System-Level Design (and Modeling for Embedded Systems)

Lecture 3 – Models of Computation

Kim Grüttner kim.gruettner@dlr.de
Henning Schlender henning.schlender@dlr.de
Jörg Walter joerg.walter@offis.de

System Evolution and Operation
German Aerospace Center (DLR)
&
Distributed Computation and Communication
OFFIS

Lecture 3: Outline

- **Models of Computation (MoCs)**
  - Introduction
  - Programming models
  - Concurrency models
- **Process-based MoCs**
  - Process networks
  - Dataflow
  - Process calculi
- **State-based MoCs**
  - Finite state machines
  - Hierarchical, concurrent state machines
  - Process state machines

Models of Computation (MoCs)

- **Conceptual, abstract description of system behavior**
  - Classification based on underlying characteristics
    - Computation and communication
    - ➢ Define a class of design models
  - Well-defined, formal definition and semantics
    - Functionality (data) and order (time)
    - ➢ Formal analysis and reasoning
    - ➢ Various degrees of complexity and expressiveness
  - Decomposition into pieces and their relationship
    - Objects and composition rules
    - Data and control flow

➢ **Analyzability and expressiveness of behavioral models**
- ➢ Inherent or analyzable properties (liveness, fairness, …)
- ➢ Restrictions (Turing completeness, halting problem)
- ➢ Efficient implementation

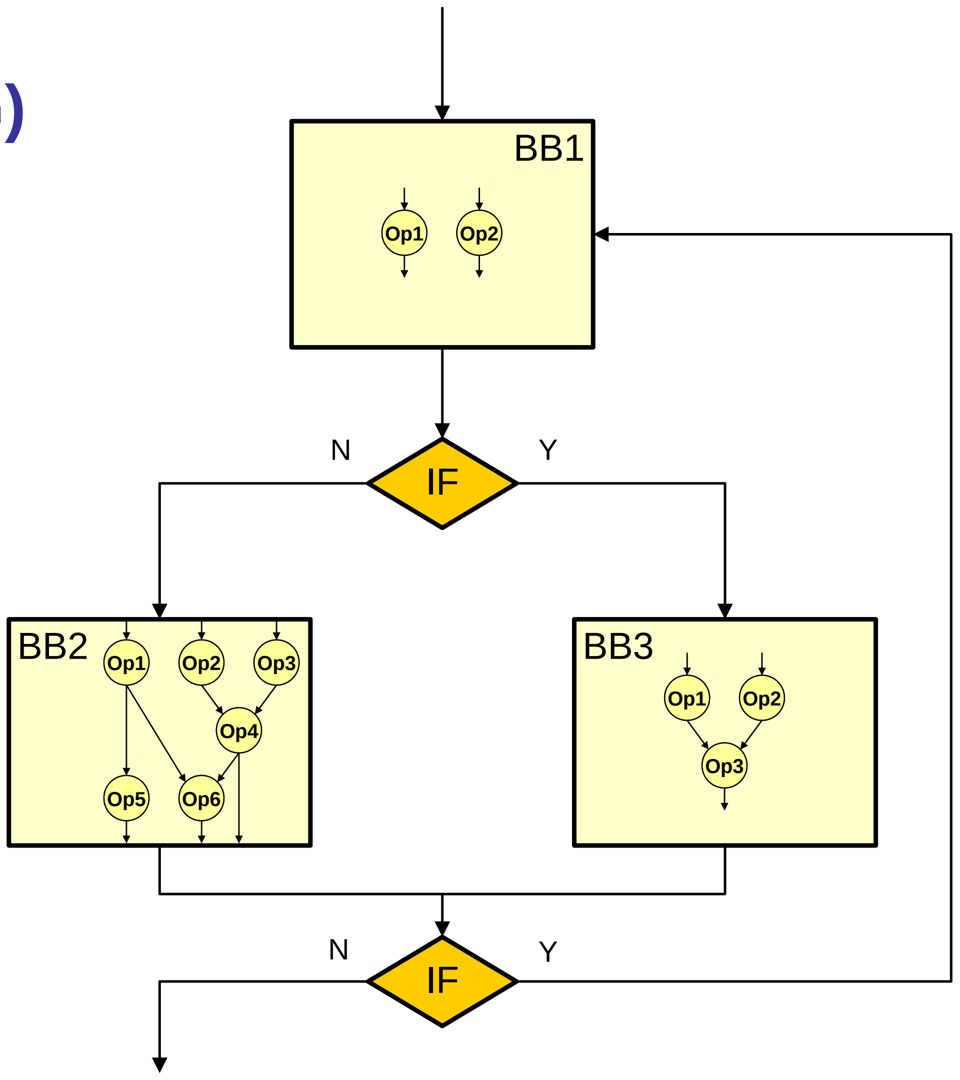
Programming Models

- **Imperative programming models**
  - Statements that manipulate program state, control flow
  - ➢ Sequential programming languages [C, C++, Java, …]
- **Declarative programming models**
  - Rules for data manipulation, data flow
  - ➢ Functional programming [Haskell, Lisp]
  - ➢ Logic programming [Prolog]

➢ **Von Neumann execution**
  - ➢ Implicit concurrency (dependency analysis)
  - ➢ Implicit ordering (time)
  - ➢ Implicit state (set of variables)
  - ➢ Implicit termination (Turing complete)

Control and Data

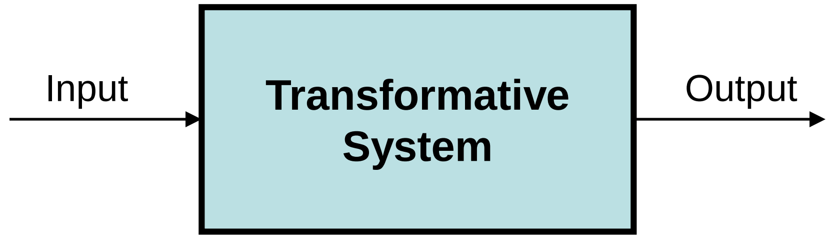
- **Control/Data Flow Graph (CDFG)**
  - Control
    - Branches, loops
    - Dependencies, order of basic blocks (BB) of computation
  - Data
    - Basic blocks (BBs) with expressions
    - Directed, acyclic graph of operations



➢ **Graphical representation (syntax) of program**

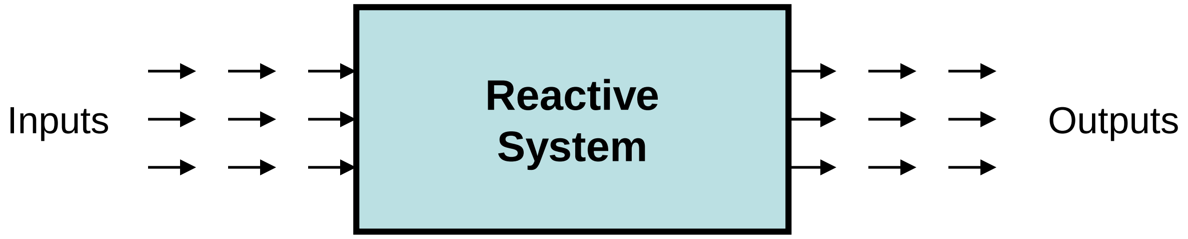
Reactive Systems

- **Transformative**



  - Output = F(Input)
  - ➢ Data processing

- **Reactive**



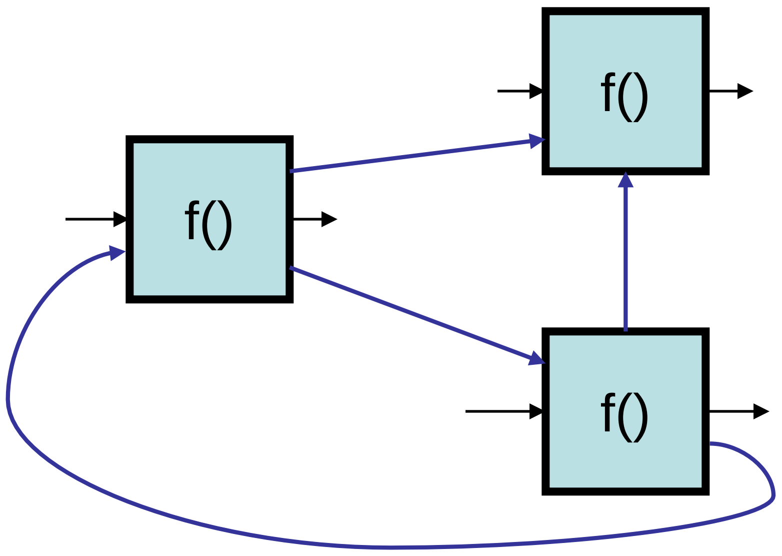
  - Continuous interaction with environment
  - ➢ Control oriented: e.g. interactive (servers, GUIs) and constrained (embedded)

➢ **Ordering, interleaving of inputs and outputs (when)?**

  - ➢ Timing defines reactive behavior (e.g. in case of feedback)

Concurrency

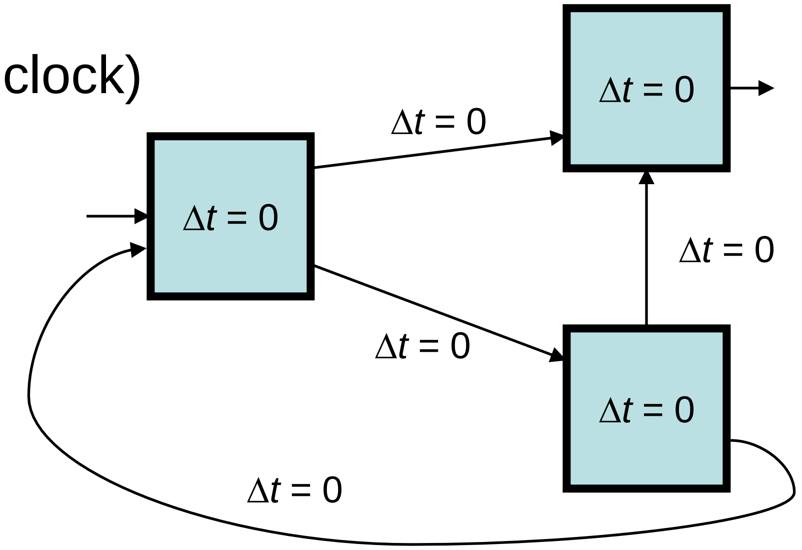
- **Interaction, ordering relationship between blocks**
  - Control or data dependencies, communication (what)
  - Synchronization, time (when)



  - Physical: total order (real time)
    - Distinct components naturally interleaved in (very fine) time
  - Logical: partial order (logical time)
    - Specification of interleaving based on causality only (implementation freedom)

Synchronous Model

- **Synchronous hypothesis**
  - Reactions are instantaneous (zero time)
  - Simultaneous (broadcast)
  - Sequence of discrete steps (logical clock)
  - ➢ Deterministic, static verifiable (independent of actual delays)

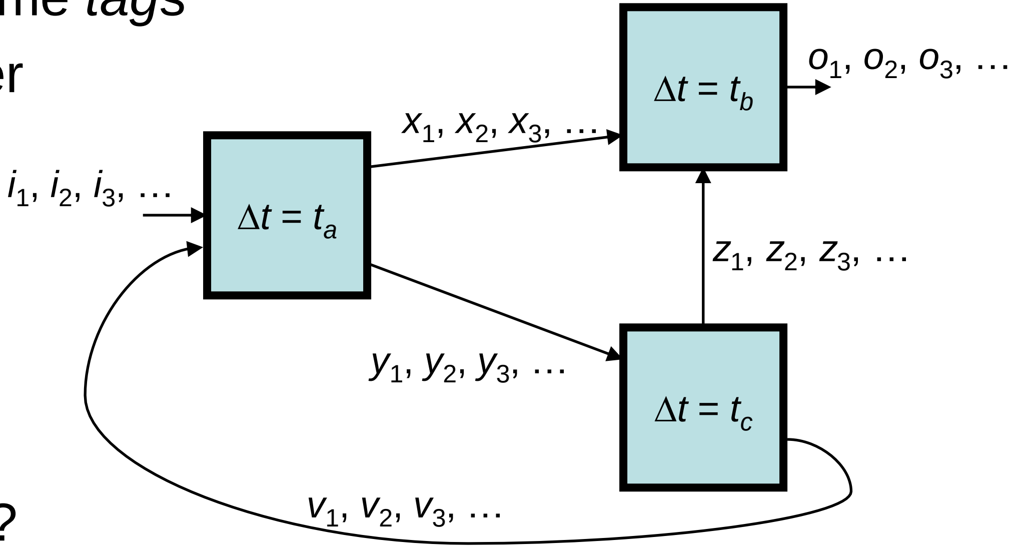


- **Challenges:**
  - Semantics: causal microsteps, but: cycles, conflicts?
  - Implementation: Δ*t* << *clock*, global clock, shared events

➢ **Synchronous languages**
  - ➢ Imperative (control) [Esterel] or declarative (data) [Lustre] style
  - ➢ Reject cycles [Lustre] or require unique fixed-point [Esterel]
  - ➢ Hardware (synchronous) or software (safety critical) compilers

Discrete Event

- **Asynchronous, relax relations**
  - Signals = streams of events
  - Event e_i = (*value*, *tag*), discrete time *tags*
  - Global event and evaluation order
  - ➢ Flexible, arbitrary dynamic delays



- **Challenges**
  - Semantics: simultaneous events?
  - Implementation: maintain delays (global time)

➢ **Design languages**
  - ➢ General, universal modeling and simulation (multi-scale)
  - ➢ Hardware-description [VHDL, Verilog], system-level [SpecC, SystemC]
  - ➢ Deterministic zero-delay delta steps, potentially endless cycles
  - ➢ Ignore delays for synthesis, only realize causality (subset)

Models of Computation (MoCs)

- **MoC examples**
  - Programming models: imperative, declarative
    - Statements for algorithmic computation, operation-level granularity
  - Concurrency models: synchronous, discrete event
    - Basic, general interaction and global relation between operations
  - Process-based models: networks, dataflow, calculi
    - Activity, data flow/dependencies
  - State-based models: evolution from FSM to PSM
    - State enumeration, control flow/dependencies

➢ **Specification and algorithm modeling**
- ➢ Ptolemy (UC Berkeley): heterogeneous mixture of MoCs
- ➢ Matlab/Simulink (Mathworks), LabView (NI): dataflow
- ➢ Statemate (IBM Rational), UML: StateCharts, HCFSM

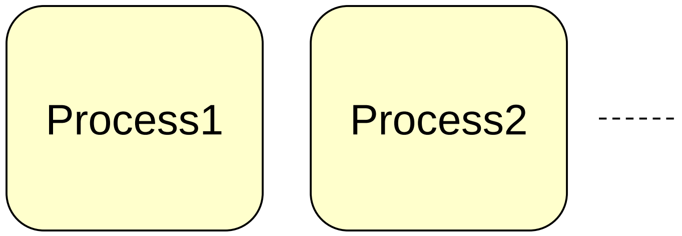
Lecture 3: Outline

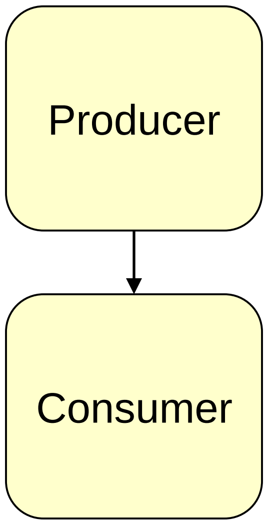
✓ Models of Computation (MoCs)

- **Process-based MoCs**
  - Process networks
  - Dataflow
  - Process calculi
- **State-based MoCs**

Process-Based Models

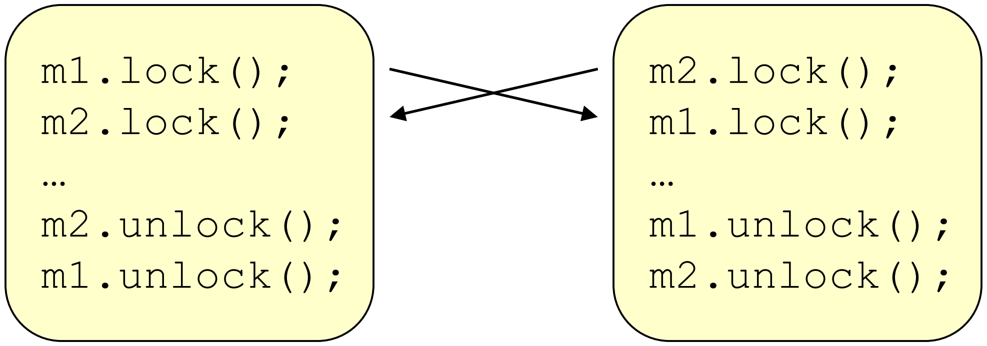
- **Activity and causality (data flow)**
  - Asynchronous, coarse-grain concurrency
- **Set of processes**
  - Processes execute in parallel
    - Concurrent composition
  - Each process is internally sequential
    - Imperative program
- **Inter-process communication**
  - Shared memory [Java]
    - Synchronization: critical section/mutex, monitor, …
  - Message passing [MPI]
    - Synchronous, rendezvous (blocking send)
    - Asynchronous, queues (non-blocking send)
- **Implementation: OS processes or threads**
  - Single or multiple processors/cores





Problem of Concurrency: Deadlocks

- **Circular chain of 2 or more processes which each hold a shared resource that the next one is waiting for**
  - Circular dependency through shared resources



```
m1.lock();
m2.lock();
…
m2.unlock();
m1.unlock();
```

```
m2.lock();
m1.lock();
…
m1.unlock();
m2.unlock();
```

  - ➢ Prevent chain by using the same precedence
  - ➢ Use timeouts (and retry), but: livelock

➢ **Dependency can be created when resources are shared**
  - ➢ Side effects, e.g. when blocking on filled queues/buffers

Problem of Concurrency: Determinism

- **Deterministic: same inputs always produce same results**
- **Random: probability of certain behavior**
- **Non-deterministic: undefined behavior (for some inputs)**
  - Undefined execution order
    - Statement evaluation in imperative languages: `f(a++, a++)`
    - Concurrent process race conditions:

```
x = a;
y = b;
```

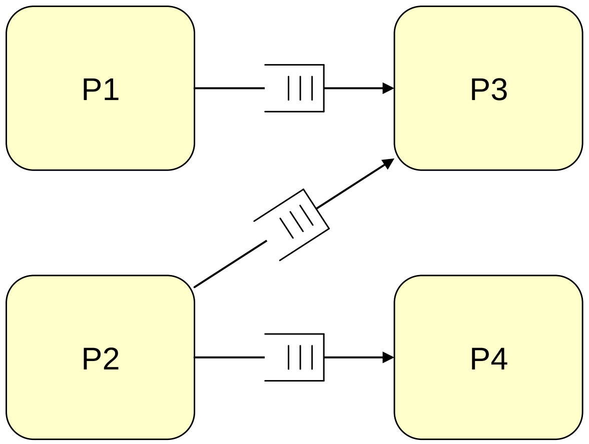
```
a = 1;
b = 2;
```

x = ?, y = ?

➢ **Can be desired or undesired**
  - ➢ How to ensure correctness?
    - ➢ Tedious and error-prone manual untangling (synchronization)
    - ➢ Simulator will typically pick only one behavior
  - ➢ But: over-specification?
    - ➢ Leave freedom of implementation choice (concurrency)

Kahn Process Network (KPN) [Kahn74]

- **C-like processes communicating via FIFO channels**
  - Unbounded, uni-directional, point-to-point queues
    - Sender (`send()`) never blocks
    - Receiver (`wait()`) blocks until data available



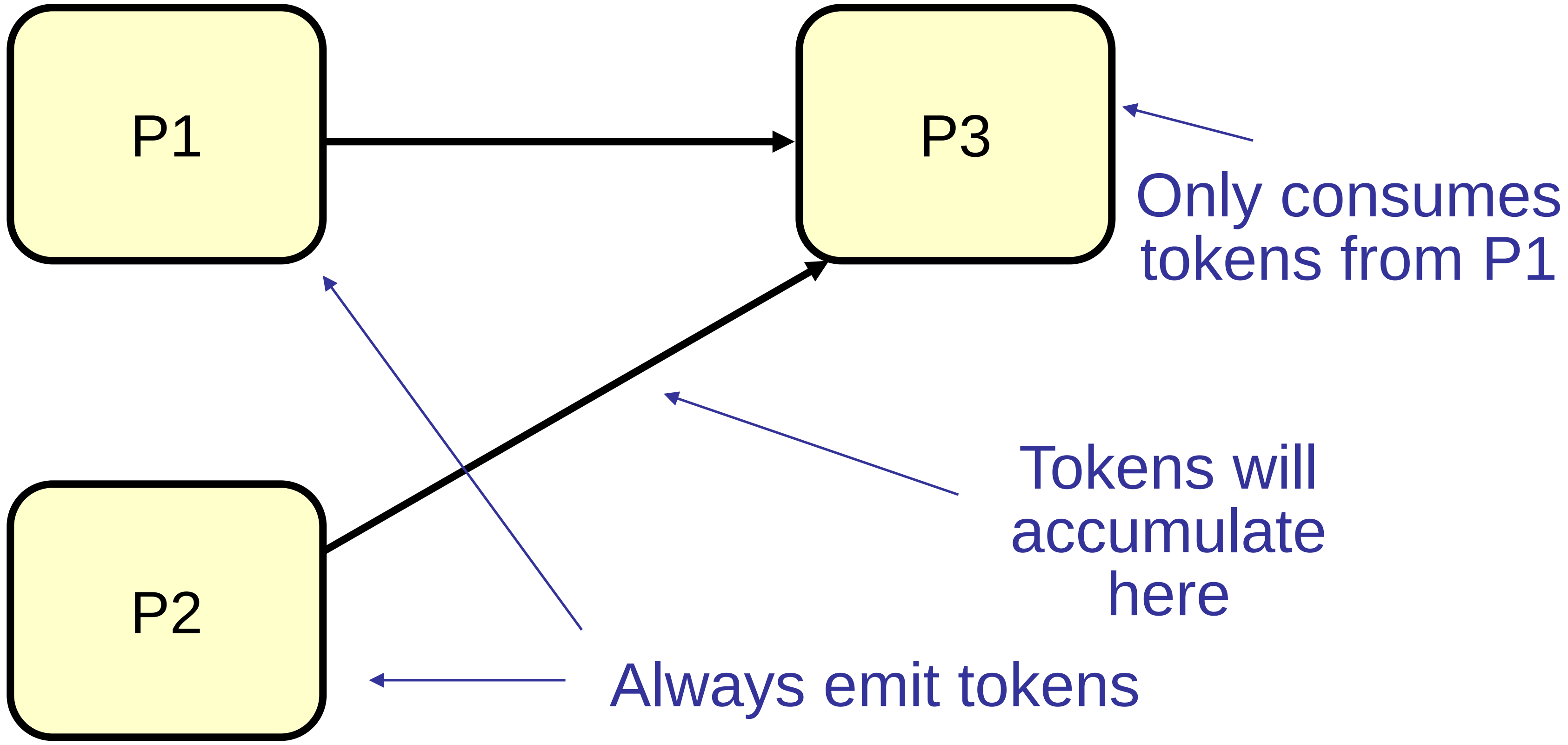
➢ **Deterministic**
  - Behavior does not depend on scheduling strategy
  - Focus on causality, not order (implementation independent)

Kahn Process Network (KPN) (2)

- **Determinism**
  - Process can’t peek into channels and can only wait on one channel at a time
  - Output data produced by a process does not depend on the order of its inputs
  - ➢ Terminates on global deadlock: all process blocked on `wait()`
- **Formal mathematical representation**
  - Process = continuous function mapping input to output streams
- **Turing-complete, undecidable (in finite time)**
  - Terminates?
  - Can run in bounded buffers (memory)?

KPN Scheduling: Data Driven

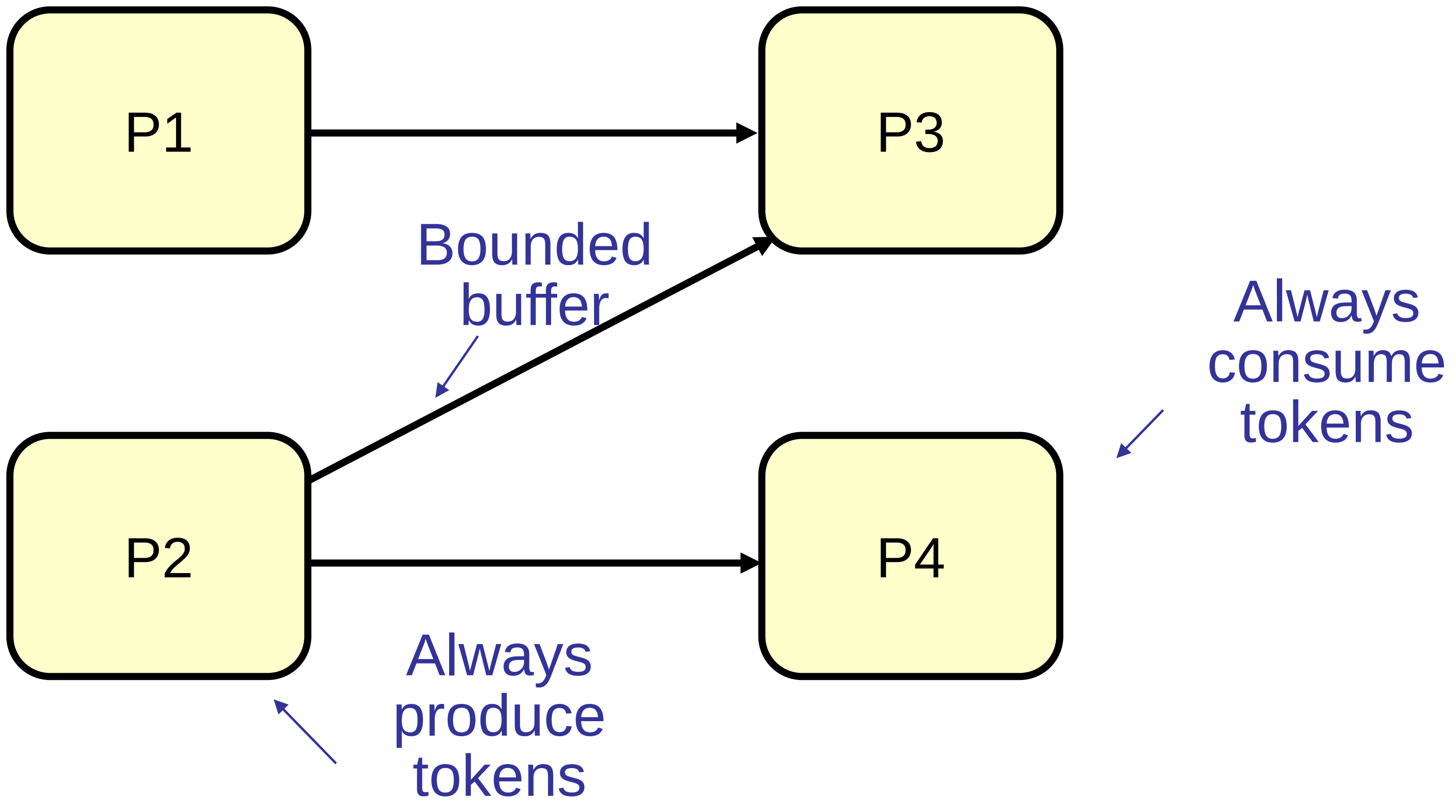
- **Scheduling determines memory requirements**
- **Data-driven scheduling**
  - Run processes whenever they are ready:



Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

KPN Scheduling: Demand-Driven

- **Only run a process whose outputs are being requested**
  - Synchronous, unbuffered message-passing
- **However...**



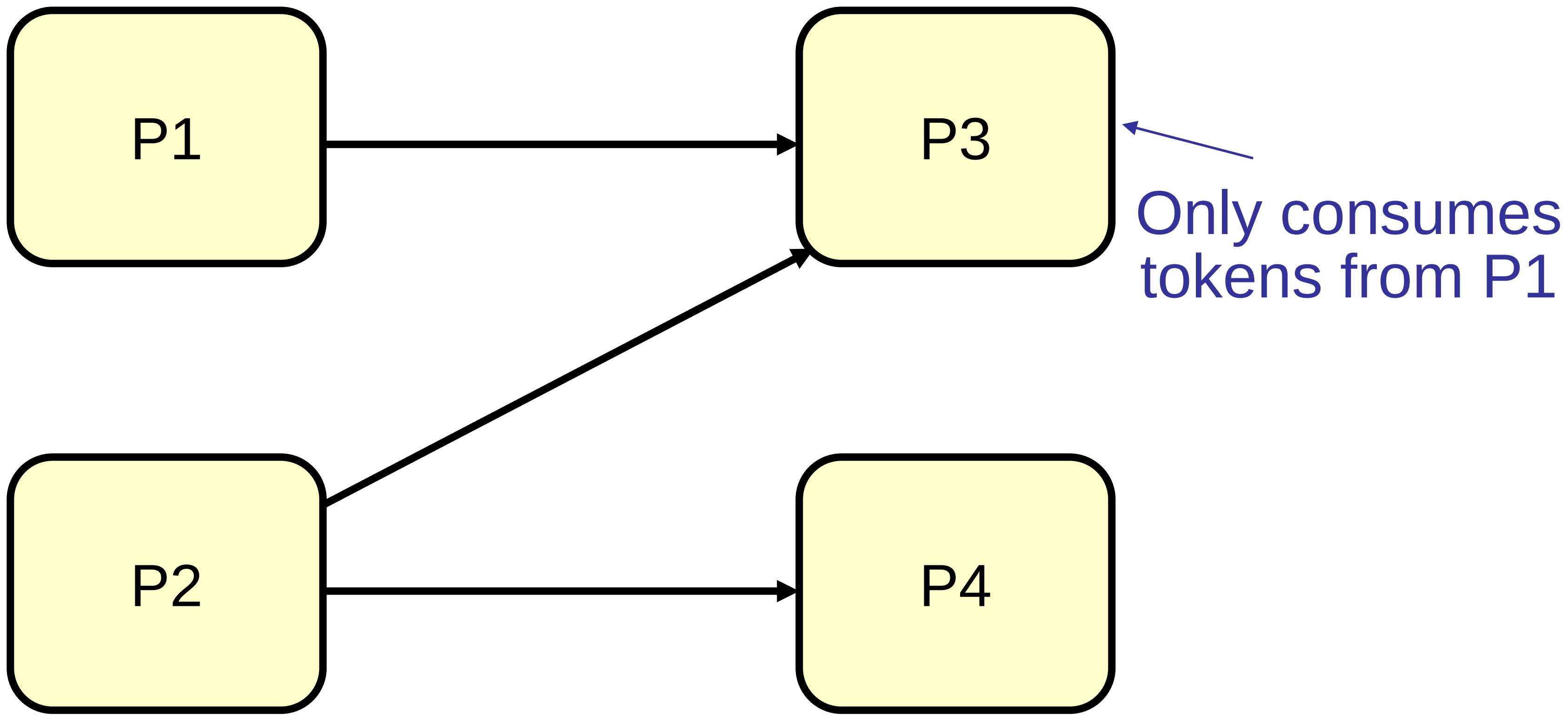
Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

KPN Scheduling

- **Inherent tradeoffs**
  - Completeness
    - Run processes as long as they are ready
    - ➢ Might require unbounded memory
  - Boundedness
    - Block senders when reaching buffer limits
    - ➢ Potentially incomplete, artificial deadlocks and early termination
  - ➢ Data driven: completeness over boundedness
  - ➢ Demand driven: boundedness over completeness and even non-termination

Motivation: Parks' Algorithm

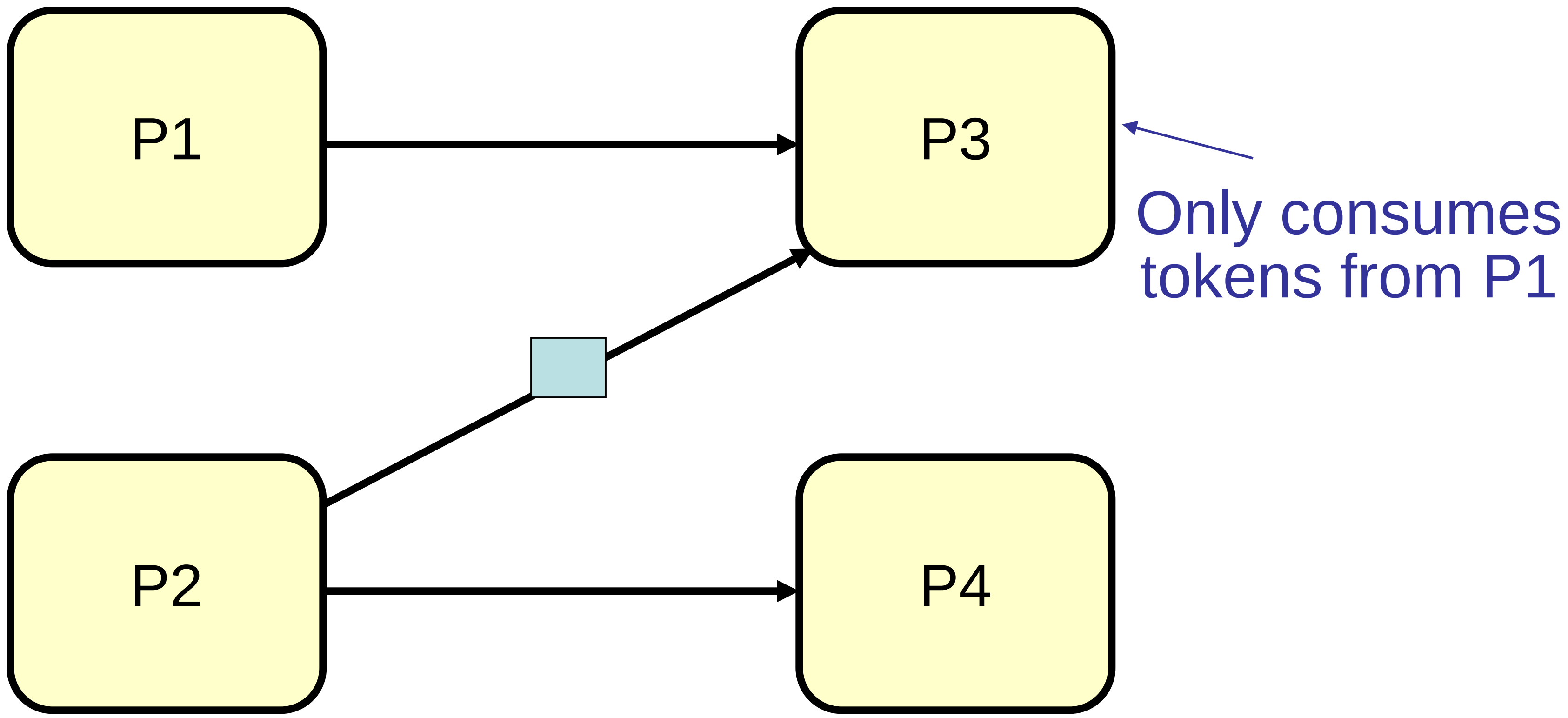
- **Start with buffer size 1**
- **Run P1, P2, P3, P4**



Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

Motivation: Parks' Algorithm

- **P2 blocked**
- **Run P1, P3, P1, … indefinitely**



Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

KPN Scheduling: Parks' Algorithm

- **Hybrid approach [Parks95]**
  - Start with smallest bounded buffers
  - Schedule with blocking `send()` until (artificial) deadlock
  - Increase size of smallest blocked buffer and continue

- **Parks' Scheduling Algorithm (1995)**
  - Set a capacity on each channel
  - Block a write if the channel is full
  - Repeat
  - Run until deadlock occurs
  - If there are no blocking writes → terminate
  - Among the channels that block writes, select the channel with least capacity and increase capacity until producer can fire

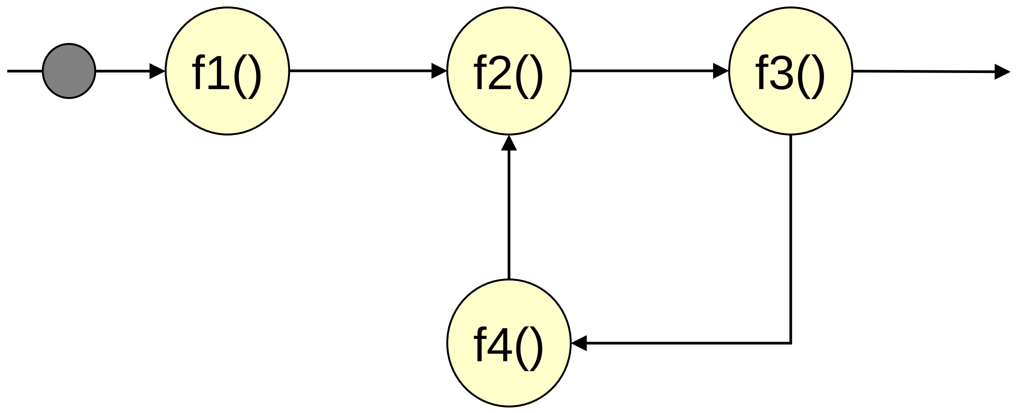
KPN Scheduling: Parks' Algorithm

- **Difficult to implement**
  - Size of infinite FIFOs in limited physical memory?
  - Dynamic memory allocation, dependent on schedule
  - Boundedness vs. completeness vs. non-termination (deadlocks)
  - Dynamic context switching

- **Parks' algorithm**
  - Non-terminating over bounded over complete execution
  - ➢ Does not find every complete, bounded schedule
  - ➢ Does not guarantee minimum memory usage
  - ➢ Deadlock detection?

Dataflow

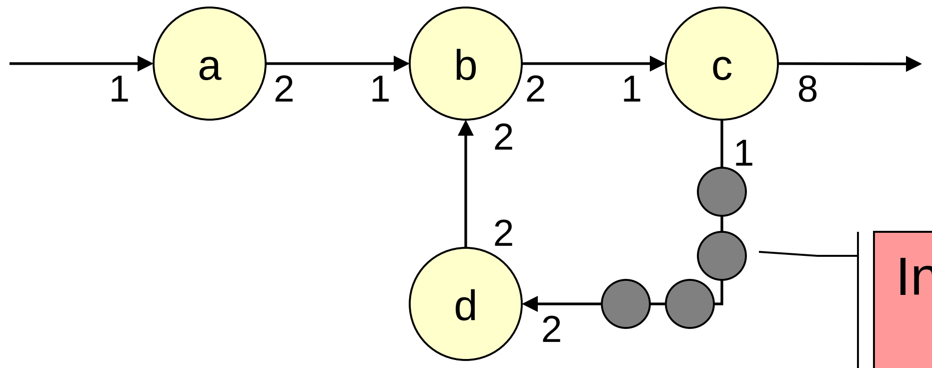
- **Breaking processes down into network of *actors***
  - Atomic blocks of computation, executed when *firing*
  - Fire when required number of input *tokens* are available
    - – Consume required number of tokens on input(s)
    - – Produce number of tokens on output(s)
  - ➢ Separate computation & communication/synchronization
    - ➢ Actors (indivisible units of computation) may fire simultaneously, any order
    - ➢ Tokens (units of communication) can carry arbitrary pieces of data
- **Directed graph of infinite FIFO arcs between actors**
  - Boundedness, completeness, non-termination?



➢ **Signal-processing applications**

Synchronous Dataflow (SDF) [Lee86]

- **Fixed number of tokens per firing**
  - Consume fixed number of inputs
  - Produce fixed number of outputs



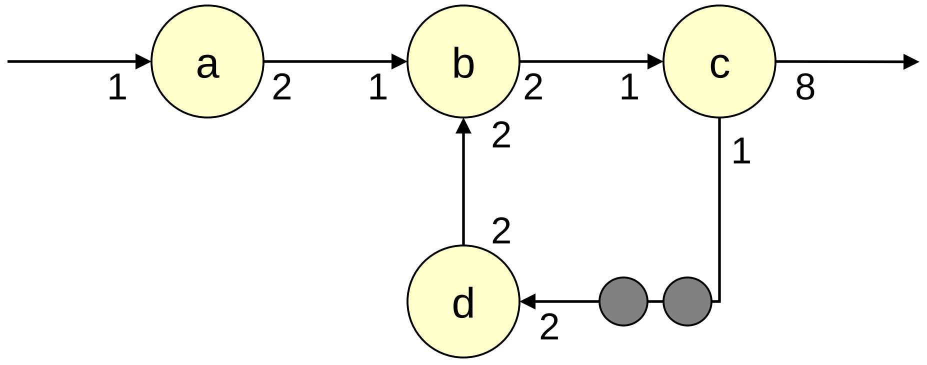
Initialization
- ➢ Delay
- ➢ Prevent deadlock

- ➢ **Can be scheduled statically**
  - ➢ Flow of data through system does not depend on values
- ➢ **Find a repeated sequence of firings**
  - ➢ Run actors in proportion to their rates
  - ➢ Fixed buffer sizes, no under- or over-flow

SDF Scheduling (1)

- **Solve system of linear rate equations**
  - Balance equations per arc
    - *2a = b*
    - *2b = c*
    - *b = d*
    - *2d = c*

    ➢ *4a = 2b = c = 2d*



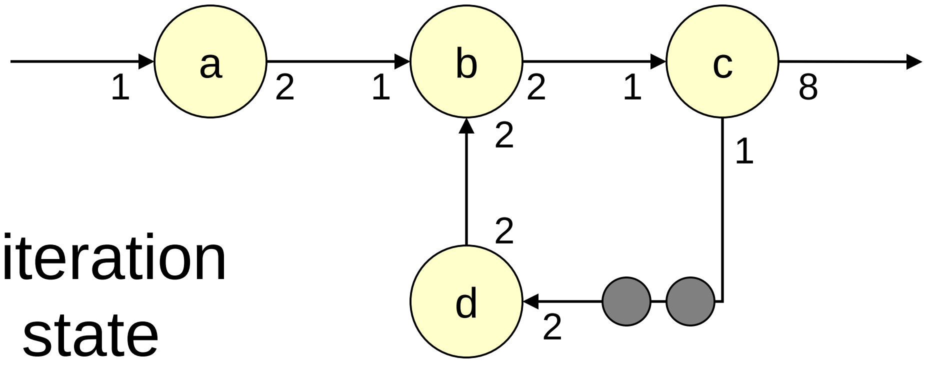
  - Inconsistent systems
    - Only solvable by setting rates to zero
    - Would otherwise (if scheduled dynamically) accumulate tokens
  - Underconstrained systems
    - Disjoint, independent parts of a design

➢ **Compute repetitions vector**

  ➢ Linear-time depth-first graph traversal algorithm

SDF Scheduling (2)

- **Periodically schedule actors in proportion to their rates**
  - Smallest integer solution
    - – *4a = 2b = c = 2d*
    - ➢ *a = 1, b = 2, c = 4, d = 2*
  - Symbolically simulate one iteration of graph until back to initial state
    - – Insert initialization tokens to avoid deadlock
    - ➢ *adbccdbcc = a(2db(2c))*
    - ➢ *a(2db)(4c)*

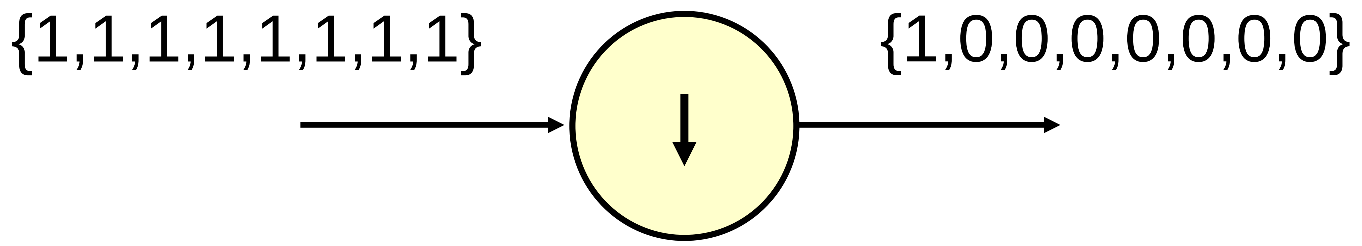


➢ Schedule determines memory requirements
  - ➢ *a(2db(2c))*: 2 token slots on each arc for total of 8 token buffers
  - ➢ *a(2db)(4c)*: extra initialization tokens, 12 token buffers

➢ Single appearance schedule to reduce code size
  - ➢ Looped code generation and compiler optimizations

Cyclo-Static Dataflow (CSDF)

- **Periodic firings (cyclic pattern of token numbers)**
  - 8:1 Downsampler
    - – Traditional SDF: store and consume 8 input tokens for one output



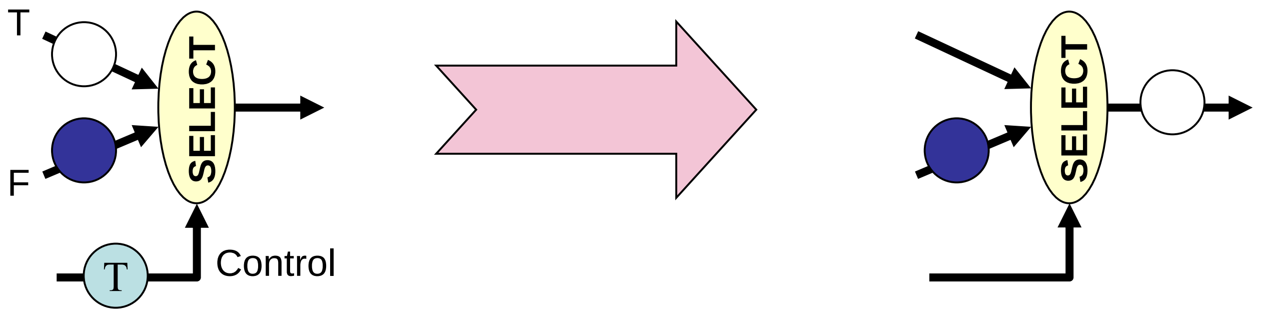
- ➢ First firing: consume 1, produce 1
- ➢ Second through eighth firing: consume 1, produce 0

➢ **Static scheduling in similar manner as SDF**

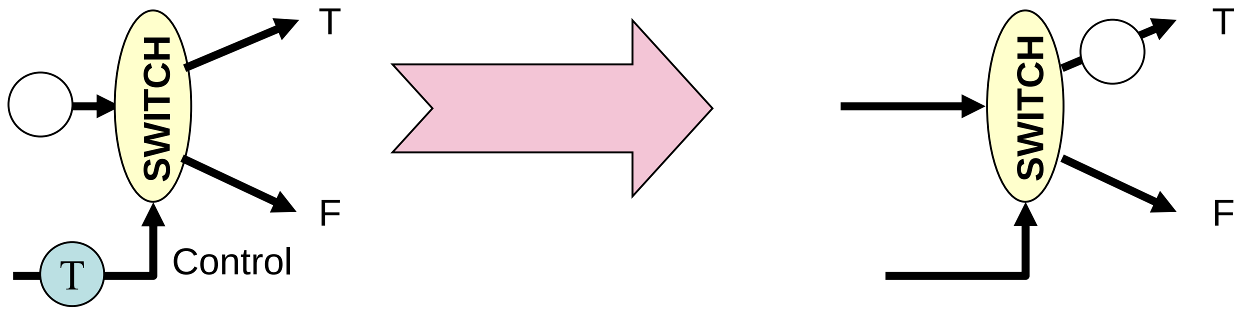
Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

Boolean Dataflow (BDF)

- **Allow actors with boolean control inputs**
  - Select actor



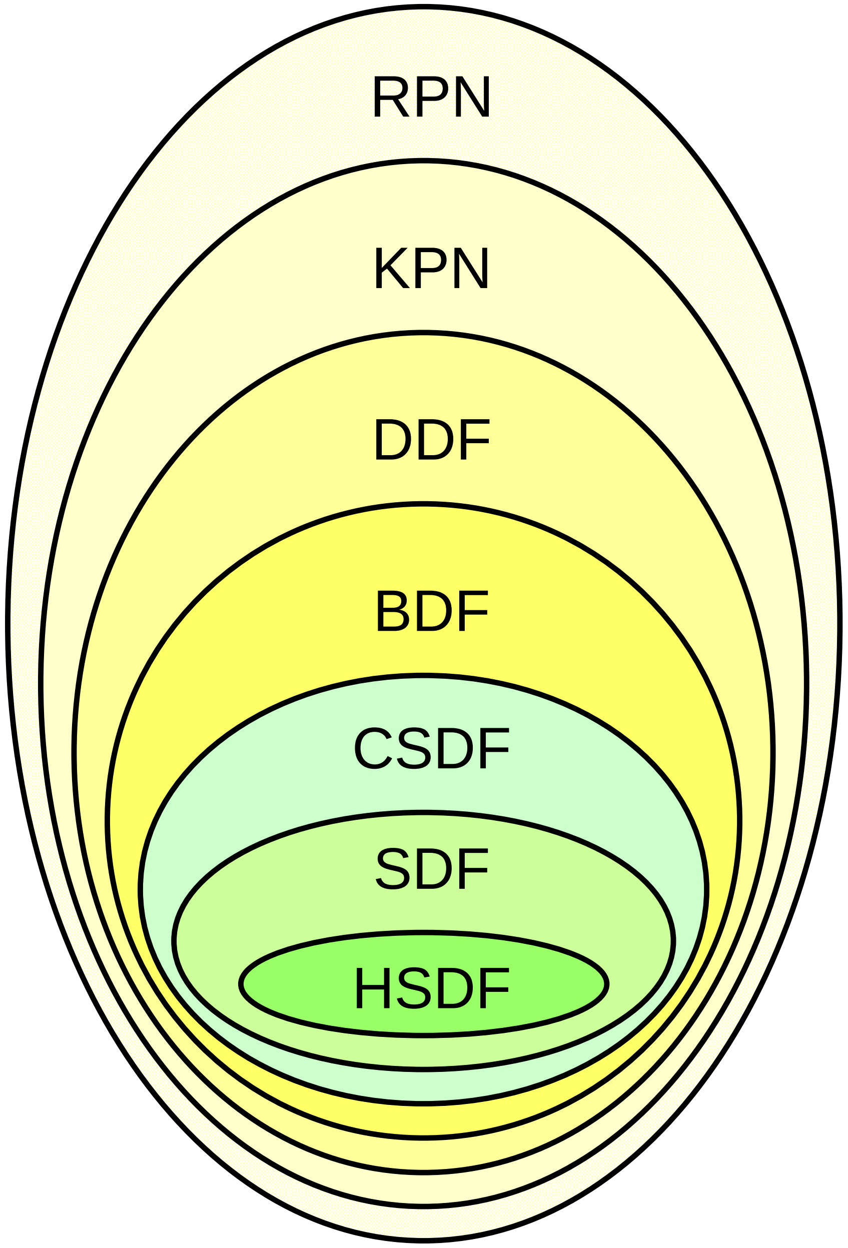
  - Switch actor



➢ **Turing complete**
  - ➢ Loops, branches
  - ➢ Quasi-static scheduling

Source: M. Jacome, UT Austin.

Process-Based MoCs



Yellow: Turing complete

RPN	Reactive Process Network
KPN	Kahn Process Network
DDF	Dynamic Dataflow
BDF	Boolean Dataflow
CSDF	Cyclo-Static Dataflow
SDF	Synchronous Dataflow
HSDF	Homogeneous SDF (DFG, with single token)

Source: T. Basten, MoCC 2008.

Process Calculi

- **Rendezvous-style, synchronous communication**
  - Communicating Sequential Processes (CSP) [Hoare78]
  - Calculus of Communicating Systems (CCS) [Milner80]
  - ➢ Restricted interactions

➢ **Formal, mathematical framework: process algebra**
  - Algebra = <objects, operations, axioms>
    - Objects: processes {*P*, *Q*, …}, channels {*a*, *b*, …}
    - Composition operators: parallel ($P \| Q$), prefix/sequential ($a \rightarrow P$), choice ($P+Q$)
    - Axioms: indemnity ($\emptyset \| P = P$), commutativity ($P+Q=Q+P$, $P \| Q = Q \| P$)
  - ➢ Manipulate processes by manipulating expressions

➢ **Parallel programming languages**
  - ➢ CSP-based [Occam/Transputer, Handle-C]

Lecture 3: Outline

✓ Models of Computation (MoCs)

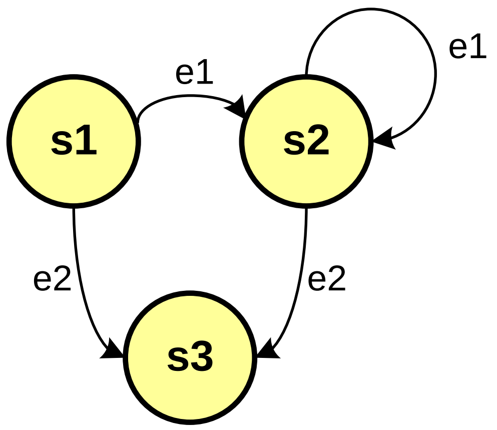
✓ Process-based MoCs

- **State-based MoCs**
  - Finite state machines
  - Hierarchical, concurrent state machines
  - Process state machines

State-Based Models

- Status and reactivity (control flow)

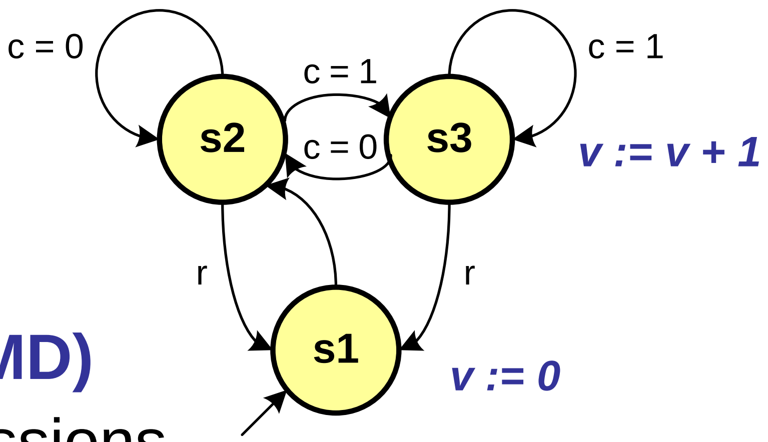
- Explicit enumeration of computational states
  - State represents captured history
- Explicit flow of control
  - Transitions in reaction to events



- Stepwise operation of a machine
  - Cycle-by-cycle hardware behavior
  - Finite number of states
    - Not Turing complete
- State-oriented imperative representation
  - State only implicit in control/data flow (CDFG)
- Formal analysis
  - Reachability, equivalence, …

Finite State Machines

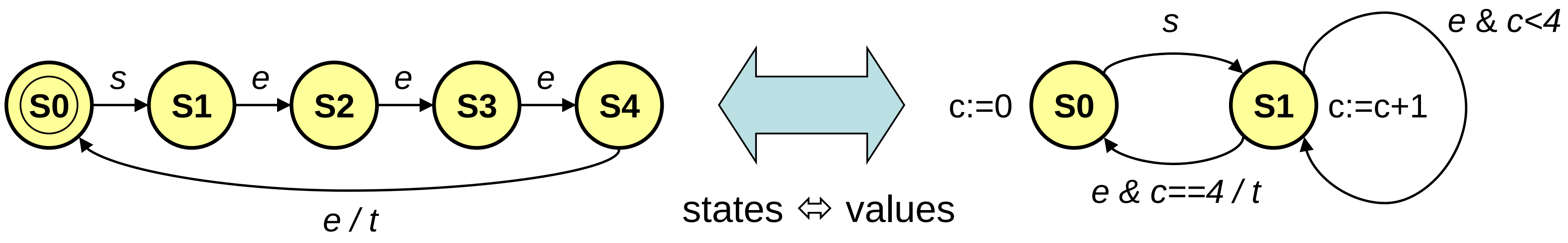
- **Finite State Machine (FSM)**
  - Basic model for describing control and automata
    - Sequential circuits
  - States *S*, inputs/outputs *I/O*, and state transitions
    - FSM: <*S*, *I*, *O*, *f*, *h*>
    - Next state function *f*: $S \times I \rightarrow S$
    - Non-deterministic: *f* is multi-valued
  - Output function *h*
    - Mealy-type (input-based), *h*: $S \times I \rightarrow O$
    - Moore-type (state-based), *h*: $S \rightarrow O$
    - ➢ Convert Mealy to Moore by splitting states per output



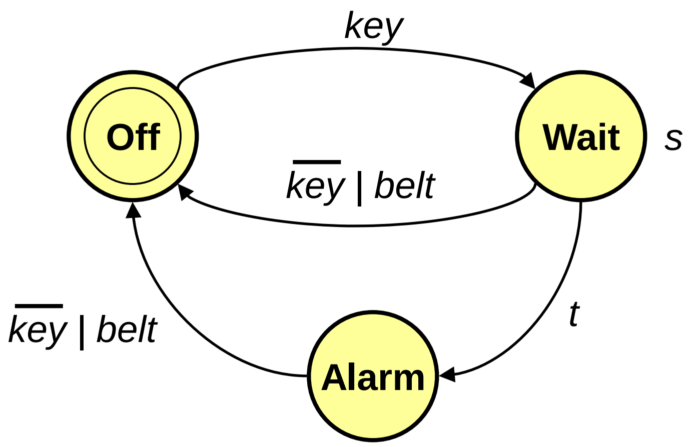
- **Finite State Machine with Data (FSMD)**
  - Computation as control and expressions
    - Controller and datapath of RTL processors
  - FSM plus variables *V*
    - FSMD: <*S*, *I*, *O*, ***V***, *f*, *h*>
    - Next state function *f*: $S \times V \times I \rightarrow S \times V$
    - Output function *h*: $S \times V \times I \rightarrow O$

Reducing Complexity

- **FSM with Data (FSMD) / Extended FSM (EFSM)**
  - Mealy counter
    - Implicit self-loops on ē (absence), $f: S \times V \times 2^I \rightarrow S \times V$, $h: S \times V \times 2^I \rightarrow 2^O$

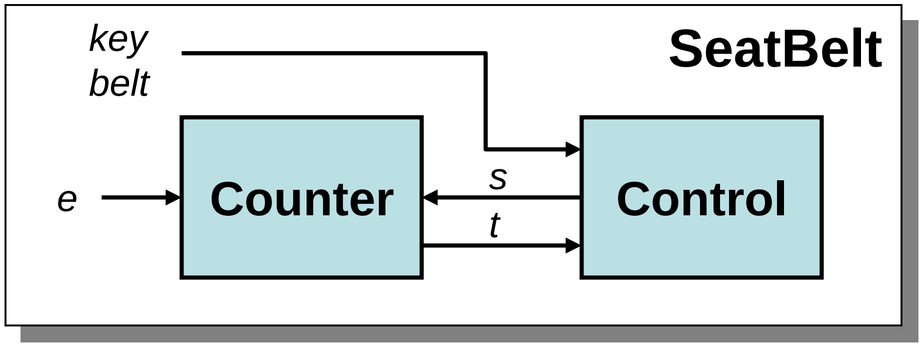


- **Non-Deterministic FSM (NFSM)**
  - Choice in control
    - Implicit self-loops for unspecified conditions? Usually!
    - **Wait**: *belt & t* ?
    - Multiple arcs for same condition?
    - Incomplete specification (undecided), unknown behavior (don't care)

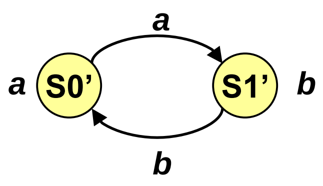


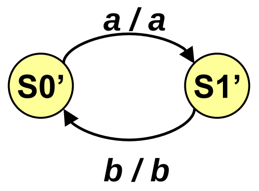
Communicating FSMs

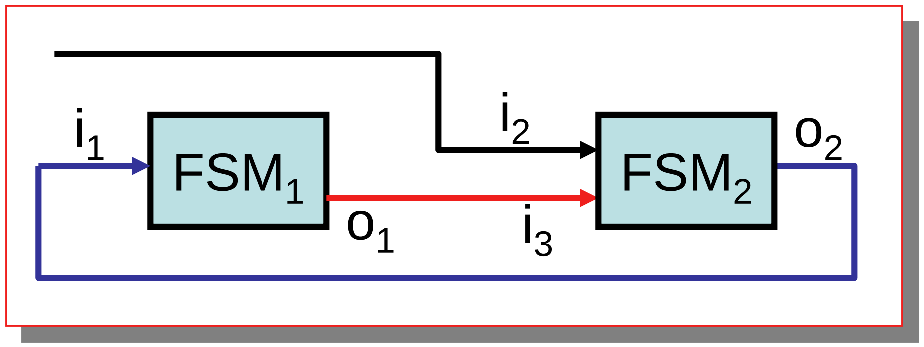
- **FSM composition**
  - SeatBelt: <S',I',O',f',h'>
    - S' = $S_1 \times S_2$ = {…,(**Wait,S1**),…}
    - $I' \subseteq I_1 \cup I_2$ = {*e*, *key*, *belt*}
    - $O' \subseteq O_1 \cup O_2$
    - f': $S_1 \times S_2 \times I' \rightarrow S_1 \times S_2$, s.t. $f' \in f_1 \times f_2$
    - h': $S_1 \times S_2 \times I' \rightarrow O'$, s.t. $h' \in h_1 \times h_2$
  - Connectivity constraints
    - Mapping of outputs to inputs: $f_i(s_i, \ldots, h_j(s_j, i_j), \ldots)$, $h_i(s_i, \ldots, h_j(s_j, i_j), \ldots)$
  - ➢ Synchronous execution
    - ➢ Simultaneous and instantaneous, zero delay hypothesis



- ➢ **Composability**
  - ➢ Moore
    - ➢ Well-defined, delayed
  - ➢ Mealy
    - ➢ Combinatorial cycles, consistency



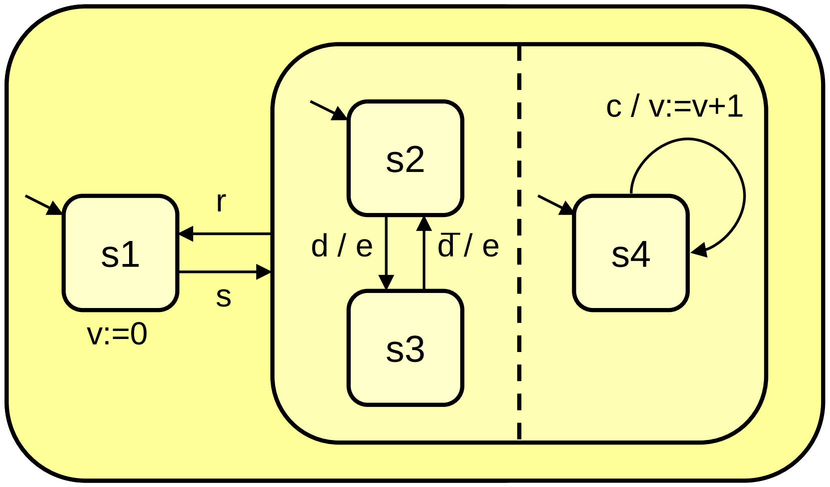




Source: M. Jacome, UT Austin.

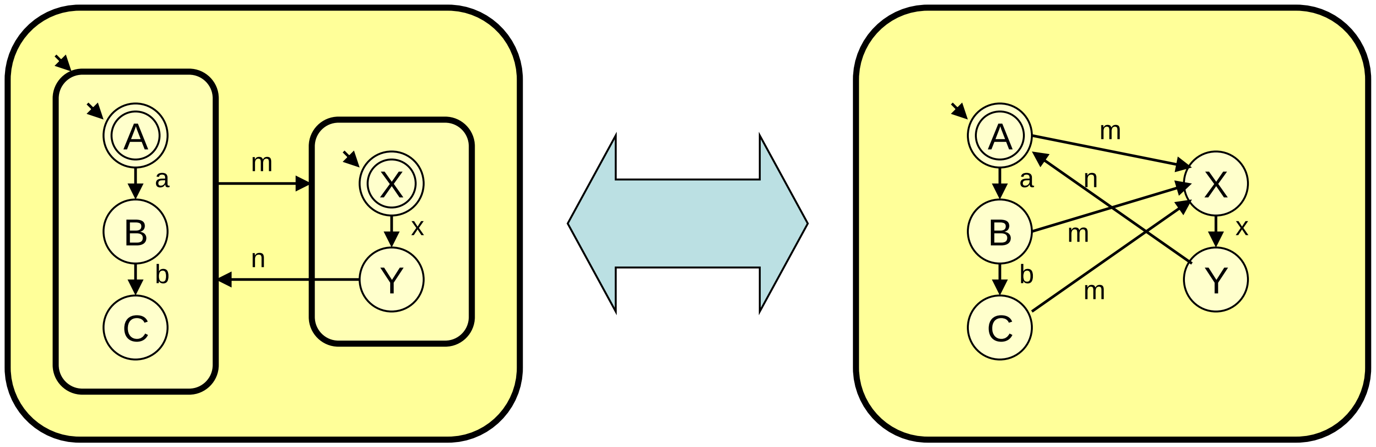
Hierarchical & Concurrent State Machines

- **Superstate FSM with Data (SFSMD)**
  - Hierarchy to organize and reduce complexity
    - – Superstates that contain complete state machines each
    - – Enter into one and exit from any substate
- **Hierarchical Concurrent FSM (HCFSM)**
  - Hierarchical and parallel state composition
    - – Lock-step concurrent composition and execution
  - Communication through global variables, signals and events
  - ➢ Graphical notation [StateCharts]

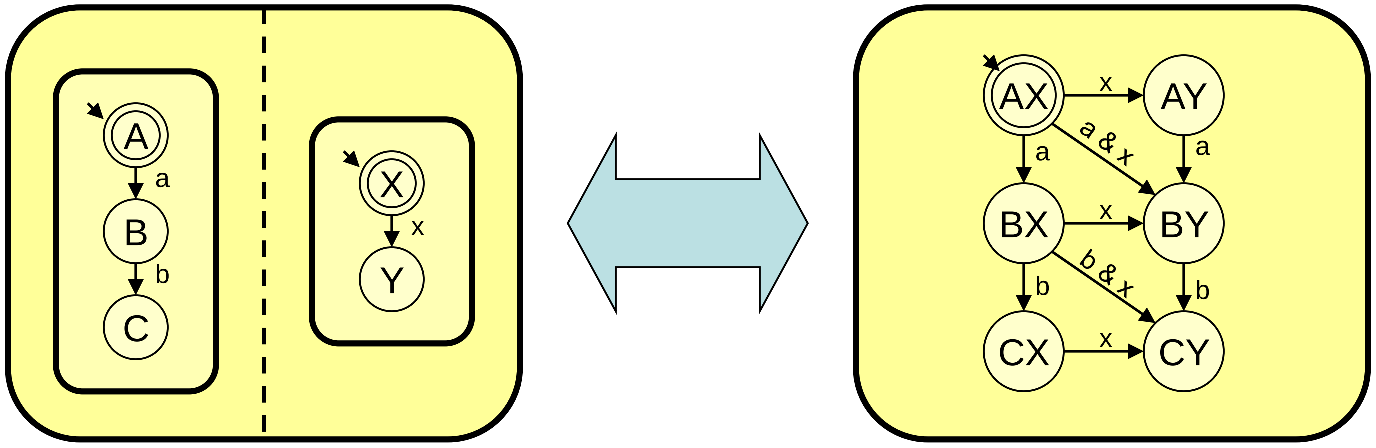


Managing Complexity and State Explosion

- **Hierarchy (OR state)**

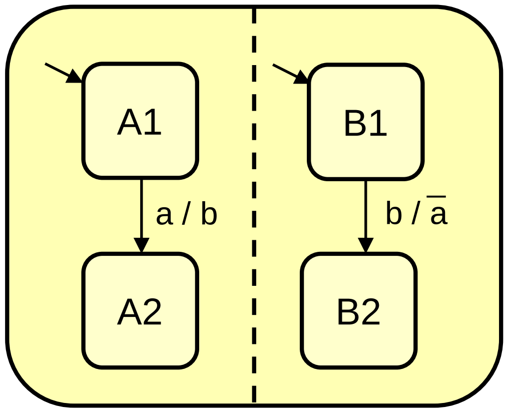


- **Concurrency (AND state)**



HCFSM Semantics

- **Reaction time upon external event?**
  - Synchronous, reactive: zero time, event broadcast



- **Event propagation?**
  - Grandfather paradox
    - Inconsistency or non-determinism
  - ➢ Synchronous
    - ➢ Reject cycles [Argos, Lustre]
    - ➢ Require fixed-point [SyncCharts/Esterel]

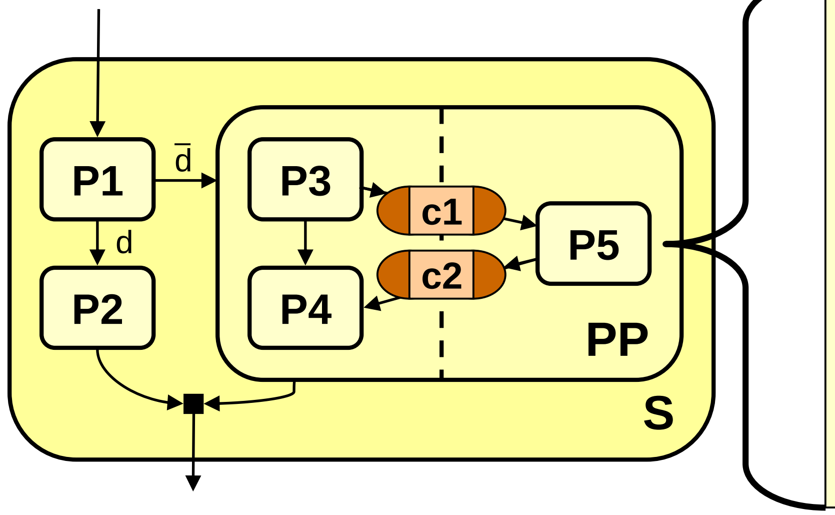
➢ ***N* micro-steps (internal) per macro-step (I/O) [Statemate]**
- Events posted in next and only in next micro step
  - “Synchronous”
    - One micro/macro step at regular times: delayed reaction, *not* synchronous (Moore)
  - “Asynchronous”
    - Zero-delay micro steps: causal chain reaction (Mealy, but: state updates in cycles)
- ➢ Deterministic
  - ➢ Together with other rules, e.g. priority of conflicting transitions

Source: “Statemate Course,” K. Baukus

Process State Machines

- **Sound combination of process and state based models**
  - Asynchronous concurrent HCFSM execution [UML]
    - Explicit event queues, deadlock analysis [PetriNet]
  - Globally asynchronous, locally synchronous (GALS)
    - Co-design Finite State Machines (CFSM) [Polis]
  - Leaf states are imperative processes
    - Program State Machine (PSM) [SpecSyn]
    - Transition-Immediately (TI) and Transition-on-Completion (TOC)

➢ Processes and abstract channels
  - Computation & communication
  - ➢ Process State Machine (PSM) [SpecC]



```
…
c1.receive(d,e);
a = 42;
while (a<100)
 { b = b + a;
   if (b > 50)
      c = c + d;
   else
      c = c + e;
   a = c;
  }
c2.send(a);
…
```

Lecture 3: Summary

- **Programming models**
  - Imperative, declarative: C, C++, Haskel, …
  - Synchronous, reactive: Esterel, Lustre
  - Asynchronous threads: Java
- **Process-based models: KPN, Dataflow, SDF**
  - ➢ Data dominated, block diagram level
- **State-based models: FSM(D), HCFSM**
  - ➢ Control dominated, machine level

➢ **Hybrid models**
  - ➢ Combination of process and state (data and control)
  - ➢ Behavior from specification down to implementation