Summary of MOCs

- Synchronous languages/synchronous EFSMs
  - syntactic determinism
  - totally ordered tags
    - explicit control and scheduling
    - explicit pipelining and buffering
  - reactive
- Data Flow networks
  - inherent determinism
  - partially ordered tags
    - scheduling, pipelining and buffering freedom
  - non-reactive (“blocking read”)
  - undecidable (above SDF)

Summary of MOCs

- Petri nets
  - syntactic determinism
  - partially ordered tags
  - reactive (multiple fanin/fanout for places)
  - decidable (uninterpreted)
- Discrete Events
  - inherently non-deterministic
  - totally ordered tags
  - “least common multiple” between MOCs
Co-design Finite State Machines

Three-level hierarchy
- top level: asynchronous, partially ordered
  (bounded buffer non-blocking single-read communication)
- middle level: synchronous FSM
  (atomic event- and condition-based transition)
- bottom level: SDF-like
  (FSM provides tokens and selects active sub-network)

Refinement (implementation) mapping:
- embed partial order into total order

Choose execution time for EFSM transitions:
- delay = 0
  - synchronous collapsing (Esterel-style)
  - determines task granularity
- delay > 0
  - asynchronous composition
  - delay determined by implementation choice
    (HW vs. SW, different processor and communication architecture, …)
Co-design Finite State Machines

- Delay choice implies (unwanted) non-determinism
  - currently: make assumptions on *scheduling* (relative priority of CFSMs -> no “event loss”)
  - future: define *abstraction mapping* yielding deterministic behavior

- Two levels of partitioning:
  - =0 vs. >0: task partitioning
  - choice of >0: HW/SW partitioning

Communication primitive: event

- One-way data communication
- Need efficient implementation (interrupts, buffers...)
- No mutual synchronization requirement, but...
  - Building block for higher-level synchronization primitives

- Examples:
  - *valued event*: temperature sample
  - *pure event*: excessive temperature alarm
Outline

- Synchrony and asynchrony
- CFSM definitions
  - Signals & networks
  - Timing behavior
  - Functional behavior
- CFSM & process networks
- Example of CFSM behaviors
  - Equivalent classes

Codesign Finite State Machine

- Underlying MOC of Polis
- Combine aspects from several other MOCs
- Preserve formality and efficiency in implementation
- Mix
  - synchronicity
    - zero and infinite time
  - asynchronicity
    - non-zero, finite, and bounded time
- Embedded systems often contain both aspects
Synchrony: Basic Operation

- Synchrony is often implemented with clocks
- At clock ticks
  - Module reads inputs, computes, and produce output
  - All synchronous events happen simultaneously
  - Zero-delay computations
- Between clock ticks
  - Infinite amount of time passed

Synchrony: Basic Operation (2)

- Practical implementation of synchrony
  - Impossible to get zero or infinite delay
  - Require: computation time $<<$ clock period
  - Computation time = 0, w.r.t. reaction time of environment
- Feature of synchrony
  - Functional behavior independent of timing
    - Simplify verification
  - Cyclic dependencies may cause problem
    - Among (simultaneous) synchronous events
Synchrony: Triggering and Ordering

- All modules are triggered at each clock tick
- Simultaneous signals
  - No a priori ordering
  - Ordering may be imposed by dependencies
    - Implemented with delta steps

Synchrony: System Solution

- System solution
  - Output reaction to a set of inputs
- Well-designed system:
  - Is completely specified and functional
  - Has an unique solution at each clock tick
  - Is equivalent to a single FSM
  - Allows efficient analysis and verification
- Well-design-ness
  - May need to be checked for each design (Esterel)
    - Cyclic dependency among simultaneous events
Synchrony: Implementation Cost

- Must verify synchronous assumption on final design
  - May be expensive
- Examples:
  - Hardware
    - Clock cycle > maximum computation time
      - Inefficient for average case
  - Software
    - Process must finish computation before
      - New input arrival
      - Another process needs to start computation

Asynchrony: Basic Operation

- Events are never simultaneous
  - No two events have the same tag
- Computation starts at a change of the input
- Delays are arbitrary, but bounded
Asynchrony: Triggering and Ordering

- Each module is triggered to run at a change of input
- No a priori ordering among triggered modules
  - May be imposed by scheduling at implementation

Asynchrony: System Solution

- Solution strongly dependent on input timing
- At implementation
  - Events may “appear” simultaneous
  - Difficult/expensive to maintain total ordering
    - Ordering at implementation decides behavior
    - Becomes DE, with the same pitfalls
Asynchrony: Implementation Cost

- Achieve low computation time (average)
  - Different parts of the system compute at different rates
- Analysis is difficult
  - Behavior depends on timing
  - Maybe be easier for designs that are insensitive to
    - Internal delay
    - External timing

Asynchrony vs. Synchrony in System Design

- They are different at least at
  - Event buffering
  - Timing of event read/write
- Asynchrony
  - Explicit buffering of events for each module
    - Vary and unknown at start-time
- Synchrony
  - One global copy of event
    - Same start time for all modules
Combining Synchrony and Asynchrony

- Wants to combine
  - Flexibility of asynchrony
  - Verifiability of synchrony

- Asynchrony
  - Globally, a timing independent style of thinking

- Synchrony
  - Local portion of design are often tightly synchronized

- Globally asynchronous, locally synchronous
  - CFSM networks

CFSM Overview

- CFSM is FSM extended with
  - Support for data handling
  - Asynchronous communication

- CFSM has
  - FSM part
    - Inputs, outputs, states, transition and output relation
  - Data computation part
    - External, instantaneous functions
CFSM Overview (2)

- CFSM has:
  - Locally synchronous behavior
    - CFSM executes based on snap-shot input assignment
    - Synchronous from its own perspective
  - Globally asynchronous behavior
    - CFSM executes in non-zero, finite amount of time
    - Asynchronous from system perspective

- GALS model
  - Globally: Scheduling mechanism
  - Locally: CFSMs

Network of CFSMs: Depth-1 Buffers

- Globally Asynchronous, Locally Synchronous (GALS) model
Introducing a CFSM

- A Finite State Machine
- Input events, output events and state events
- Initial values (for state events)
- A transition function
  - Transitions may involve complex, memory-less, instantaneous arithmetic and/or Boolean functions
  - All the state of the system is under form of events
- Need rules that define the CFSM behavior

CFSM Rules: phases

- Four-phase cycle:
  1. Idle
  2. Detect input events
  3. Execute one transition
  4. Emit output events
- Discrete time
  - Sufficiently accurate for synchronous systems
  - Feasible formal verification
- Model semantics: Timed Traces i.e. sequences of events labeled by time of occurrence
CFSM Rules: phases

- Implicit *unbounded delay* between phases
- *Non-zero* reaction time
  (avoid *inconsistencies* when interconnected)
- *Causal model based on partial order*
  *(global asynchronicity)*
  - potential verification speed-up
- Phases *may not overlap*
- Transitions always *clear input buffers*
  *(local synchronicity)*

Communication Primitives

- **Signals**
  - Carry information in the form of events and/or values
    - Event signals: present/absence
    - Data signals: arbitrary values
      - Event, data may be paired
  - Communicate between two CFSMs
    - 1 input buffer / signal / receiver
  - Emitted by a sender CFSM
  - Consumed by a receiver CFSM by setting buffer to 0
  - “Present” if emitted but not consumed
Communication Primitives (2)

- **Input assignment**
  - A set of values for the input signals of a CFSM
- **Captured input assignment**
  - A set of input values read by a CFSM at a particular time
- **Input stimulus**
  - Input assignment with at least one event present

Signals and CFSM

- **CFSM**
  - Initiates communication through events
  - Reacts only to input stimulus
    - except initial reaction
  - Writes data first, then emits associated event
  - Reads event first, then reads associated data
CFSM networks

- **Net**
  - A set of connections on the same signal
  - Associated with single sender and multiple receivers
  - An input buffer for each receiver on a net
    - Multi-cast communication

- **Network of CFSMs**
  - A set of CFSMs, nets, and a scheduling mechanism
  - Can be implemented as
    - A set of CFSMs in SW (program/compiler/OS/uC)
    - A set of CFSMs in HW (HDL/gate/clocking)
    - Interface (polling/interrupt/memory-mapped)

Scheduling Mechanism

- **At the specification level**
  - Should be as abstract as possible to allow optimization
  - Not fixed in any way by CFSM MOC

- **May be implemented as**
  - RTOS for single processor
  - Concurrent execution for HW
  - Set of RTOSs for multi-processor
  - Set of scheduling FSMs for HW
Timing Behavior

- **Scheduling Mechanism**
  - Globally controls the interaction of CFSMs
  - Continually deciding which CFSMs can be executed

- **CFSM can be**
  - Idle
    - Waiting for input events
    - Waiting to be executed by scheduler
  - Executing
    - Generate a single reaction
    - Reads its inputs, computes, writes outputs

Timing Behavior: Mathematical Model

- **Transition Point**
  - Point in time a CFSM starts executing

- **For each execution**
  - Input signals are read and cleared
  - Partial order between input and output
  - Event is read before data
  - Data is written before event emission
Timing Behavior: Transition Point

- A transition point $t_i$
  - Input may be read between $t_i$ and $t_{i+1}$
  - Event that is read may have occurred between $t_{i-1}$ and $t_{i+1}$
  - Data that is read may have occurred between $t_0$ and $t_{i+1}$
  - Outputs are written between $t_i$ and $t_{i+1}$

- CFSM allow loose synchronization of event & data
  - Less restrictive implementation
  - May lead to non-intuitive behavior

Event/Data Separation

- Value $v_1$ is lost even though
  - It is sent with an event
  - Event may not be lost
- Need atomicity
Atomicity

- Group of actions considered as a single entity
- May be costly to implement
- Only atomicity requirement of CFSM
  - Input event are read atomically
    - Can be enforced in SW (bit vector) HW (buffer)
    - CFSM is guarantee to see a snapshot of input events
- Non-atomicity of event and data
  - May lead to undesirable behavior
  - Atomized as an implementation trade-off decision

Non Atomic Data Value

Reading

- Receiver R1 gets (X=4, Y=5), R2 gets (X=5 Y=4)
- X=4 Y=5 never occurs
- Can be remedied if values are sent with events
  - still suffers from separation of data and event
**Atomicity of Event Reading**

- R1 sees no events, R2 sees X, R3 sees X, Y
- Each sees a snapshot of events in time
- Different captured input assignment
  - Because of scheduling and delay

---

**Functional Behavior**

- Transition and output relations
  - input, present_state, next_state, output
- At each execution, a CFSM
  - Reads a captured input assignment
  - If there is a match in transition relation
    - consume inputs, transition to next_state, write outputs
  - Otherwise
    - consume no inputs, no transition, no outputs
Functional Behavior (2)

- Empty Transition
  - No matching transition is found

- Trivial Transition
  - A transition that has no output and no state changes
  - Effectively throw away inputs

- Initial transition
  - Transition to the init (reset) state
  - No input event needed for this transition

CFSM and Process Networks

- CFSM
  - An asynchronous extended FSM model
  - Communication via bounded non-blocking buffers
    - Versus CSP and CCS (rendezvous)
    - Versus SDL (unbounded queue & variable topology)
  - LTS w/ edge involving presence/absence of multiple signals
    - Versus CSP, CCS, SDL (one symbol)
    - Versus dataflow (presence only)
  - Not continuous in Kahn’s sense
    - Different event ordering may change behavior
      - Versus dataflow (ordering insensitive)
CFSM Networks

- Defined based on a global notion of time
  - Total order of events
  - Synchronous with relaxed timing
    - Global consistent state of signals is required
    - No local ordering involving of tags among
      - Input signals
      - Output signals
    - Input and output are in partial order

Buffer Overwrite

- CFSM Network has
  - Finite Buffering
  - Non-blocking write
    - Events can be overwritten
      - if the sender is “faster” than receiver

- To ensure no overwrite
  - Explicit handshaking mechanism
  - Scheduling
Example of CFSM Behaviors

- A and B produce i1 and i2 at every i
- C produce err or o at every i1,i2
- Delay (i to o) for normal operation is \( n_r \), err operation 2\( n_r \)
- Minimum input interval is \( n_i \)
- Intuitive “correct” behavior
  - No events are lost

Equivalent Classes of CFSM Behavior

- Assume parallel execution (HW, 1 CFSM/processor)
- Equivalent classes of behaviors are:
  - Zero Delay
    - \( n_r = 0 \)
  - Input buffer overwrite
    - \( n_i < n_r \)
  - Time critical operation
    - \( n_i/2 < n_r \leq n_i \)
  - Normal operation
    - \( n_r < n_i/2 \)
Equivalent Classes of CFSM Behavior (2)

- **Zero delay:** $n_r = 0$
  - If C emits an error on some input
    - A, B can react instantaneously & output differently
  - May be logically inconsistent

- **Input buffers overwrite:** $n_i < n_r$
  - Execution delay of A, B is larger than arrival interval
    - always loss of event and requirements not satisfied

Equivalent Classes of CFSM Behavior (3)

- **Time critical operation:** $n_i/2 < n_r \leq n_i$
  - Normal operation results in no loss of event
  - Error operation may cause lost input

- **Normal operation:** $n_r < n_i/2$
  - No events are lost
  - May be expensive to implement

- **If error is infrequent**
  - Designer may accept also time critical operation
    - Can result in lower-cost implementation
Equivalent Classes of CFSM Behavior (4)

- Implementation on a single processor
  - Loss of Event may be caused by
    - Timing constraints
      - $n_t < 3n_r$
    - Incorrect scheduling
      - If empty transition also takes $n_r$
        - ACBC round robin will miss event
        - ABC round robin will not

Some Possibility of Equivalent Classes

- Given 2 arbitrary implementations, 1 input stream:
  - Dataflow equivalence
    - Output streams are the same ordering
  - Petri net equivalence
    - Output streams satisfied some partial order
  - Golden model equivalence
    - Output streams are the same ordering
      - Except reordering of concurrent events
    - One of the implementations is a reference specification
  - Filtered equivalence
    - Output streams are the same after filtered by observer
Conclusion

- **CFSM**
  - Initially unbounded FIFO buffers
    - Bounds on buffers are imposed by refinement
  - Delay is also refined by implementation
  - Local synchrony
    - Relatively large atomic synchronous entities
  - Global asynchrony
    - Break synchrony, no compositional problem
    - Allow efficient mapping to heterogeneous architectures

---

Conclusion (2)

- **Lossy buffers**
  - Desirable for efficient implementation
    - Especially for exceptions and resets
  - Sometime problematic
    - Need to verify that lossy behavior does not occur
    - Enforced by implementation

- **CFSM networks**
  - Capture wide range of heterogeneous implementation
  - Keep computation, communication, and timing separate
  - After architecture mapping, becomes a DE model
    - Lend itself to analysis with VHDL or Verilog tools
  - Equivalent classes of behavior