EE382V: Embedded System Design and Modeling

Lecture 5 – Models of Computation

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

Lecture 5: Outline

- **Models of Computation (MoCs)**
  - Concurrency & communication
- **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 ways of describing system behavior**
  - Distinguish abstract classes of behavioral modeling
    - Concurrency and time (order)
    - Computation and communication
  - Decomposition into objects and their relationship
    - Composition rules
    - Data and control flow
  - Unambiguous, formal definition and semantics
    - Formal analysis and reasoning for synthesis and verification
    - Various degrees of complexity and expressiveness
- **Analyzability and expressiveness of specification models**
  - Fundamental tradeoffs
    - Turing complete models are proofably not analyzable
  - Implementability & predictability
    - Low overhead, early exploration

Models of Computation (MoCs)

- **MoC examples**
  - Programming models: imperative, declarative
    - Statements for algorithmic computation, operation-level granularity
  - Simulation models: synchronous, discrete event
    - Basic concurrency, general interaction 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
- **Circuit and logic design**
  - Sequential circuit optimization: FSM

Models of Computation (MoCs)

- **A MoC is a framework in which to express what actions must be taken to complete a computation**
  - Objects and their relationships
- **MoCs need to**
  - Be *powerful/expressive enough* for the application domain
  - Have appropriate *synthesis* and *validation* semantics
- **Why different models?**
  - Different models ⇒ different properties
  - Turing complete models are too powerful!
  - Imperative programming models are poor match
    - Reactive instead of transformation systems
  - ➢ Domain-specific models

Source: M. Jacome, UT Austin.

Properties

- **A property is an assertion about the behavior, rather than a description of the behavior**
  - It is an abstraction of the behavior along a particular axis
- **Examples:**
  - *Liveness* property: when designing a network protocol, one may require that the design never *deadlocks*
  - *Fairness* property: when designing a network protocol, one may require that *any request will eventually be satisfied*

  The above properties do not completely specify the behavior of the protocol, they are instead properties we require the protocol to have
- **Can include other non-functional requirements**
  - *Timeliness*: guarantees about meeting deadlines in the worst case (*real-time*)

Source: M. Jacome, UT Austin.

Properties & MoCs

- **Properties can be classified in three groups:**
  1. Properties that are *inherent* to the model (i.e., that can be shown formally to hold for *all specifications* described using that model)
  2. Properties that can be verified *syntactically* for a given specification (i.e., that can be shown to hold with a simple, usually polynomial-time analysis of the specification)
  3. Properties that must be verified *semantically* for a given specification (i.e., that can be shown to hold by executing, at least implicitly, the specification for all inputs that can occur)

Source: M. Jacome, UT Austin.

Model Validation

- **By construction**
  - property is inherent
- **By verification**
  - property is provable syntactically
- **By simulation**
  - check behavior for all inputs
- **By intuition**
  - property is true, I just know it is…

better be higher in this list…

Source: M. Jacome, UT Austin.

(Engineering) Models vs. Reality

- **"You can't strike oil by drilling through a map" [Solomon'68]**
  - Yet, maps are incredibly useful

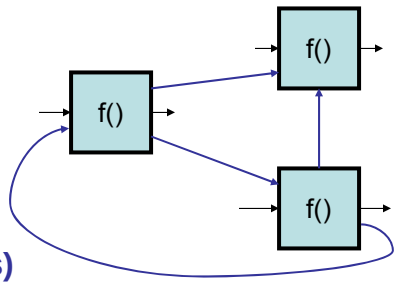
➢ **We can make definitive statements about models from which we can *infer* properties of system realizations [Kopetz]**
  - ➢ Validity of inference depends on model fidelity
  - ➢ Always approximate

➢ **Assertions about (predicted) properties are always assertions about a model of the system**
  - ➢ Never truly properties of the final implemented system

Source: E. Lee, CEDA Keynote, DAC'13.

Recap: Concurrency and Time

- **Logical concurrency (time)**
  - Partial order (undefined)
- **Synchronization (dependencies)**
  - Restrictions on order (causality)



➢ **Fundamental issues**
  - ➢ Non-determinism
  - ➢ Deadlocks

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)

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

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?

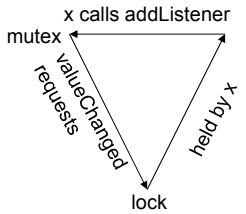
Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

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

Lecture 5: Outline

✓ Models of Computation (MoCs)

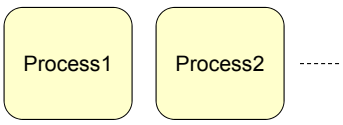
- **Process-based MoCs**
  - Process networks
  - Dataflow
  - Process calculi
- State-based MoCs

Types of Parallelism

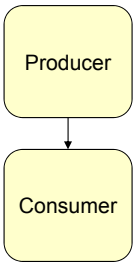
- **Task parallelism (MIMD)**
  - Multiple independent processes
    - Separate code and data
  - Asynchronous operation
    - Explicit data communication & synchronization
- **Data parallelism (SIMD/SIMT)**
  - Multiple instances of same thread
    - Operating on independent pieces of data
  - Bulk synchronous operation
    - Implicit barrier type of synchronization (fork-join)
- ➢ **Ideally independent of implementation model**
  - Shared vs. distributed memory
  - ➢ Some combinations better implementable than others

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



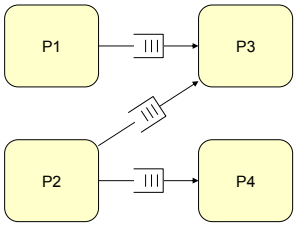
- **Inter-process communication**
  - Shared memory [OpenMP]
    - Synchronization: critical section/mutex, monitor, …
  - Message passing [MPI]
    - Synchronous, rendezvous (blocking send)
    - Asynchronous, queues (non-blocking send)



- ➢ **Implementation: OS processes or threads**
  - ➢ Single or multiple processors/cores

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



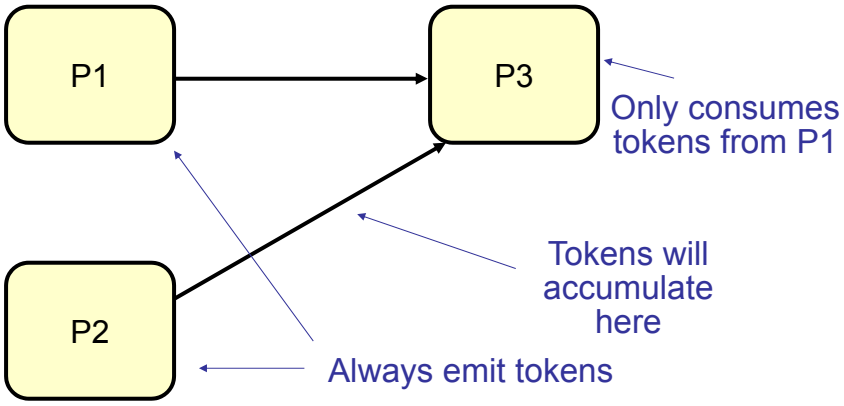
- **Deterministic**
  - Behavior does not depend on scheduling strategy
  - Focus on causality, not order (implementation independent)

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?
  - Can run in bounded buffers (memory)?

KPN Scheduling

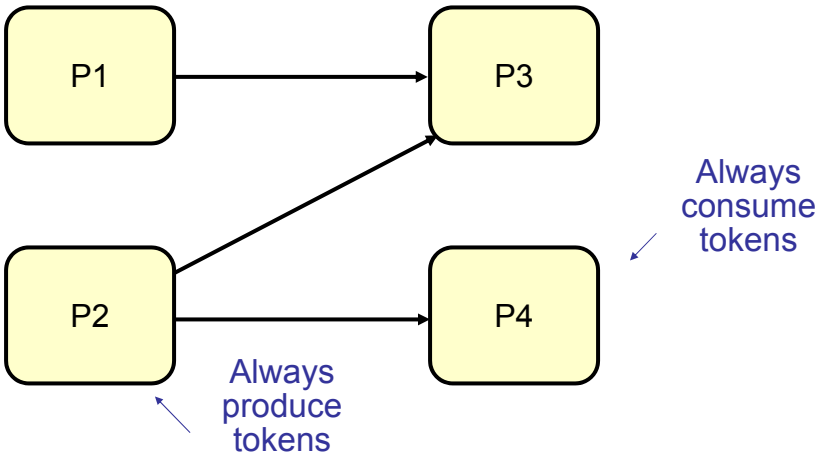
- **Scheduling determines memory requirements**
- **Data-driven scheduling**
  - Run processes whenever they are ready:



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...**



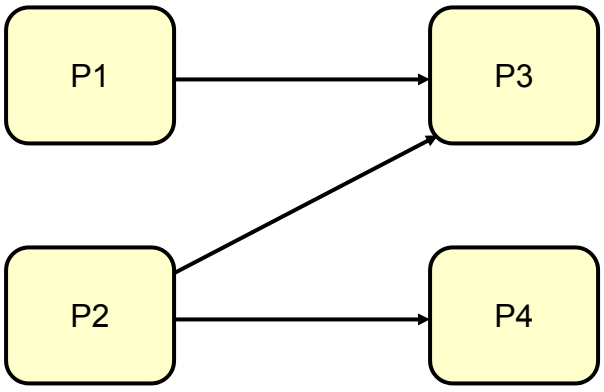
Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

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

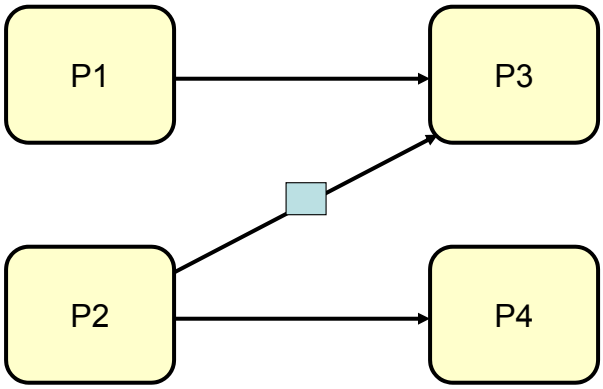
- **Start with buffer size 1**
- **Run P1, P2, P3, P4**



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

Parks' Algorithm

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



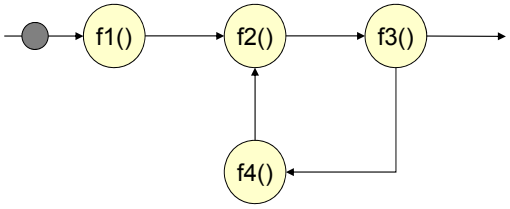
Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

Kahn Process Networks (KPN)

- **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 [Dennis'74]

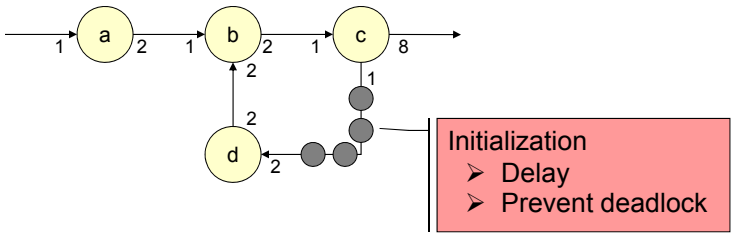
- **Breaking processes down into network of *actors***
  - Atomic blocks of computation, executed when *firing*
  - Fire when required number of input *tokens* are available
    - Consume required number of tokens on input(s)
    - Produce number of tokens on output(s)
  - ➢ Separate computation & communication/synchronization
    - ➢ 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]

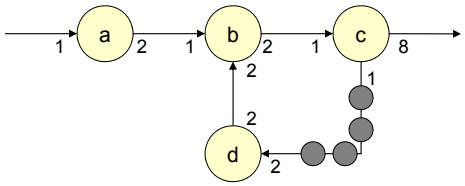
- **Fixed number of tokens per firing**
  - Consume fixed number of inputs
  - Produce fixed number of outputs



- ➢ **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)

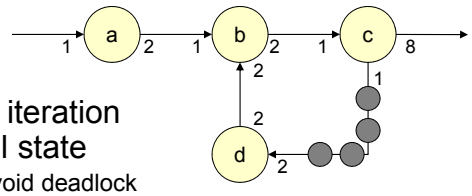
- **Solve system of linear rate equations**
  - Balance equations per arc
    - $2a = b$
    - $2b = c$
    - $b = d$
    - $2d = c$
    - ➢ $4a = 2b = c = 2d$



  - 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)

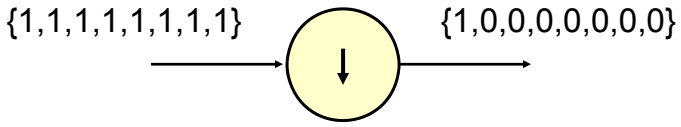
- **Periodically schedule actors in proportion to their rates**
  - Smallest integer solution
    - $4a = 2b = c = 2d$
    - ➢ $a = 1, b = 2, c = 4, d = 2$



  - Symbolically simulate one iteration of graph until back to initial state
    - Insert initialization tokens to avoid deadlock
    - ➢ *adbccdbcc = a(2db(2c))*
    - ➢ *a(2db)(4c)*
  - ➢ Single-processor/sequential scheduling (PASS)
    - ➢ Memory requirements 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
  - ➢ Multi-processor/parallel scheduling (PAPS)
    - ➢ Latency/throughput vs. buffer sizes

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



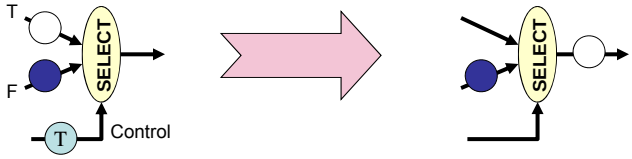
  - ➢ First firing: consume 1, produce 1
  - ➢ Second through eighth firing: consume 1, produce 0

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

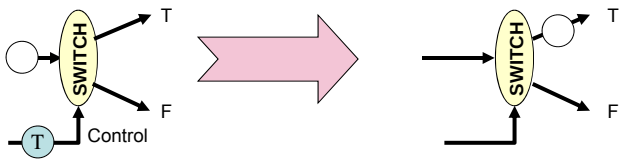
Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

Boolean Dataflow (BDF)

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



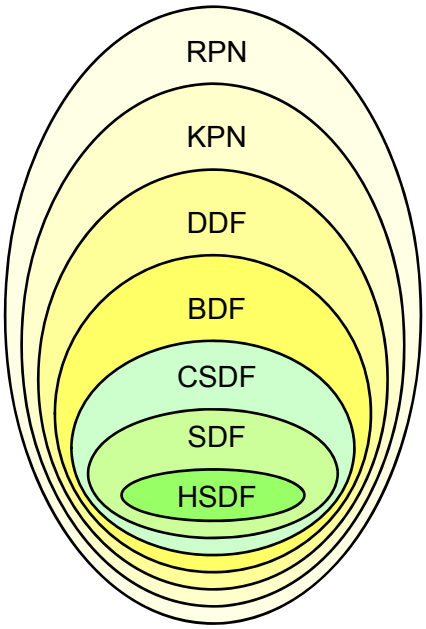
  - Switch actor



- ➢ **Touring 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

Source: T. Basten, MoCC 2008.

Dataflow Variants

- **Dynamic dataflow models**
  - Structured dataflow [LabView's G language]
    - If-then-else, switch-case with analyzable semantics
  - Modal models
    - 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
- **Timed 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]
  - ➢ Cyber-physical systems (CPS)

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 5: Outline

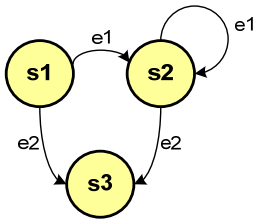
✓ System specification

✓ Process-based MoCs

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

State-Based Models

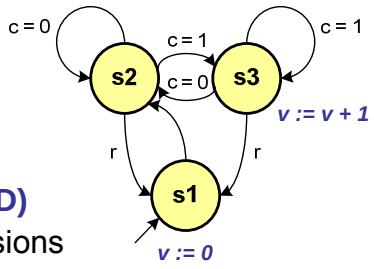
- **Status and reactivity (control flow)**
- **Explicit enumeration of computational states**
  - State represents captured history
- **Explicit flow of control**
  - Transitions in reaction to events



- **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

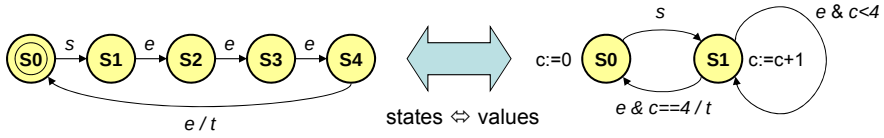
- **Finite State Machine (FSM)**
  - Basic model for describing control and automata
    - Sequential circuits
  - States *S*, inputs/outputs *I/O*, and state transitions
    - FSM: <*S*, *I*, *O*, *f*, *h*>
    - Next state function *f*: $S \times I \rightarrow S$
    - Non-deterministic: *f* is multi-valued
  - 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



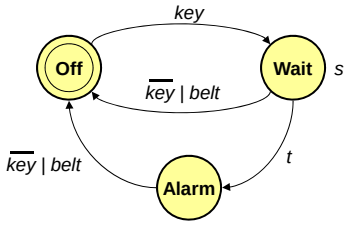
- **Finite State Machine with Data (FSMD)**
  - Computation as control and expressions
    - Controller and datapath of RTL processors
  - FSM plus variables *V*
    - FSMD: <*S*, *I*, *O*, *V*, *f*, *h*>
    - Next state function *f*: $S \times V \times I \rightarrow S \times V$
    - Output function *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), $f: S \times V \times 2^I \rightarrow S \times V$, $h: S \times V \times 2^I \rightarrow 2^O$

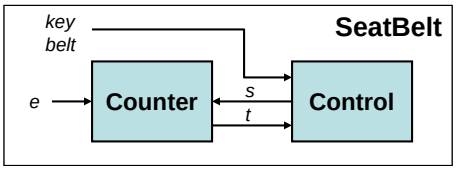


- **Non-Deterministic FSM (NFSM)**
  - Choice in control
    - Implicit self-loops for unspecified conditions? Usually!
    - **Wait**: *belt* & *t* ?
    - Multiple arcs for same condition?
    - Incomplete specification (undecided), unknown behavior (don't care)

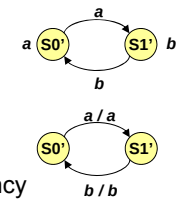


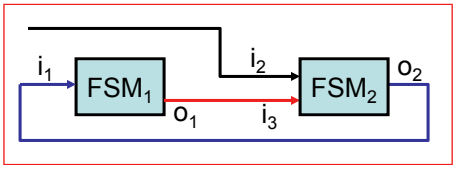
Communicating FSMs

- **FSM composition**
  - SeatBelt: <S',I',O',f',h'>
    - $S' = S_1 \times S_2 = \{\ldots,(\textbf{Wait,S1}),\ldots\}$
    - $I' \subseteq I_1 \cup I_2 = \{e, key, belt\}$
    - $O' \subseteq O_1 \cup O_2$
    - $f': S_1 \times S_2 \times I' \rightarrow S_1 \times S_2$, 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 concurrency
    - Simultaneous, lock step, zero delay hypothesis



- **Composability**
  - Moore
    - Delayed
    - Well-defined
  - Mealy
    - Instantaneous
    - Cycles, consistency

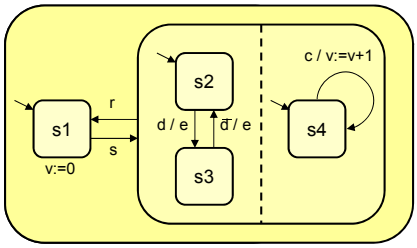




Source: M. Jacome, UT Austin.

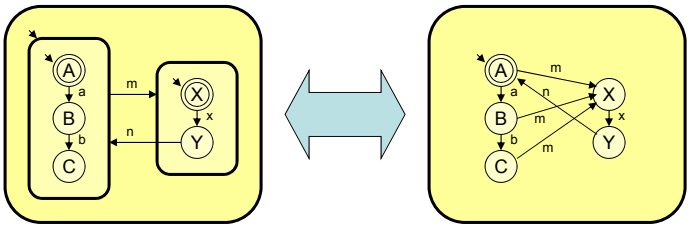
Hierarchical & Concurrent State Machines

- **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]

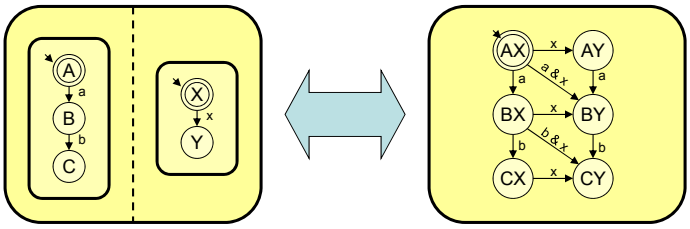


Managing Complexity and State Explosion

- **Hierarchy (OR state)**

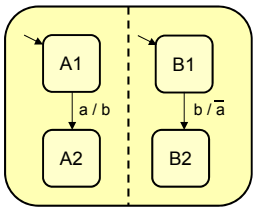


- **Concurrency (AND state)**



HCFSM Semantics

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



  - **Event propagation?**
    - Grandfather paradox
      - Inconsistent cycles, non-determinism
    - ➢ Synchronous reactive (SR) model
      - ➢ Reject cycles [Argos, Lustre]
      - ➢ Require fixed-point [SyncCharts/Esterel]

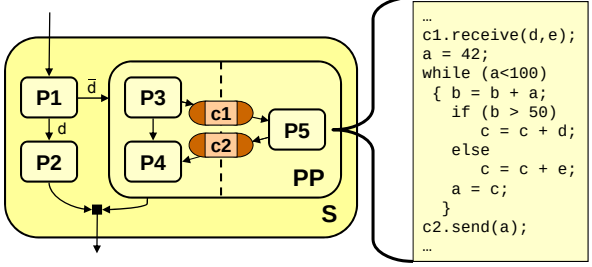
- ➢ ***N* micro-steps (int.) per macro-step (ext.) [Statemate]**
  - Events posted in next and only in next micro step
    - "Synchronous"
      - One micro/macro step at regular times: delayed reaction, *not* synchronous (Moore)
    - "Asynchronous"
      - Zero-delay micro steps: causal chain reaction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

Source: "Statemate Course," K. Baukus

Process and State Based Models

- **From synchronous to asynchronous compositions…**
  - Asynchronous concurrency in HCFSMs [UML]
    - Explicit event queues, deadlock analysis [PetriNet]
  - Processes are state machines
    - Globally asynchronous, locally synchronous (GALS) systems
    - Co-design Finite State Machines (CFSM) [Polis]
  - States are processes
    - Imperative leaf states, transition-immediately (TI) and on completion (TOC): Program State Machine (PSM) [SpecCharts]
    - States with continuous process networks [*Charts], hybrid continuous/discrete time models

- ➢ Arbitrary hierarchy
  - ➢ Process State Machine (PSM) [SpecC]
  - ➢ Heterogeneous MoCs [Ptolemy]



Lecture 5: Summary

- **Models of Computation (MoCs)**
  - Formally express behavior
- **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