COMP 409
NOV. 23, 2010

PAUL MISSMAN

Last Time

Message passing: another paradigm for concurrent programming.
- uses messages instead of shared memory.
- must distinguish between
    asynchronous: sender does not block, receiver blocks (snail-mail)
    synchronus: sender blocks, reciever blocks (more info is known) (phone call - wait until someone answers).
- recall muddy children problem.
- there's a heirarchy of distributed knowledge
    $D_G\psi$ → "everyone collectively knows ψ"
    $S_G\psi$ → "someone knows ψ".
    $E_G\psi$ → "everyone knows ψ"
    $E^k_G\psi$ → "k-levels of you know that I know that you know, etc..." note $E^1_G\psi = E_G\psi$.
    $E^{k+1}_G = E_G E^k_G\psi$.
    ⋮
    $C_G\psi$ → "everyone knows that everyone knows ... ψ". COMMON KNOWLEDGE.

Muddy Children Problem
- K muddy children ⇒ a proof exists
- trying to figure out $E^k_G M$.

2-army Problem
- needs common knowledge.

} different behaviors/properties with different types of knowledge.

PROCESS ALGEBRA
→ High-level abstractions of concurrent programs.
    → Focus is on the concurrent activity ⇒ communication, synchronization
→ Process algebrae was invented to analyze this idea.
    Algebraic formalism for representing concurrent processes. - Bergstra & Klop '82
    CSP [T. Hoare 1978] Communicating sequential Processes (became occum (a language)).
    CCS [R. Milner 1980]
    Meije
    ACP
    π Calculus
    Ambient Calculus
    Action Systems
- the above are all message-passing-based, and most are synchronous. (For some of them, asynchronous versions have been defined → a bit more realistic for stuff like the internet)
- Interleaving semantics ⇒ major task.
    → people look for process equivalence in terms of patterns of communication.
        → do they respond in the same way? → hard to tell!
→ We will see the CSP formalism (thats most like concurrent languages).

C.S.P. Communicating Sequential Processes
- process expressions
- primitives: P :: Q
    process ↗ ↖ halt (does nothing).
    P :: Q!e (P sends some message 'e' and sends it to Q)

    Q :: P?x (Process Q stores message from P into x)

    P || Q (These two things execute concurrently) (|| is a parallel composition operator)

    ex: P :: Q!s || Q :: P?x
    these two must match up in a SYNCHRONOUS way.

PAUL HESSMAN

$P ; Q$ ⇒ sequential composition,

so $P: Q!_s ; \emptyset \,\|\, Q: P?x ; \emptyset$ can be reduced to ⇒ $\emptyset \| \emptyset$

- idea is to execute / keep reducing as much as possible.
- a few more operators:
  - we need a way of making choices,

  Guards. eg: $G \rightarrow C$ ⇒ G must be TRUE before C can be executed. eg: boolean condition,
  - we can consider $A?x$ to be true if A is ready to receive x.
  - so if a communication is ready to happen, the guard is considered TRUE.

EXAMPLE

A wants to send to B, with buffer C in between.

A:: C!m

B:: C?x

C:: $A?y \rightarrow B!y$ or C can be this with sequential composition; C:: $A?y ; B!y$ (the same thing happens in this case, but not always)

SO:

$A \| C \| B$

reduce → $\emptyset \| B!y \| C?x$

reduce → $\emptyset \| \emptyset \| \emptyset$

} Notice: This only works once, what if we want to loop? (with Recursion).

Recursive definition:

C:: $A?y \rightarrow B!y ; C$ (old way) or alternatively: C:: $*[A?y \rightarrow B!y]$ ← repetition while possible.

- how do you actually recognize when looping stops?
- Introduce explicit 'choice' operator.

  symbol ⇒ [] (external choice). used with guards & iteration.

⇒ Iterated Guarded Choice

$*[G_1 \rightarrow C_1$ []
$G_2 \rightarrow C_2$ []
$G_3 \rightarrow C_3$ []
(etc)]

(when a guard is chosen, the others get thrown away until next iteration)

Simple examples

V:: *[inslot ? \$1.00 → make tea []
inslot ? \$1.10 → make coffee] } ⇒ external choice.

V:: *[inslot ? \$1.00 → make tea []
inslot ? \$1.00 → make coffee] } same price. ⇒ becomes an internal choice.

in the second choice, V chooses what to dispense. ⇒ different behavior (arbitrary choice).

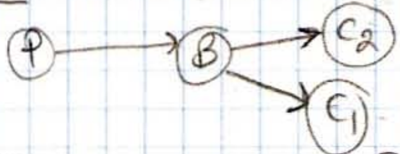
There is also an internal choice operator to do this. symbol: ⊓

V:: *[inslot ? \$1.00 → tea ⊓
inslot ? \$1.10 → coffee] ⇒ it decides what to make for you regardless of what money you put in. If you put the wrong amount, it may never do anything if it decided to make the other thing.

X:: $A?x \rightarrow \emptyset$ [] $B?x \rightarrow \emptyset$
A:: X!2 ‖ X executes
B:: X!8 ‖ X executes

→ X:: $A?x \rightarrow \emptyset$ ⊓ $B?x \rightarrow \emptyset$
- this can be stuck with no communication (deadlock) if it commits to the wrong side.

→ Usually we want external choice, but internal choice does exist.
EXAMPLE: buffered communication with multiple consumers

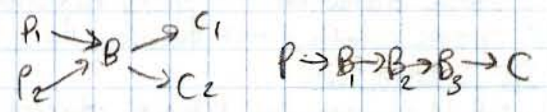


P:: *[int x = produce(); B! x]
C1:: *[int y; B?y]
C2:: *[int y; B?y]

B:: *[int z; P?z → (C1!z [] C2!z)]

internal choice, or external choice → could be different although not here
- usually external unless you want to represent some internal arbitrary choice

→ this can be used as a basis for more complex channels:



A4 due dec 3

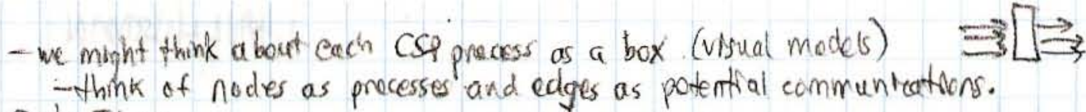
Last time
- → message passing → process algebra → concurrent structures/behaviors.
- → we look at CSP (includes some stuff to do with programming languages like booleans, variables, conditional tests...)
  - → we define processes, and use operators on them. sequential → A;B or parallel A||B w/ synchronous communication.
  - → syntax A!17 (send) A?x receive into x.
  - → we can restrict by types/constant values,
    - eg: A:: B!17 || B:: A?17 can be reduced (ie do the communication).
  - → can have choice through Guards. eg: G → C "if G then C",
    - ↳ [G1 → C1 [] ← external choice operator. → makes choice based on environment. (Guard becomes true based on current context)
      G2 → C2 []
      G3 → C3 []]
  - → there's an internal choice operator. A ⊓ B ⇒ chooses one or the other (arbitrarily), irrespective of environment.
  - → saw 1-cell buffer
- → we can get a multi-buffer by linking single buffers together, eg: (P) → (B1) → (B2) → (C) (more capacity).
  nb: there are naming issues due to specificity. we'd like to pass arguments to make it more modular.

example: ELEVATOR

```
    int floor = 1, target = Ø; // initialized to have no target.
E:: *[b1 ? pressed && target == Ø → target = 1 []
      b2 ? pressed && target == Ø → target = 2 []
      ... etc for each floor...
      target > Ø && floor < target → floor = floor + 1 []
      target > Ø && floor > target → floor = floor - 1 []
      target = floor → door!open → door?closed ; target = Ø]
```

```
door:: *[E?open → E!closed]
                ^
        (Passengers get on)
```

b1:: →
(listen for potential passenger to press button & send a pressed message to Elevator.)

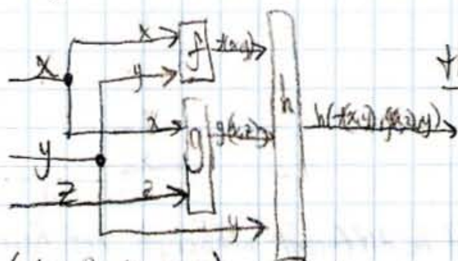


- we might think about each CSP process as a box (visual models)
  - think of nodes as processes and edges as potential communications.

Data Flow

→ Comes from functional programming. ⇒ receives input → gives output (ie a function)

eg: $h(f(x,y), g(x,z), y)$



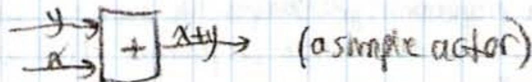
(circuit digram)

this suggests a strategy
→ each datum is a 'token'
- tokens move along edges from function to function.
- functions wait for all their inputs to be there before outputting. (function fires)
  → they consume inputs & emit outputs instantaneously.

2 flavors of data flow
- static dataflow → strong restrictions for what functions can do (deterministic behavior).
- dynamic dataflow → functions do arbitrary things

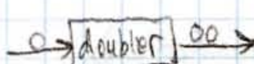
STATIC DATA FLOW
- field in computer science started by Jack Dennis
  - functions are actors
    - block & wait until all input data is available ⇒ enabled
    - can fire (consume input tokens and emit a token on every output line).
    - assume FIFO, lossless, bounded.

example

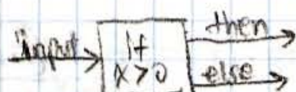


(a simple actor)

- one token on input → one token on (every) output
  = homogenous actor



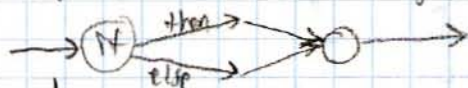
(recieves one piece of data, and spits it out twice
- this emits 2 tokens for each input. (generalizes the concept)
- we allow this eventhough its not strictly following the definition.
- takes any fixed # of inputs and emits any fixed # of outputs.
  = REGULAR actor
- there are some things we can't do with regular actor ⇒ CHOICE.

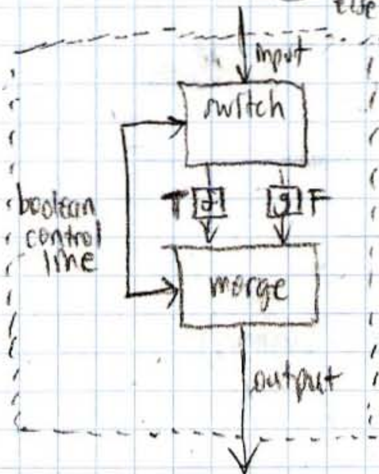


- this is not a regular actor, output depends on input. (switching).
- this is a fundamental thing we need to do!

- so we add SOME irregular actors.

Two Irregular actors (split & merge).





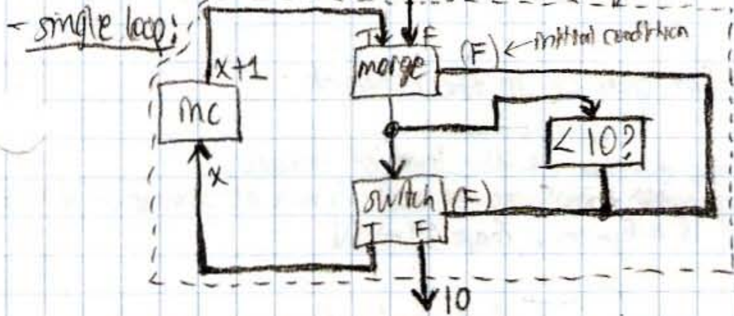
- consumes boolean token to decide where to pass the output.

- if treated as a black box, it becomes homogenous!

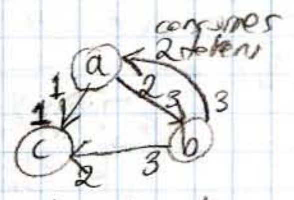
COMP 409
NOV 25 CONT'D

PAUL NUSSMAN

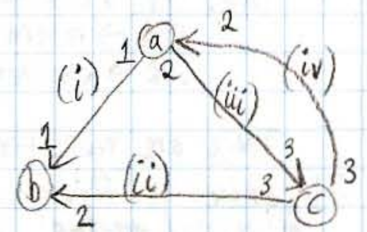
- now we can make non-trivial programs!
- single loop:



(this program counts to 10)
- the original conditions of this circuit are restored in the end.
- can be treated as a black box.



- It's useful to know how much data can fit on each circuit?
- How big are our channels? (in terms of resources)
- with regular actors, we can figure it out. ——— example →
- how many channels allow the above network to work? → we want a bound on # of channels
  - we can derive a system of equations from this.
  - every edge gives us an equation:
    - (i) $a = c$
    - (ii) $3b = 2c$
    - (iii) $2a = 3b$
    - (iv) $3b = 2a$
- what values of a, b, c make this true?



COMP 409
NOV 30 2010

epitome.cim.mcgill.ca:8080/Amjed/Survey } see moodle.

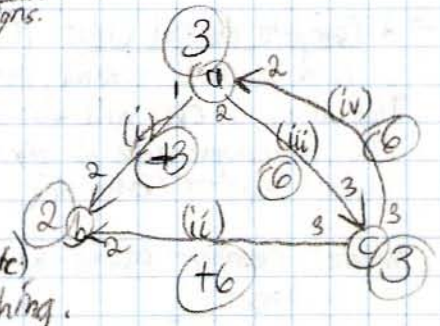
PAUL NUSSMAN

A4 demo/overview

- code given needs to be fixed
- actors should be obvious what they do.
- tokens → what's flowing along the lines
- speed bar along the bottom
- red - buffer capacity
- input → type a number

Last Time

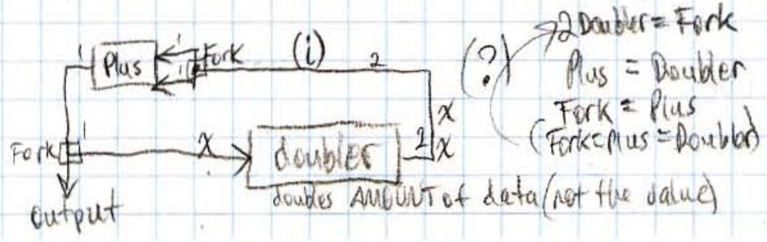
→ data flow → circuit-model with semantics → static dataflow. (restrict it to simple constructs). Simplest = homogeneous
actors have inputs & outputs. Regular actors (every time it fires, it does the same thing). Irregular actors. (for ifs), switch and merge only. ⇒ now we get a real language. Need to make sure that any use of irregular actors looks regular from the outside. Then these circuits can be analyzed.
- For practical implementation we need some finite bound on channel sizes.
- we make a system of equations to determine buffer capacities.



Back to example from last time → Find buffer capacities

(i) $a = c$ (a & c have to fire the same # of times) ⇒ steady state.
(ii) $3b = 2c$
(iii) $2a = 3b$ } ⇒ solving: $3b = 2a$ ⇒ eg: $(a, b) = (3, 2)$ is true.
~~(iv) $3b = 2a$~~ (duplicate) (# times a is fired at a steady state)

- Note: trivial solution is (0,0) but this isn't useful to show anything. we also have every multiple of any non-trivial solution.
- the (0,0) solution → what does it mean?



(?) 2 Doubler = Fork
Plus = Doubler
Fork = Plus
(Fork = Plus = Doubler)

} only solution that works for this is the trivial solution
∴ This is not a well behaved network
line (i) accumulates tokens.

Definition: **Well Behavedness** is when a non-trivial solution can be found.

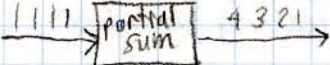
Dynamic Data Flow

-> we generalize actors with no restrictions

-> more complex, but there's a way to analyze what goes on in the network.

Gilles Kahn

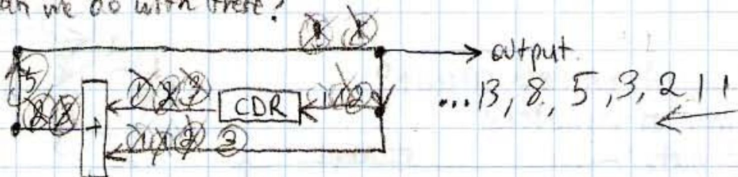
eg: Partial Sum



- not a deterministic function at the level of tokens.
- it does however compute something => on the domain of streams of tokens
  -> in this case it is a function. maps $1^\infty \rightarrow \mathbb{N}$.

-> what can we do with these?

eg:



Produces the fibonacci sequence!

Note: CAR & CDR from functional languages are trivial stream functions.
- CAR -> return first element, eliminate the rest.
- CDR -> eliminates the first element, returns the rest.

There are more sensible concepts with irregular actors.

- In order to figure out what our circuit is actually doing, we have to look at each function (individual agents) each of which is a stream function, and we need to know how to compose these agents to determine what they do together.
- To do that:

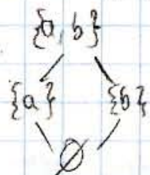
partial-order

-> total order $(\mathbb{Z}, \leq)$

partial order -> partial.

-> LUB, GLB (least/greatest upper/lower bound)

subsets of $\{a, b\}, \subseteq$

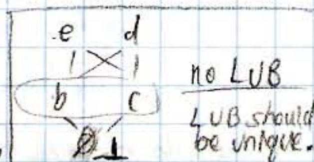


least upper bound: $a \sqcup b$ is the element at least as big as a and b, and smaller than any other upper bound of a & b.

eg: $\{a\} \sqcup \{\emptyset\} = \{a\}$ or $\{a, b\}$.
↑ this one's the LEAST upper bound.

Greatest lower bound: $a \sqcap b$ is the largest element smaller than both a & b.

Note: not every partially ordered set has GLBs or LUBs for any pair.



-> A Complete Partial Order (CPO)

is a partially ordered set (poset) with a LUB for all increasing sequences. Typically, a CPO will also have a least element (element smaller than everything else, but comparable to anything else). symbol: "$\perp$" called "bottom".

note: "top", "T" exists too

Natural numbers ordered by $\leq$ => does this form a CPO?
- -> poset
- -> 1, 2, 3, 4, 5
- -> 1, 2, 3, 4, 5, 6, 7, 8, 9, ..., etc => infinite sequence => LUB doesn't exist (note $\infty \notin \mathbb{N}$)

Sets of natural numbers ordered by $\subseteq$

$\perp = \emptyset$, union can be used to find LUB.

COMP 409
NOV 30 CONT'D

PAUL HUSSMAN

- We are trying to think of streams.
- streams ordered by prefix order ($\sqsubseteq$)
  - → empty stream = $\perp$ (is a prefix for any stream).
  - → stream 1 is a prefix of a stream 12 is a prefix for 127 $\sqsubseteq$ 1277 $\sqsubseteq$ ... etc ("is a prefix for")
- streams, prefixed ordered.
  - $\perp$
    - ↳ actually is a CPO. "for every possible sequence stream, there's a LUB" ⇐ ★ show this.
- What functions make sense on CPO's? ⇒ streams?
  - monotonic functions.
    - ⇒ we don't mean functions like this: $f(x) = x+1$
    - ⇒ we mean order-preserving functions.

$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ if f is monotonic.

COMP 409
Dec 2 2010

[LAST CLASS] (A4 due Dec 3 but accepted until dec 8 without penalty)

PAUL HUSMAN

Last time → Dataflow → static → regular, homogeneous, + a few special: switch, merge, CONS (initialize data in channels)
→ can look for well behavedness ⇒ all edge capacities are bounded. (no infinite data in any channels)
⇒ solve system of equations
→ dynamic → arbitrary actors eg: fibonacci generator (can do useful operations).

(maps one stream to another)
(it's a function in the domain of streams)

eg: 11111... → partial sum → ... 4321
→ what is sensible?

→ the domain of streams actually forms a partial order
all streams, ordered by prefix order
so $S_1 \sqsubseteq S_2$ if S_1 is a prefix of S_2 (ie $S_2 = S_1 x$).

- what kinds of functions should we allow in streams?
  - monotonic functions → if $x \leq y$ then $f(x) \leq f(y)$. [ORDER PRESERVING]
    - → most 'reasonable' functions are monotonic.
    - ★EXERCISE: come up with a NON-monotonic function. (trig functions?)
  - continuous functions
    - → Preserves Least Upper Bounds (LUBs) for all increasing sequences.
    - eg: $f(a) \sqcup f(b) = f(a \sqcup b)$
    - more generally, we have:
      $\bigsqcup\{f(a), f(b), f(c), \ldots\} = f(a \sqcup b \sqcup c \sqcup \ldots)$
    - continuous ⇒ monotonic (continuous is a generalization of monotonic).
      but monotonic ⇏ continuous.
    - → continuous functions are still a 'reasonable' class of functions
    - → it's reasonable even in dataflow.

- once something is outputted, it can't be changed

→ to make sure of that, we need monotonicity
we actually have continuous ⇒ monotonic, so we get that for free.

- here's a non-continuous function
  - say f is a function mapping to 2 streams, T & F.

f: map streams → True if input stream is infinite → True; → False if input is finite → False

- it detects infinity ⇒ unreasonable.

$f(a.) = F$
$f(aa.) = F$
$f(aaa.) = F$
$f(aaaa.) = F$
$\vdots$
$f(a^\infty) = T$

so we find out:

$\sqcup(f(a.), f(aa.), f(aaa.), f(aaaa.), f(\ldots))$ — all F

so LUB of $\{F, F, F, \ldots\}$ (for all possible finite streams)
$= F$
$= F(a. \sqcup aa. \sqcup aaa. \sqcup \ldots \sqcup \ldots)$ — limit is a^∞

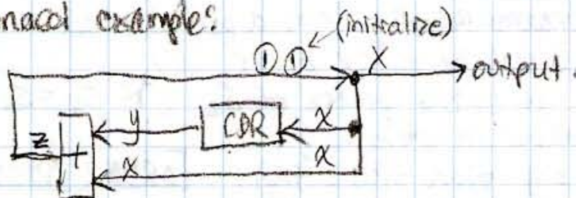
not same ∴ not continuous,

$F(a^\infty) = T$ (by definition)

- we will have ONLY continuous functions in our dataflow.

Theorem: A dataflow network is described by a system of (recursive) stream flow equations, the least solution of which describes the temporal history of the network.
- we need to show that a LEAST solution always exists. (we won't prove it here).

recall fibonacci example:



- name the streams: x, y, z (x=z, but this will be more clear)
  - x: z ① ①
  - y: CDR(x) (everything but the first thing in x)
  - z: x ⊕ y
- so we get a (recursive) system of equations
- how do we solve the systems?
- we use a grid:

	0	1	2.	3	4 →
x	⊥	11	211	3211	etc →
y	⊥	1	21	321	etc →
z	⊥	2	32	532	etc →

⇒ least: - iteratively start calculating each system
- begin from the smallest element; ⊥ ⇒ bottom, empty stream

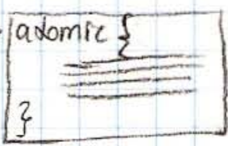
- So as we keep going, we expose the actual stream at each point.
(- nb: if the stream is infinite, this goes on forever.)

COMP 409
Dec 2 CONT'D

PAUL HUSMAN

What is current/important in concurrency research

→ nb: concurrency is different.
  → it actually does give more expressiveness.
  → threads can arbitrarily inspect each other at different points in time.
  → major issues:
    - correctness → how can we improve performance while preserving atomicity and preventing races.
      → can we come up with new languages / paradigms that ensure certainly correct programs.
    → Transactional programming.
      → machine with transactional memory
      → uses a new primitive called atomic →
        - makes body of that clause appear to be atomic to everyone else.



      → we can treat the atomic blocks as transactions
        eg: have various "undo" levels
          → try to execute the atomic block
          → detect any atomicity violations
          → roll-back if necessary.
      - drawback / missing piece.
        → still a very hard problem → we rely on developers to get it right -
        → nested atomic blocks? → semantics?
        → no built in mechanism for thread communication. (no signalling).

→ FINAL EXAM dec 15
  - open book but no electronics

Course Summary
- What is a thread and what is it good for
- Amdahl's law
- latency hiding
- parallel hardware → CMP, CMT, FMT, SMT etc.
- Atomicity → interleavings, independence, at-most-once (AMO)
- Java & PThreads → won't need to know complex models
  → know basic aspects of them
  → volatile → subtle difference in Java
  → Pthreads attributes
- Mutual Exclusion → critical sections, several algorithms (some work, some don't), liveliness
  → fairness → ticket / bakery. Peterson's (ideal properties)
  → hardware support FA, TS, CAS, LL/SC
- Semaphores → counting, binary → signalling, mutex, split
- Deadlock → Coffman's laws, dining philosophers.
- Monitors & CVs → how CVs actually function
  → spurious wake-ups can lead to bad things
  → monitor semantics
  → Readers & writers → common synchronization problem!
  → Reader/Writer Preference/fair
- General Concurrency Problems ⇒ cigarette, roller coaster, sleeping barber, Producer/Consumer
- Memory Models & Consistency → message passing (synchronous vs asynchronous), CSP process algebra
- DataFlow → static, dynamic

ALSO
- priorities
- JSR 166
- Java Util Concurrent