EE382N.23: Embedded System Design and Modeling

Lecture 4 – Process-Based MoCs

Andreas Gerstlauer
Electrical and Computer Engineering
University of Texas at Austin
gerstl@ece.utexas.edu

Lecture 4: Outline

- **Process-based Models of Computation (MoCs)**
  - Processes and threads
  - (Kahn) Process networks
  - Dataflow
  - Process calculi

Types of Parallelism

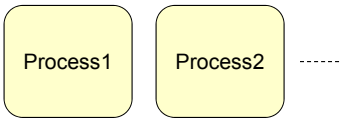
- **Task parallelism (MIMD)**
  - Multiple independent processes/threads
    - Separate code and data
  - Asynchronous operation
    - Explicit data communication & synchronization
- **Data parallelism (SIMD/SPMD)**
  - Multiple instances of same thread
    - Operating on independent pieces of data
  - Lockstep or bulk synchronous operation
    - Implicit barrier type of synchronization (fork-join)

➢ **Ideally independent of implementation model**
- Shared (SMP) vs. distributed (AMP) memory, SMT vs. SIMT
- ➢ Some combinations better implementable than others

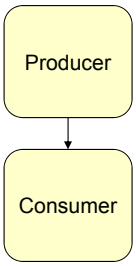
Process-Based Models

➢ **Activity and causality (data flow)**
- ➢ Asynchronous, coarse-grain concurrency

- **Set of processes/threads**
  - Execute in parallel
    - Concurrent composition
  - Each process is internally sequential
    - Imperative program



- **Inter-process communication**
  - Shared memory
    - Synchronization: critical section/mutex, monitor, …
  - Incomprehensible [Lee'06]
    - Non-determinism, race conditions, deadlocks, …



➢ **Traditional models are poor match [OS, Java]**
- Best effort, no correctness guarantees

Consider a Simple Example

"The Observer pattern defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically."

Eric Gamman Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*, Addision-Wesley, 1995

Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

Example: Observer Pattern in Java

```
public void addListener(listener) {…}

public void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```

Will this work in a multithreaded context?

Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

Observer Pattern with Mutexes

```
public synchronized void addListener(listener)
  {…}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```

Javasoft recommends against this.
What's wrong with it?

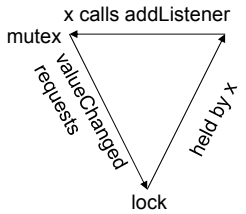
Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

Mutexes using Monitors are Minefields

```
public synchronized void addListener(listener)
  {…}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```

- **valueChanged() may attempt to acquire a lock on some other object and stall.**
- **If the holder of that lock calls addListener(): deadlock!**



Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

Observer Pattern Gets Complicated

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized (this) {
        myValue=newValue;
        listeners=myListeners.clone();
    }
    for (int i=0; i<listeners.length; i++) {
      listeners[i].valueChanged(newValue)
    }
}
```

while holding lock, make a copy of listeners to avoid race conditions

notify each listener outside of the synchronized block to avoid deadlock

This still isn't right. What's wrong with it?

Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

How to Make it Right?

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized (this) {
        myValue=newValue;
        listeners=myListeners.clone();
    }
    for (int i=0; i<listeners.length; i++) {
      listeners[i].valueChanged(newValue)
    }
}
```

Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object, but listeners may be notified in the opposite order. The listeners may be alerted to the value-changes in the wrong order!

Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

Problems with Thread-Based Concurrency

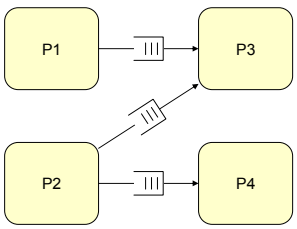
- ***Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans***
  - Nondeterministic, best effort
    - Explicitly prune away nondeterminism
  - Poor match for embedded systems
    - Lack of timing abstraction
  - Termination in reactive systems
    - Composability?

➢ Search for non-thread-based models: which are the requirements for appropriate specification techniques?

Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

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



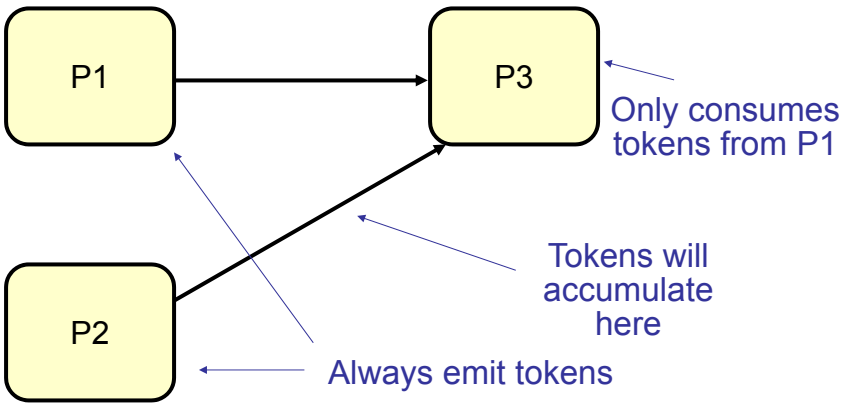
➢ **Deterministic**
  - Behavior does not depend on scheduling strategy
  - Focus on causality, not order (implementation independent)

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 `receive()` (or have otherwise ended)
- **Formal mathematical representation**
  - Process = continuous function mapping input to output streams
- **Turing-complete, undecidable (in finite time)**
  - Terminates (deadlocks)?
  - Can run in bounded buffers (memory)?

KPN Scheduling

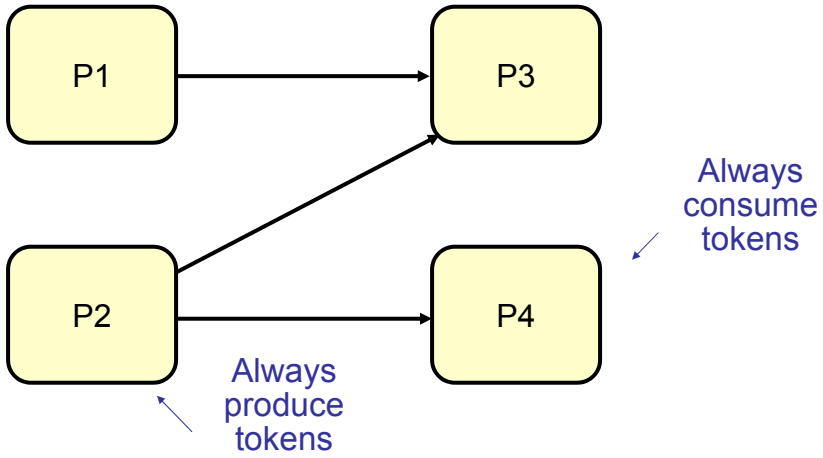
- **Scheduling determines memory requirements**
- **Data-driven scheduling**
  - Run processes whenever they are ready:



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

Demand-Driven Scheduling

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



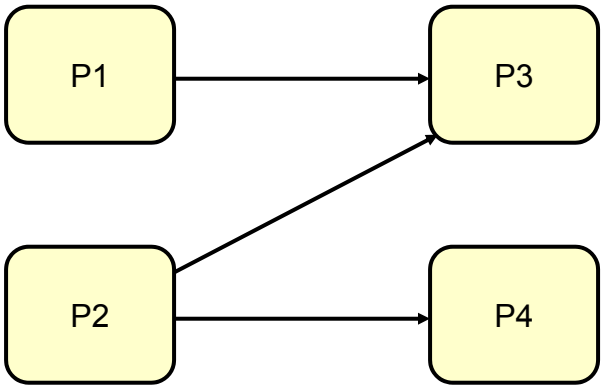
Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

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

Parks' Algorithm

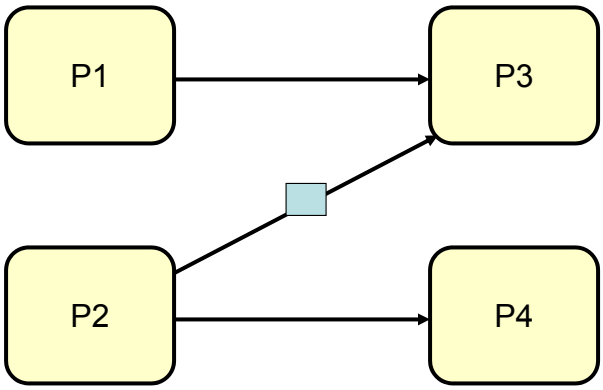
- **Start with buffer size 1**
- **Run P1, P2, P3, P4**



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

Parks' Algorithm

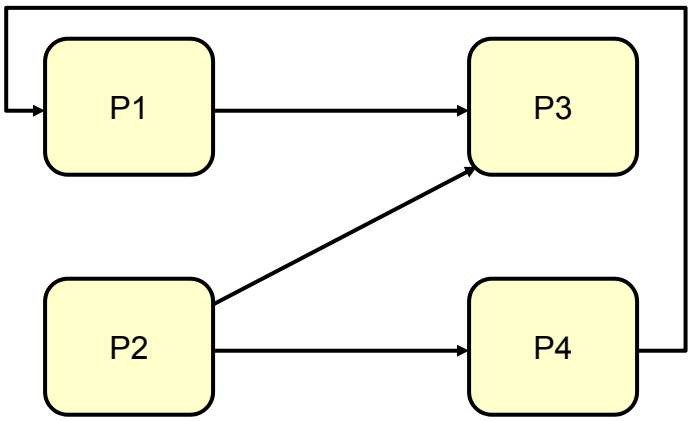
- **P2 blocked**
- **Run P1, P3, P1, … indefinitely**



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

Parks' Algorithm

- **But …**

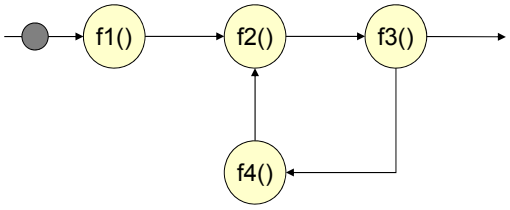


Kahn Process Networks (KPN)

- **Difficult to implement right**
  - Size of infinite FIFOs in limited physical memory?
  - Dynamic memory allocation, dependent on schedule
  - Dynamic scheduling & context switching
  - Boundedness vs. completeness vs. non-termination (deadlocks), are undecidable, depend on runtime schedule
  - Message-passing communication [MPI, Unix pipes]
  - How to model non-determinism? (e.g. merge process)
- **Parks' algorithm**
  - Bounded over complete (non-terminating) execution
  - ➢ Does not find every complete, bounded schedule [Geilen03]
  - ➢ Does not guarantee minimum memory usage
  - ➢ Deadlock detection?

Dataflow [Dennis74]

- **Breaking processes down into network of *actors***
  - Atomic blocks of computation, executed when *firing*
    - *Functional,* no side effects, no state: outputs purely a function of inputs
  - 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**



➢ **Signal-processing, dataflow machines [MIT in the 80s]**

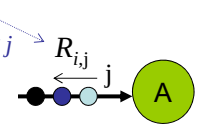
Firing Rules (1)

- **An actor with $p \geq 1$ input streams can have N firing rules**

$$\Re = \{\boldsymbol{R}_1, \boldsymbol{R}_2, \ldots, \boldsymbol{R}_N\}$$

  - The actor can fire if and only if one or more of the firing rules is satisfied
  - Each firing rule constitutes a set of patterns, one for each of the p inputs

$$\boldsymbol{R}_i = \{R_{i,1}, R_{i,2}, \ldots, R_{i,p}\}$$



  - A pattern $R_{i,j}$ is a (finite) sequence

For firing rule i to be satisfied, each pattern $R_{i,j}$ must form a **prefix** ⊑ of the **sequence of unconsumed tokens** at input j.

- **An actor with $p = 0$ input streams is always enabled (*source* actor)**

Source: M. Jacome, UT Austin.

Firing Rules (2)

- **For some firing rules, some patterns may be empty lists, $R_{i,j} = \perp$**
  - This means that any available sequence at input j is acceptable, because $\perp \sqsubseteq X$ for any sequence X
  - It does NOT mean that input j must be empty
- **Generalization of a *prefix algebra***
  - The symbol "*" denotes a token wildcard
  - The sequence [*] is a prefix of any sequence with at least one token
  - The sequence [*,*] is a prefix of any sequence with at least two tokens
  - …

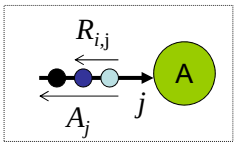
Source: M. Jacome, UT Austin.

Firing Rules (3)

- **Let A_j, for $j = 1,\ldots,p$, denote the sequence of available unconsumed tokens on the jth input of actor A**
- **Then the firing rule R_i is enabled if**

$$R_{i,j} \sqsubseteq A_j, \quad \text{for all } j = 1,\ldots,p$$

sequence $R_{i,j}$ is a prefix of sequence A_j

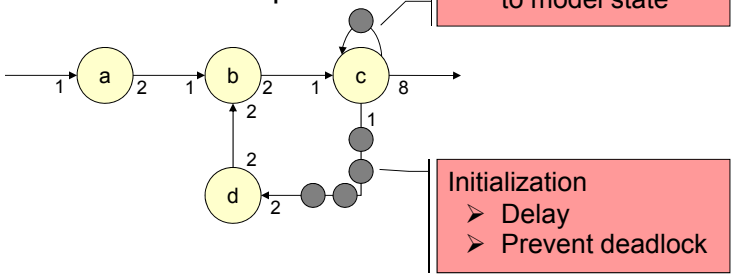


- **Dataflow is deterministic if firing rules are *sequential***
  - Sequence of blocking reads determines rule to apply
    - Blocking read of input one, followed by read of input two, etc.
  - Execute by repeatedly checking rules and firing
    - In some actor order → scheduling of actor firings (by some "interpreter")
- **Boundedness, completeness, non-termination?**
  - Still undecidable, still Turing-complete…

Source: M. Jacome, UT Austin.

Synchronous Dataflow (SDF) [Lee86]

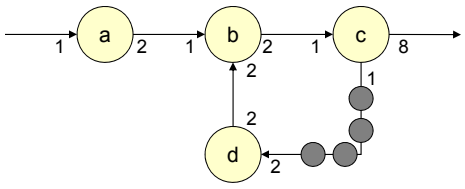
- **Fixed number of tokens per firing**
  - Consume fixed number of inputs
    - Single firing rule with fixed wildcard patterns
  - Produce fixed number of outputs



- **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)

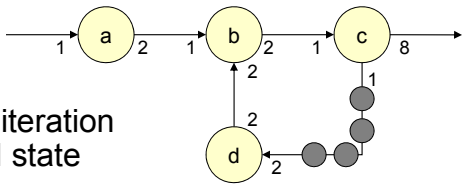
- **Solve system of linear rate equations**
  - Balance equations per arc
    - $2a = b$
    - $2b = c$
    - $b = d$
    - $2d = c$
    - $4a = 2b = c = 2d$



  - 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)

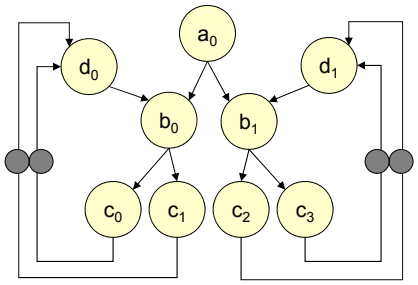
- **Periodically schedule actors in proportion to their rates**
  - Smallest integer solution
    - $4a = 2b = c = 2d$
    - ➢ $a = 1, b = 2, c = 4, d = 2$
  - Symbolically simulate one iteration of graph until back to initial state
    - *Admissible* iff no deadlock
    - Repeatedly execute this schedule
    - ➢ *adbccdbcc* = *a(2db(2c))*
    - ➢ *a(2db)(4c)*



  - ➢ Periodic admissible sequential scheduling (PASS)
    - ➢ Single processor: memory requirements (buffer size) vs. code size
      - *a(2db(2c))*: 2 token slots on each arc for total of 8 token buffers
      - *a(2db)(4c)*: 12 token buffers
      - ➢ Single appearance schedule & looped code generation
  - ➢ Periodic admissible parallel scheduling (PAPS)
    - ➢ Multi-processor: latency/throughput vs. buffer sizes

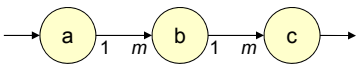
SDF Scheduling (3)

- **Precedence graph**
  - Homogeneous SDF (HSDF) conversion
    - All rates are 1, each node represents one actor instance/firing



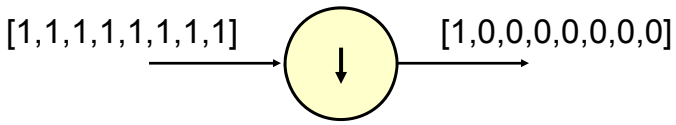
  - ➢ Scheduling = graph traversal

- ➢ **Worst-case exponential complexity**
  - Number of nodes in HSDF vs. SDF

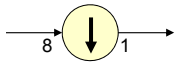


Cyclo-Static Dataflow (CSDF)

- **Periodic firings (cyclic pattern of token rates)**
  - Example: 8:1 Downsampler



    - First firing (phase): consume 1, produce 1
    - Second through eighth firing (phase): consume 1, produce 0 (discard)
  - Downsampler in SDF
    - Wait for and store all 8 input tokens before output

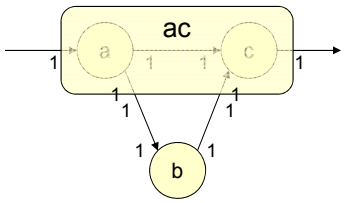


- **Static scheduling in similar manner as SDF**
  - Convert to SDF by lumping phases into actor cycles
  - Compute cycle-level repetition vector
  - Schedule phase-level precedence graph

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

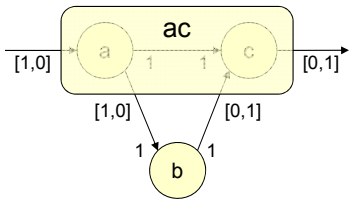
Composability

- **Hierarchically compose subgraphs into super-actors**



  - Deadlocks!
  - Not a valid composition

- **SDF is not composable, but CSDF is**

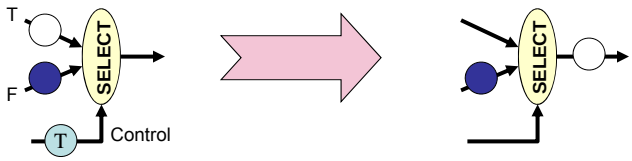


  - SDF → CSDF conversion can similarly introduce deadlock (see here)

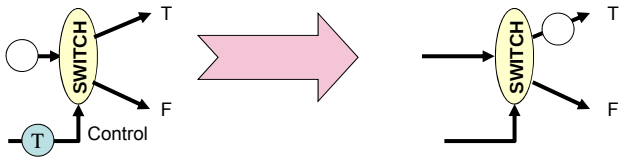
Source: T. Parks et al.. "A Comparison of synchronous and Cyclo-Static Dataflow", Asilomar, 1995.

Boolean Dataflow (BDF)

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



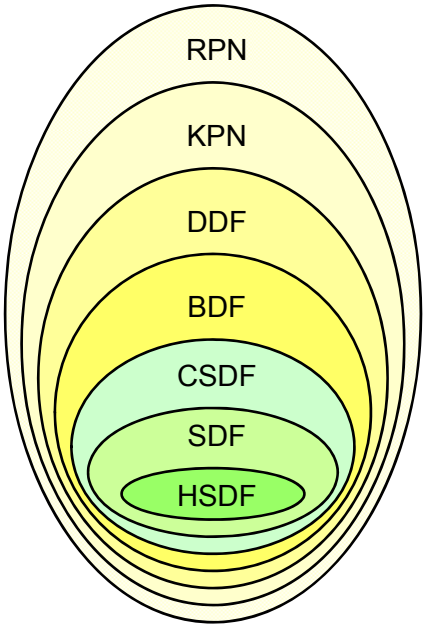
  - Switch actor



- **Touring complete**
  - Loops, branches
  - Quasi-static scheduling [Buck'93]

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

Source: T. Basten, MoCC 2008.

Dataflow Variants

- **Dynamic dataflow extensions**
  - Structured dataflow [LabView's G language]
    - If-then-else, switch-case with analyzable semantics
  - Modal models
    - Reactive process networks (RPN) [Geilen'04]
    - Parameterized dataflow (PDF) [Bhattacharya'01]
    - Heterochronous dataflow (HDF) [Lee'05]
    - Scenario-aware dataflow (SADF) [Theelen'06]
    - ➢ Parameter changes between iterations driven by state machine model
  - Dataflow languages [StreamIt]
    - Deterministic peeking, teleport messages that bypass regular flow
- **Timed (cyber-physical) dataflow extensions**
  - Time synchronous dataflow (TSDF) [Agilent ADS]
    - Fixed sampling/execution rates on arcs and actors
  - Hybrid continuous-discrete time models
    - Discrete models as piecewise constant continuous signals [Simulink]
    - Sampling at discrete/continuous interfaces [SystemC-AMS]

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 4: Summary

- **Models of Computation (MoCs)**
  - Formally express behavior
- **Process-based models**
  - OS processes/threads not manageable
  - Restricted models of communication (KPN)
  - Restricted models of computation (dataflow)
  - ➢ Tradeoffs between analyzability vs. expressiveness
- ➢ **Task-level parallelism, dataflow driven**