Models Of Computation for reactive systems

• Main MOCs:
  – Communicating Finite State Machines
  – Dataflow Process Networks
  – Petri Nets
  – Discrete Event
  – (Abstract) Codesign Finite State Machines

• Main languages:
  – StateCharts
  – Esterel
  – Dataflow networks
Finite State Machines

• Functional decomposition into states of operation
• Typical domains of application:
  – control functions
  – protocols (telecom, computers, ...)
• Different communication mechanisms:
  – synchronous
    – (classical FSMs, Moore ‘64, Kurshan ‘90)
  – asynchronous
    – (CCS, Milner ‘80; CSP, Hoare ‘85)
• Informal specification:
  – If the driver
    – turns on the key, and
    – does not fasten the seat belt within 5 seconds
  – then an alarm beeps
    – for 5 seconds, or
    – until the driver fastens the seat belt, or
    – until the driver turns off the key
FSM Example

- **KEY_ON** => **START_TIMER**
- **END_TIMER_5** => **ALARM_ON**
- **KEY_OFF** or **BELT_ON** => **END_TIMER_10** or **BELT_ON** or **KEY_OFF** => **ALARM_OFF**

If no condition is satisfied, implicit self-loop in the current state
FSM Definition

- FSM = (I, O, S, r, δ, λ)
- I = {KEY_ON, KEY_OFF, BELT_ON, END_TIMER_5, END_TIMER_10}
- O = {START_TIMER, ALARM_ON, ALARM_OFF}
- S = {OFF, WAIT, ALARM}
- r = OFF

δ : 2^I x S → S
  e.g. δ({KEY_OFF}, WAIT) = OFF

λ : 2^I x S → 2^O
  e.g. λ({KEY_ON}, OFF) = {START_TIMER}

Set of all subsets of I (implicit "and")
All other inputs are implicitly absent
Non-deterministic FSMs

- $\delta$ and $\lambda$ may be *relations* instead of *functions*:
  \[ \delta \subseteq 2^I \times S \times S \]
  \[
  \text{e.g. } \delta(\{\text{KEY_OFF, END_TIMER_5}\}, \text{WAIT}) = \{\text{OFF}, \{\text{ALARM}\}\}
  \]
  \[ \lambda \subseteq 2^I \times S \times 2^O \]

- Non-determinism can be used to describe:
  - an unspecified behavior
    (incomplete specification)
  - an unknown behavior
    (environment modeling)
**NDFSM: incomplete specification**

- E.g. error checking first partially specified:

  ![Transition Diagram](image)

- Then completed as *even parity*:

  ![Transition Diagram](image)
NDFSM: unknown behavior

• Modeling the environment
• Useful to:
  – optimize (don’t care conditions)
  – verify (exclude impossible cases)
• E.g. driver model:

  s0

  => KEY_ON or
  KEY_OFF or
  BELT_ON

• Can be refined
  – E.g. introduce timing constraints
  – (minimum reaction time 0.1 s)
NDFSM: time range

• Special case of unspecified/unknown behavior, but so common to deserve special treatment for efficiency

• E.g. delay between 6 and 10 s
NDFSMs and FSMs

• Formally FSMs and NDFSMs are equivalent
  – (Rabin-Scott construction, Rabin ‘59)
• In practice, NDFSMs are often more compact
  – (exponential blowup for determinization)
Finite State Machines

• Advantages:
  – Easy to use (graphical languages)
  – Powerful algorithms for
    – synthesis (SW and HW)
    – verification

• Disadvantages:
  – Sometimes over-specify implementation
    – (sequencing is fully specified)
  – Number of states can be unmanageable
  – Numerical computations cannot be specified compactly (need Extended FSMs)
Modeling Concurrency

• Need to compose parts described by FSMs
• Describe the system using a number of FSMs and interconnect them
• How do the interconnected FSMs talk to each other?
FSM Composition

• Bridle complexity via hierarchy: FSM product yields an FSM
• Fundamental hypothesis:
  – all the FSMs change state together (synchronicity)
• System state = Cartesian product of component states
  – (state explosion may be a problem...)
• E.g. seat belt control + timer
FSM Composition

KEY_ON and START_TIMER =>
START_TIMER

must be coherent

not SEC and
(KEY_OFF or BELT_ON) =>

not SEC and
(KEY_OFF or BELT_ON) =>

SEC and
(KEY_OFF or BELT_ON) =>

SEC and
(KEY_OFF or BELT_ON) =>

OFF, 0
WAIT, 1
OFF, 1
WAIT, 2
OFF, 2

Timer

Belt Control
Given

\[ M_1 = ( I_1, O_1, S_1, r_1, \delta_1, \lambda_1 ) \] and
\[ M_2 = ( I_2, O_2, S_2, r_2, \delta_2, \lambda_2 ) \]

Find the composition

\[ M = ( I, O, S, r, \delta, \lambda ) \]

given a set of constraints of the form:

\[ C = \{ ( o, i_1, \ldots, i_n ) : o \text{ is connected to } i_1, \ldots, i_n \} \]
FSM Composition

• Unconditional product $M' = ( I', O', S', r', \delta', \lambda' )$
  - $I' = I_1 \cup I_2$
  - $O' = O_1 \cup O_2$
  - $S' = S_1 \times S_2$
  - $r' = r_1 \times r_2$
  $\delta' = \{ ( A_1, A_2, s_1, s_2, t_1, t_2 ) : ( A_1, s_1, t_1 ) \in \delta_1 \text{ and } ( A_2, s_2, t_2 ) \in \delta_2 \}$
  $\lambda' = \{ ( A_1, A_2, s_1, s_2, B_1, B_2 ) : ( A_1, s_1, B_1 ) \in \lambda_1 \text{ and } ( A_2, s_2, B_2 ) \in \lambda_2 \}$

• Note:
  - $A_1 \subseteq I_1, A_2 \subseteq I_2, B_1 \subseteq O_1, B_2 \subseteq O_2$
  - $2^{X \cup Y} = 2^X \times 2^Y$
FSM Composition

• Constraint application

\[ \lambda = \{ ( A_1, A_2, s_1, s_2, B_1, B_2 ) \in \lambda' : \text{for all } (o, i_1, \ldots, i_n) \in C \quad o \in B_1 \cup B_2 \quad \text{if and only if} \quad i_j \in A_1 \cup A_2 \text{ for all } j \} \]

• The application of the constraint rules out the cases where the connected input and output have different values (present/absent).
**FSM Composition**

\[ I = I_1 \cup I_2 \]
\[ O = O_1 \cup O_2 \]
\[ S = S_1 \times S_2 \]

Assume that
\[ o_1 \in O_1, \quad i_3 \in I_2, \quad o_1 = i_3 \text{ (communication)} \]

\( \delta \) and \( \lambda \) are such that, e.g., for each pair:

\[ \delta_1(\{i_1\}, s_1) = t_1, \quad \lambda_1(\{i_1\}, s_1) = \{o_1\} \]
\[ \delta_2(\{i_2, i_3\}, s_2) = t_2, \quad \lambda_2(\{i_2, i_3\}, s_2) = \{o_2\} \]

we have:

\[ \delta(\{i_1, i_2, i_3\}, (s_1, s_2)) = (t_1, t_2) \]
\[ \lambda(\{i_1, i_2, i_3\}, (s_1, s_2)) = \{o_1, o_2\} \]

i.e. \( i_3 \) is in input pattern iff \( o_2 \) is in output pattern
• Problem: what if there is a cycle?
  – Moore machine: $\delta$ depends on input and state, $\lambda$ only on state
    composition is always well-defined
  – Mealy machine: $\delta$ and $\lambda$ depend on input and state
    composition may be undefined
    what if $\lambda_1(\{i_1\}, s_1) = \{o_1\}$ but $o_2 \notin \lambda_2(\{i_3\}, s_2)$?
• Causality analysis in Mealy FSMs (Berry ‘98)
Moore vs. Mealy

• Theoretically, same computational power (almost)
• In practice, different characteristics
• Moore machines:
  – non-reactive
    (response delayed by 1 cycle)
  – easy to compose
    (always well-defined)
  – good for implementation
    – software is always “slow”
    – hardware is better when I/O is latched
Moore vs. Mealy

- Mealy machines:
  - reactive
    (0 response time)
  - hard to compose
    (problem with combinational cycles)
  - problematic for implementation
    - software must be “fast enough”
      (synchronous hypothesis)
    - may be needed in hardware, for speed
Hierarchical FSM models

• Problem: how to reduce the size of the representation?

• Harel’s classical papers on StateCharts (language) and bounded concurrency (model): 3 orthogonal exponential reductions

• Hierarchy:
  – state a “encloses” an FSM
  – being in a means FSM in a is active
  – states of a are called OR states
  – used to model pre-emption and exceptions

• Concurrency:
  – two or more FSMs are simultaneously active
  – states are called AND states

• Non-determinism:
  – used to abstract behavior
The vending machine

• A machine that sells coffee
  – Accepts one dollar (d1) bills
  – Maximum two dollars
  – Quarters change
  – Sells two products
    – Small coffee for $1
    – Large coffee for $1.25
Denotational descriptions are “implicit” in the sense that they describe the properties that the system must have. They often are given as a system of equalities and inequalities that must be satisfied by the system.

- The controller is denoted by a set of traces of symbols from an alphabet.
- Non all-capital letters names belong to the alphabet of a process.
- Capital letters names denote processes (CTRL is the controller process).
- A process is a letter followed by a process: $P = x \rightarrow Q$.
- SKIP is a process that successfully completes execution (it does nothing, it just completes the execution).
- If $P$ and $Q$ are processes then $Z = P ; Q$ is a process that behaves like $P$ until it completes and then like $Q$.
- If $P$ and $Q$ are processes then $P | Q$ denotes a choice between $P$ and $Q$. 
Vending machine description

- Alphabet

\{d_1, sc, lc, q\}

- \$1 dollar bill
- Small coffee
- Large coffee
- Quarter
Vending machine description

• Vending machine process

\[ VM = (\text{SMALL} | \text{LARGE}) ; VM \]

Behaves as (small “choice” large) until successful completion and then like VM

• It is a recursive definition of the form

\[ X = F(X) \]

• For a large coffee:

\[ \text{LARGE} = d1 \rightarrow (d1 \rightarrow (l c \rightarrow \text{CHANGE3})) \]

\[ \text{CHANGE3} = q \rightarrow (q \rightarrow (q \rightarrow \text{STOP})) \]
Vending machine FSM

- The encoding of the behaviors with a labeled directed graph
- No inputs/outputs yet (as in the denotational description)
Vending machine I/O description

\[ I = \{d1, sc, lc\} \]
\[ O = \{serves, servel, q\} \]
\[ S = \{idle, $1, $2, c1, c2, c3, c4\} \]

(deterministic description)

\[ \delta : I \times S \rightarrow S \]
\[ \lambda : I \times S \rightarrow O \]

State transition function
Output function

Examples:
\[ \delta(d1, idle) = $1 \quad \rightarrow \quad \text{If waiting and one dollar is inserted, change state to$1 credit} \]
\[ \delta(sc, $1) = idle \quad \rightarrow \quad \text{If$1 credit and small coffee is requested, change state to idle and serve the coffee} \]
\[ \lambda(sc, $1) = serves \]
Vending machine I/O description

Labels: \textit{input/output}, where input and output can both be empty.
Communication with the rest of the system

Our state machine does not live in isolation

– What is the communication semantics?
– The serving system and the change return are electromechanical system with their own evolution dynamics
The Nokia 3120 User Interface

- Keypad
- Events
- Controller
- Control software
Controller description: Denotational

- The controller is denoted by a set of traces of symbols from an alphabet
- Non all-capital letters names belong to the alphabet of a process
- Capital letters names denote processes (CTRL is the controller process)
- A process is a letter followed by a process: \( P = x \rightarrow Q \)
- SKIP is a process that successfully completes execution (it does nothing, it just completes the execution)
- If \( P \) and \( Q \) are processes then \( Z = P ; Q \) is a process that behaves like \( P \) until it completes and then like \( Q \)
- \(*P\) is a finite number of repetition of process \( P\)
Controller description: Denotational

To lock or unlock a Nokia phone press “Menu” followed by the Star key

\[ LKUNLK = \text{Menu} \rightarrow \text{Star} \rightarrow \text{SKIP} \]

Process  Letter of the alphabet  Successful

Once unlocked, pick something from the menu and perform some action (for instance, choose “Contacts->Find->Alberto) and perform the action “Call”

\[ SELECTION = \text{Menu} \rightarrow (\text{CHOICE}; \text{ACTION}) \]

Sequential composition

\[ CHOICE = (1 \rightarrow \text{SKIP}) \parallel (2 \rightarrow \text{SKIP}) \parallel ... \]

A complete operation is an unlock followed by a selection followed by a lock

\[ OP = LKUNLK; SELECTION; LKUNLK \]

A controller is a finite (the phone breaks at some point) sequences of operations

\[ CTRL = \ast OP \]
Controller description: Denotational Implicit

A tuple is the mathematical object that denotes the controller

\[(I, O, S, \delta, \lambda, s_0)\]

\[I = (Menu, Star, 1, 2...)]
\[O = (Call, SMS, ...)]
\[S = (Lk, Lk\_Menu, UnLk, MainMenu, Contacts, ...)]

These two functions encode the possible traces

\[\delta : 2^I \times S \rightarrow S\]
\[\lambda : 2^I \times S \rightarrow O\]

Example: To describe the unlock sequence

\[\delta(\text{Menu}, Lk) = Lk\_Menu\]
\[\delta(\text{Star}, Lk\_Menu) = UnLk\]
An operational description is “explicit” in the sense that it defines:

- The meaning of enabled transitions, events etc.
- What happens when a transition is enabled
- How a state transition is accomplished
Composition with synchronization labels

The Lock/Unlock FSM

Start
Menu
Un_Menu

Done
UnLk

Star/Lock

key ≠ Star

Menu

Lk

Star/Unlock

Lk_Menu

key ≠ Star

The Phone is executing the requested service

Event notification
An example of service

The Select Contacts FSM

In service: the phone cannot be locked

Coming from The lock/unlock FSM
Transitions with the same synchronization labels must happen concurrently. There is no notion of time.
StateCharts: a Language to Capture FSMs

- An extension of conventional FSMs
- **Conventional FSMs** are inappropriate for the behavioral description of complex control
  - flat and unstructured
  - inherently sequential in nature
- **StateCharts** supports repeated decomposition of states into sub-states in an AND/OR fashion, combined with a **synchronous** (instantaneous broadcast) communication mechanism
State Decomposition

- **OR-States** have sub-states that are related to each other by exclusive-or

- **AND-States** have orthogonal state components (synchronous FSM composition)
  - AND-decomposition can be carried out on any level of states (more convenient than allowing only one level of communicating FSMs)

- **Basic States** have no sub-states (bottom of hierarchy)

- **Root State**: no parent states (top of hierarchy)
To be in state U the system must be either in state S or in state T.

StateChart OR-decomposition
To be in state U the system must be both in states S and T.
StateCharts Syntax

• The general syntax of an expression labeling a transition in a StateChart is \( e[c]/a \), where
  
  – \( e \) is the *event* that triggers the transition
  
  – \( c \) is the *condition* that guards the transition
    (cannot be taken unless \( c \) is true when \( e \) occurs)
  
  – \( a \) is the *action* that is carried out if and when the transition is taken

• For each transition label:
  
  – event condition and action are optional
  
  – an event can be the changing of a value
  
  – standard comparisons are allowed as conditions and assignment statements as actions
StateCharts Actions and Events

• An action $a$ on the edge leaving a state may also appear as an event triggering a transition going into an orthogonal state:
  – a state transition broadcasts an event visible immediately to all other FSMs, that can make transitions immediately and so on
  – executing the first transition will immediately cause the second transition to be taken *simultaneously*

• Actions and events may be associated to the execution of orthogonal components: $\text{start}(A)$, $\text{stopped}(B)$
Graphical Hierarchical FSM Languages

• Multitude of commercial and non-commercial variants:
  – StateCharts, UML, StateFlow, …

• Easy to use for control-dominated systems

• Simulation (animated), SW and HW synthesis

• Original StateCharts have problems with causality loops and instantaneous events:
  – circular dependencies can lead to paradoxes
  – behavior is implementation-dependent
  – not a truly synchronous language

• Hierarchical states necessary for complex reactive system specification
Synchronous vs. Asynchronous FSMs

• Synchronous (Esterel, StateCharts):
  – communication by shared variables that are read and written in zero time
  – communication and computation happens instantaneously at discrete time instants
  – all FSMs make a transition simultaneously (lock-step)
  – may be difficult to implement
    – multi-rate specifications
    – distributed/heterogeneous architectures
Synchronous vs. Asynchronous FSMs

• A-synchronous FSMs:
  – free to proceed independently
  – do not execute a transition at the same time (except for CSP rendezvous)
  – may need to share notion of time: synchronization
  – easy to implement
Synchronization
Handover

- A Mobile Station moving across the cell boundary needs to maintain active connections without interruptions or degradations

- Handover
  - tight inter-base-station synchronization (in GSM achieved using GPS)
  - asynchronous base station operation (UMTS)
Frame Synchronization

• Medium Access Control Layer: TDMA
  – each active connection is assigned a number of time slots (channel)

• A common notion of time is needed
  – each Base Station sends a frame synchronization pilot (FS) at the beginning of every frame to ensure that all Mobile Stations have the same slot counts
Bit Synchronization

- Transmitter interleaves the payload data with a pilot sequence known by the receiver.

- Receiver extracts the clock from the pilot sequence and uses it to sample the payload data.
Asynchronous communication

- Blocking vs. non-Blocking
  - Blocking read
    - process can not test for emptiness of input
    - must wait for input to arrive before proceed
  - Blocking write
    - process must wait for successful write before continue
  - blocking write/blocking read (CSP, CCS)
  - non-blocking write/blocking read (FIFO, CFSMs, SDL)
  - non-blocking write/non-blocking read (shared variables)
Asynchronous communication

- Buffers used to adapt when sender and receiver have different rate
  - what size?
- Lossless vs. lossy
  - events/tokens may be lost
  - bounded memory: overflow or overwriting
  - need to block the sender
- Single vs. multiple read
  - result of each write can be read at most once or several times
Communication Mechanisms

• Rendez-Vous (CSP)
  – No space is allocated for the data, processes need to synchronize in some specific points to exchange data
  – Read and write occur simultaneously

• FIFO
  – Bounded (ECFSMs, CFSMs)
  – Unbounded (SDL, ACFSMs, Kahn Process Networks, Petri Nets)

• Shared memory
  – Multiple non-destructive reads are possible
  – Writes delete previously stored data
# Communication models

<table>
<thead>
<tr>
<th></th>
<th>Transmitters</th>
<th>Receivers</th>
<th>Buffer Size</th>
<th>Blocking Reads</th>
<th>Blocking Writes</th>
<th>Single Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsynchronized</td>
<td>many</td>
<td>many</td>
<td>one</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Read-Modify-write</td>
<td>many</td>
<td>many</td>
<td>one</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Unbounded FIFO</td>
<td>one</td>
<td>one</td>
<td>unbounded</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Bounded FIFO</td>
<td>one</td>
<td>one</td>
<td>bounded</td>
<td>no</td>
<td>maybe</td>
<td>yes</td>
</tr>
<tr>
<td>Single Rendezvous</td>
<td>one</td>
<td>one</td>
<td>one</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Multiple Rendezvous</td>
<td>many</td>
<td>many</td>
<td>one</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Outline

- Part 3: Models of Computation
  - FSMs
  - Discrete Event Systems
  - CFSMs
  - Data Flow Models
  - Petri Nets
  - The Tagged Signal Model
Discrete Event

• Explicit notion of time (global order…)
• Events can happen at any time asynchronously
• As soon as an input appears at a block, it may be executed
• The execution may take non zero time, the output is marked with a time that is the sum of the arrival time plus the execution time
• Time determines the order with which events are processed
• DE simulator maintains a global event queue (Verilog and VHDL)

• Drawbacks
  – global event queue => tight coordination between parts
  – Simultaneous events => non-deterministic behavior
  – Some simulators use delta delay to prevent non-determinacy
Simultaneous Events in DE

Fire B or C?

B has 0 delay

B has delta delay

Fire C once? or twice?

Fire C twice.

Can be refined

E.g. introduce timing constraints
(minimum reaction time 0.1 s)

Still have problem with 0-delay (causality) loop
Outline

- Part 3: Models of Computation
  - FSMs
  - Discrete Event Systems
  - CFSMs
  - Data Flow Models
  - Petri Nets
  - The Tagged Signal Model
Co-Design Finite State Machines: Combining FSM and Discrete Event

- 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 and VCC
- 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 \(<\!<\!<\!\text{ clock period}\)
  – 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

![Diagram showing ticks and delta steps in continuous time](image)
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-designed-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
Pure 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*)
• 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 v1 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 events are read atomically
    – Can be enforced in SW (bit vector) HW (buffer)
    – CFSM is guaranteed to see a snapshot of input events
• Non-atomicity of event and data
  – May lead to undesirable behavior
  – Atomicized 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)
  - 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
    - 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 nr, err operation 2nr
- Minimum input interval is ni
- 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
    – nr = 0
  – Input buffer overwrite
    – ni < nr
  – Time critical operation
    – ni/2 < nr \leq ni
  – Normal operation
    – nr < ni/2
Equivalent Classes of CFSM Behavior (2)

• Zero delay: \( nr = 0 \)
  – If C emits an error on some input
    – A, B can react instantaneously & output differently
    – May be logically inconsistent

• Input buffers overwrite: \( ni < nr \)
  – Execution delay of A, B is larger than arrival interval
    – always loss of event
    – requirements not satisfied
Equivalent Classes of CFSM Behavior (3)

- Time critical operation: \( \frac{ni}{2} < nr \leq ni \)
  - Normal operation results in no loss of event
  - Error operation may cause lost input

- Normal operation: \( nr < \frac{ni}{2} \)
  - 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
      - $n_i < 3n_r$
    - Incorrect scheduling
      - If empty transition also takes $n_r$
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 satisfy some partial order
  – Golden model equivalence
    – Output streams have 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
  - Extension: ACFSM: Initially unbounded FIFO buffers
    - Bounds on buffers are imposed by refinement to yield ECFSM
  - 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