: the former latch of FF stores the next state computed in the previous clock cycle, which is assumed to be stable by this time
- Falling clock edge: the latter latch of FF copies the state from the former latch, causing the following logic to start computing the next state
- The clock period is chosen to be sufficiently long for the logic to finish its computation by the next rising edge of the clock

118

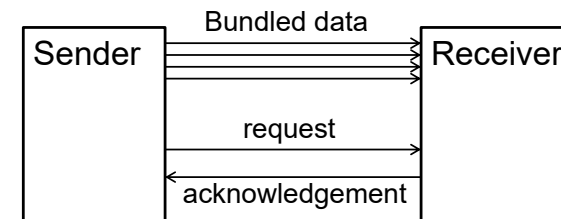
## Synchronous (clocked) circuits

- Pros:
  - Greatly simplifies the design
- Cons:
  - Latency**: response time is several clock periods
  - Variability (manufacturing, temperature, voltage)**: has to be addressed by increasing the clock period, deteriorating the performance
  - Clock skew**: in large high-frequency designs there is a noticeable difference in arrival time of the clock signal to the different parts of the silicon die – great care has to be taken to avoid malfunction
  - Reliability**: failures when synchronising external inputs – the circuit is effectively a hard-real-time system
  - Power**: clock consumes up to 45% of power; ticks even when there is no data to process (can be mitigated by *clock gating*)
  - Electromagnetic interference**: clock is a strong high-frequency signal, so can interfere with other signals, impose shielding requirements, etc.

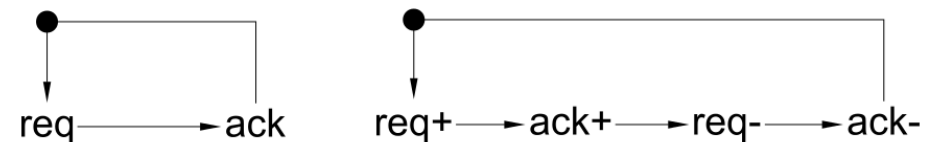
119

## Asynchronous circuits

- No clock, time becomes elastic
- Handshakes** are used for synchronisation



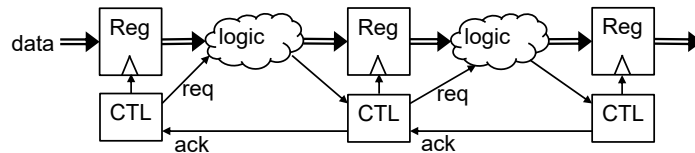
- 2-phase and 4-phase handshaking protocols



120

## Asynchronous pipeline

- Global clock is replaced by small blocks providing local “clocks” controlling the registers
- Completion detection is provided by logic – hence average-case rather than worst-case performance
- Asynchronous circuits are much more difficult to design, but they allow for more flexibility and alleviate the problems associated with the global clock

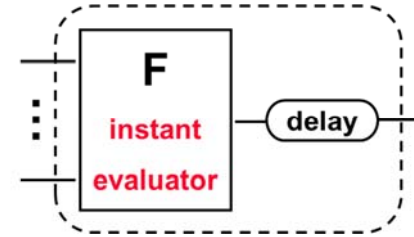


121

## Asynchronous styles

- There are many styles that differ by their delay models, e.g. [quasi-]delay-insensitive ([Q]DI), speed-independent (SI), fundamental-mode, burst-mode, etc.; we focus on SI≈QDI

- Gates/latches are **atomic** (in particular, no internal glitches):



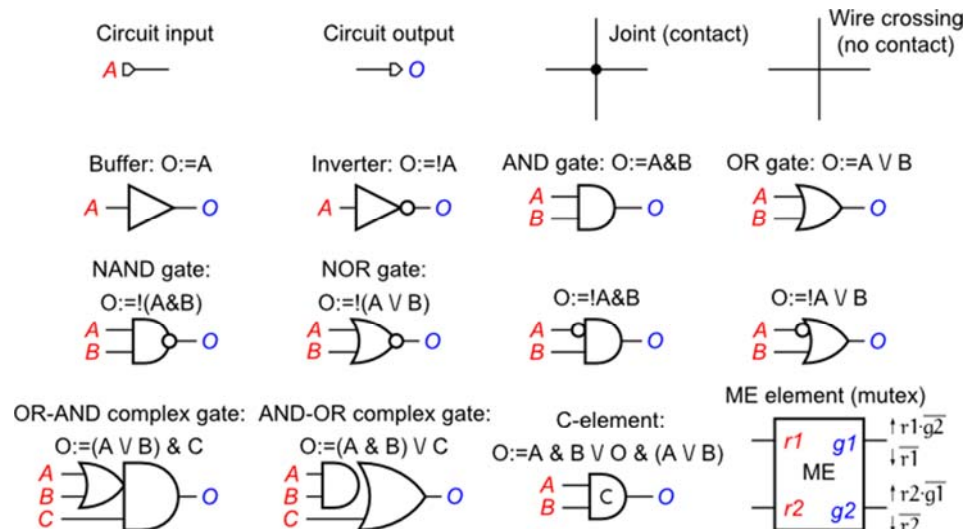
Good citizens: **unate** gates/latches, e.g. BUF, NOT, AND, OR, NAND, NOR, AND-OR, C-element,...

Suspects: **binate** gates, e.g. XOR, MUX, D-latch – may have internal glitches, but may still be useful

- Gate delays are positive and finite, but variable and unbounded
- Wire delays are negligible (SI), or [some] wire forks are **isochronic** (QDI), i.e. wire delays can be added to gate delays

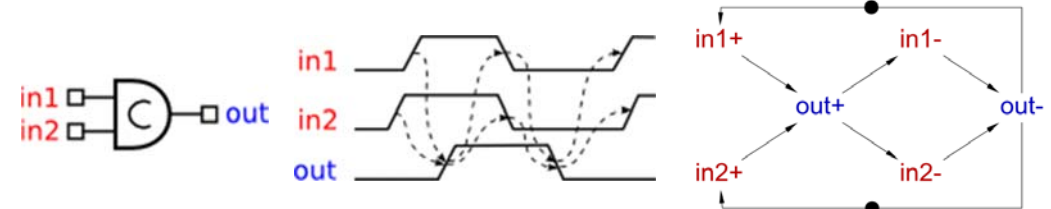
122

## Notation: Gates and latches



123

## Example: C-element

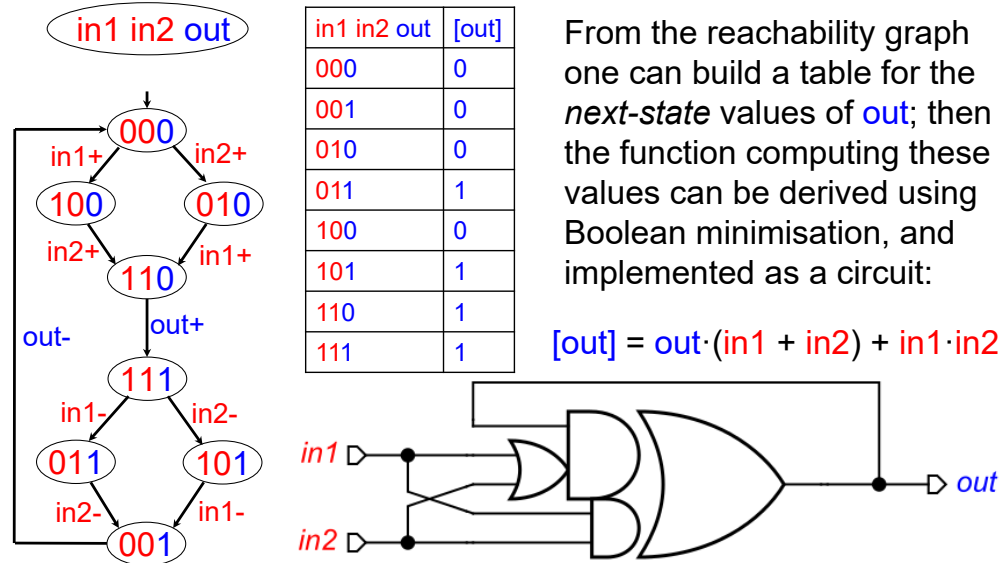


Synthesise a circuit with the following behaviour:

- if both inputs are 0 then the output should switch to 0
- if both inputs are 1 then the output should switch to 1
- otherwise the output should maintain its current value (i.e. the output is not completely determined by the current inputs – the circuit has **memory**)
- assumption:** the environment ‘fulfils its contract’ specified by the above PN (**GIGO principle** – **G**arbage **I**n, **G**arbage **O**ut)

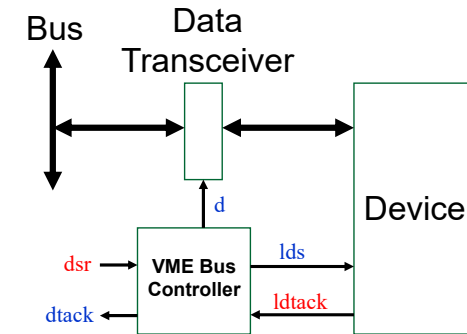
124

## Example: Reachability graph / synthesis



125

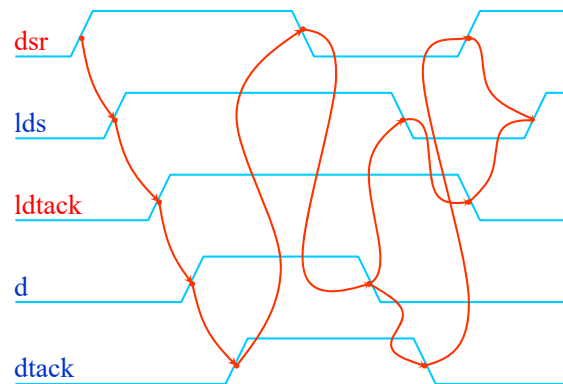
## Example: VME bus controller (read cycle)



**Objective:** synthesise an asynchronous controller decoupling Device from Bus, thus allowing higher degree of concurrency between them; in particular, allow Bus to send new requests while Device is still resetting

126

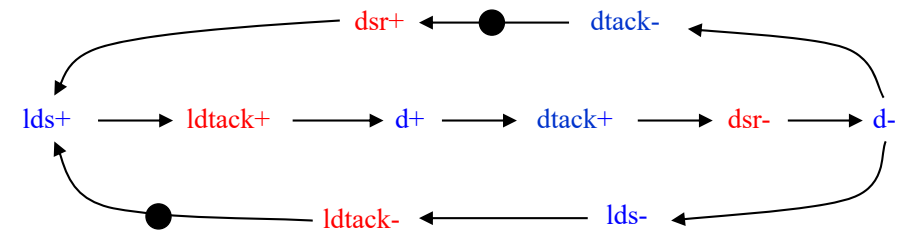
## Example: Timing diagram



**Timing diagrams** are traditionally used by h/w engineers. They are informal, and may miss some important scenarios. We use Petri nets instead to formally specify the behaviour.

127

## Example: Petri net specification

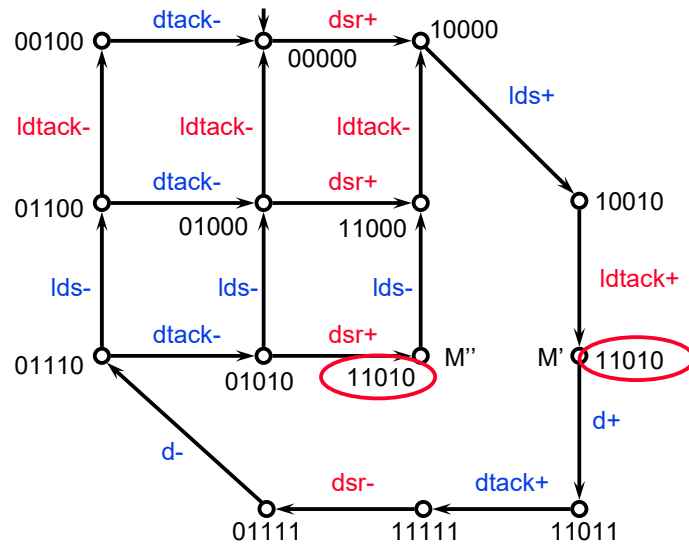


- This PN model captures both the **behaviour of the circuit** and the **assumptions about its environment**, and contains sufficient information to **synthesise** a circuit
- The model is **executable**, i.e. we can simulate various scenarios by executing this PN and thus **test / validate** the model
- Can **formally verify** various correctness properties on this model, e.g. deadlock-freeness and **output-persistence**

128



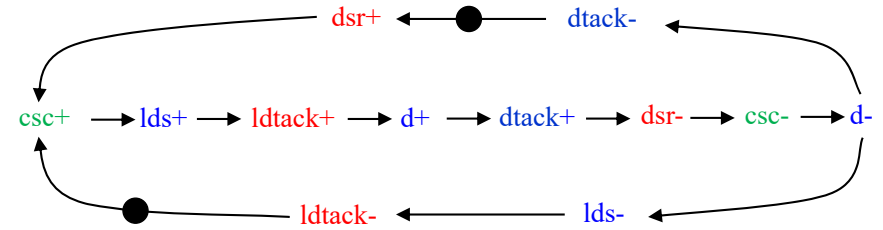
## Example: Reachability graph / CSC conflict



Two reachable states are in **Complete State Coding (CSC) conflict** if their encodings are the same but they enable different sets of output signals. CSC conflicts must be resolved before logic synthesis.

129

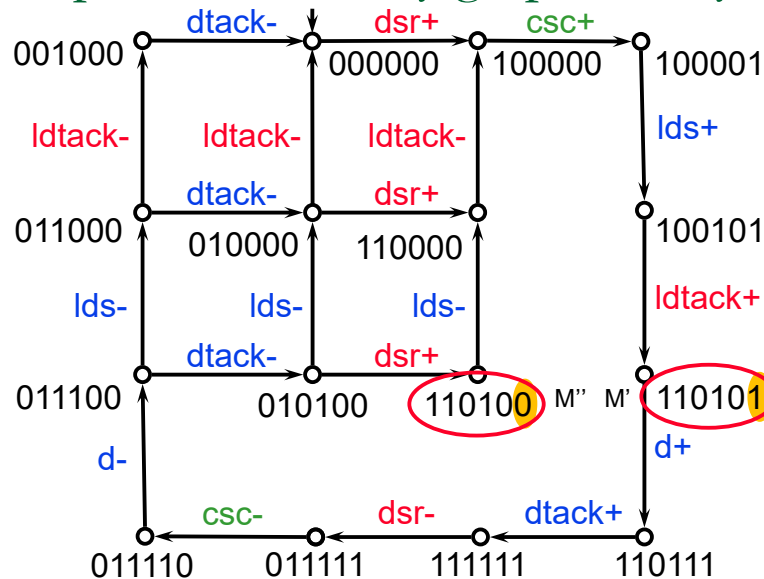
## Example: Resolving the CSC conflict



- **Idea:** Insert a new **internal** signal **csc** to help the circuit trace its state, i.e. memory is added
- **Note:** This signal is not visible to the environment, so its transitions must not **trigger** (i.e. directly enable) or disable input transitions – this corresponds to the **input-properness** property
- **Note:** Resolution of CSC conflicts can be automated

130

## Example: Reachability graph satisfying CSC



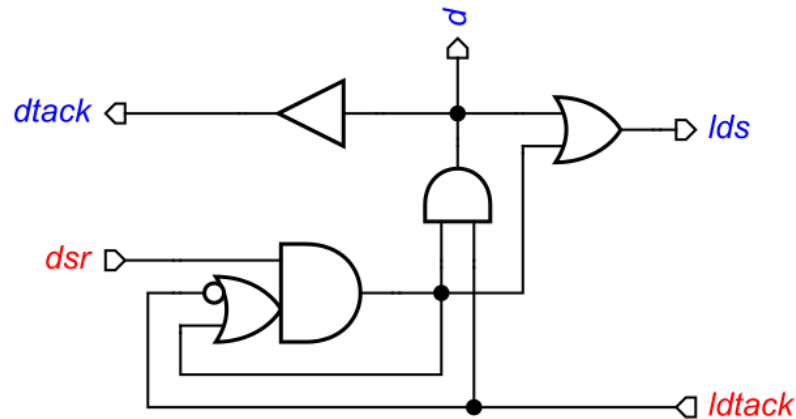
131

## Example: Logic synthesis

Code	[dtack]	[lds]	[d]	[csc]
001000	0	0	0	0
000000	0	0	0	0
100000	0	0	0	1
100001	0	1	0	1
011000	0	0	0	0
010000	0	0	0	0
110000	0	0	0	0
100101	0	1	0	1
011100	0	0	0	0
010100	0	0	0	0
110100	0	0	0	0
110101	0	1	1	1
011110	1	1	0	0
011111	1	1	1	0
111111	1	1	1	1
110111	1	1	1	1
unreachable	any	any	any	any
<b>Implementation</b>	<b>d</b>	<b>d+csc</b>	<b>csc·ldtack</b>	<b>dsr·(-ldtack+csc)</b>

132

## Example: Resulting circuit



133

## Gate-level modelling

**Problem statement:** model the given digital circuit as a PN

- Can represent each signal  $s$  by two places,  $p_{s=0}$  and  $p_{s=1}$ ; exactly one of these places is marked at any time, representing the current value of  $s$
- Since there is no information about the environment's behaviour, it is taken to be the most general (i.e., it can always change the value of any input); this is modelled for each input signal  $i$  by adding transitions  $i+$  (with the arcs  $p_{i=0} \rightarrow i+ \rightarrow p_{i=1}$ ) and  $i-$  (with the arcs  $p_{i=1} \rightarrow i- \rightarrow p_{i=0}$ )

134

## Gate-level modelling (continued)

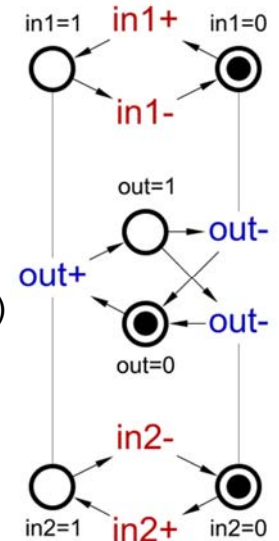
- For each output (or internal) signal  $o$  with the next-state function  $[o]=E$  we compute the **set** and **reset** functions as  $[o\uparrow]=E|_{o=0}$  and  $[o\downarrow]=\neg E|_{o=1}$ ; they should be represented in the minimised **disjunctive normal form (DNF)**
- For each term  $m$  of the set function we add a transition  $o+$  and arcs  $p_{o=0} \rightarrow o+ \rightarrow p_{o=1}$ . For each literal  $s$  (resp.  $\neg s$ ) in  $m$ , we connect  $o+$  to the place  $p_{s=1}$  (resp.  $p_{s=0}$ ) by a pair of arcs going in opposite directions (to test for the presence of a token on a place without consuming it).
- A similar process is used to define the transitions  $o-$  using the reset function

135

## Gate-level modelling: Example (AND-gate)

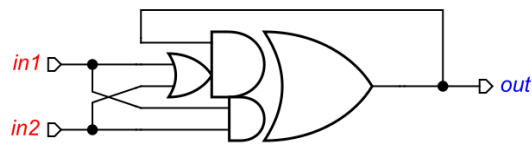
$$\begin{aligned}
 [out] &= in1 \cdot in2 & \text{in1} \quad \text{in2} \quad \text{out} \\
 [out\uparrow] &= in1 \cdot in2 & \text{in1} \quad \text{in2} \quad \text{out} \\
 [out\downarrow] &= \neg(in1 \cdot in2) = \neg in1 + \neg in2
 \end{aligned}$$

- Output-persistence** is violated, e.g. after  $in1+ in2+$  output  $out+$  can be disabled by either  $in1-$  or  $in2-$ , which leads to a non-digital pulse on  $out$
- This is because the circuit (and thus this PN) **lacks information about the environment's behaviour!**
- The output-persistence holds in some restricted environments (e.g. VME bus controller has an AND gate)



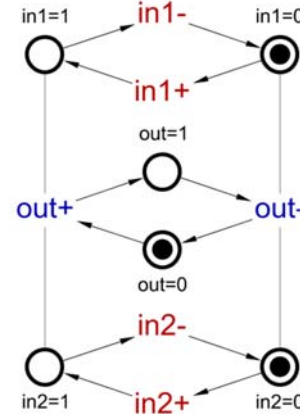
136

## Gate-level modelling: Example (C-element)



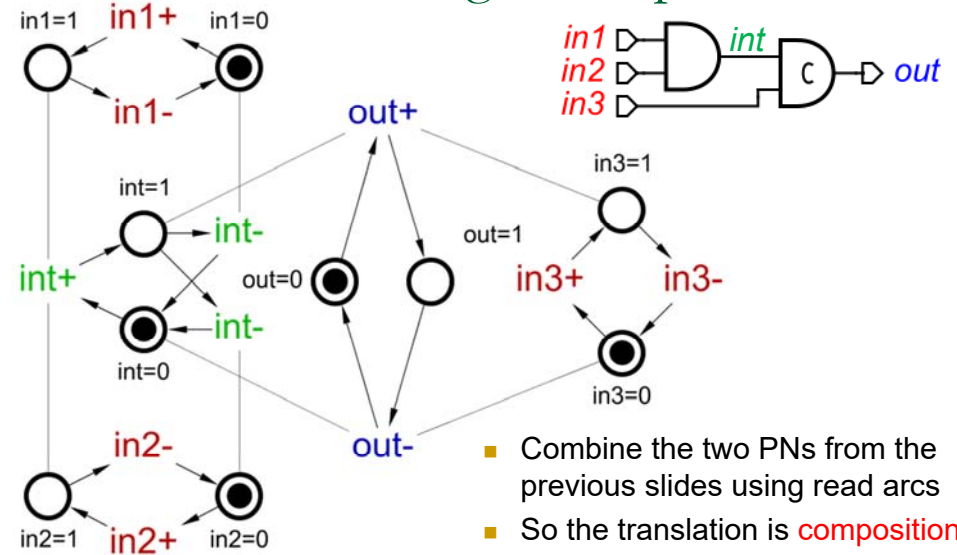
$$\begin{aligned}
 [out] &= out \cdot (in1 + in2) + in1 \cdot in2 \\
 [out\uparrow] &= 0 \cdot (in1 + in2) + in1 \cdot in2 = in1 \cdot in2 \\
 [out\downarrow] &= \neg(1 \cdot (in1 + in2) + in1 \cdot in2) = \\
 &= \neg(in1 + in2 + in1 \cdot in2) = \neg(in1 + in2) = \\
 &= \neg in1 \cdot \neg in2
 \end{aligned}$$

- This PN has more behaviour than the specification of C-element
- **Output-persistence** is violated, e.g. after  $in1+$  or  $in2+$  output  $out+$  can be disabled by either  $in1-$  or  $in2-$ , which leads to a non-digital pulse on  $out$
- This is because the circuit (and thus this PN) lacks information about the environment's behaviour!
- The circuit works correctly (**output-persistence + conformation**) in an environment that fulfils the original contract



137

## Gate-level modelling: Example AND-C



- Combine the two PN from the previous slides using read arcs
- So the translation is **compositional**

138

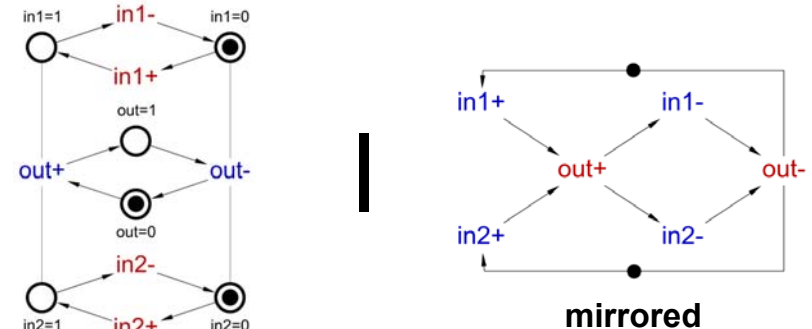
## Gate-level modelling: Verification

- Gate-level circuit has no information about its environment, so naïve verification will **almost always** reveal violations of output-persistence
- Hence need to supply the environment's behaviour (i.e. the original PN spec) during verification: **Assuming the environment fulfils the contract**, the circuit should satisfy the following standard properties:
  - **output-persistence**: no output or internal signal can be disabled by another signal (except in special elements like **mutex** or **WAIT** designed to handle non-persistence); otherwise non-digital pulses may be produced
  - **conformation**: the circuit does not break its environment by producing an unexpected output, i.e. the circuit must fulfil its part of the contract
  - **input-properness**: an internal signal cannot trigger or disable an input (note that the environment is oblivious to the internal signals)
  - **deadlock-freeness**
- Additional standard or custom properties may be required

139

## Gate-level modelling: Verification

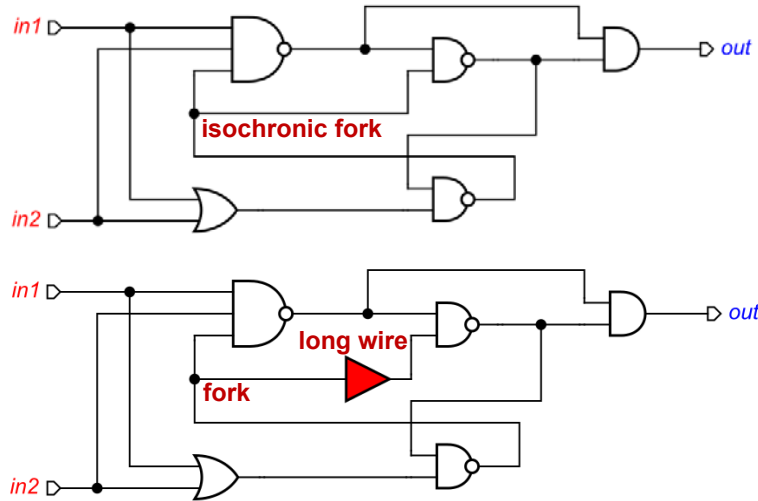
- **Problem**: how to restrict the behaviour of the circuit by the behaviour of the environment to verify the properties?
- **Idea**: can use the parallel composition! First, convert the circuit to a PN, and then compose this PN with the **mirror** (i.e. inputs and outputs are swapped) of the original PN spec:



140

## Example: Mayevsky's C-element

Can a C-element be implemented by the following circuits?



141

## Concurrent computation

142

## Amdahl's law

- Limits the theoretical maximum speedup due to parallelisation
- **Observation:** usually only some parts of the program can be parallelised, i.e. a parallel program usually contains some sequential parts
- Let  $0 < s \leq 1$  be the fraction of time the sequential program spends in its non-parallelisable part and  $n$  be the number of processors, then

$$\text{Speedup} = \frac{1}{s + \frac{1-s}{n}} < \frac{1}{s}$$

- Hence the speedup is bounded, no matter how many processors are deployed!

143

## Amdahl's law: example

- Suppose a digger has to dig a 9 metres long trench, and it takes an hour per metre. Moreover, it takes an hour to negotiate the driver's wage. The overall sequential time is then  $1 + 1 \cdot 9 = 10$  hours.
- Suppose now we have an unbounded number of diggers (it still takes an hour to negotiate wages with the drivers' rep). Amdahl's law says that the maximum speedup is smaller than  $1/0.1 = 10$ , as the fraction of time spent in the sequential part of the process (negotiation phase) is 0.1, and the overall time is always greater than an hour (the length of the sequential negotiation phase).

144

## Gustafson's law

- Amdahl's law: considers a problem of fixed size
- **Observation:** In practice, the problem size is usually set to use the available processing power within some fixed time, i.e. if more processing power is available then larger problems can be solved in the same time
- Let  $s'$  be the sequential fraction of the overall execution time of a parallel program, and  $n$  be the number of processes, then
$$\text{Speedup} = n - s'(n - 1)$$
- Thus, the speedup is unbounded if the problem is scaled (provided  $s' < 0.99$ )

145

## Gustafson's law: example

- Suppose the setup is the same as in the example for Amdahl's law, i.e. for one digger it takes  $1 + 1 \cdot 9 = 10$  hours to dig a 9 metres trench.
- Now suppose one would like to dig as long a trench as possible by deploying more diggers, but the time is limited to 10 hours. By Gustafson's law:  $s' = 1/(1+9) = 0.1$  and the speedup for  $n$  diggers is  $n - 0.1(n - 1) = 0.9n + 0.1$ .
  - e.g. for  $n=10$  the formula gives the speedup of 9.1. Note that the length of the trench dug in 10 hours is 90 metres, and for a single digger it would take 91 hours, i.e. the speedup is  $91/10 = 9.1$ , which is consistent with the formula.
- Thus **the increase in speedup is unbounded**, and so an arbitrarily long trench can be dug in 10 hours by deploying sufficiently many diggers!

146

## Flynn's taxonomy

- Classification of computer architectures

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

- **Single Instruction, Single Data stream (SISD)** – the usual sequential execution
- **Single Instruction, Multiple Data stream (SIMD)** – e.g. GPU
- **Multiple Instruction, Single Data stream (MISD)** – rare, occasionally used for fault tolerance
- **Multiple Instruction, Multiple Data stream (MIMD)** – e.g. multicore, distributed (network of workstations, Cloud)

147

## Memory: shared vs. distributed

- Classification of computer architectures
- **Shared memory:** values stored by any processor are visible to all processors; e.g. multicore
  - Less difficult to program
  - Truly concurrent memories are expensive
  - Cache coherence and instruction re-ordering issues – see *weak memory models*, e.g. <http://preshing.com/20120930/weak-vs-strong-memory-models>
- **Distributed memory:** each processor has private memory and communicates with the other processors by sending messages (usually over a network); e.g. NoW, Cloud
  - Hardware is cheaper
  - More difficult to program
  - Not always possible to partition the data efficiently
- **Hybrid architectures:** e.g. a network of multicore PCs

148

## Dependencies & parallelisation

- Dependencies in the code hinder parallelisation; some kinds of dependencies can be eliminated
- **Data dependencies:** reordering of instructions affects the final values of variables

$x := 0$	$x := 0$	$x' := 0$
$y := x + 1$	$x' := x$	$y := x' + 1$
$z := y + 1$	$y := x' + 1$	$x := 1$
	$x := 1$	

read-after-write    write-after-read    write-after-write

- **Control dependencies:** an instruction can be executed or not depending on the result of a preceding instruction

```
if x < 0 then y := -x
```

149

## Types of concurrency: Basic

- Have several tasks that are largely independent (apart from occasional synchronisation) and so can be executed in parallel
- **Example:** Digging a long trench using several diggers
- Common in software development:
  - Operating System running multiple processes
  - Server spawning a new session process for each client
  - Multiple threads working on sub-tasks within the same processes (**thread pools** are commonly used)

150

## Types of concurrency: Pipeline

- Items of work arrive sequentially
- Each item is processed in stages, with the stages being concurrent to each other
- **Examples:** RISC instruction pipeline, graphics pipeline in GPU, protein manufacturing in living cells, etc.
- Common in hardware, manufacturing, biological systems, etc., but used also in software – e.g. some OSes allow one to create ‘pipes’ between processes

151

## Types of concurrency: Speculative execution

- **Idea:** do the work **before** it is known whether it has to be done or not, using spare processing capabilities which otherwise would have been idle
- If the work turns out to be necessary, can get the result quicker; otherwise, the result is discarded
- Can cancel a speculative branch as soon as it is found to be unnecessary, to free up its resources, save power, etc.
- **Example:** can concurrently compute `slowCheck()` and one or both of `doSomething()` and `doSomethingElse()`; after the result of `slowCheck()` becomes known, one of the branches is cancelled, but some performance gain is likely

```
if(slowCheck()) doSomething();  
else doSomethingElse();
```

152



## Race conditions and synchronisation

- **Race condition**: a resource is concurrently accessed in a 'bad' way (the definition of 'bad' depends on the resource), e.g.:
  - several threads accessing the same memory location, with at least one of them writing to it
  - several threads concurrently printing text to a console
  - several threads accessing the same file
    - may or may not be ok if all the threads are readers
    - advanced file systems allow concurrent access to a file, even with writing, provided the threads work with different records in the file (used by databases)
- Race conditions result in 'undefined behaviour'
- **Synchronisation** is used to prevent race conditions: the resource is 'protected' (in a weak sense) by some synchronisation primitive (lock/mutex, semaphore, etc.), and a process should 'ask for permission' before accessing the resource

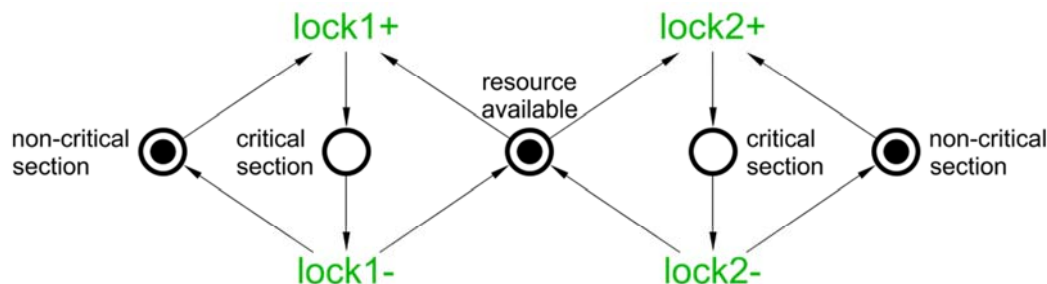
153

## Sync primitives: Lock (a.k.a. mutex)

- Used to 'protect' resources that must be accessed by at most one process at a time, i.e. in a *mutually exclusive* way
- **Idea**: each process must **acquire** the lock before accessing the resource (race condition if they don't), and **release** it afterwards (deadlock or livelock if they don't)
- The **acquire** operation may block the process for some time, e.g. if another process is using the resource or wins arbitration
- Lock operations are expensive: cause contention, require system calls, contain **memory barriers** to ensure **cache coherence** (i.e. consistent view of memory state) and prevent CPU out-of-order execution and compiler reordering optimisations
- There are many kinds of locks: spin vs. sleep locks, extra operations like try\_acquire, re-entrant locks, time locks, etc.

154

## Sync primitives: Lock (a.k.a. mutex)



155

## Sync primitives: Lock (a.k.a. mutex)

Issues to take care of when using locks:

```
procedure Transfer(accFrom, accTo, amount):
    // 1. what if accFrom==accTo?
    // 2. can deadlock
    acquire accFrom.lock;
    acquire accTo.lock;
    // 3. what if canTransfer throws an exception?
    if canTransfer(accFrom, accTo, amount) then
        accFrom.amount -= amount;
        accTo.amount += amount;
    release accFrom.lock;
    release accTo.lock;
```

156



## Sync primitives: Lock (a.k.a. mutex)

**Multiple acquires on the same lock:** easy to prevent – just do a check or ensure that such a situation is impossible (or use re-entrant locks – but they may be less efficient):

```
procedure Transfer(accFrom, accTo, amount):  
    if(accFrom==accTo){ special_processing(); return; }  
    acquire accFrom.lock;  
    acquire accTo.lock; // still can deadlock  
    ...
```

```
procedure Transfer(accFrom, accTo, amount):  
    assert (accFrom!=accTo); // assume cannot happen  
    acquire accFrom.lock;  
    acquire accTo.lock; // still can deadlock  
    ...
```

157

## Sync primitives: Lock (a.k.a. mutex)

- **Preventing deadlocks when acquiring multiple locks:** acquire the locks in a consistent order, e.g. according to their memory addresses, or better use a special function that does it for you (e.g. `std::lock` or `std::scoped_lock` in C++)
- Releasing the locks is a non-blocking operation and so can be done in any order

```
procedure Transfer(accFrom, accTo, amount):  
    assert (accFrom!=accTo); // assume accounts are distinct  
    if(&accFrom<&accTo){  
        acquire accFrom.lock;  
        acquire accTo.lock;  
    }  
    else{  
        acquire accTo.lock;  
        acquire accFrom.lock;  
    }  
    ...
```

158

## Sync primitives: Lock (a.k.a. mutex)

- **Preventing deadlocks due to an exception in the critical section**

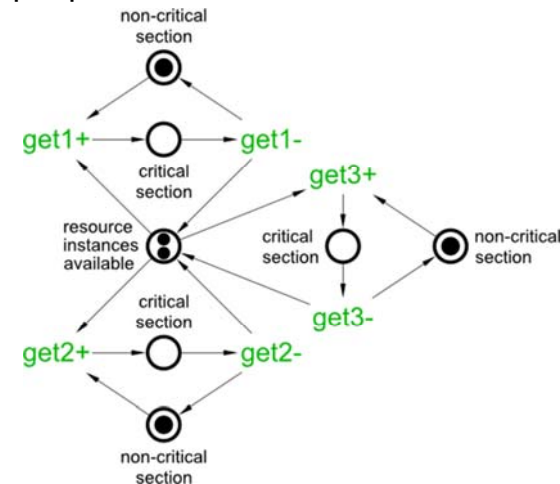
- Java: **synchronised** blocks ensure this; can also use `try{...} finally{...}` (works, but ugly)
- In C++ can use the **RAII idiom** (Resource Acquisition Is Initialization):

```
mutex lock;  
{ // start a scope  
    // constructor of lock_guard acquires the lock  
    lock_guard<mutex> _(lock);  
    ... // critical section  
    // destructor of lock_guard is automatically  
    // called when exiting the scope (even if  
    // the critical section throws an exception!)  
    // releasing the lock  
}
```

159

## Sync primitives: Semaphore

- Like a lock, but allows up to N processes to access a resource, e.g. up to 8 people in a lift



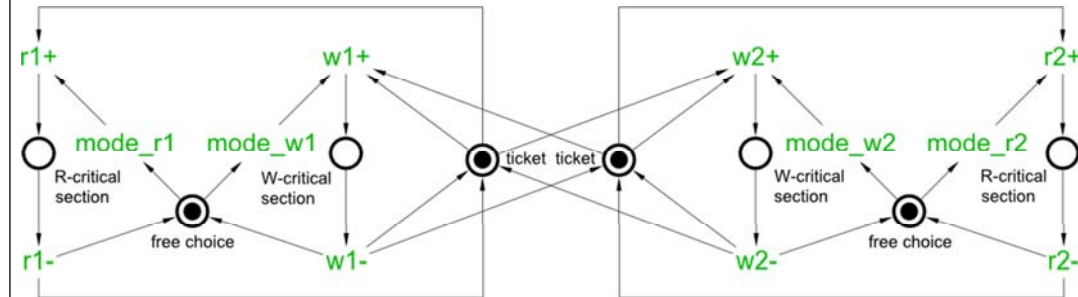
160

## Sync primitives: Read/Write lock

- Used to protect resources that can be read concurrently, but written exclusively
- **Idea:** each process, when acquiring the lock, specifies whether it is for writing or for reading; read/read concurrency is allowed, but read/write and write/write concurrency is forbidden
- **Idea:** Suppose there are N processes, each can be either a reader or a writer
  - we create N 'tickets'
  - a reader needs one ticket to enter the critical section
  - a writer needs all N tickets to enter the critical section

161

## Sync primitives: Read/Write lock



162

## Sync primitives: Read/Write lock

- RW locks are prone to **writer starvation** – the scenarios when there are always some readers, and hence the writer is blocked for long time or even indefinitely
- Many implementations are designed specifically to prevent such scenarios, e.g. new readers can be blocked if there is a waiting writer, i.e. the writer first acquires a write lock (competing with the other writers), sets a flag that there is a waiting writer (thus blocking new readers), and waits for the current readers to leave the critical section

163

## Sync primitives: Atomics

- Contemporary CPUs have instructions that perform non-blocking atomic operations with small (up to 8 bytes) objects (e.g. bool, int, pointers, etc.): read, write, swap, compare-and-swap, test-and-set, increment, etc.
- The programmer can specify the ordering constraint
- Basis of **lock-free algorithms**
- More details:

<http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2>

<http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-2-of-2>

164

# Pipelines

165

## What is a pipeline?

- A set of processing **stages** connected in series, i.e. a stage feeds into the next stage
- Stages work concurrently
- Process a stream of incoming 'items', producing an stream of outgoing 'items'
- May be re-configurable and have feedbacks and bypasses
- **Examples:**
  - assembly line
  - RISC instruction pipeline
  - digital circuits: synchronous (RTL) or asynchronous

166

## Abstract model of a pipeline

- Abstract away the nature of items being processed and the operations performed on them – items are modelled by **tokens**, and stages are modelled by 'boxes' that can hold a single token for some period of time
- Tokens must not collide
- Tokens normally do not appear/disappear spontaneously (at the ends of the pipeline tokens move into / out of the environment), though there are exceptions
- This correct operation can be achieved by the following policy: *a stage can accept a new token only after its previous token has moved to the next stage*

167

## Bubbles

- A stage whose previous token has moved away but whose next token has not arrived yet is said to contain a **bubble**, e.g.:
  - an empty slot in an assembly line
  - an old value in a register that is no longer required and can be overwritten
- A token can move only to a stage containing a bubble (and the bubble moves in the opposite direction), so **bubbles are essential**: a pipeline without bubbles (i.e. full of tokens) stalls, as only the last token can move
- Good performance of the pipeline as a whole depends on the right number of tokens and bubbles in it!


168

## Performance – key parameters & metrics

- **Latency** – the time required for a token to propagate through the whole pipeline
- **Throughput** – the rate at which tokens enter (or exit) the pipeline, assuming the environment does not cause any delay
- Pipelining is aimed at improving the throughput, not the latency (in fact, the latency is usually increased)
- **Cycle period** =  $1 / \text{throughput}$

169

## Performance analysis & optimisation

- **Observation:** The throughput of a linear pipeline is no higher than the throughput of its slowest stage (the **bottleneck**)
- **Example:** these two pipelines have the same throughput  


```
graph LR; A1[delay=1] --> A2[delay=3]; A2 --> A3[delay=2]; B1[delay=3] --> B2[delay=3]; B2 --> B3[delay=3]
```
- Hence, no point to optimise faster stages, and no harm to slow them down (unless they become new bottlenecks)!
- **Balancing stages** – try to make stage delays similar, e.g.:
  - off-load work from slow stages and redistribute it between faster stages (making a fast stage slightly slower does not affect the throughput)
  - optimise or parallelise slow stages, or use **wagging**
  - split a slow stage into several ones
- If the delays are variable, **buffering** between stages is useful

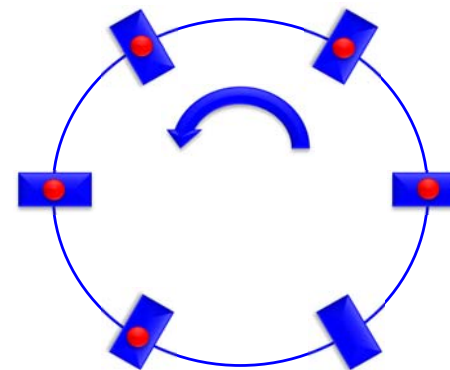
170

## Performance analysis & optimisation

- A common technique for performance analysis of a pipeline is to connect its ends turning it into a **ring**
- For maximal concurrency, as many tokens as possible should be able to move simultaneously
  - if there are too few tokens, concurrency is limited (**token-limited operation**)
  - if there are too few bubbles, few tokens can move and so concurrency is limited (**bubble-limited operation**)
  - ideally, #tokens = #bubbles
  - note that it is easy to add new bubbles (but not tokens) by **buffering**

171

## Performance analysis & optimisation

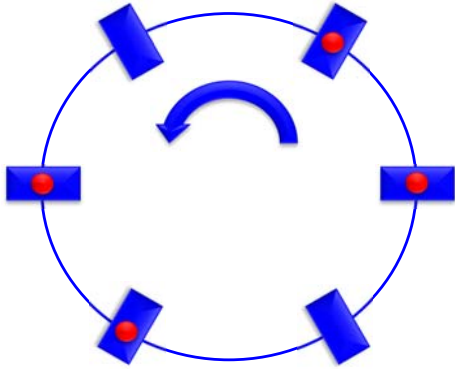


bubble-limited operation,  
throughput=1/6

Adapted from: J. Cortadella. Elastic circuits. Advanced course at EMICRO/SIM 2013  
[http://www.cs.upc.edu/~jordicf/gavina/BIB/files/ElasticCircuits\\_EMicro2013.pptx](http://www.cs.upc.edu/~jordicf/gavina/BIB/files/ElasticCircuits_EMicro2013.pptx)

172

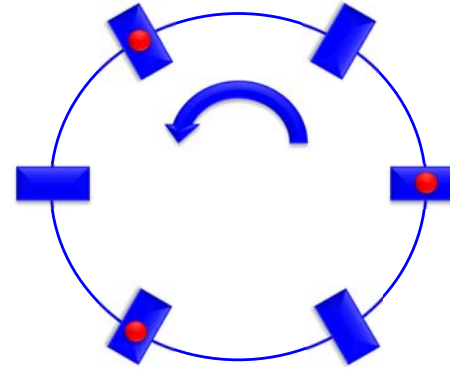
## Performance analysis & optimisation



bubble-limited operation,  
throughput=2/6

173

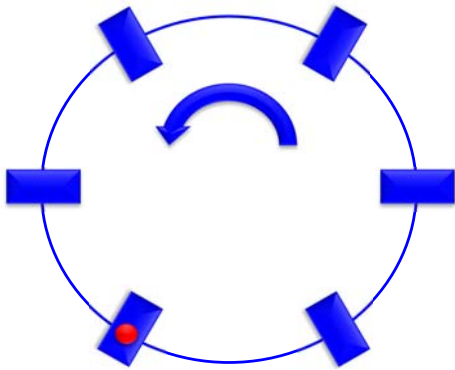
## Performance analysis & optimisation



#tokens = #bubbles  
throughput=3/6

174

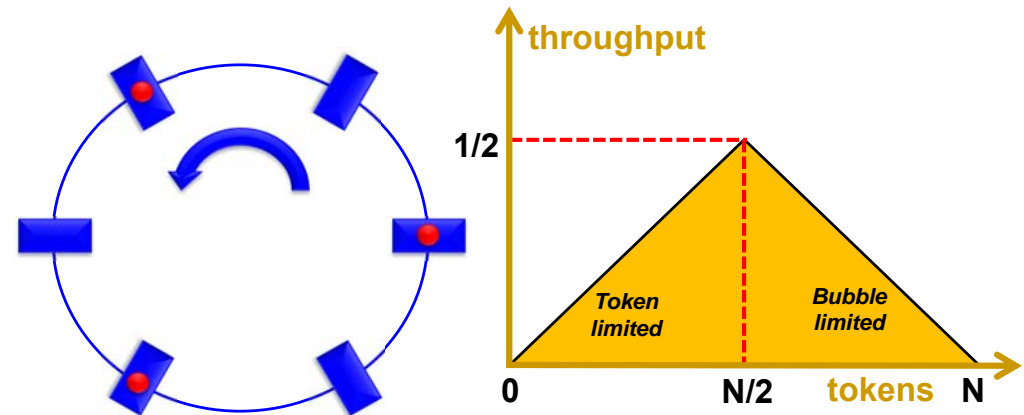
## Performance analysis & optimisation



token-limited operation,  
throughput=1/6

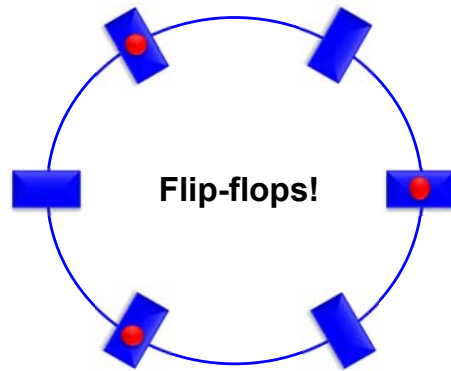
175

## Performance analysis & optimisation



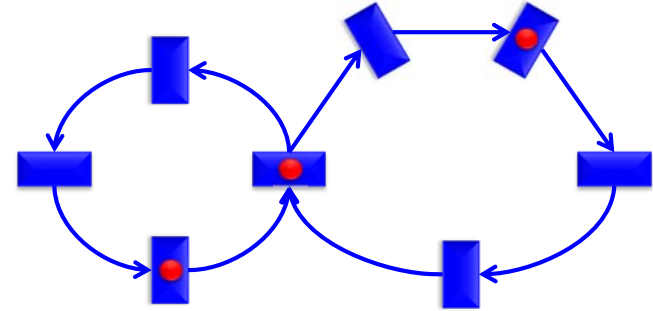
176

## Synchronous pipeline



177

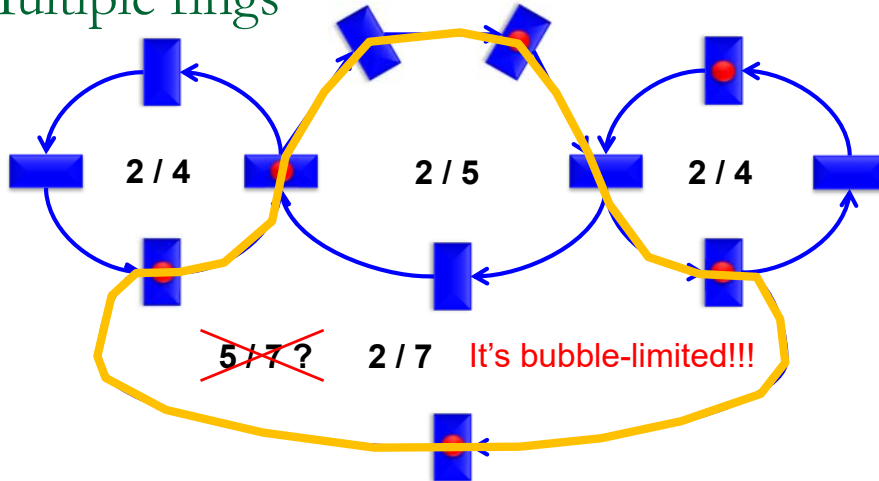
## Multiple interlocked rings



- Assume that the delay of any stage does not dominate the delays of the rings including it (otherwise need to model this stage as a cycle with two transitions and a token)
- Tokens are merged/split for stages with multiple inputs/outputs
- The number of tokens in each ring is always the same

178

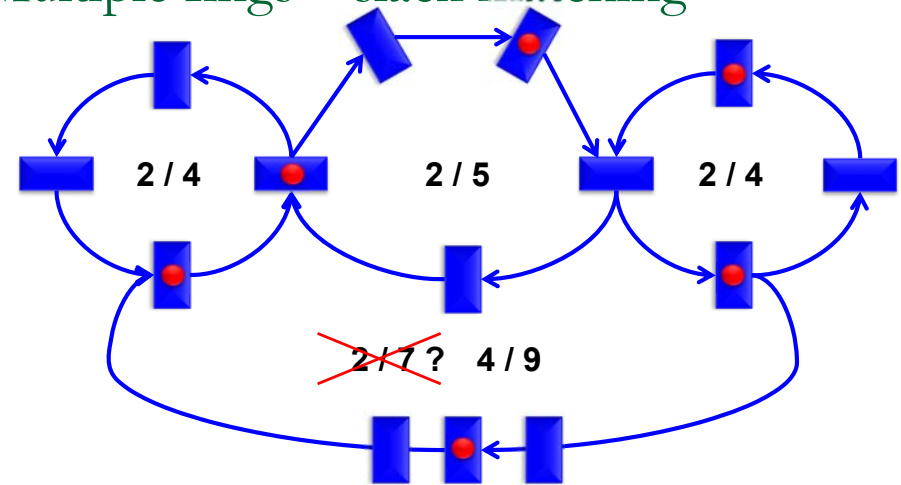
## Multiple rings



Adapted from: J. Cortadella. Elastic circuits. Advanced course at EMICRO/SIM 2013  
[http://www.cs.upc.edu/~jordicf/gavina/BIB/files/ElasticCircuits\\_EMicro2013.pptx](http://www.cs.upc.edu/~jordicf/gavina/BIB/files/ElasticCircuits_EMicro2013.pptx)

179

## Multiple rings – slack matching



- The cycle period is determined by the slowest ring
- Can easily add bubbles (but not tokens) by buffering

180