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

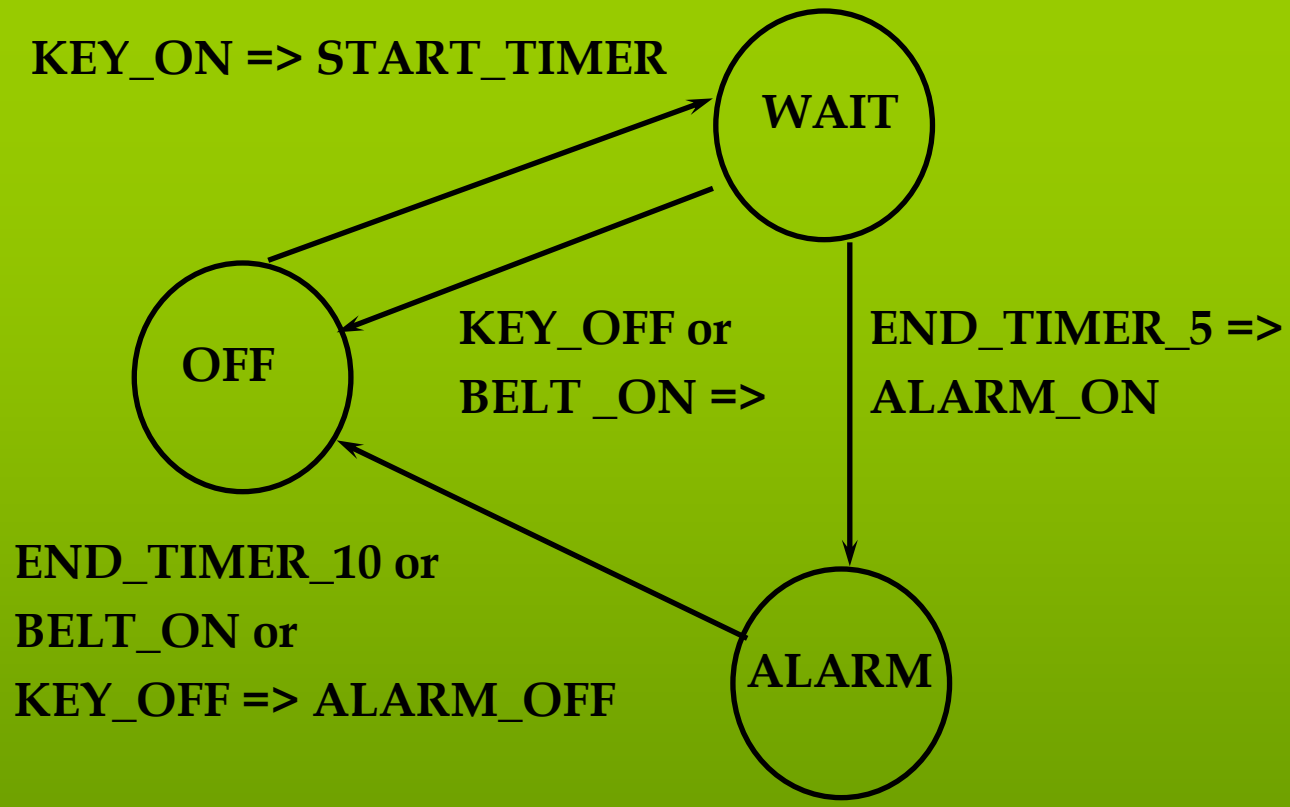
Finite State Machines

- 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)

FSM Example

- 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



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

Set of all subsets of I (implicit "and")

All other inputs are implicitly absent

$\delta : 2^I \times S \rightarrow S$

e.g. δ({ KEY_OFF }, WAIT) = OFF

$\lambda : 2^I \times S \rightarrow 2^O$

e.g. λ ({ KEY_ON }, OFF) = { START_TIMER }

Non-deterministic FSMs

- δ and λ may be *relations* instead of *functions*:

  $\delta \subseteq 2^I \times S \times S$

  implicit "and" *implicit "or"*

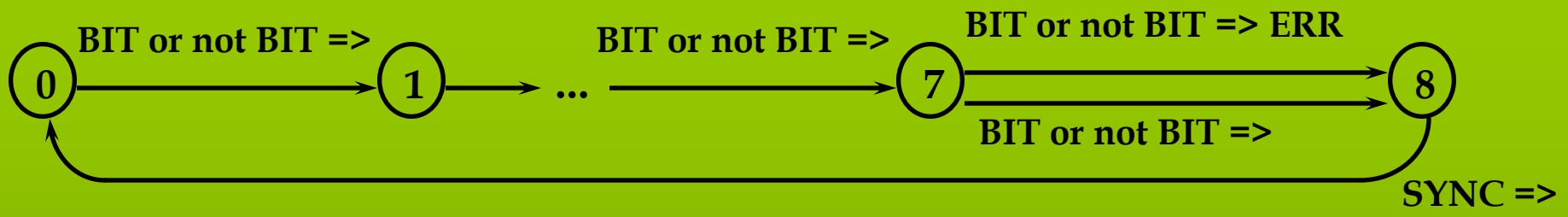
  e.g. δ({KEY_OFF, END_TIMER_5}, WAIT) = {{OFF}, {ALARM}}

  $\lambda \subseteq 2^I \times S \times 2^O$

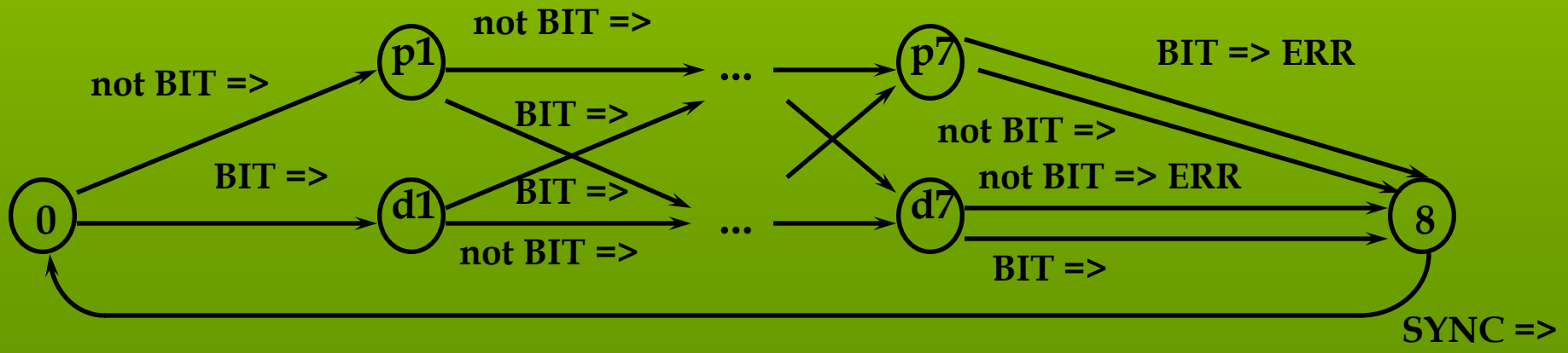
- Non-determinism can be used to describe:
  - an unspecified behavior
    (incomplete specification)
  - an unknown behavior
    (environment modeling)

NDFSM: incomplete specification

- E.g. error checking first partially specified:

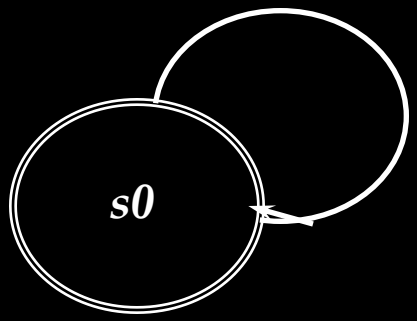


- Then completed as *even parity*:



NDFSM: unknown behavior

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

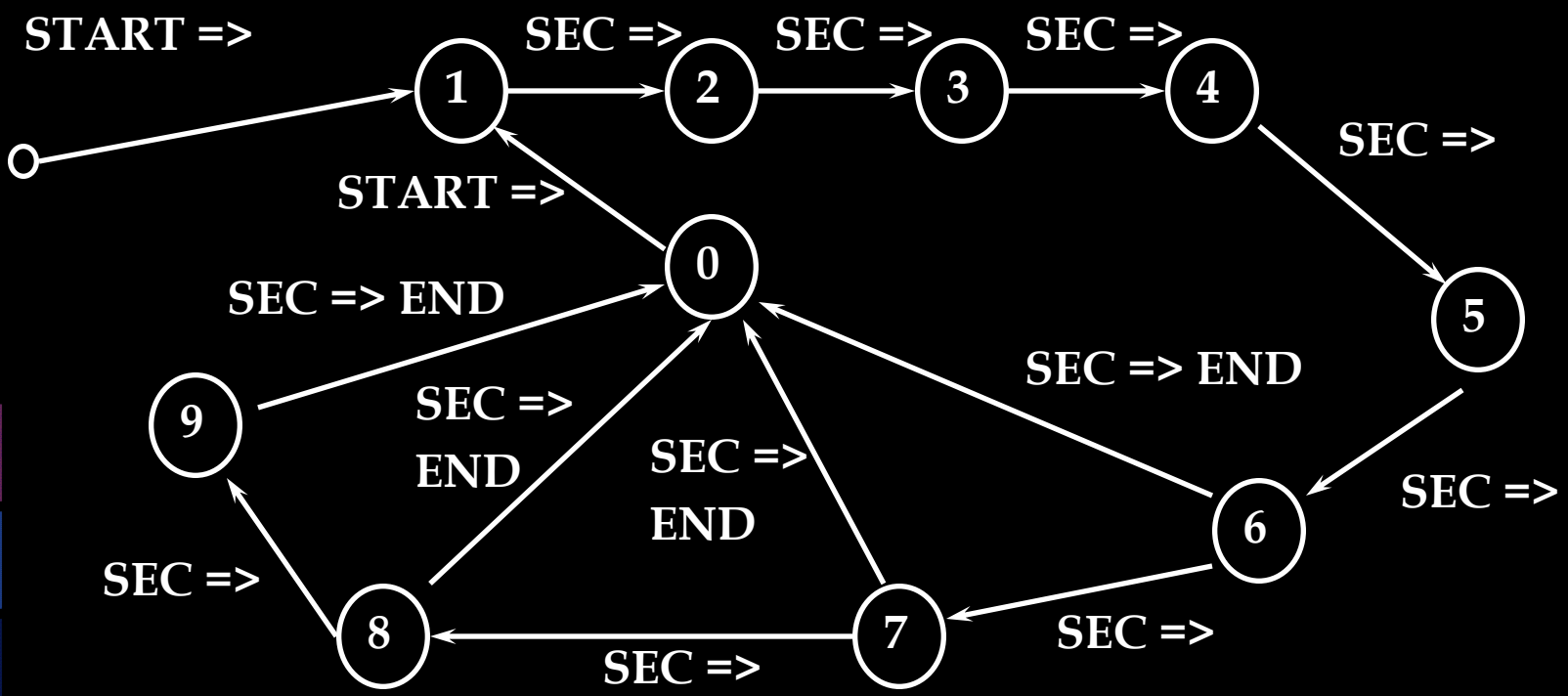


=> KEY_ON or KEY_OFF or BELT_ON

- Can be refined
  - E.g. introduce timing constraints
  - (minimum reaction time 0.1 s)

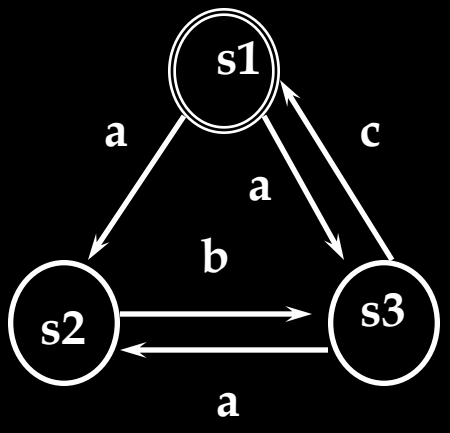
NDFSM: time range

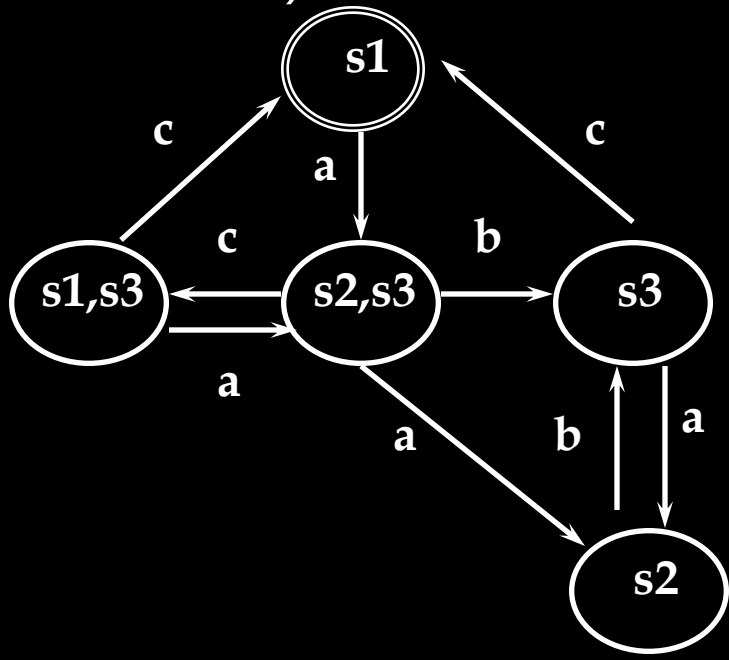
- Special case of unspecified/unknown behavior, but so common to deserve special treatment for efficiency
- E.g. delay between 6 and 10 s



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

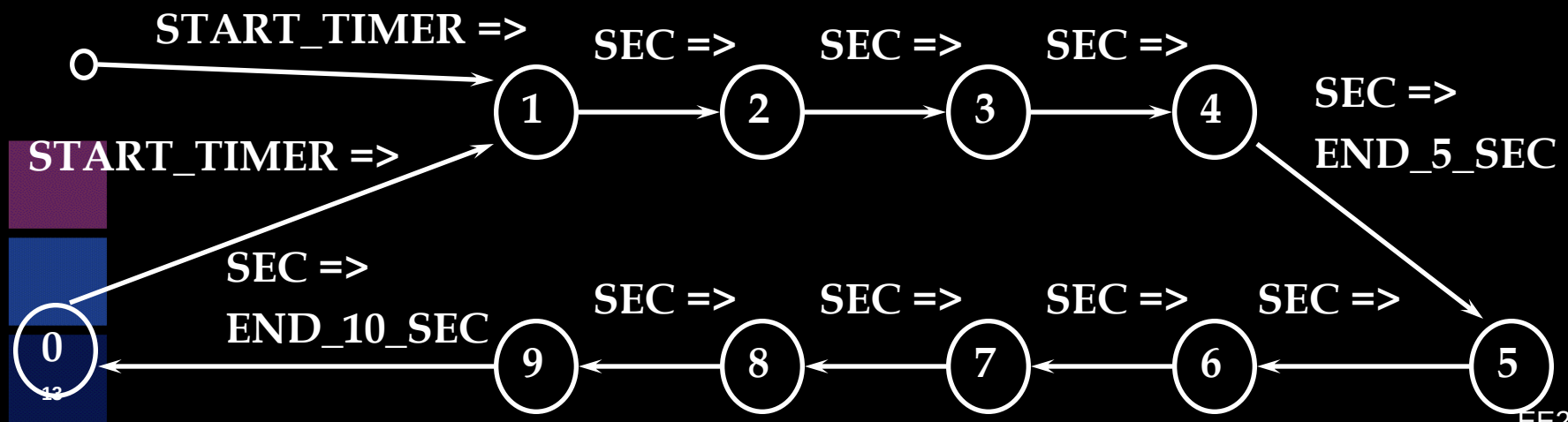
- Advantages:
  - Easy to use (graphical languages)
  - Powerful algorithms for
    - synthesis (SW and HW)
    - verification
- Disadvantages:
  - Sometimes over-specify implementation
    - (sequencing is fully specified)
  - Number of states can be unmanageable
  - Numerical computations cannot be specified compactly (need Extended FSMs)

Modeling Concurrency

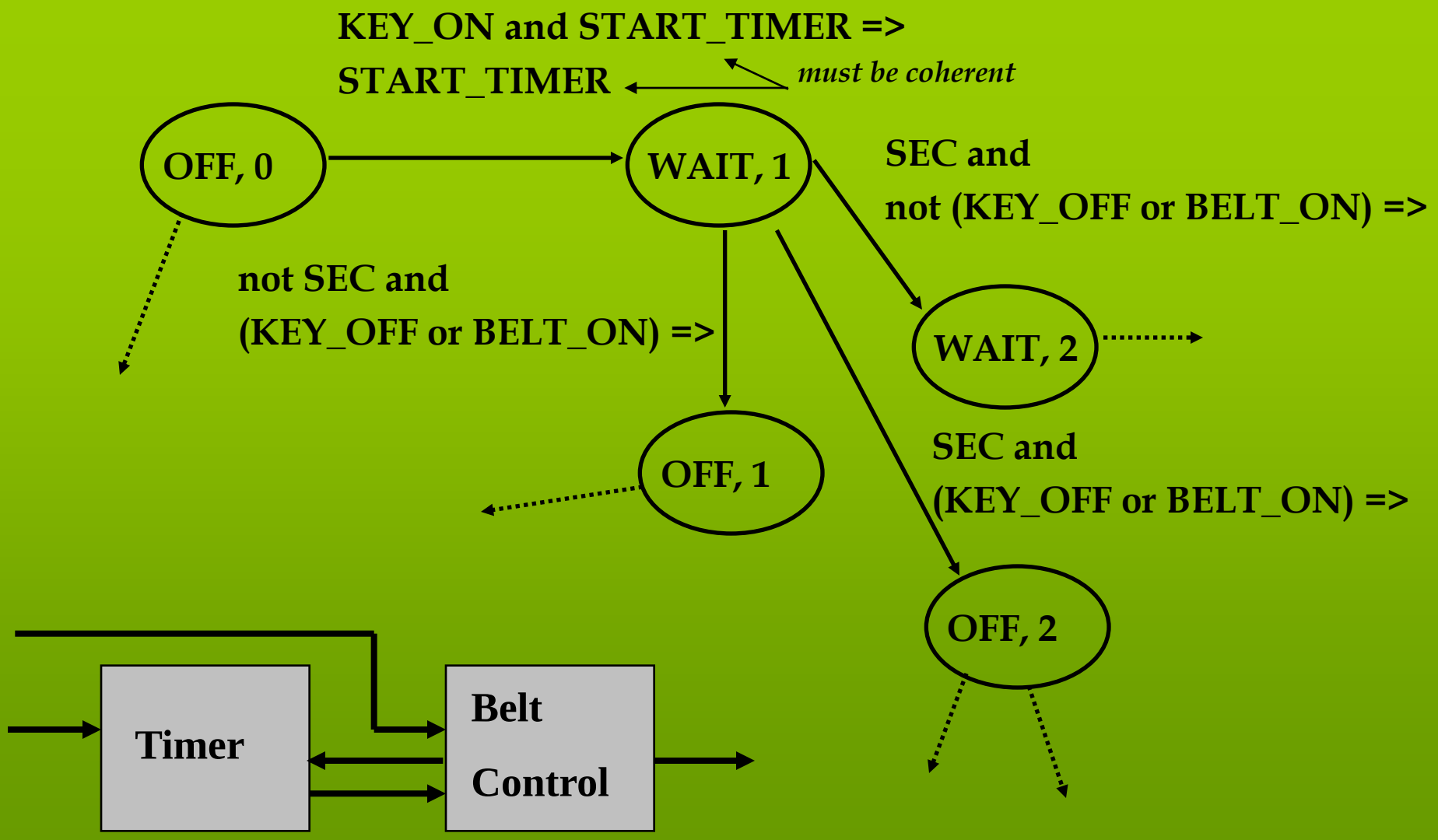
- Need to compose parts described by FSMs
- Describe the system using a number of FSMs and interconnect them
- How do the interconnected FSMs talk to each other?

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



FSM Composition

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 $M' = (I', O', S', r', \delta', \lambda')$
  - $I' = I_1 \cup I_2$
  - $O' = O_1 \cup O_2$
  - $S' = S_1 \times S_2$
  - $r' = r_1 \times r_2$

$\delta' = \{ (A_1, A_2, s_1, s_2, t_1, t_2) : \quad (A_1, s_1, t_1) \in \delta_1 \quad$ and
$(A_2, s_2, t_2) \in \delta_2 \}$

$\lambda' = \{ (A_1, A_2, s_1, s_2, B_1, B_2) : \; (A_1, s_1, B_1) \in \lambda_1 \quad$ 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

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

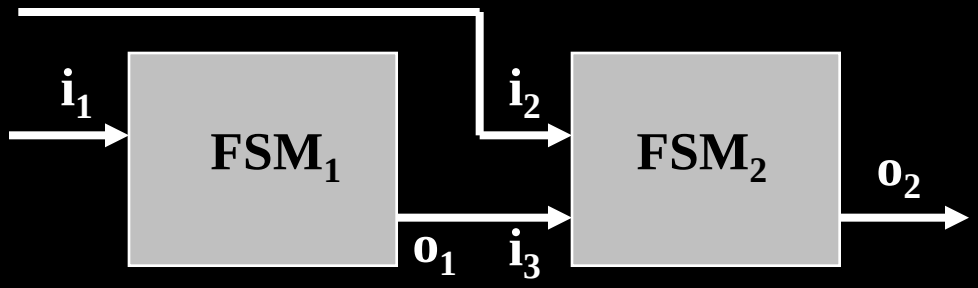
- The application of the constraint rules out the cases where the connected input and output have different values (present/absent).

FSM Composition

$I = I_1 \cup I_2$

$O = O_1 \cup O_2$

$S = S_1 \times S_2$



Assume that

$o_1 \in O_1$, $i_3 \in I_2$, $o_1 = i_3$ (communication)

δ and λ 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

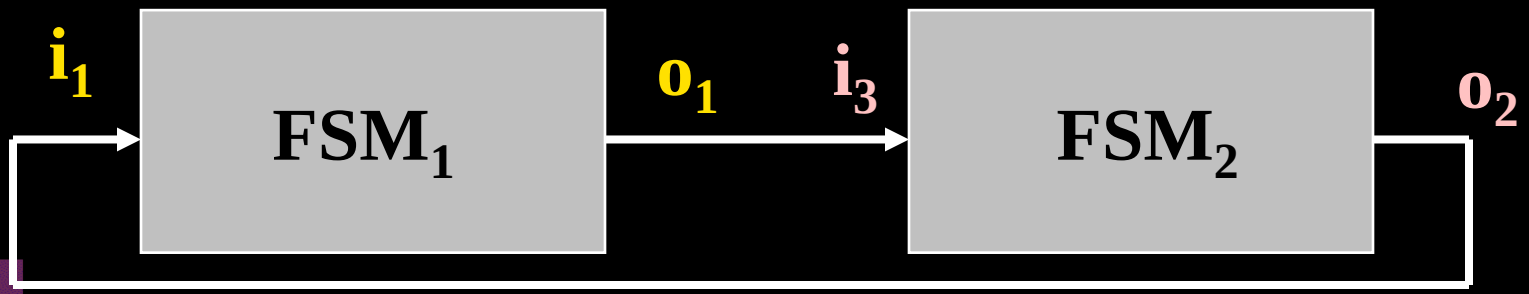
FSM Composition

- Problem: what if there is a cycle?
  - Moore machine: δ depends on input and state, λ only on state

    composition is always *well-defined*
  - Mealy machine: δ and λ 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

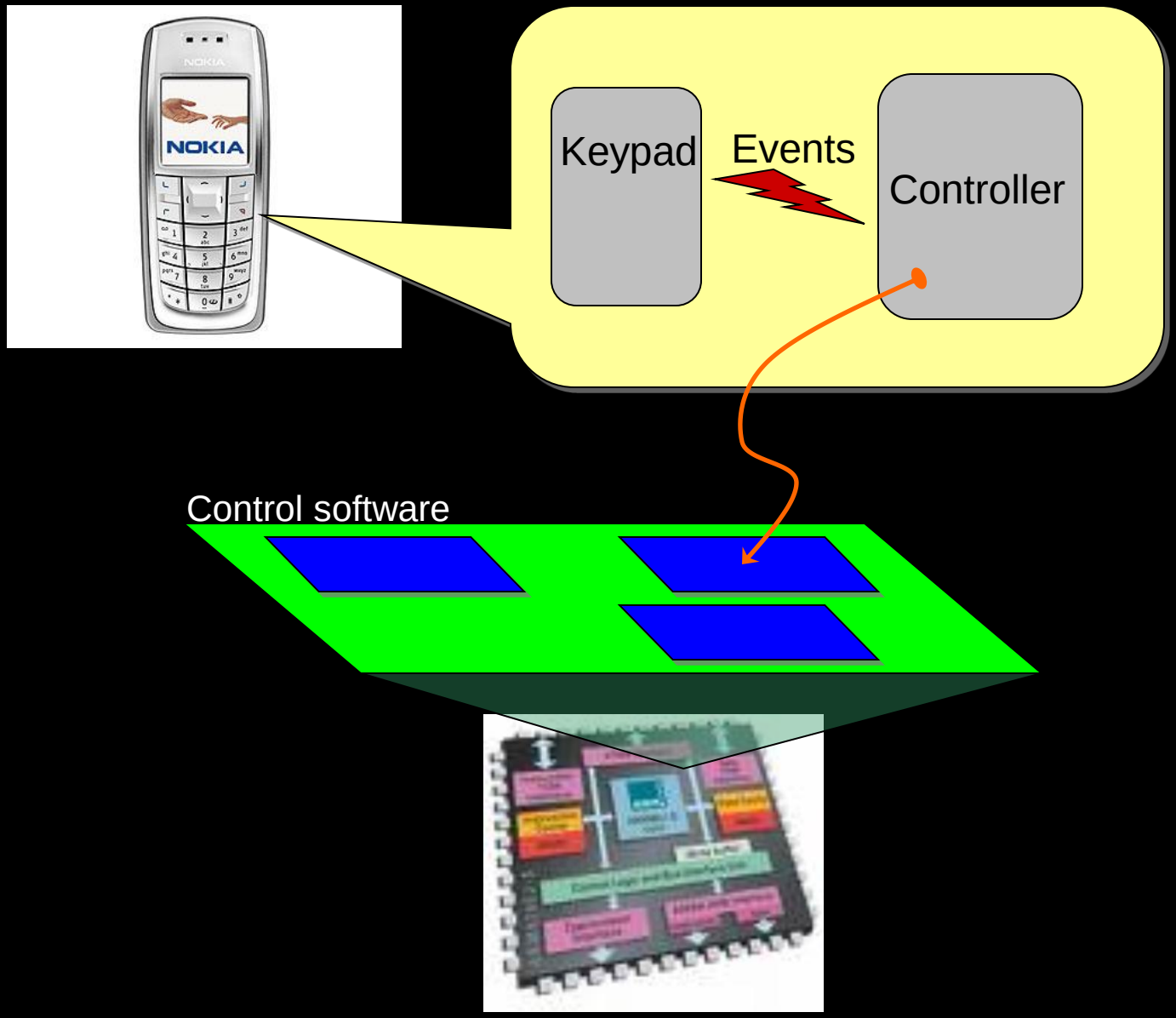
Moore vs. Mealy

- 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 Nokia 3120 User Interface



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

$$LKUNLK = Menu \rightarrow Star \rightarrow SKIP$$

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”

$$SELECTION = Menu \rightarrow (CHOICE; ACTION)$$

Sequential composition

$$CHOICE = (1 \rightarrow SKIP)|(2 \rightarrow SKIP)|...$$

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

$$CRTL = {}^{*}OP$$

Controller description: Denotational Implicit

A tuple is the mathematical object that denotes the controller

$$(I, O, S, \delta, \lambda, s_0)$$

$$\begin{aligned} I &= (Menu, Star, 1, 2...) \\ O &= (Call, SMS, ...) \\ S &= (Lk, Lk\_Menu, UnLk, MainMenu, Contacts, ...) \end{aligned}$$

These two functions encode the possible traces

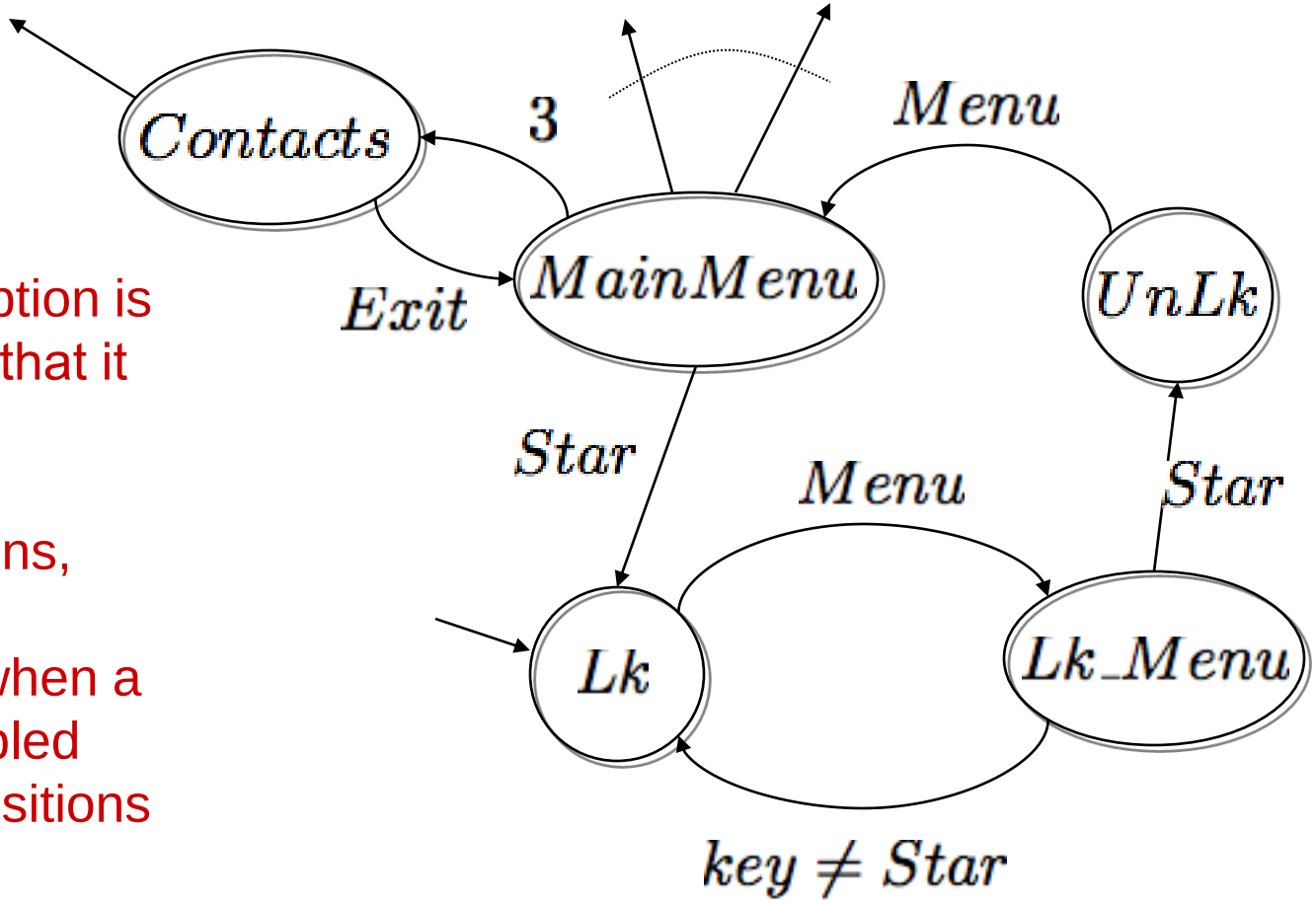
$$\begin{aligned} \delta &: 2^I \times S \rightarrow S \\ \lambda &: 2^I \times S \rightarrow O \end{aligned}$$

Example: To describe the unlock sequence

$$\begin{aligned} \delta(Menu, Lk) &= Lk\_Menu \\ \delta(Star, Lk\_Menu) &= UnLk \end{aligned}$$

Controller Description: Operational

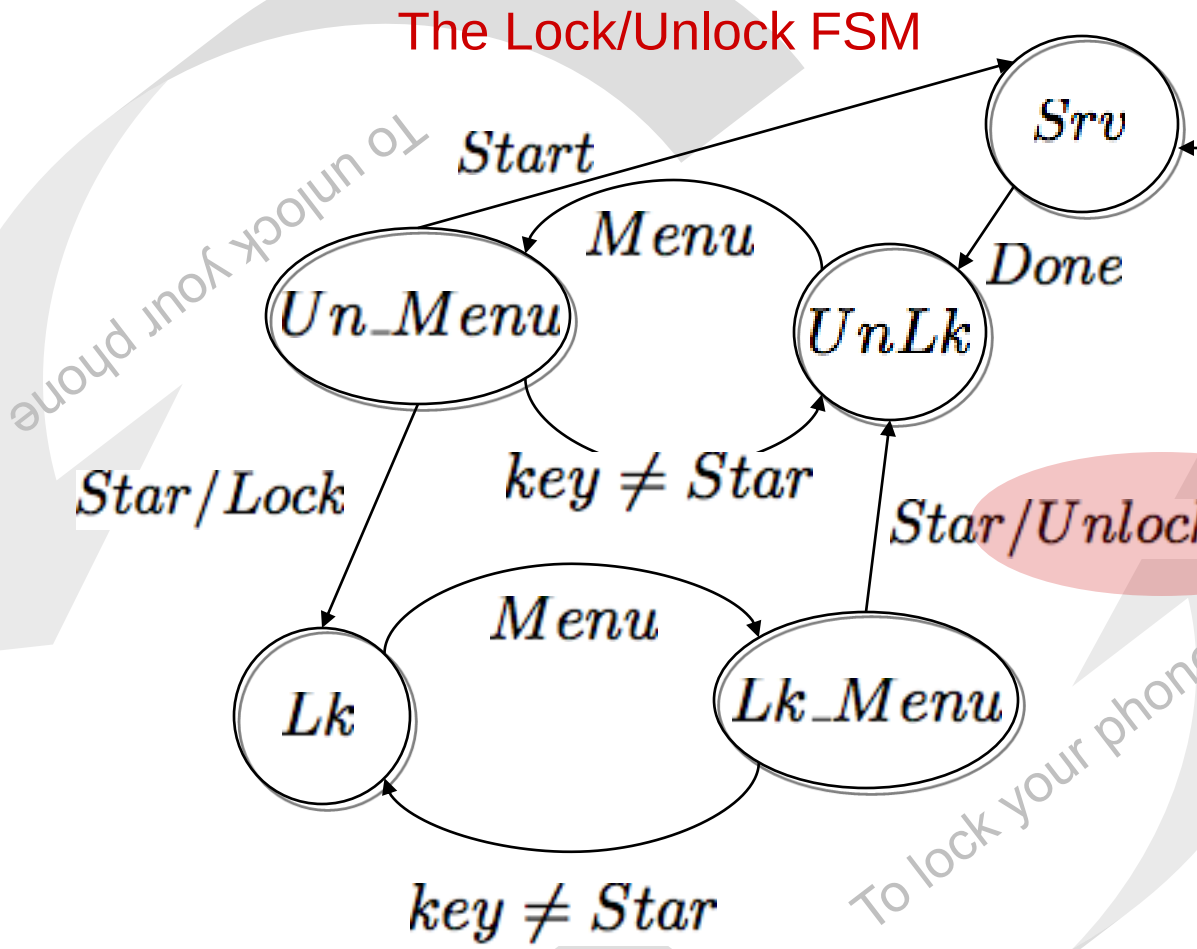
State transition graph



An operational description is "explicit" in the sense that it defines:

- The meaning of enabled transitions, events etc.
- What happens when a transitions is enabled
- How a state transitions is accomplished

Composition with synchronization labels



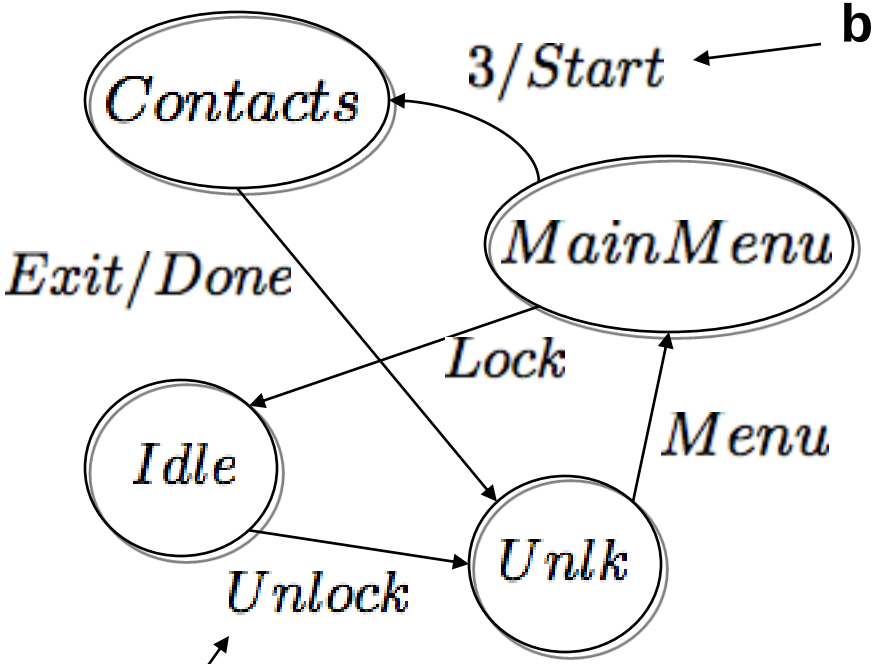
The Phone is executing the requested service

Event notification

An example of service

The Select Contacts FSM

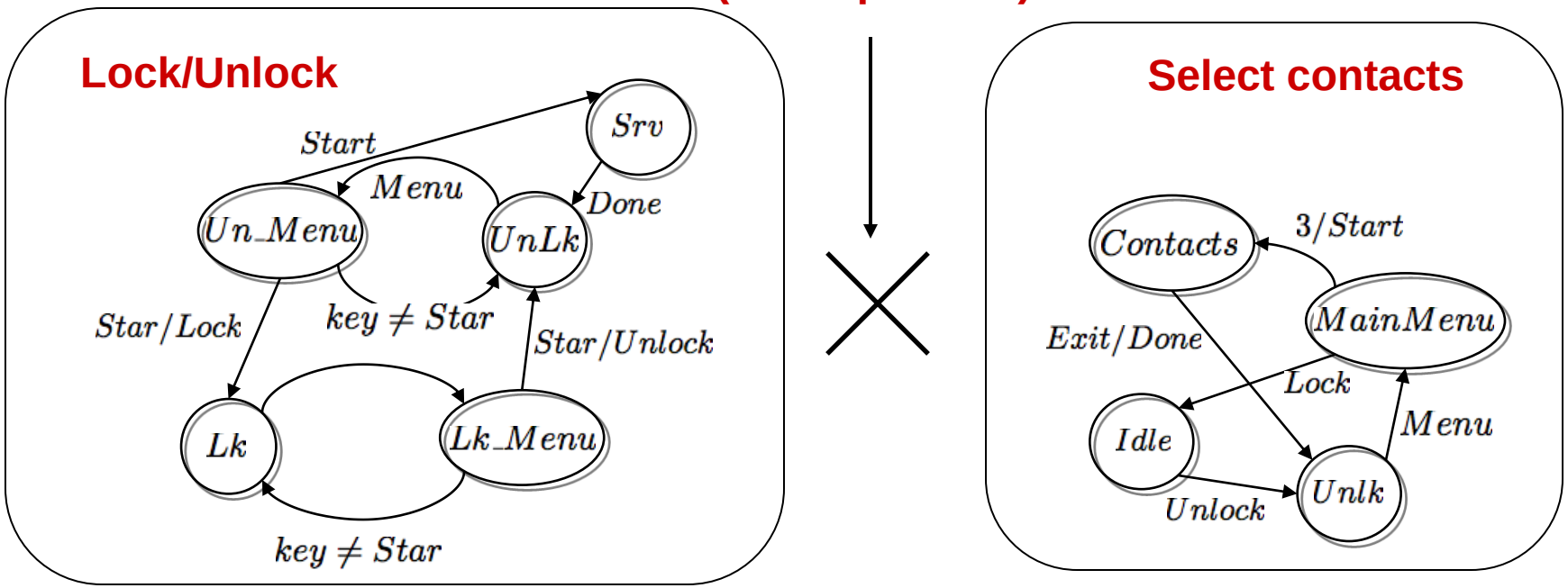
In service: the phone cannot be locked



Coming from The lock/unlock FSM

Communication by synchronization

Operation of composition (cross product)



Transitions with same synchronization labels must happen "simultaneously"

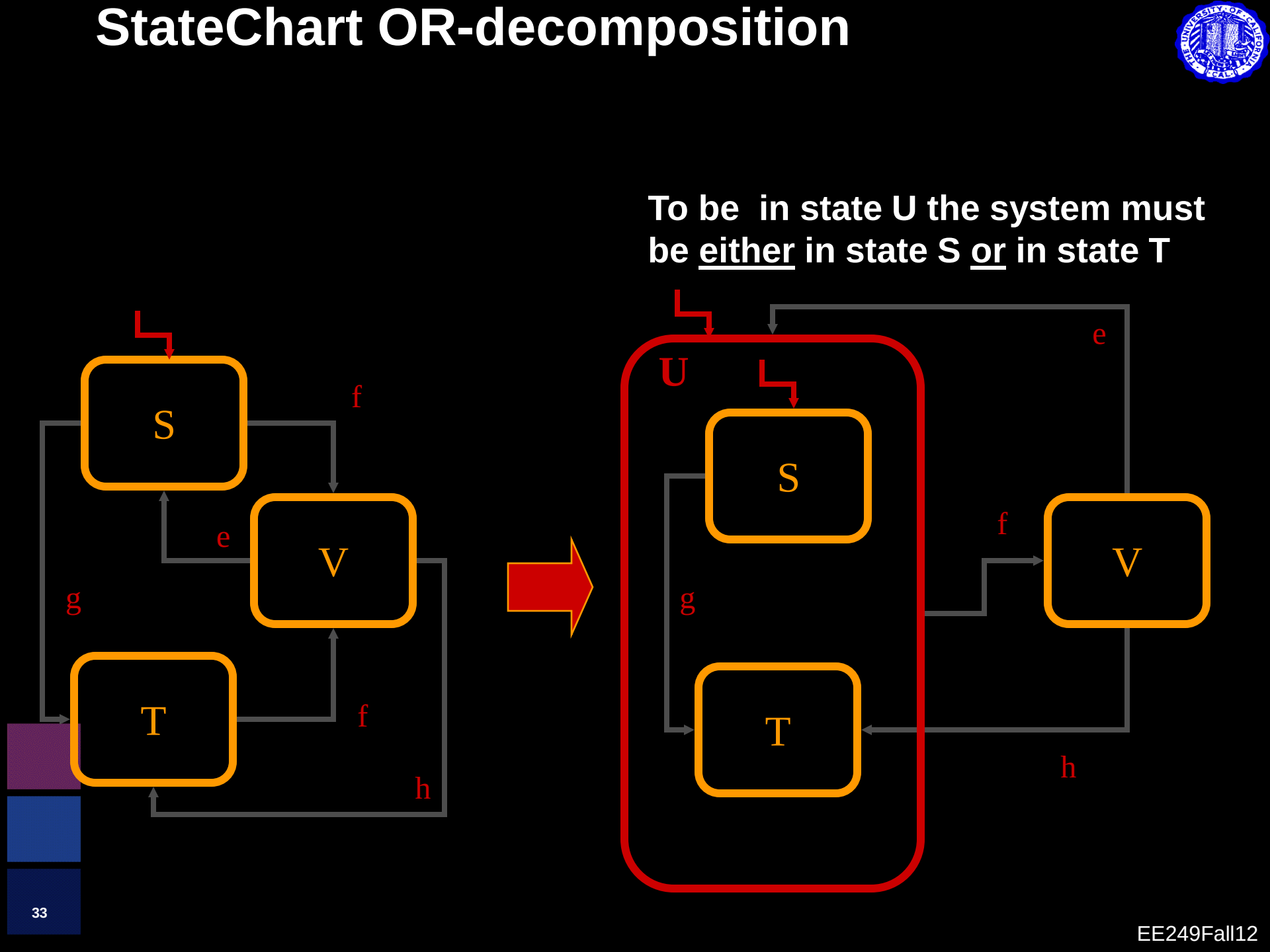
StateCharts: a Language to Capture FSMs

- 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

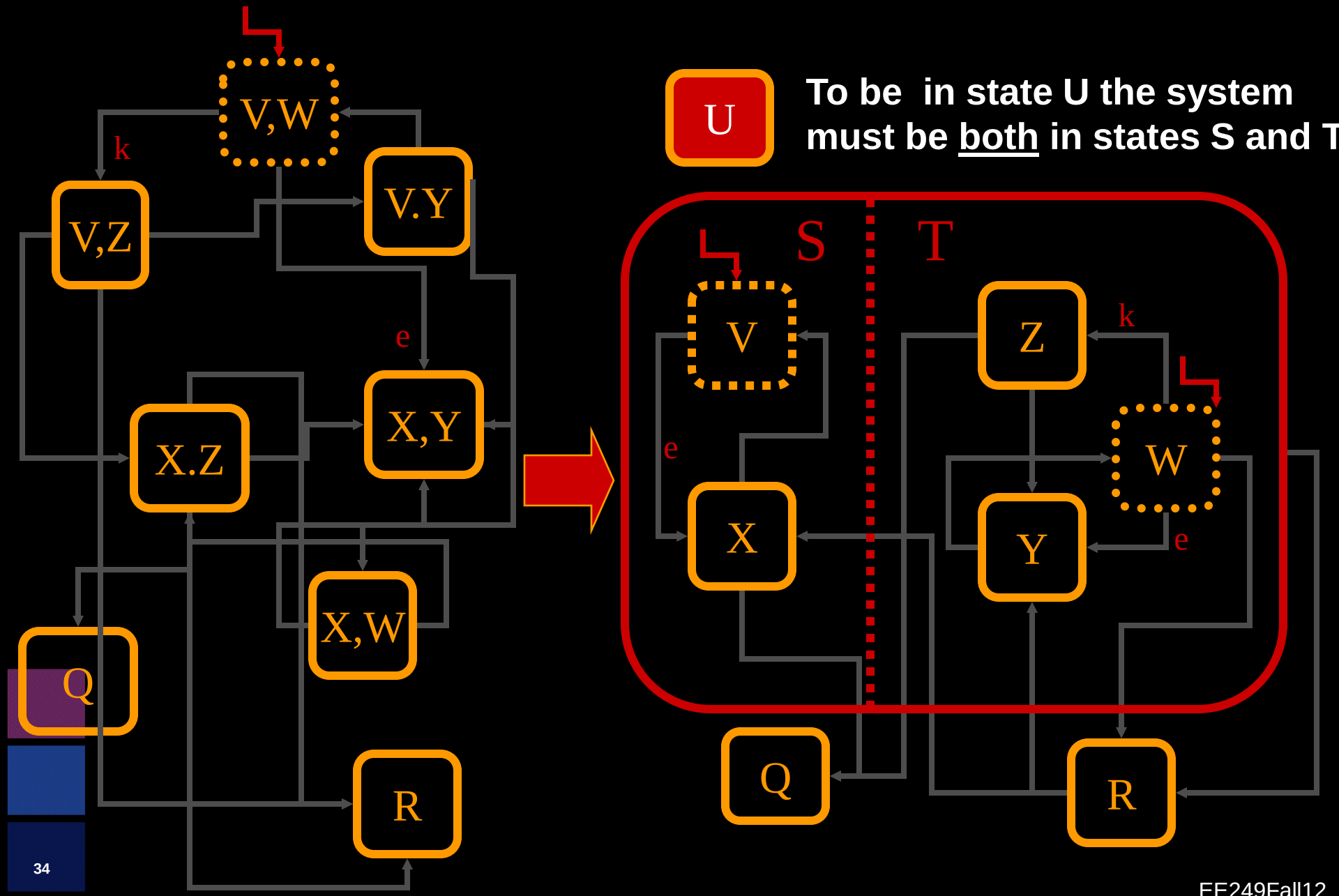
State Decomposition

- 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)

StateChart OR-decomposition



StateChart AND-decomposition



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 : *start(A)* , *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

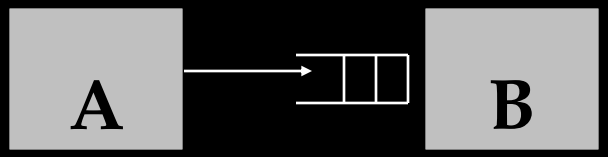
- A-synchronous FSMs:
  - free to proceed independently
  - do not execute a transition at the same time (except for CSP rendezvous)
  - may need to share notion of time: synchronization
  - easy to implement

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

	Transmitters	Receivers	Buffer Size	Blocking Reads	Blocking Writes	Single Reads
Unsynchronized	many	many	one	no	no	no
Read-Modify-write	many	many	one	yes	yes	no
Unbounded FIFO	one	one	unbounded	yes	no	yes
Bounded FIFO	one	one	bounded	no	maybe	yes
Single Rendezvous	one	one	one	yes	yes	yes
Multiple Rendezvous	many	many	one	no	no	yes

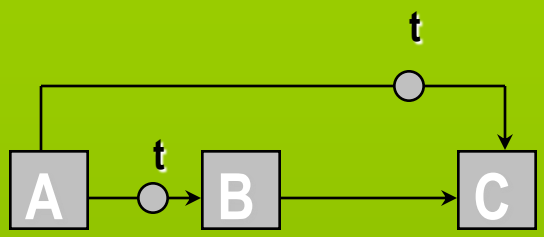
Outline

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

Discrete Event

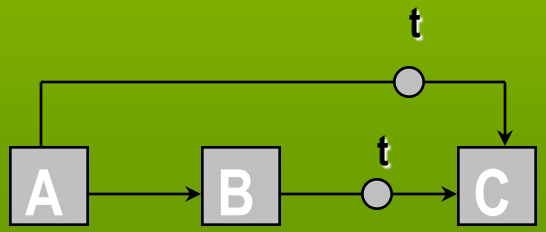
- Explicit notion of time (global order…)
- Events can happen at any time asynchronously
- As soon as an input appears at a block, it may be executed
- The execution may take non zero time, the output is marked with a time that is the sum of the arrival time plus the execution time
- Time determines 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



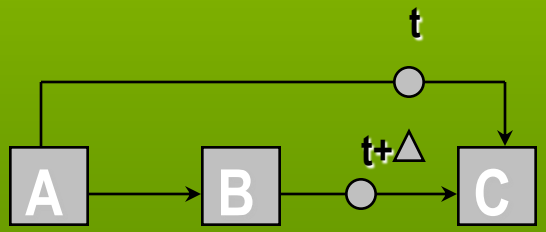
Fire B or C?

B has 0 delay



Fire C once? or twice?

B has delta delay



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

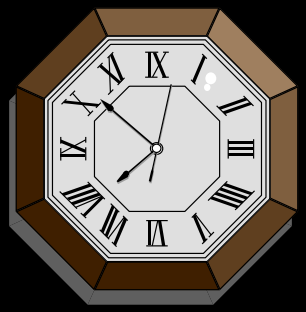
- Synchrony and asynchrony
- CFSM definitions
  - Signals & networks
  - Timing behavior
  - Functional behavior
- CFSM & process networks
- Example of CFSM behaviors
  - Equivalent classes

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 <<< clock period
  - Computation time = 0, w.r.t. reaction time of environment
- Features of synchrony
  - Functional behavior independent of timing
    - Simplify verification
  - Cyclic dependencies may cause problem
    - Among (simultaneous) synchronous events

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)
    - Problematic when 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 with different labels occur at the same time
- Computation starts at a change of the input
- Delays are arbitrary, but bounded
- 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

- Behavior strongly dependent on input timing
- At the implementation level:
  - 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 in terms of
  - Event buffering
  - Timing of event read/write
- Asynchrony
  - Explicit buffering of events for each module
    - Buffer size may be 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

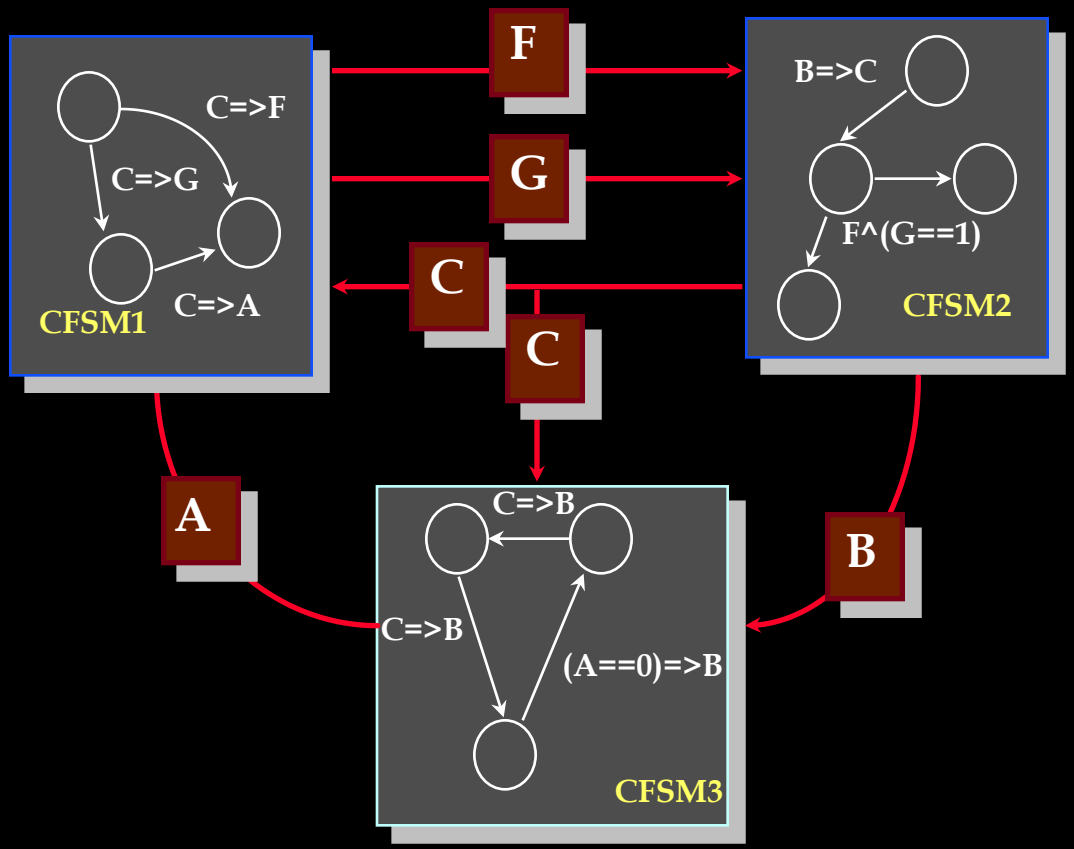
- CFSM is FSM extended with
  - Support for data handling
  - Asynchronous communication
- CFSM has
  - FSM part
    - Inputs, outputs, states, transition and output relation
  - Data computation part
    - External, instantaneous functions

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:
  - ☆ Idle
  - 🕐 Detect input events
  - 🕑 Execute one transition
  - 🕒 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)

Conclusion

- CFSM
  - Delay, and hence detailed behavior, is defined by implementation
  - Local synchrony
    - Relatively large atomic synchronous entities
  - Global asynchrony
    - Break synchrony, no compositional problem
    - Allow efficient mapping to heterogeneous architectures