

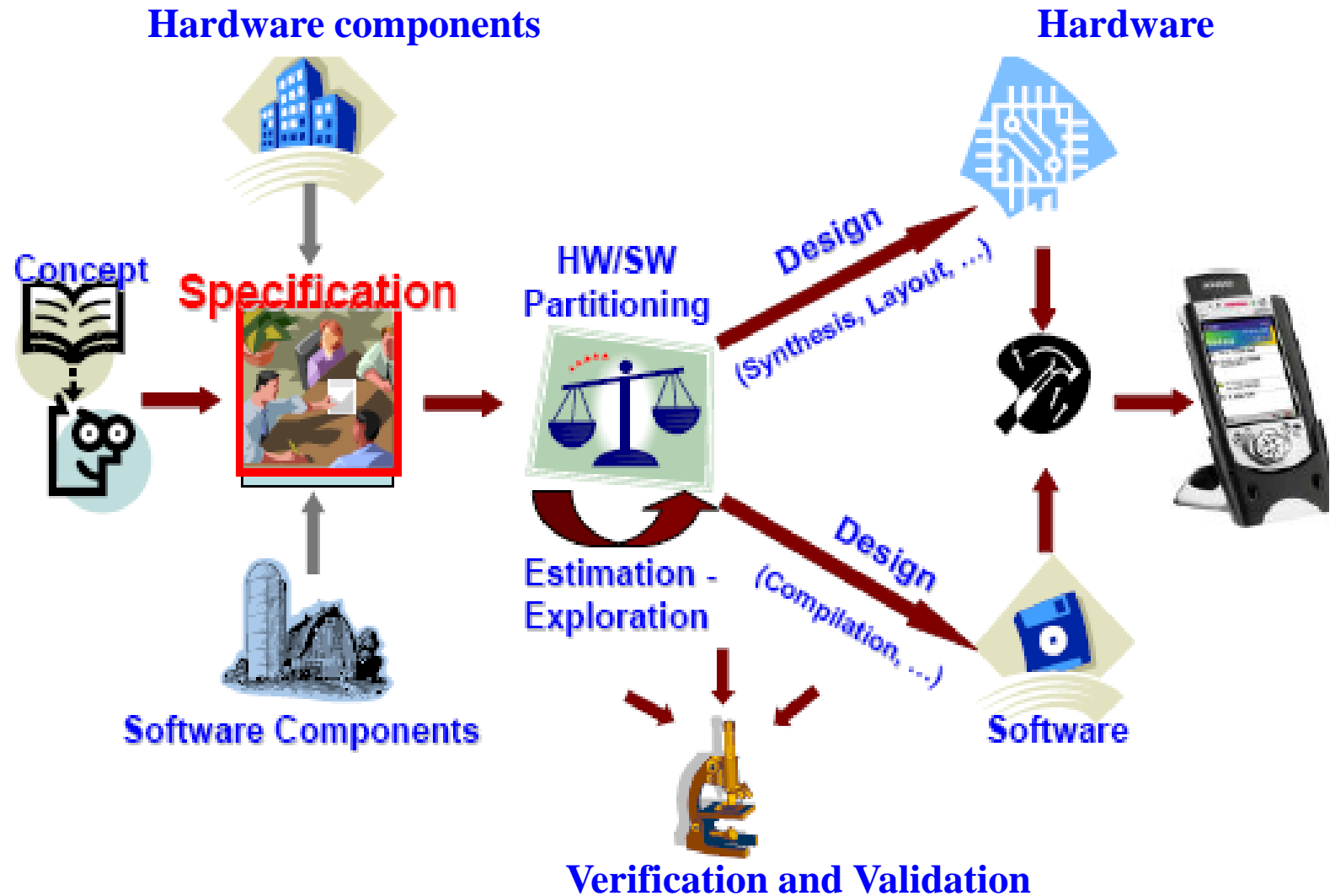
Embedded System Modeling

Ke Huang

Department of Electrical and Computer Engineering
San Diego State University

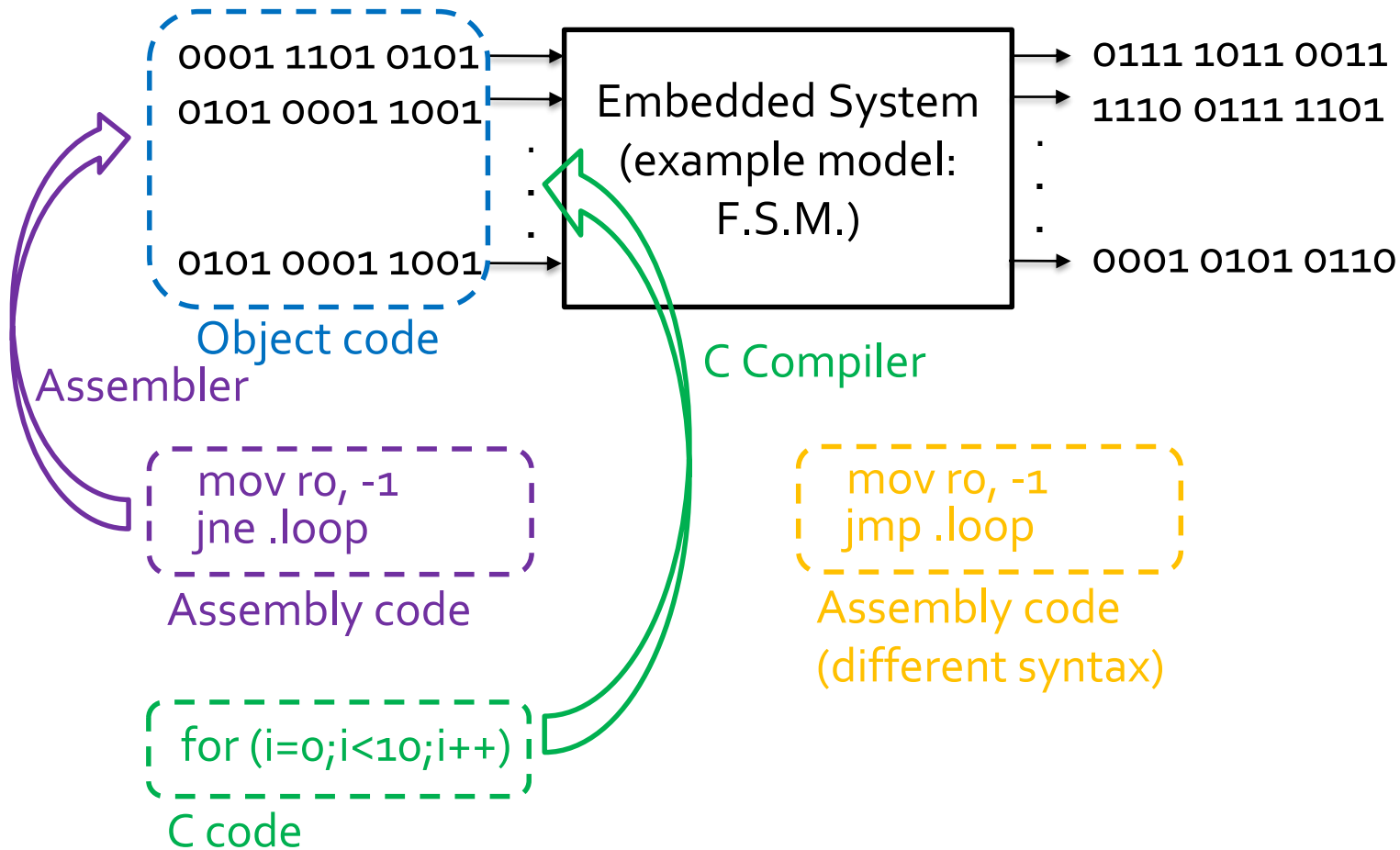


ES Design

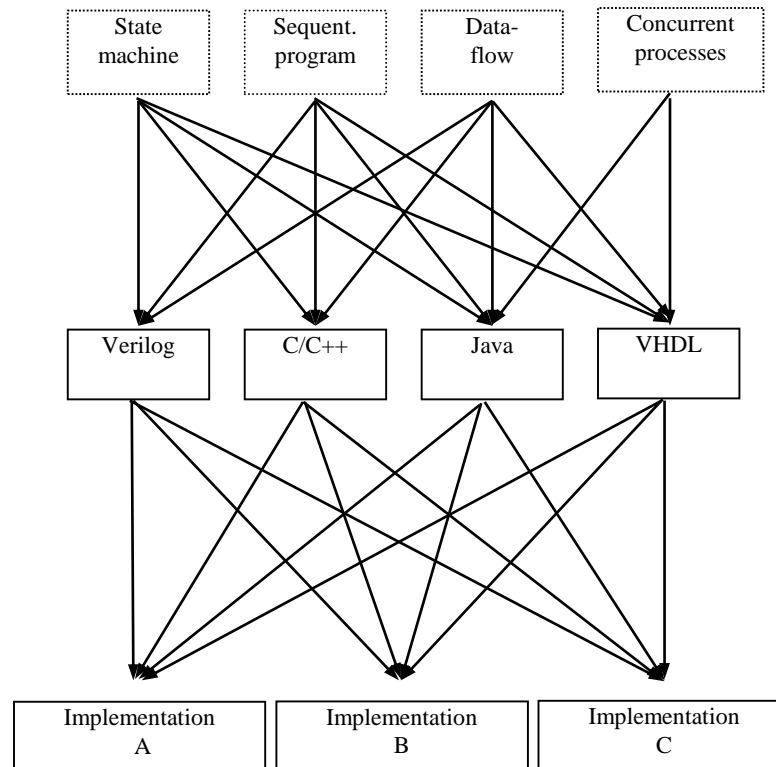
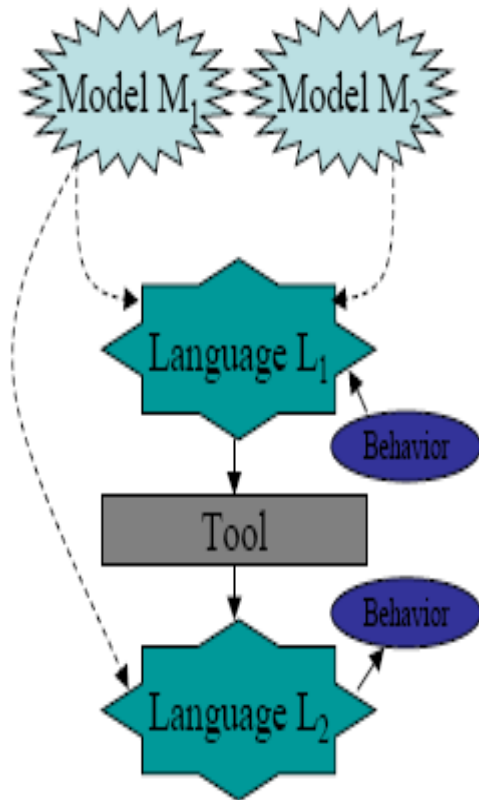


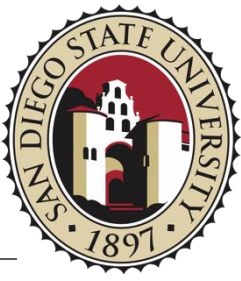


Models, Languages and Tools



Models, Languages and Tools





Design process

Design:

- A set of components interacting with each other and with the environment that is not a part of design

Model of Computation (MOC):

- Defines the behavior and interaction of the design blocks

Design process:

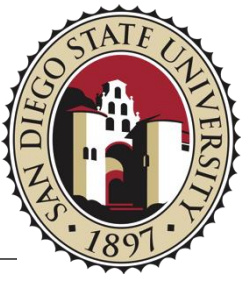
- Takes a model at a higher level of abstraction and refines it to a lower level along with mapping constraints, performance indices and properties to the same level

Validation:

- Process of checking if design is correct
 - Simulation/emulation, formal verification of specification or implementation

Synthesis:

- Design refinement where more abstract specifications are translated into less abstract specifications



Models of Computation Elements

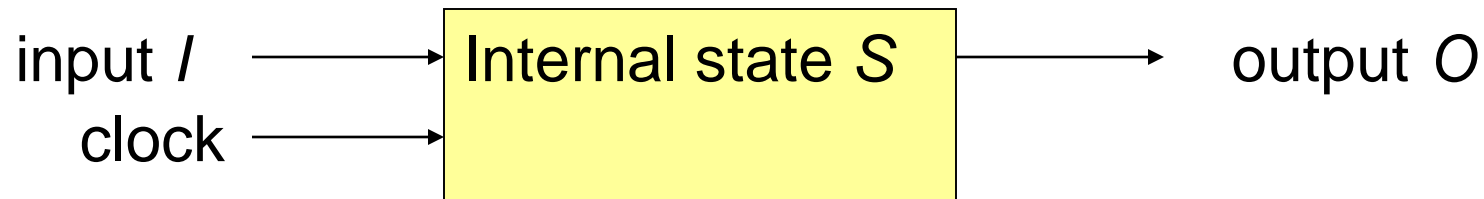
- State
 - e.g. in HW :
 - Combinational states: multiple states possible for time t
 - Sequential states: one state for a given time t
- Decidability
 - Can a property be determined in a finite amount of time?
- Concurrency and communication
 - Embedded systems usually have coordinated concurrent processes -> communication required



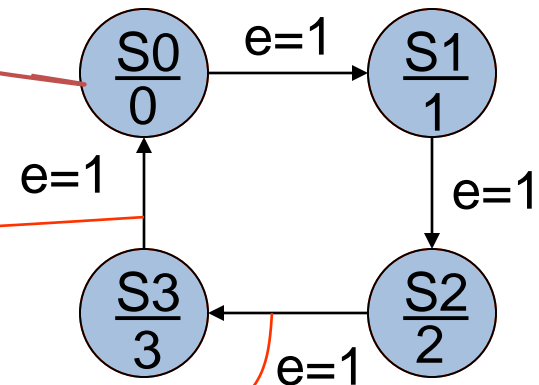
Model of Computation Examples

- **State machine models**
 - **FSMs**, Statecharts, SDL
- Petri nets
- Communicating processes
 - Kahn processes, Communicating Sequential Processes
- Ada
- Dataflow models
 - DFG, SDFG
- Discrete event models
 - VHDL, Verilog, SystemC, SpecC

Classical automata

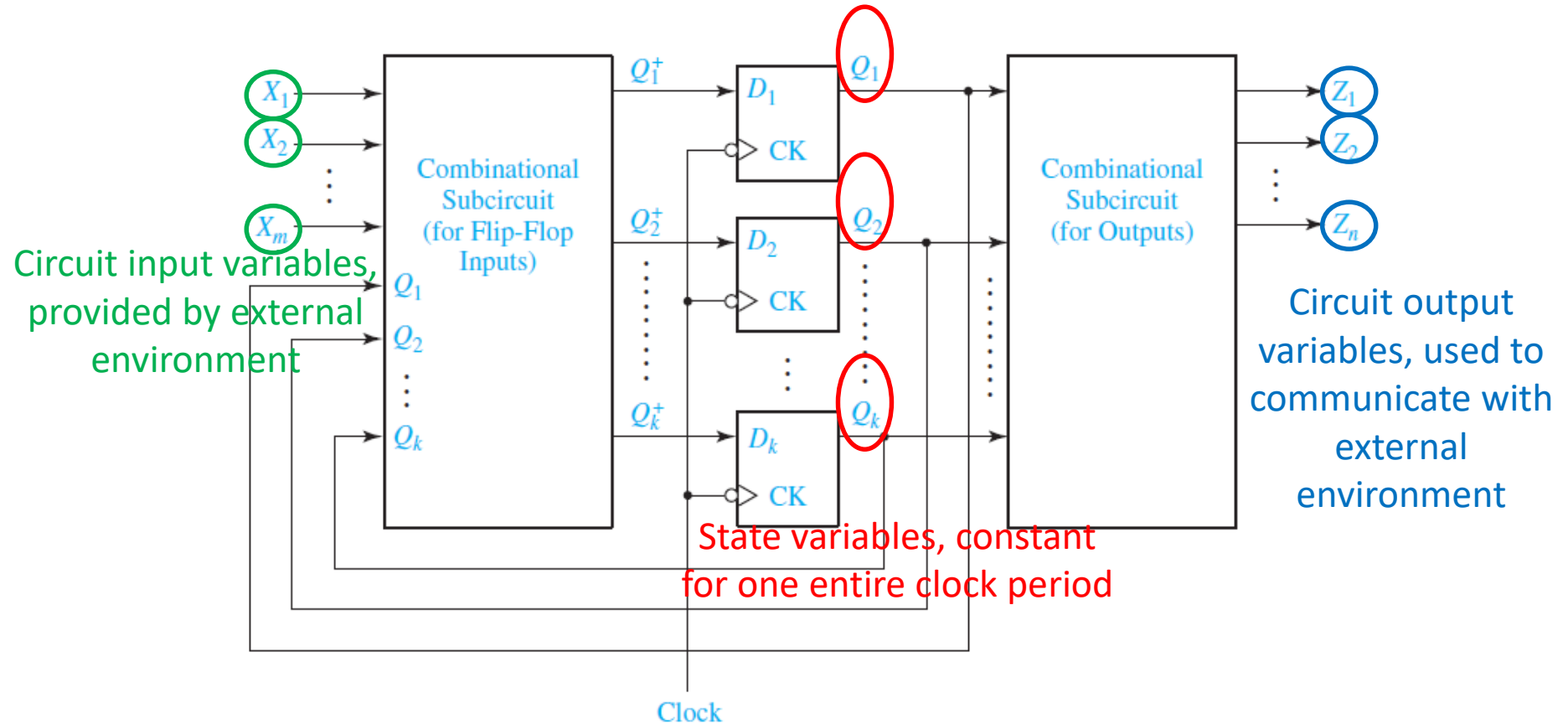


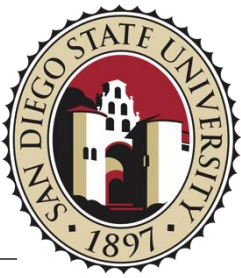
- Moore-automata:
 $O = H(S); \quad S^+ = f(I, S)$
- Mealy-automata
 $O = H(I, S); \quad S^+ = f(I, S)$



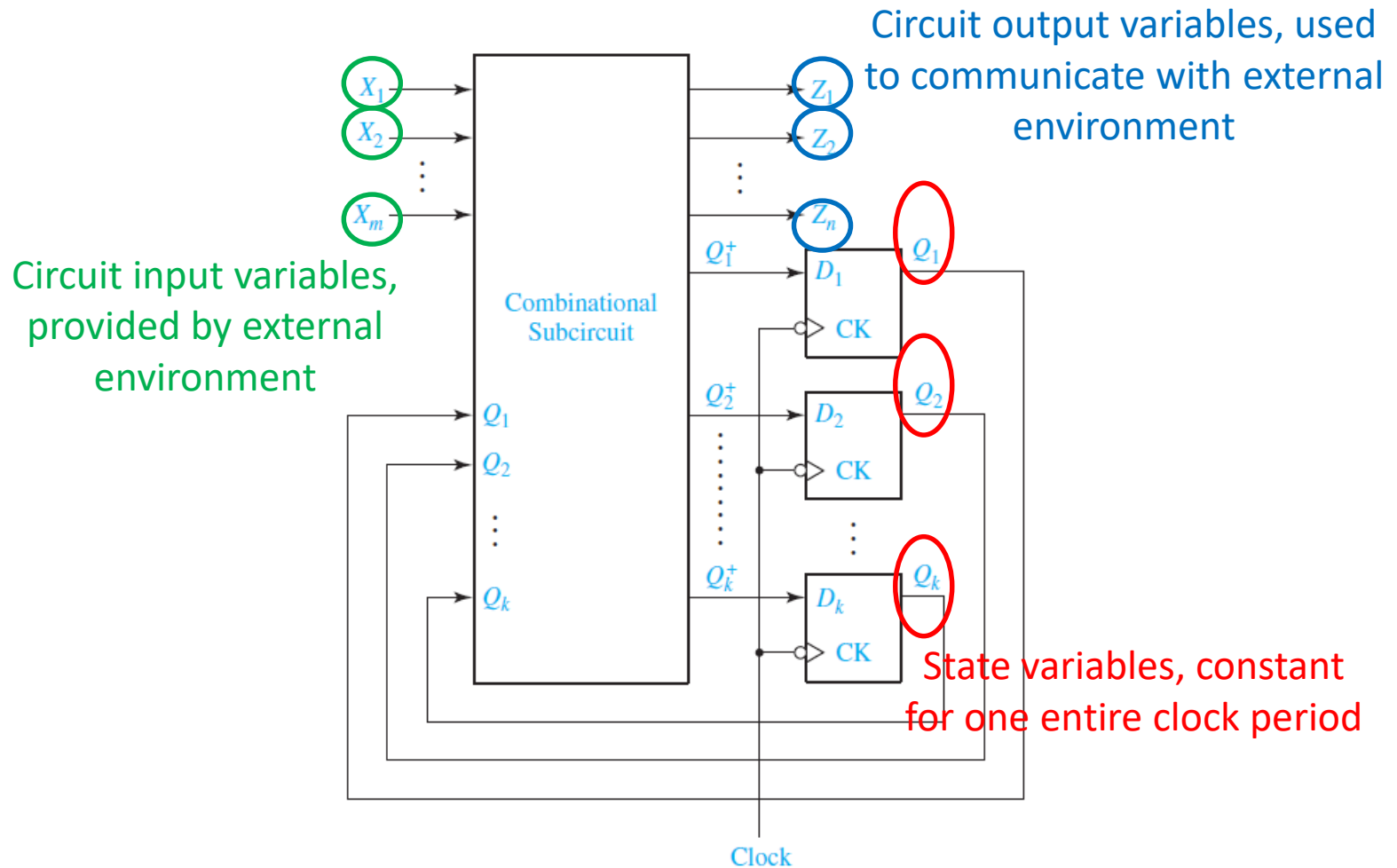


Moore FSM



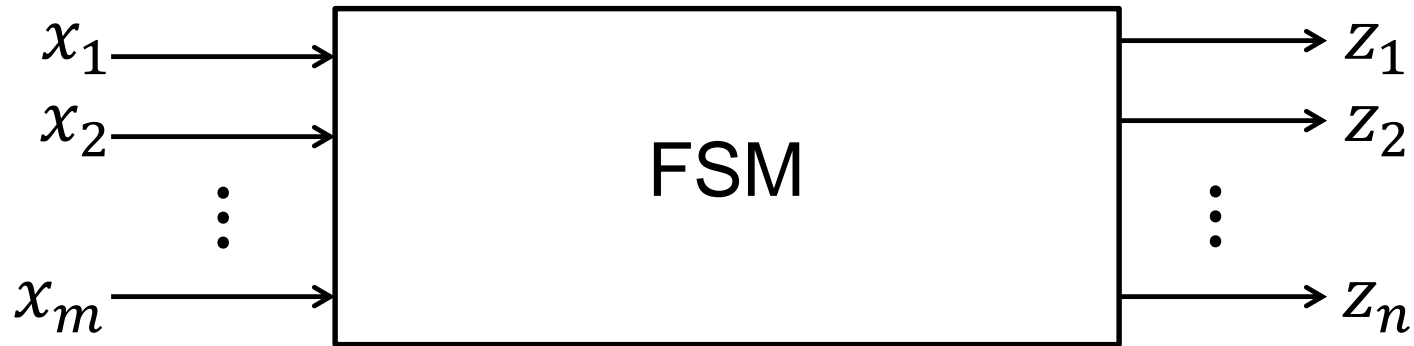


Mealy FSM





General FSM I/O



Input variables:
controllable

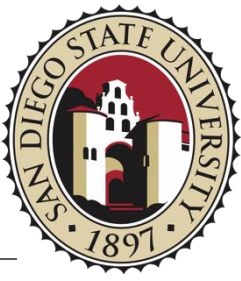
↑
In general, internal states
are not controllable and
observable

Output variables:
observable



Mealy vs. Moore Machines

- Moore machine guarantees the outputs are steady for a full clock cycle
- A change in the input takes at least once cycle to affect the output in a Moore machine
 - In Mealy machines, input change can cause output change as soon as logic is done—a big problem when two machines are interconnected –
- Moore machine might require more states since the output does not depend on input
- Moore machine is mostly seen as simpler to design



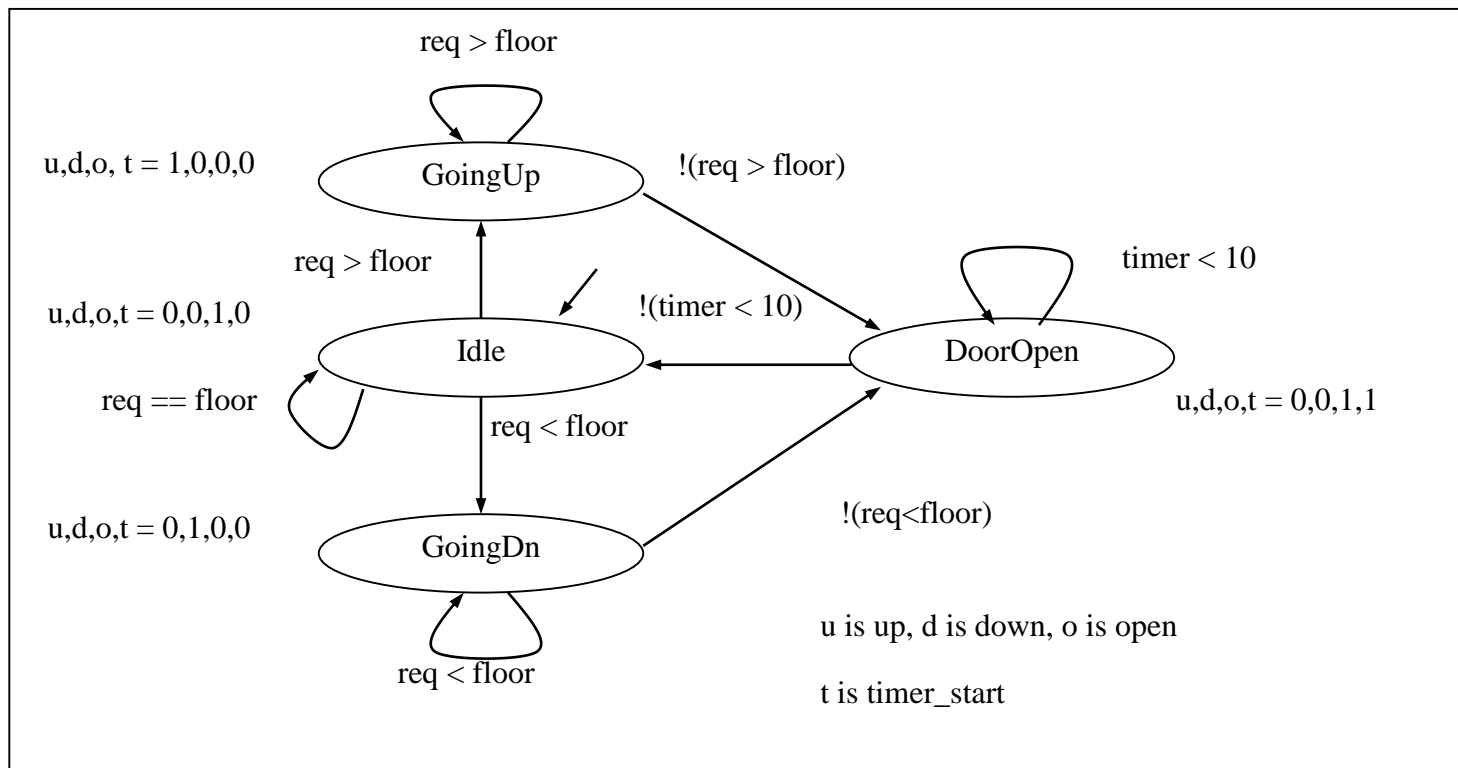
State Machine Design Process

1. Determination of inputs and outputs
2. Determination of machine states
3. Create state diagram – Moore vs. Mealy?
4. State assignment – assign a particular value for each state
5. Create transition/output table
6. Derive next state logic for each state element
7. Derive output logic
8. Implementation



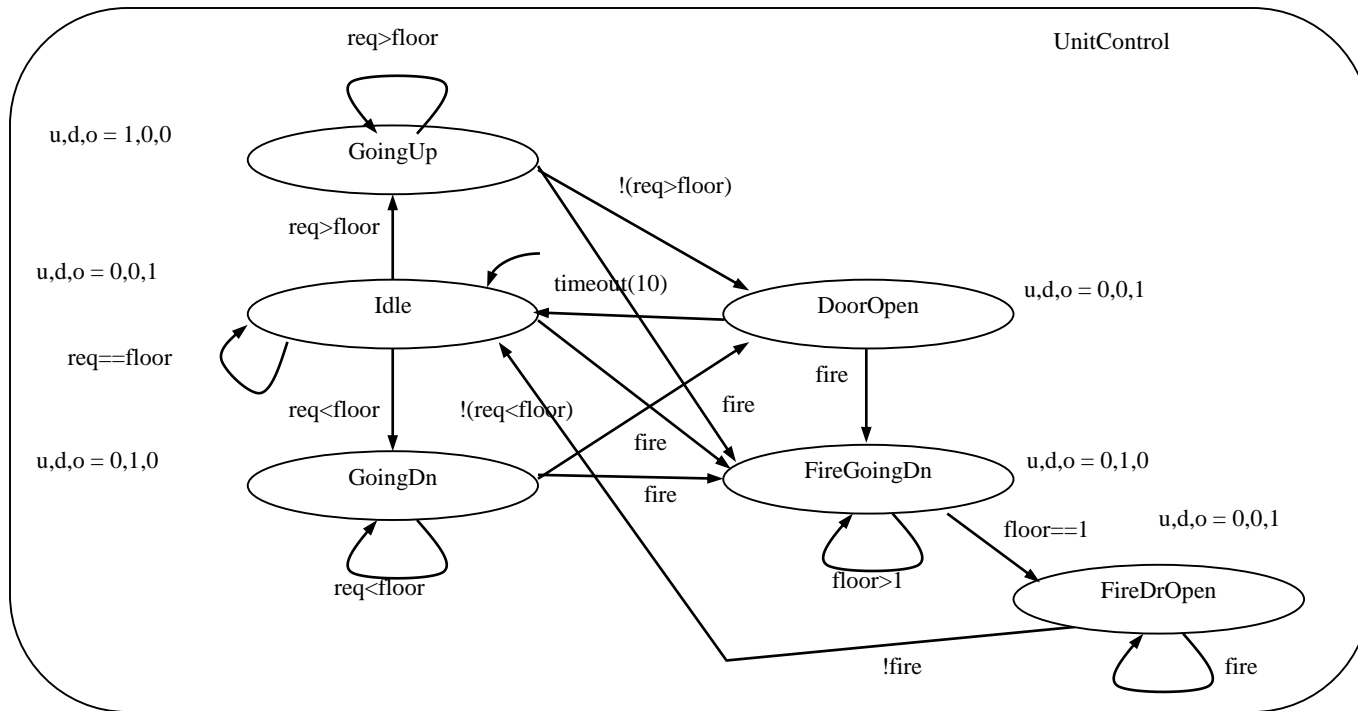
Finite-state machines (FSMs)

Elevator Control process using a state machine





Elevator Control with Fire Mode



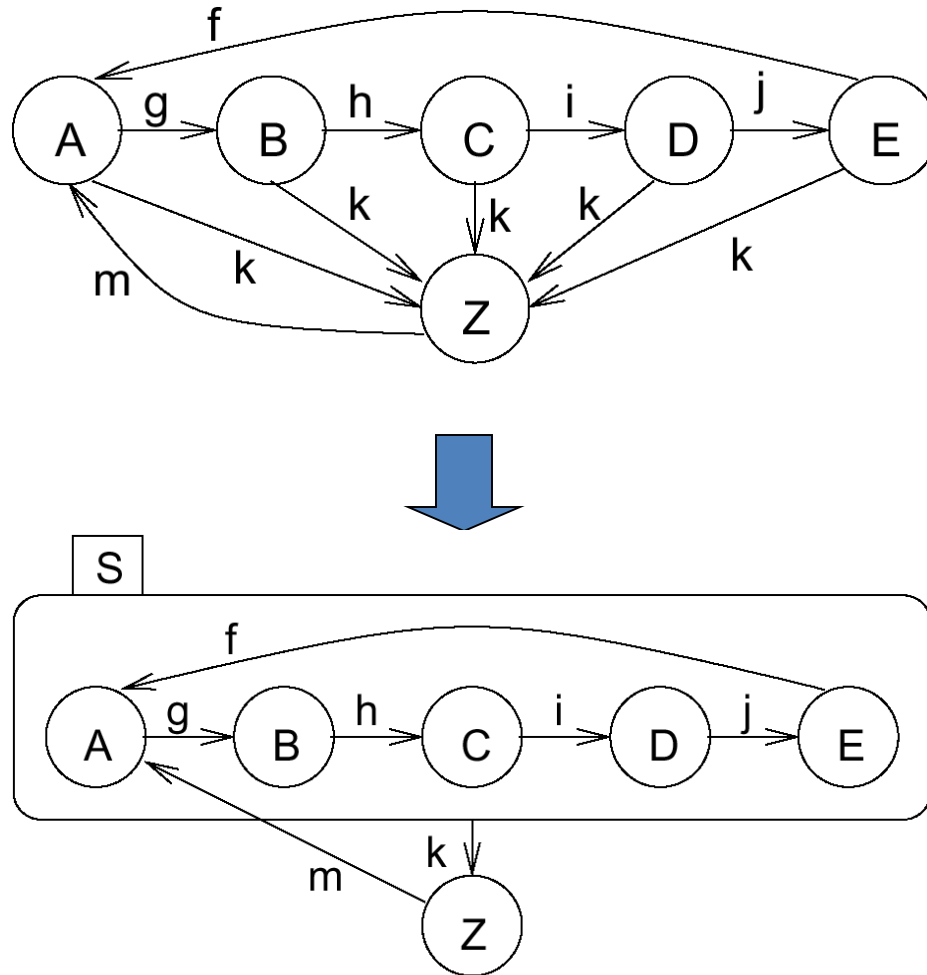
- FireMode
 - When *fire* is true, move elevator to 1st floor and open door



Models of Computation

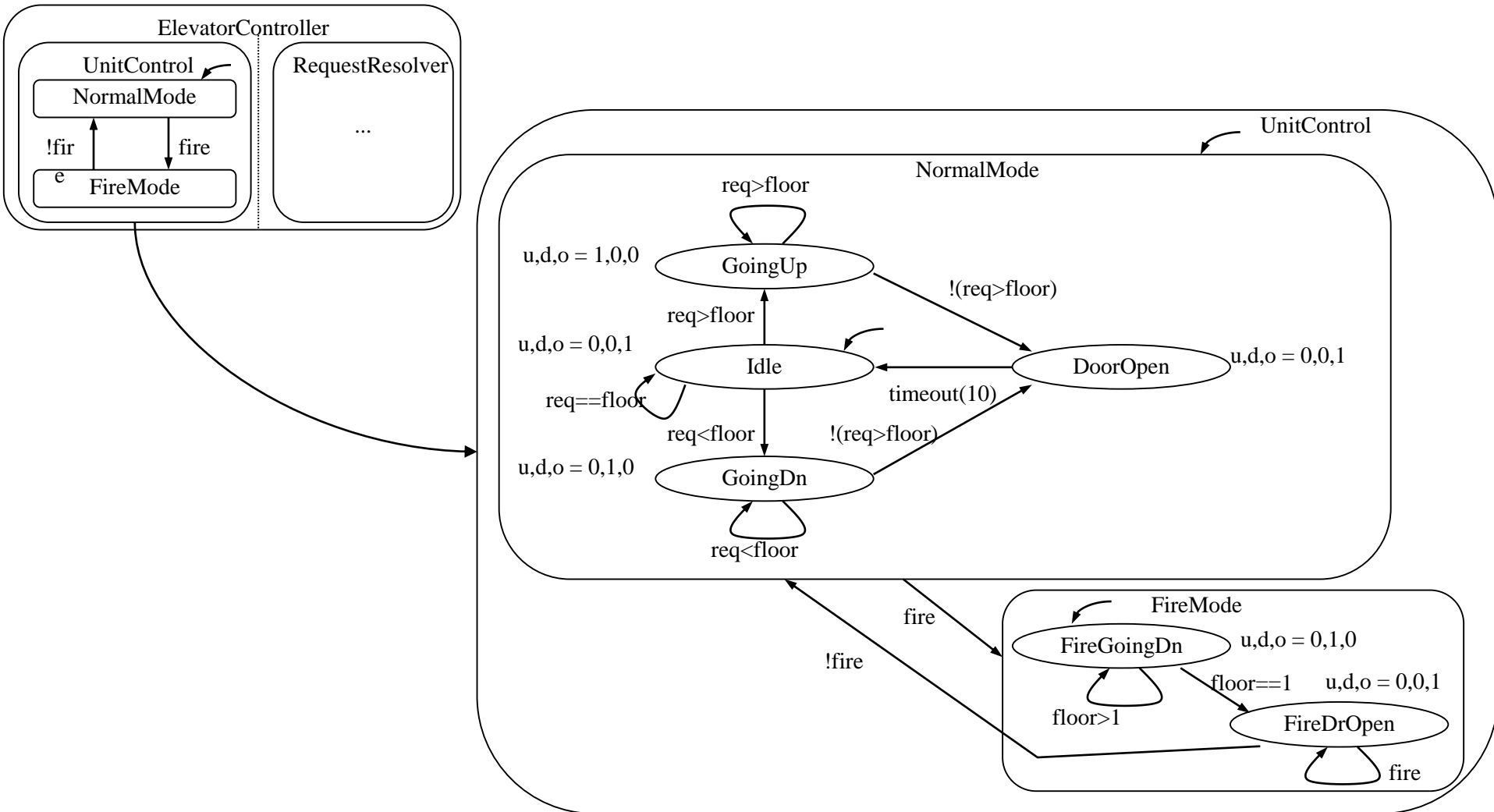
- **State Machine Models**
 - FSM, **StateCharts**, CFSM, SDL
- Petri nets
- Communicating Processes
 - Kahn processes, Communicating Sequential Processes
- Ada
- Dataflow models
 - DFG, SDFG
- Discrete Event Systems
 - VHDL, Verilog, SystemC, SpecC

StateCharts: Hierarchy



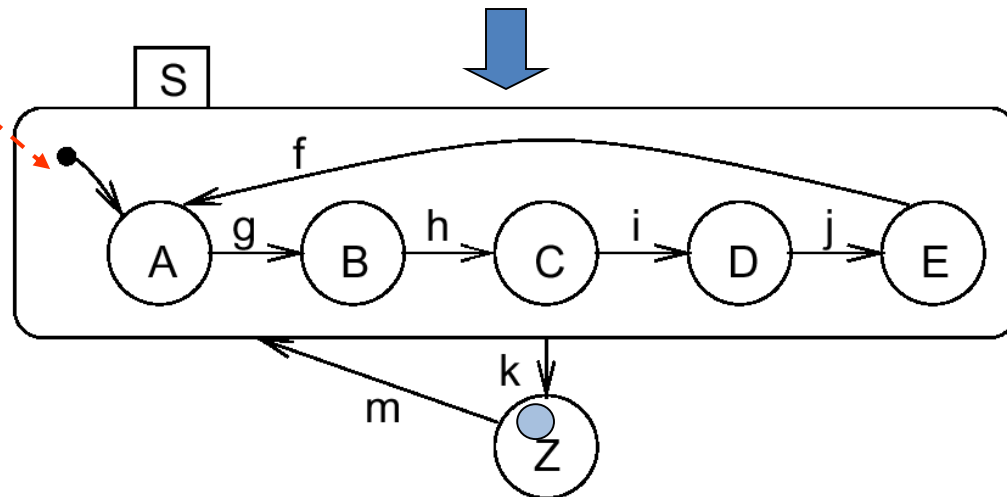
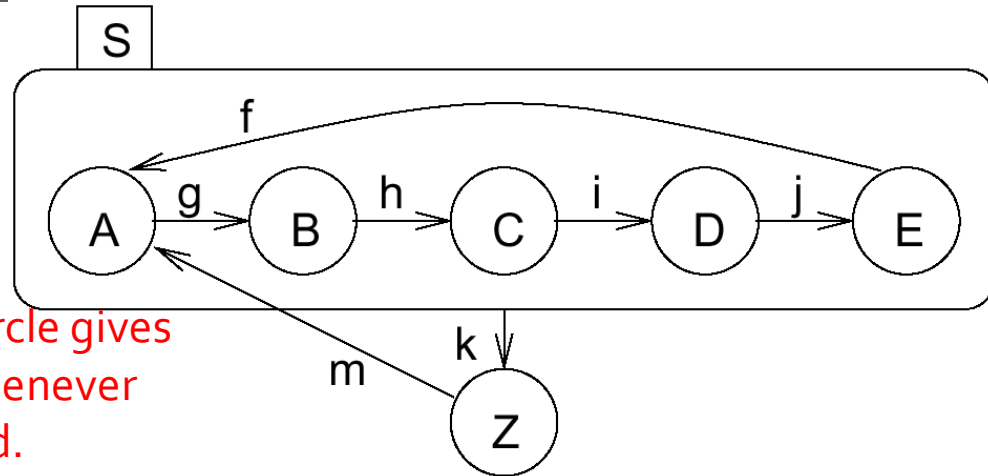


Back to Elevator Example

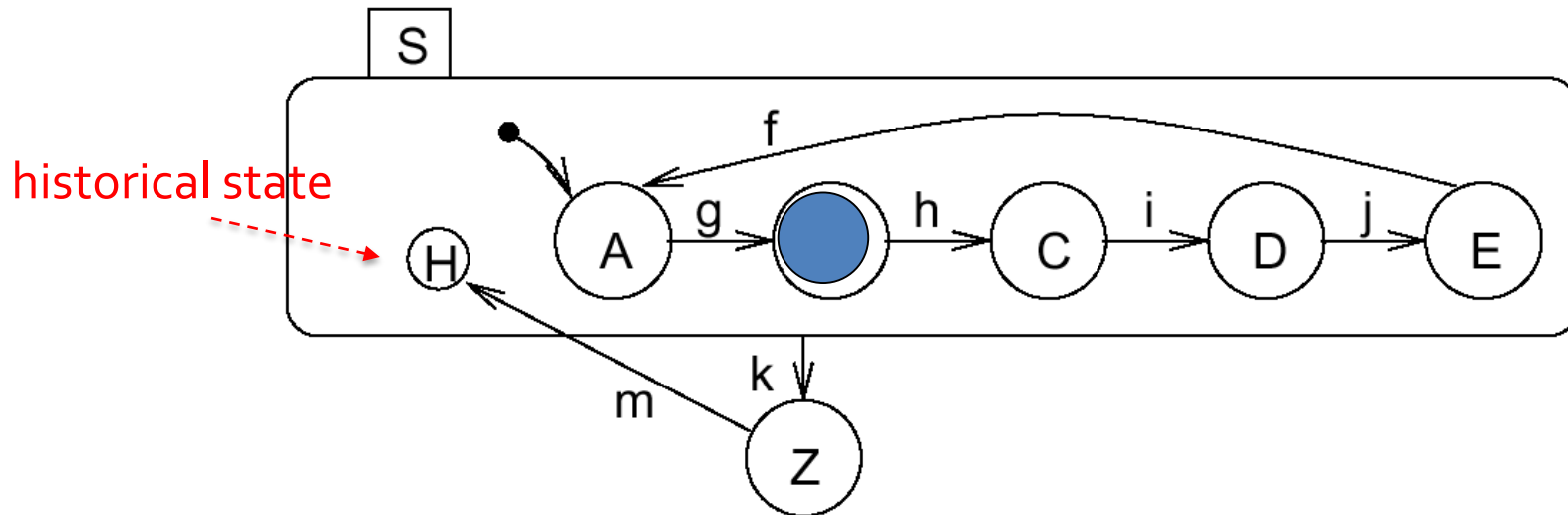


StateCharts: Default state

- Default state filled circle gives sub-state entered whenever super-state is entered.
- Not a state by itself!

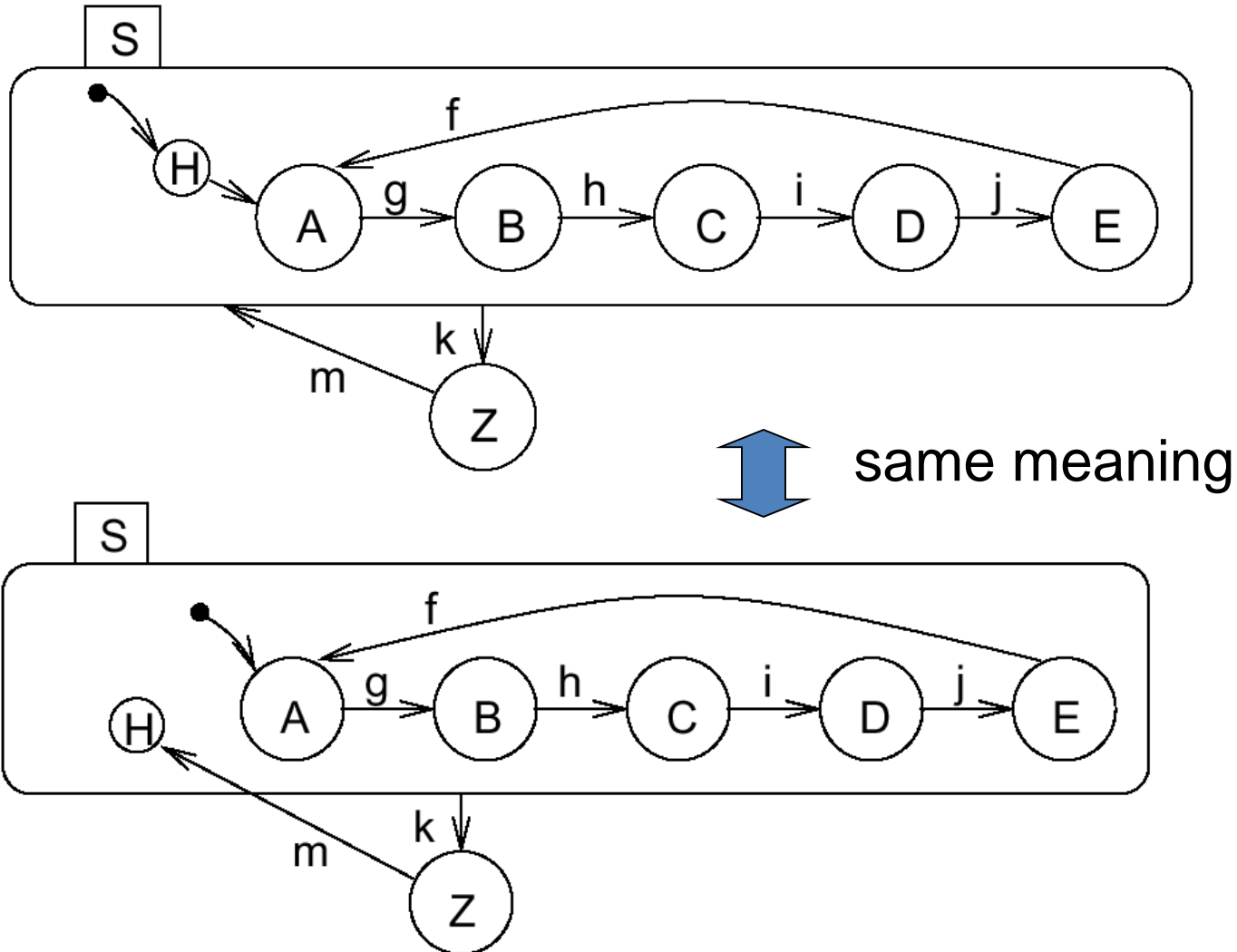


StateCharts: History



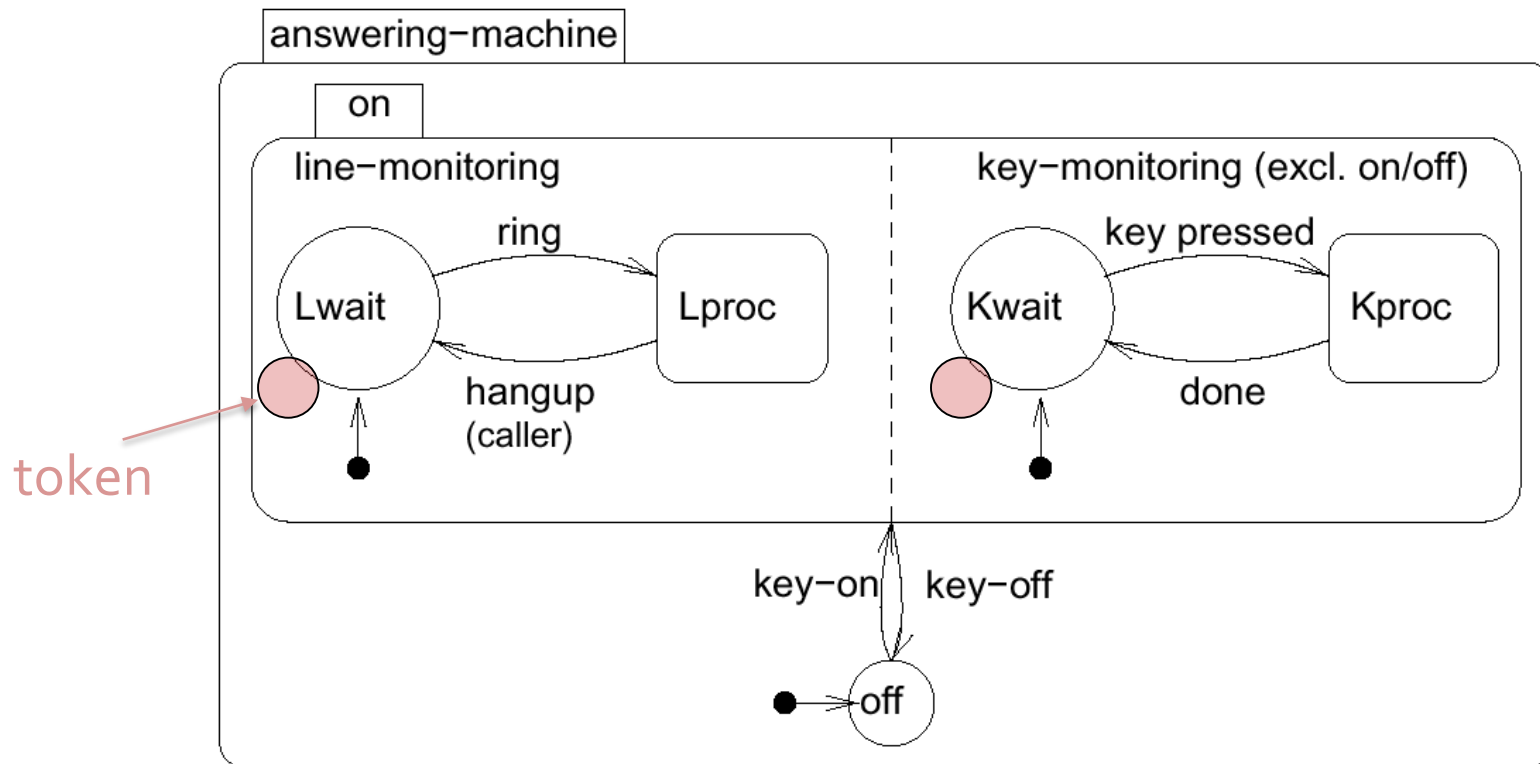
- For input m , S enters the state it was in before S was left (can be A , B , C , D , or E). If S is entered for the very first time, the default mechanism applies. History and default mechanisms can be used hierarchically.

History & default state





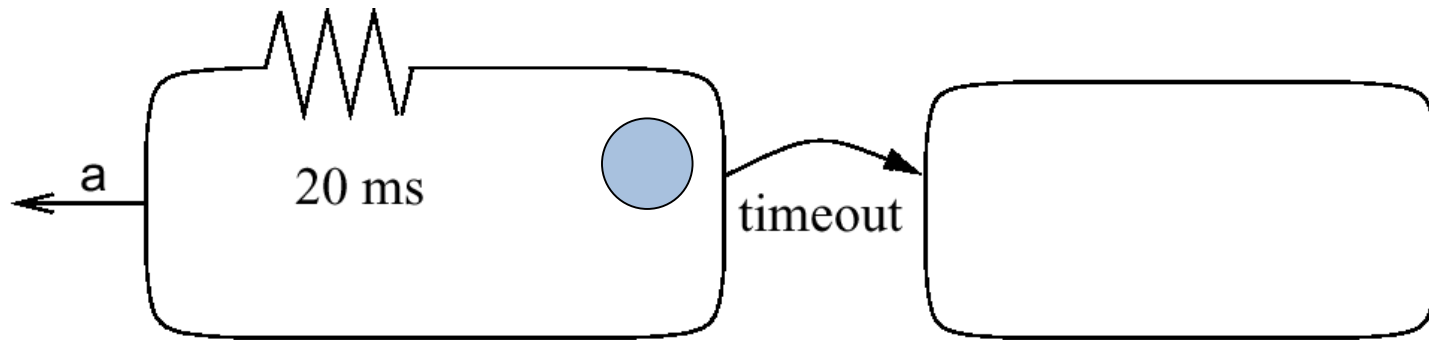
StateCharts: Concurrency



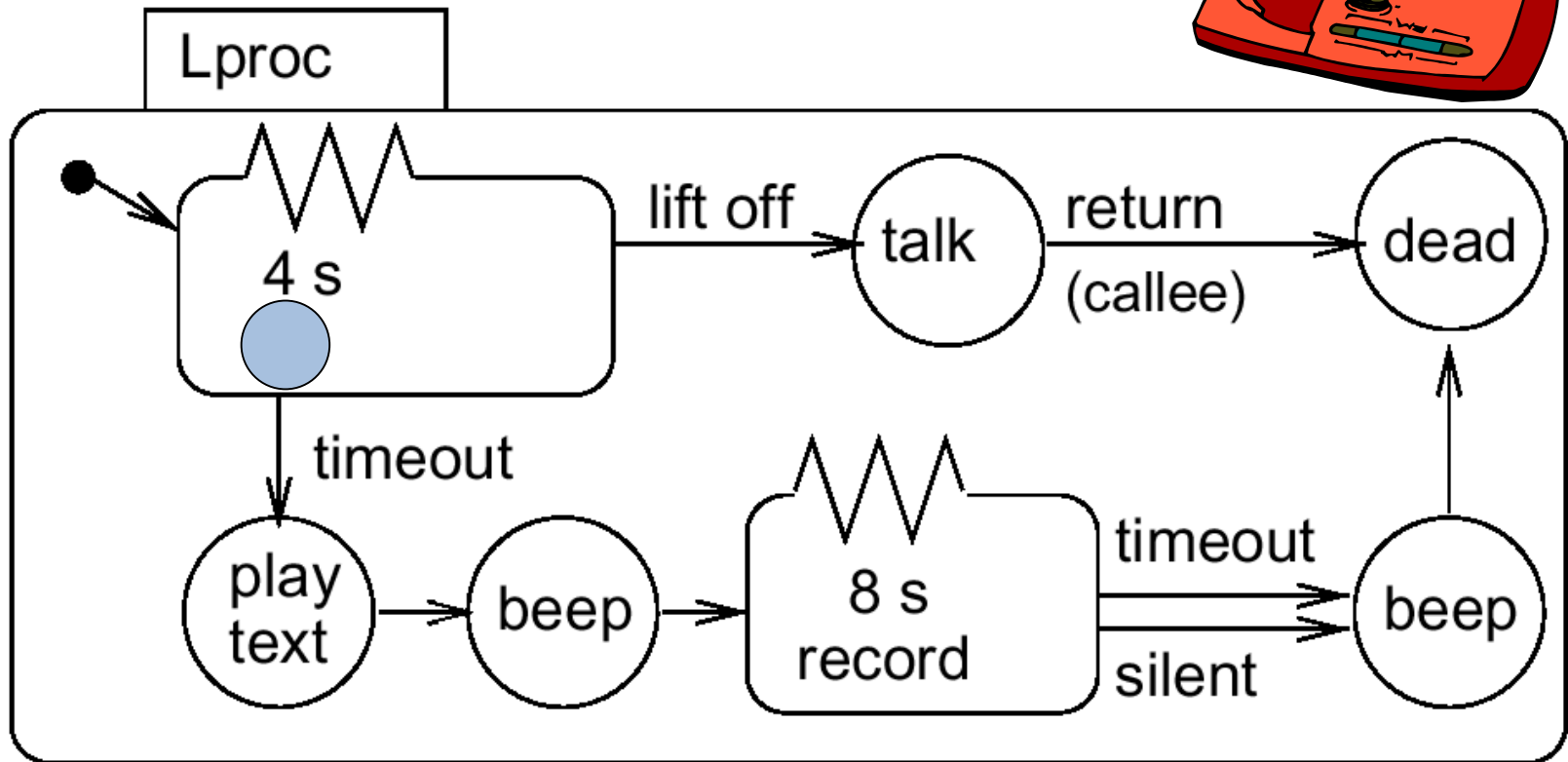
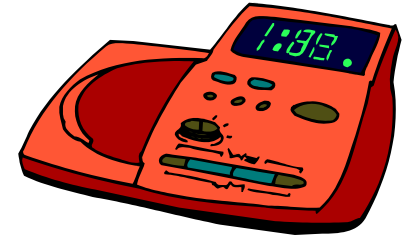
- Concurrency: being in more than one state at a time.
- Places of **tokens** denote current states.



StateCharts: Timers

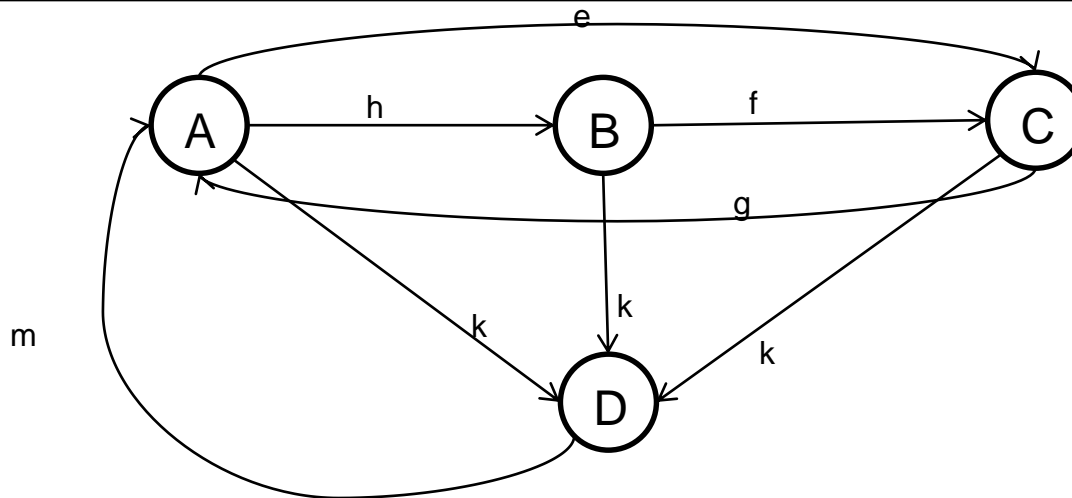
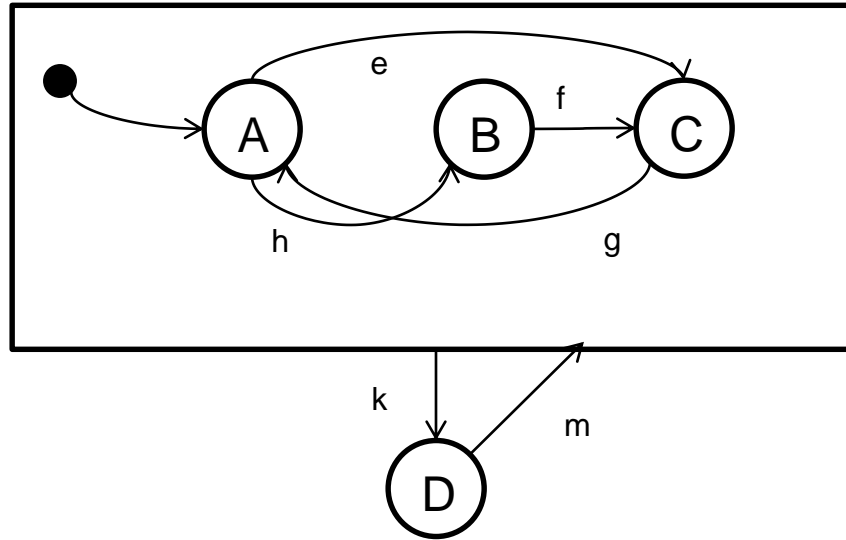


Example: Answering machine



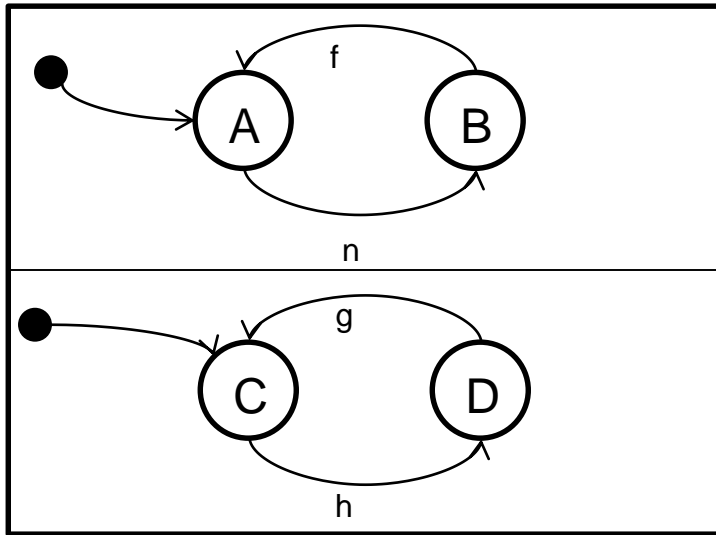
Statecharts – Example 1

Statechart
Example

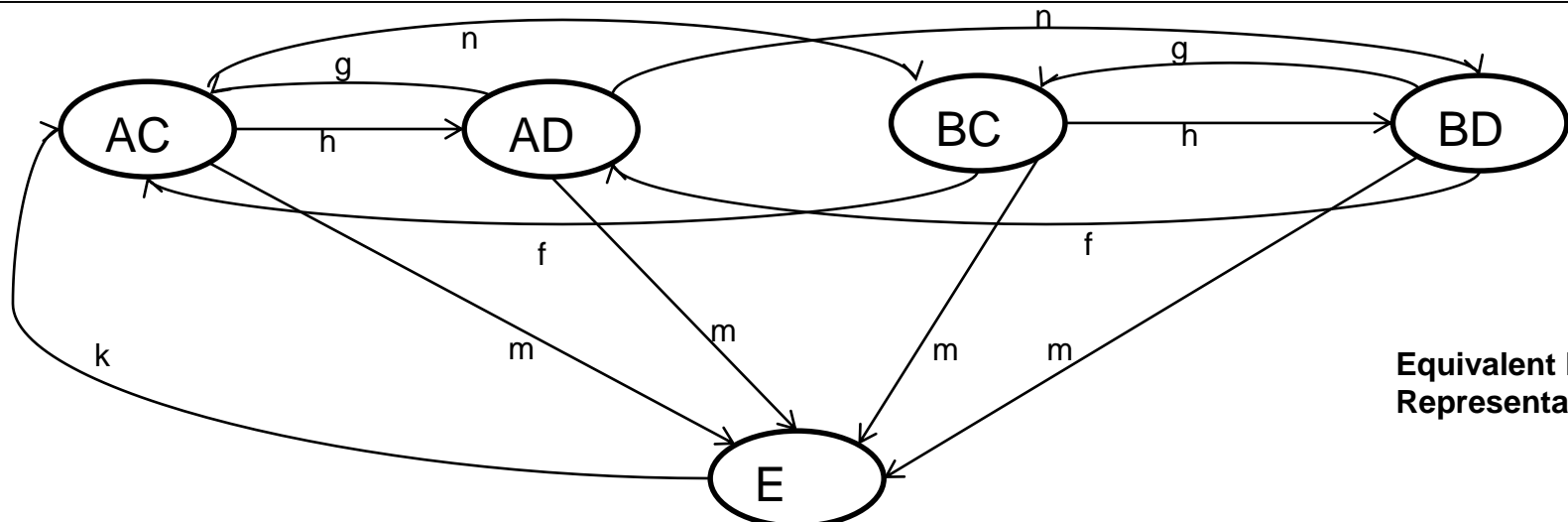


Equivalent FSM
Representation

Statecharts – Example 2

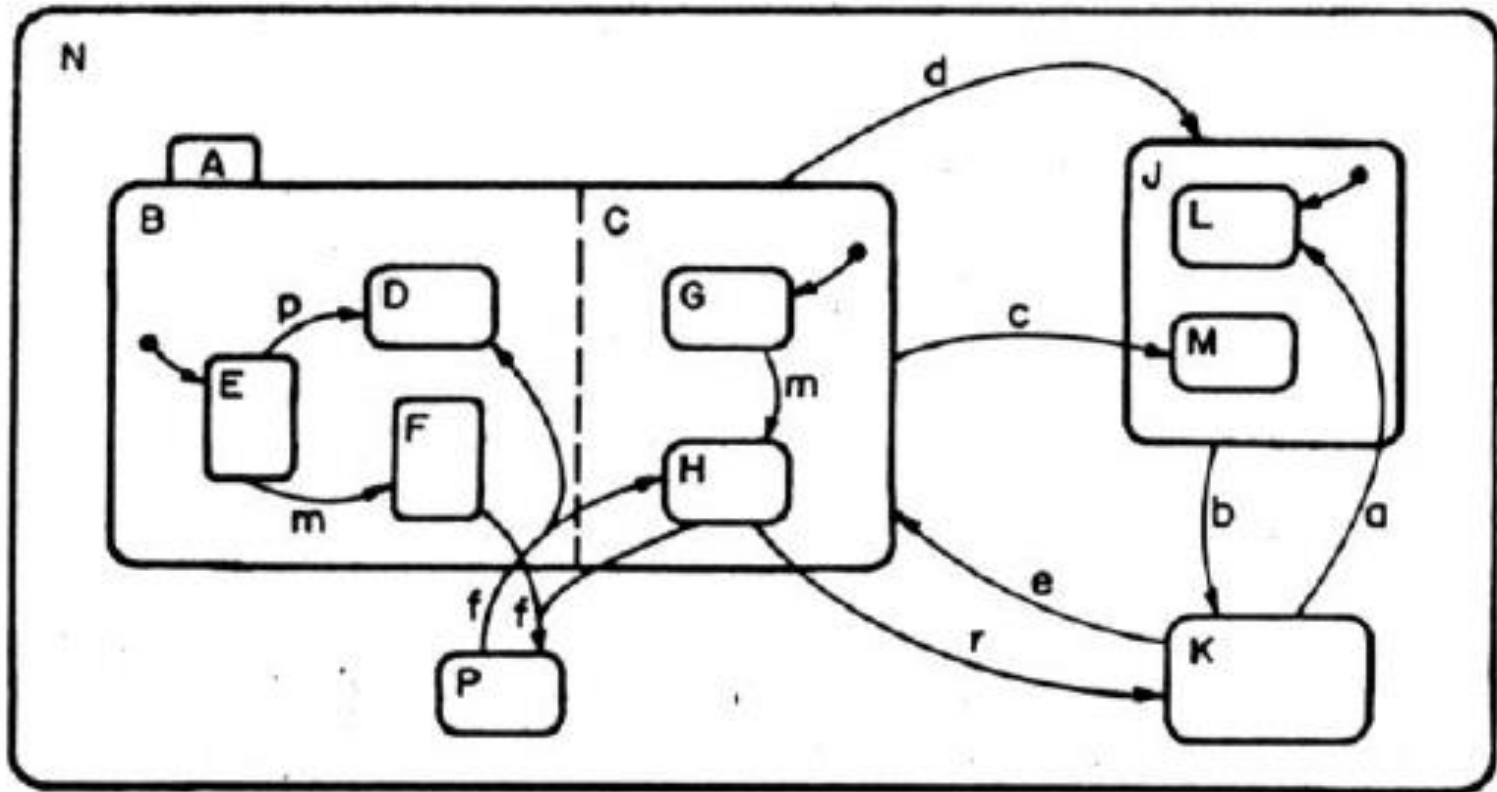


**Statechart
Example**



**Equivalent FSM
Representation**

StateCharts to FSM – Example 3





Design Problem – Example 4

We are designing a car monitoring and emergency detection system using sensors and actuators to safely operate a vehicle. The user will be able to turn on and off the ignition, and the system will monitor several sensors and actuate warnings/alarms for emergency scenarios.

The follows are the description of the system:

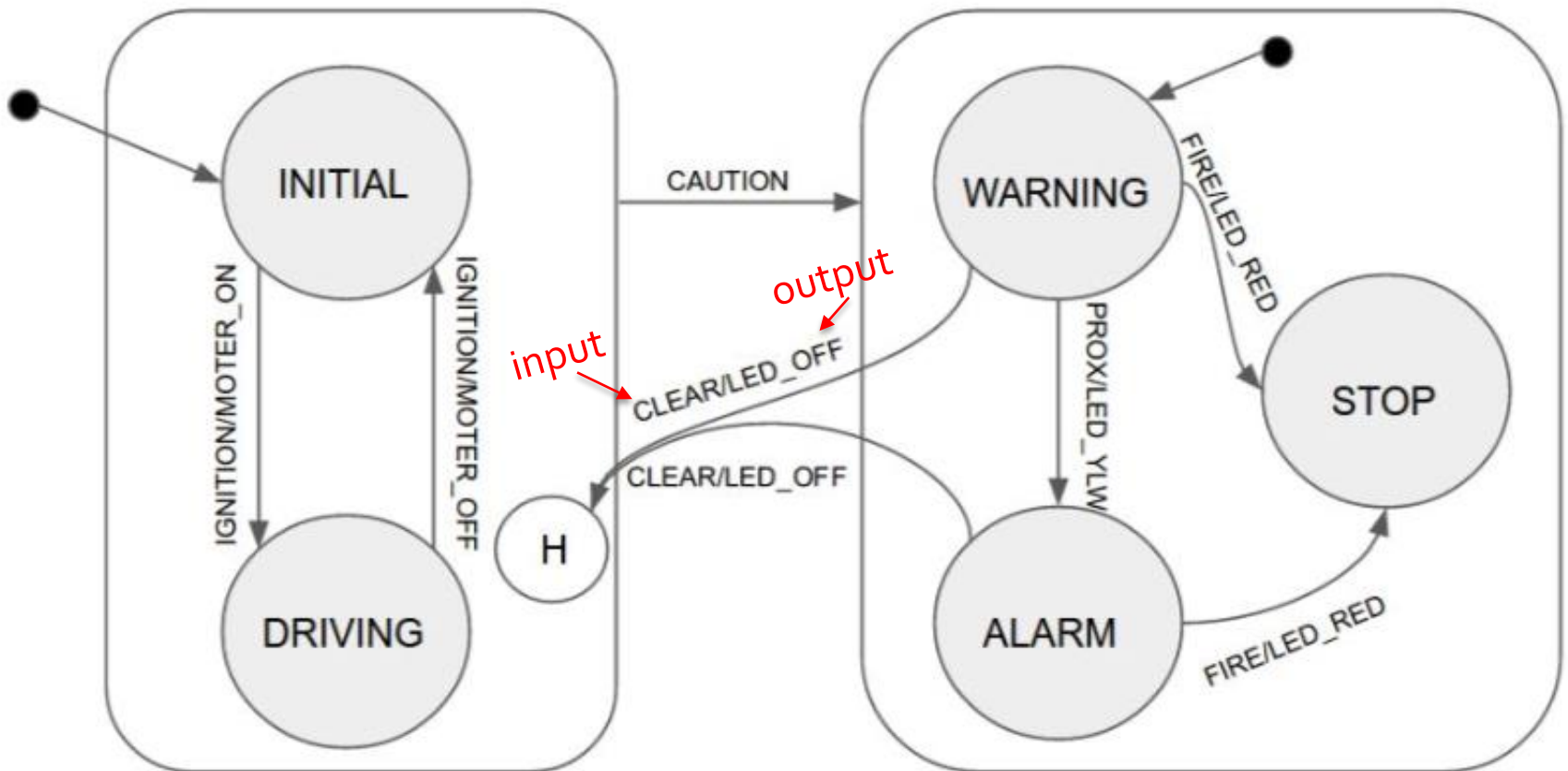
- 1) Before pushing the ignition button, the system is in the *"INITIAL"* state.
- 2) After pushing the ignition button, the systems receives the input message *"IGNITION"*, and sends the output message, *"MOTOR_ON"*, to start the motor, and goes to a *"DRIVING"* state.
- 3) If the ignition button is pushed (i.e., *"IGNITION"*) in the *"DRIVING"* state again, the system sends *"MOTOR_OFF"*, and reverts back to the *"INITIAL"* state.
- 4) When in either the *"INITIAL"* or *"DRIVING"* state, it may identify any anticipated emergency situations. Once the system receives the input message, *"CAUTION"*, it goes to the *"WARNING"* state.
- 5) In the *"WARNING"* state, a proximity sensor can detect the distances to surroundings. If the corresponding *"PROX"* message occurs, the system outputs the *"LED_YLW"* message, and goes into the *"ALARM"* state.
- 6) If high temperature is detected in either the *"WARNING"* or *"ALARM"* state, the system receives the *"FIRE"* message, and goes into the *"STOP"* state.
- 7) Before going to the *"STOP"* state, it should change the color of the LED with *"LED_RED"*.
- 8) If *"CLEAR"* message is in either the *"WARNING"* or *"ALARM"* states, the system turns off the led with *"LED_OFF"*, and should go back to the original state, i.e., one of *"INITIAL"* or *"DRIVING"*. For example, if the vehicle was in *"DRIVING"*, it should be still in the same state.
- 9) When in the *"STOP"* state, it cannot change its states anymore.



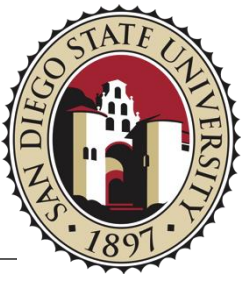
Example 4

- In this system, there are 5 states and 10 input/output messages to operate sensors/actuators as follows:
- State: " *INITIAL* ", " *DRIVING* ", " *WARNING* ", " *ALARM* ", " *STOP* "
- Message (Signal): " *IGNITION* ", " *MOTOR_ON* ", " *MOTOR_OFF* ", " *CAUTION* ", " *PROX* ", " *FIRE* ", " *CLEAR* ", " *LED_YLW* ", " *LED_RED* ", " *LED_OFF* "
- Draw a StateChart that accurately represents the functionality of the system. Use **two** superstates where appropriate to make your state diagram more readable. Each of all 5 states has to belong to one of the superstates.

Example 4



StateCharts: Application Examples



- Power converter system for trams, metros & trains
 - System-level modeling and automated code generation
 - Guarantee latencies less than 10 microseconds
 - Cut development time by 50%
 - Time from design completion to first prototype down to 1hr from 3 months
 - Defect free code automatically generated for a number of RTOS implementations

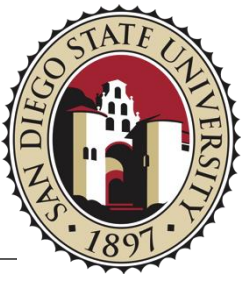


StateCharts: Application Examples



- Development of defibrillator and pacemaker technology
 - Model system behavior before requirements are finalized
 - Check that SW provides mathematically consistent representation of the product's system – correct and unambiguous representation of model behavior
 - More accurate and extensive verification – guarantee device works 100% of the time for at least 7-10yrs
 - Cut product verification costs by 20%
 - 15-20% overall cost reduction per projects on future development

StateCharts: Application Examples



- Jet engine electronic controller design
 - System level specification and consistency check
 - Construction of on-screen simulation of the cockpit display
 - Evaluate correctness in normal and faulty operation modes
 - Project so successful that now StateCharts are used for:
 - Independent overspeed protection system
 - Thrust reverse control system
 - Engine fuel control system



By BMW



StateCharts: Summary

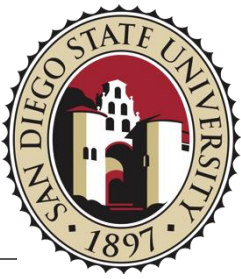
- Hierarchy
- AND- and OR-super states
- Default state, History
- Timing behavior
- State oriented behavior
- Edge labels
- Concurrency
- Synchronization & communication
 - Broadcast, shared memory
- Simulation
- Cross compiling



Models of Computation

- **State Machine Models**
 - FSM, StateCharts, **SDL**
- Petri nets
- Communicating Processes
 - Kahn processes, Communicating Sequential Processes
- Ada
- Dataflow models
 - DFG, SDFG
- Discrete Event Systems
 - VHDL, Verilog, SystemC, SpecC

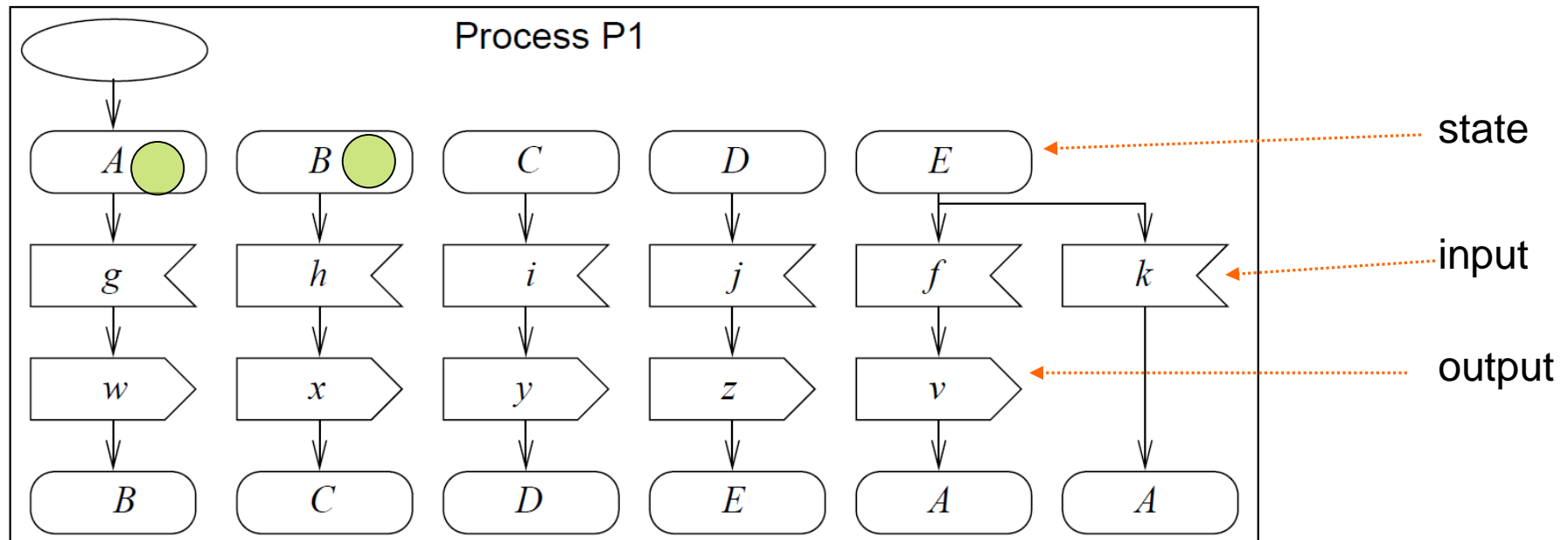
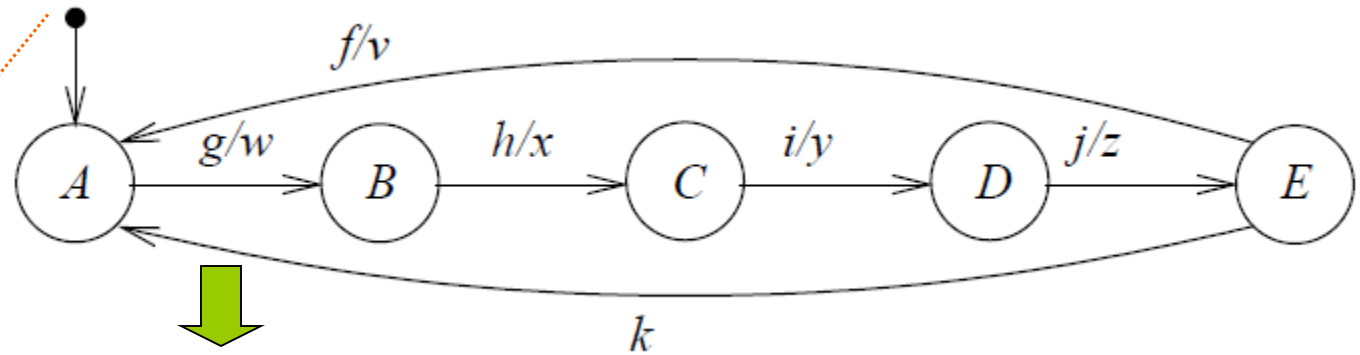
SDL: Specification and Description Language



- Designed for specification of distributed systems
 - Dates back to early 70s, formal semantics defined in the late 80s, updates from 1984 to 1999 by International Telecommunication Union (ITU)
 - Provides textual and graphical formats
- Similar to Statecharts, each FSM is called a **process**, but it uses message passing instead of shared memory for communication
- Supports operations on data

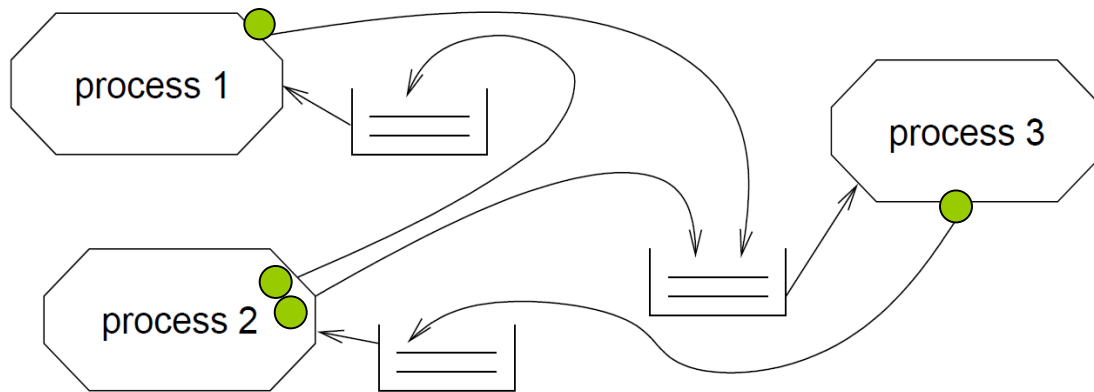


SDL-representation of FSMs/processes



Communication among SDL-FSMs

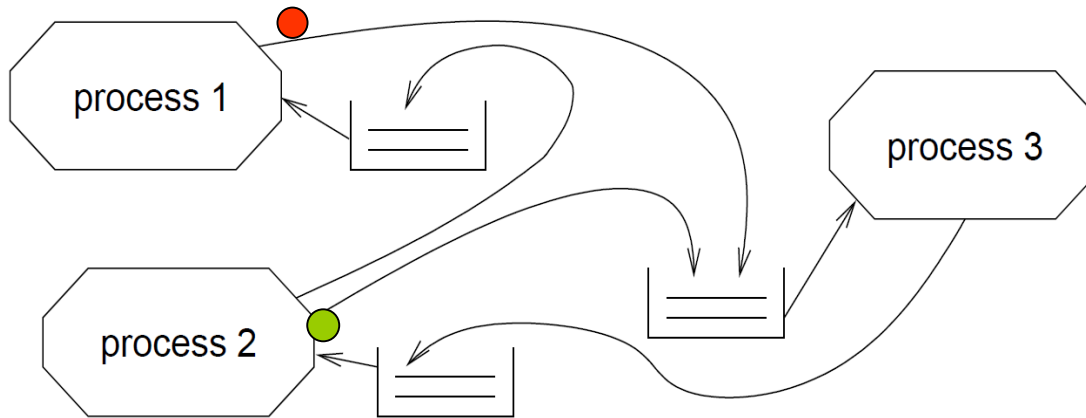
- Communication between FSMs (or “processes”) is based on **message-passing** (inputs + outputs), assuming a **potentially indefinitely large FIFO-queue**



- Each process fetches next entry from FIFO
- checks if input enables transition
- if yes: transition takes place
- if no: input is ignored (exception: SAVE-mechanism)

Determinate?

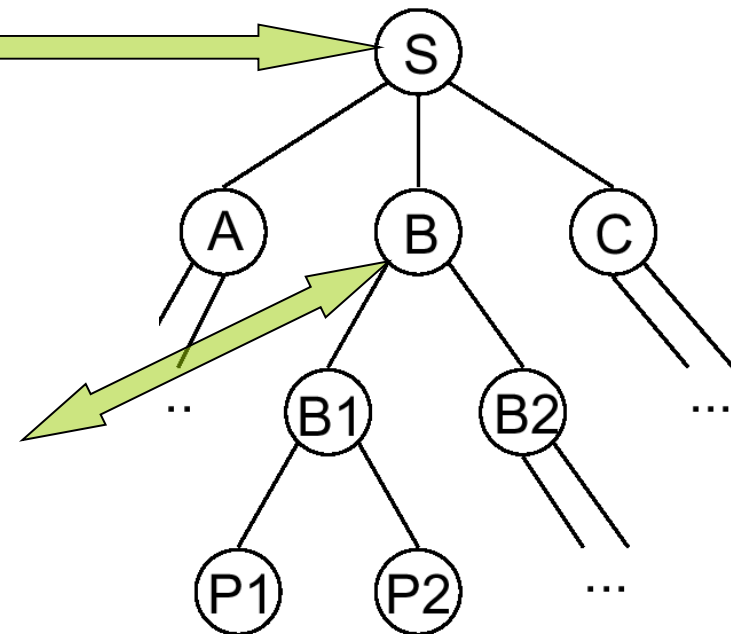
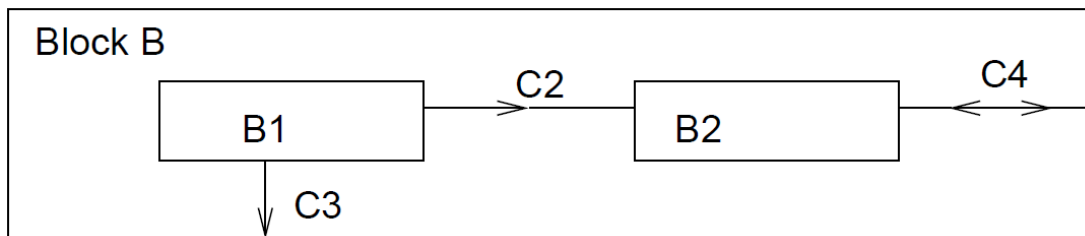
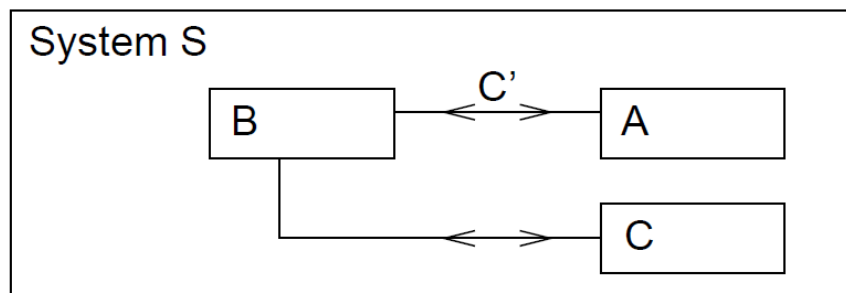
- Let tokens be arriving at FIFO at the same time:
 - Order in which they are stored is unknown:



All orders are legal so simulators can show different behaviors for the same input, all of which are correct

Hierarchy in SDL

- Process interaction diagrams can be included in **blocks**. The root block is called **system**

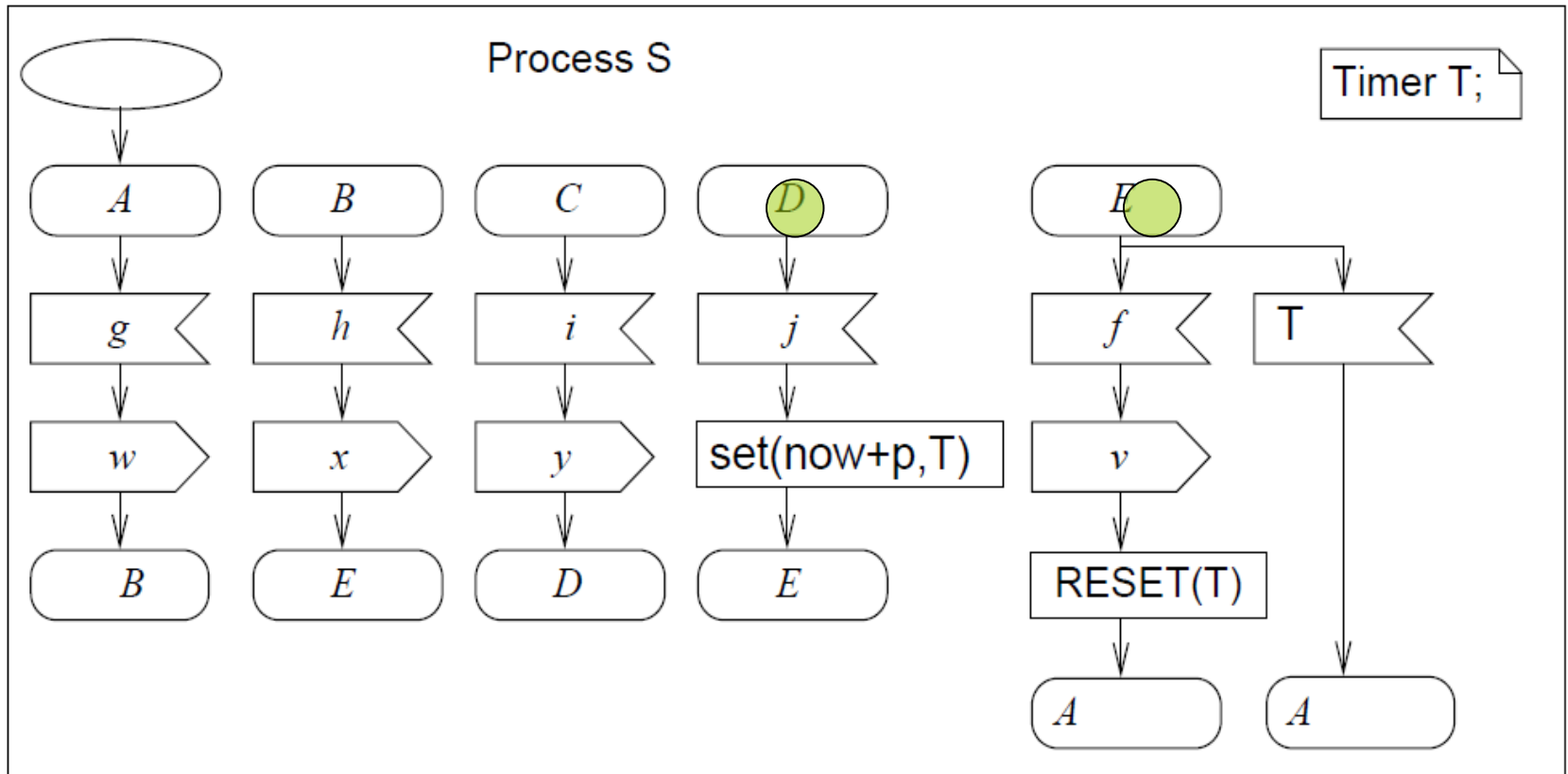


Processes cannot contain other processes, unlike in StateCharts.



Timers

- Timers can be declared locally
- Elapsed timers put signal into queue, but are not necessarily processed immediately; RESET removes a timer from FIFO-queue





SDL: Real World Example

- ADSL design
 - Ideal language for telecom design; communication between components and their different states of operation can be easily modeled with SDL
 - Object orientation and automatic code generation significantly simplified system design verification
 - Early testing done on SDL – significant savings in the cost of expensive test equipment



SDL summary

- FSM model for the components
- Non-blocking message passing for communication
- Implementation requires bound for the maximum length of FIFOs; may be very difficult to compute
- Not necessarily determinate
- Timer concept adequate just for soft deadlines
- Limited way of using hierarchies
- Limited programming language support
- No description of non-functional properties
- Excellent for distributed applications (used for ISDN)
- Commercial tools available (see <http://www.sdl-forum.org>)

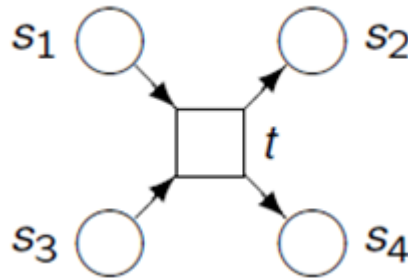




Models of Computation

- State Machine Models
 - FSM, StateCharts, SDL, CFSM
- Petri nets
- Communicating Processes
 - Kahn processes, Communicating Sequential Processes
- Ada
- Dataflow models
 - DFG, SDFG
- Discrete Event Systems
 - VHDL, Verilog, SystemC, SpecC

Petri net definitions

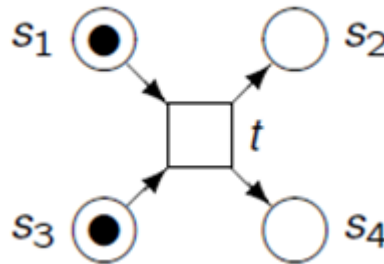
- Petri nets (PNs) are a basic model of parallel and distributed systems (named after Carl Adam Petri). The basic idea is to describe state changes in a system with transitions.



- Petri nets contain places  and transitions  that may be connected by directed arcs.
- Places symbolize **states, conditions, or resources** that need to be met/be available before an action can be carried out.
- Transitions symbolize **actions**.

Petri net definitions

- Places may contain **tokens** that may move to other places by executing (“firing”) actions.
- A **token** on a place means that the corresponding condition is fulfilled or that a resource is available:



- In the example, transition **t** may “**fire**” if there are tokens on places **s1** and **s3**. Firing **t** will remove those tokens and place new tokens on **s2** and **s4**.



Why Petri net?

- Parallelism and concurrency is modeled better in a Petri net than FSM.
- The composition of state machines is complex; composition of Petri net is simple.
- Petri nets are asynchronous in nature, however when synchronization is needed, that is also easy to model.

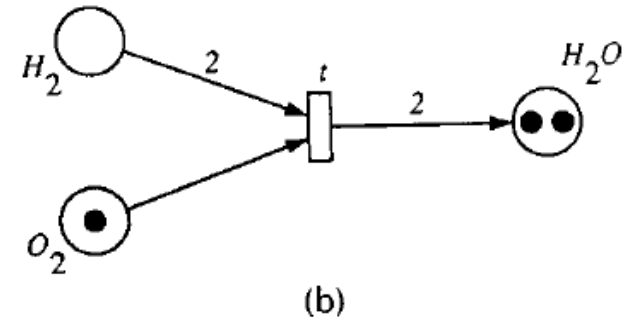
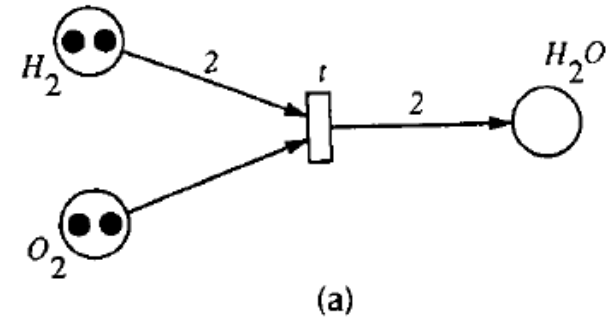
Petri net definitions

A Petri net is a 5-tuple, $PN = (P, T, F, W, M_0)$ where:

$P = \{p_1, p_2, \dots, p_m\}$ is a finite set of places,
 $T = \{t_1, t_2, \dots, t_n\}$ is a finite set of transitions,
 $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation),
 $W: F \rightarrow \{1, 2, 3, \dots\}$ is a weight function,
 $M_0: P \rightarrow \{0, 1, 2, 3, \dots\}$ is the initial marking,
 $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

A Petri net structure $N = (P, T, F, W)$ without any specific initial marking is denoted by N .

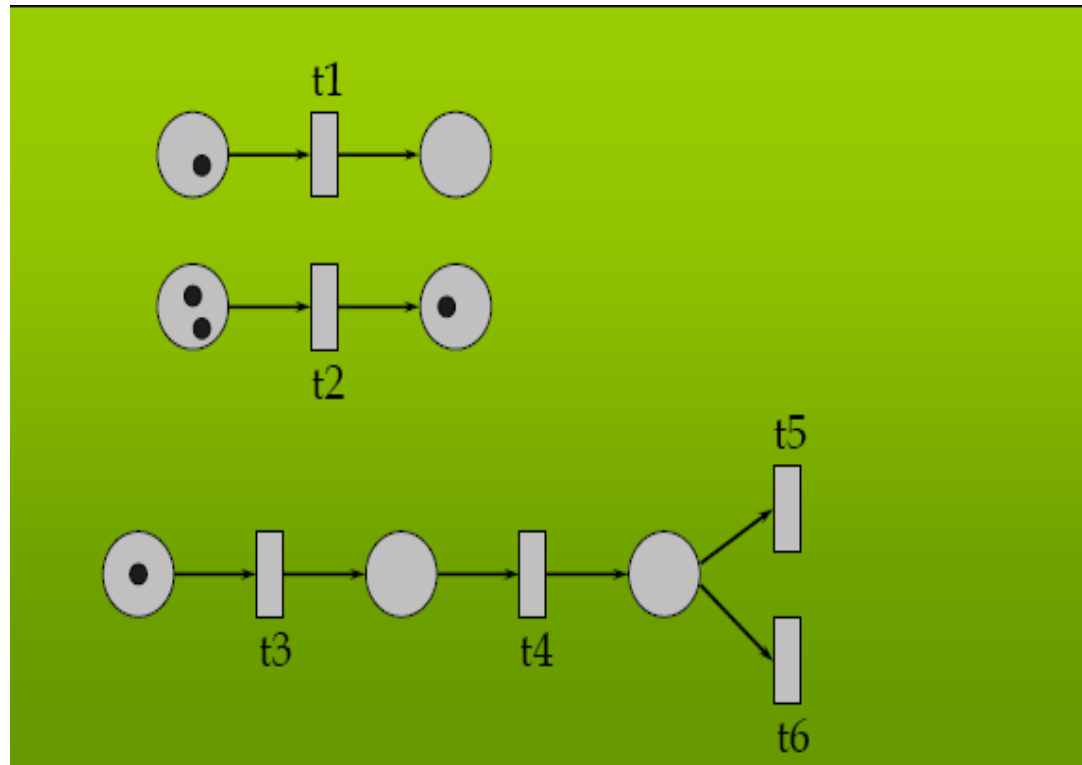
A Petri net with the given initial marking is denoted by (N, M_0) .



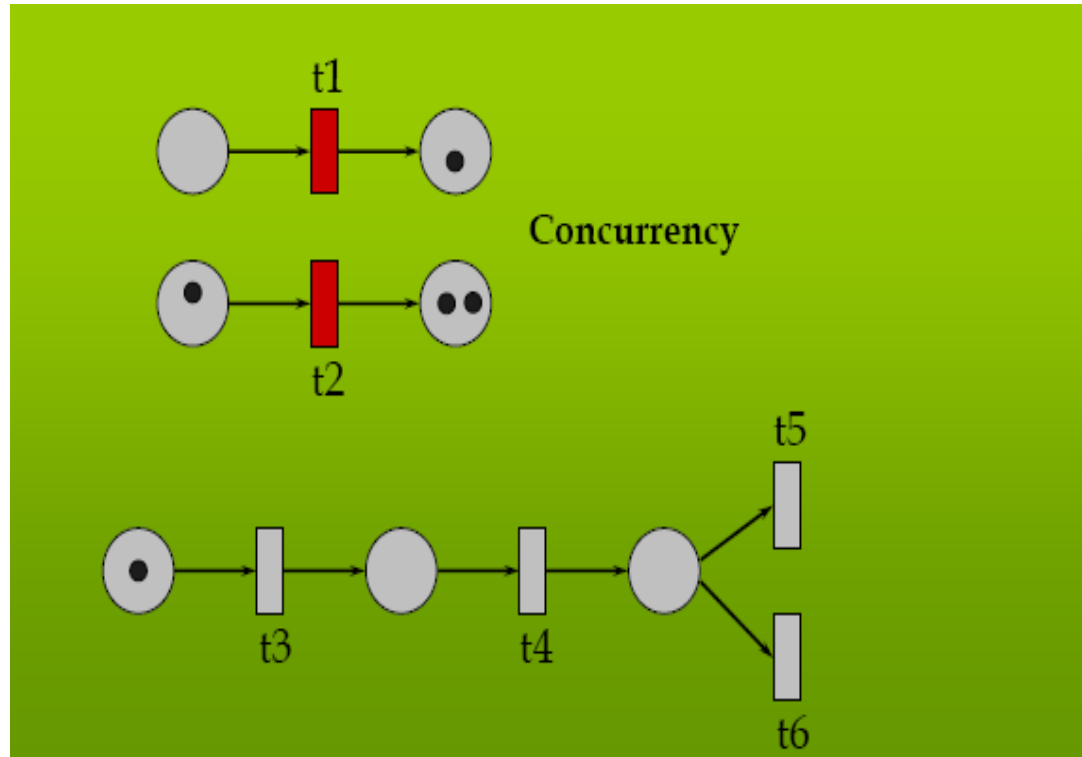
▪ H₂O Example

Input Places	Transition	Output Places
Preconditions	Event	Postconditions
Input data	Computation step	Output data
Input signals	Signal processor	Output signals
Resources needed	Task or job	Resources released
Conditions	Clause in logic	Conclusion(s)
Buffers	Processor	Buffers

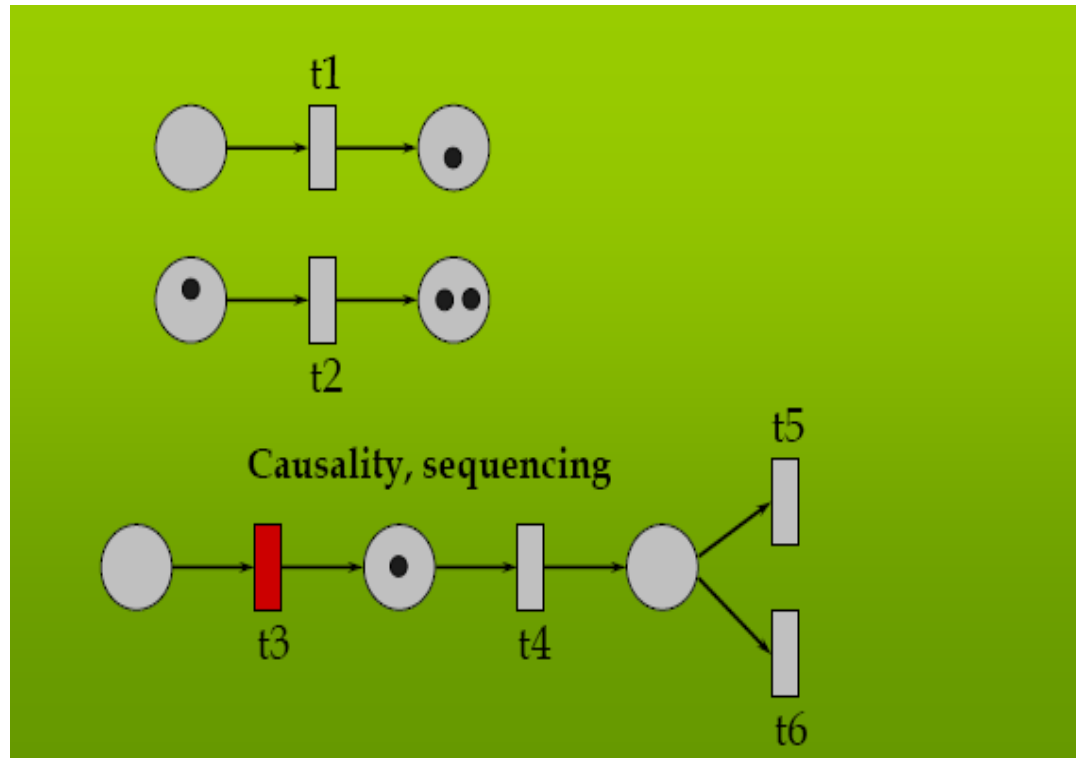
Concurrency, Causality, Choice



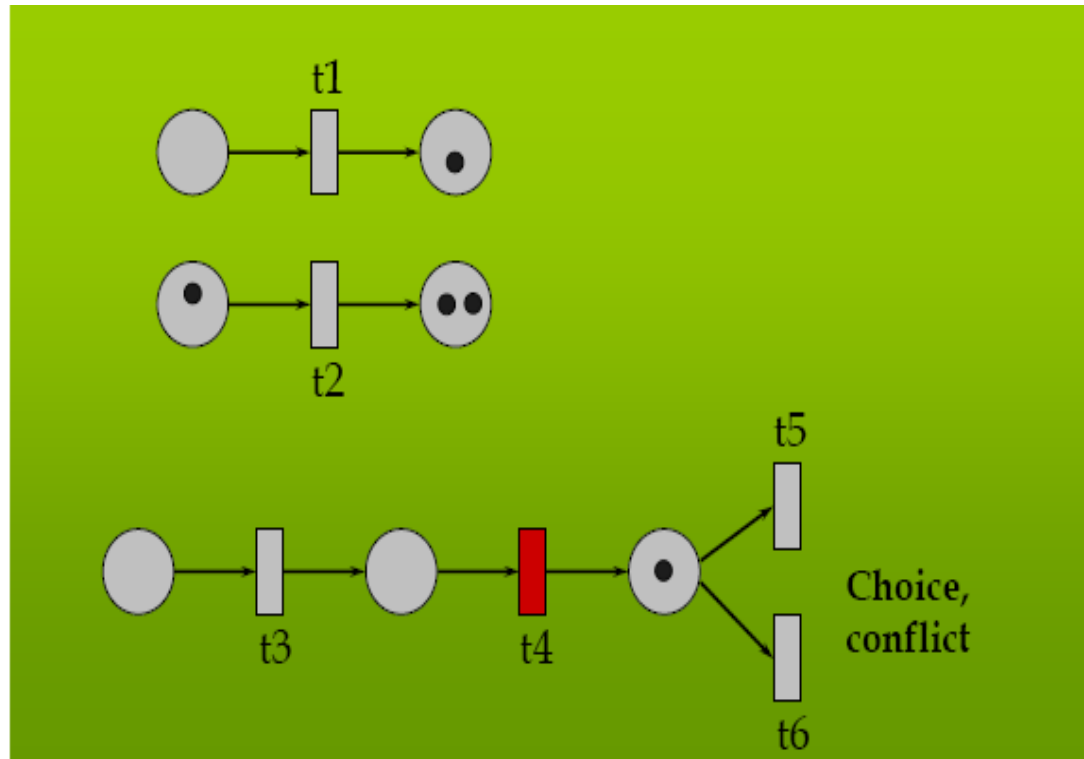
Concurrency, Causality, Choice



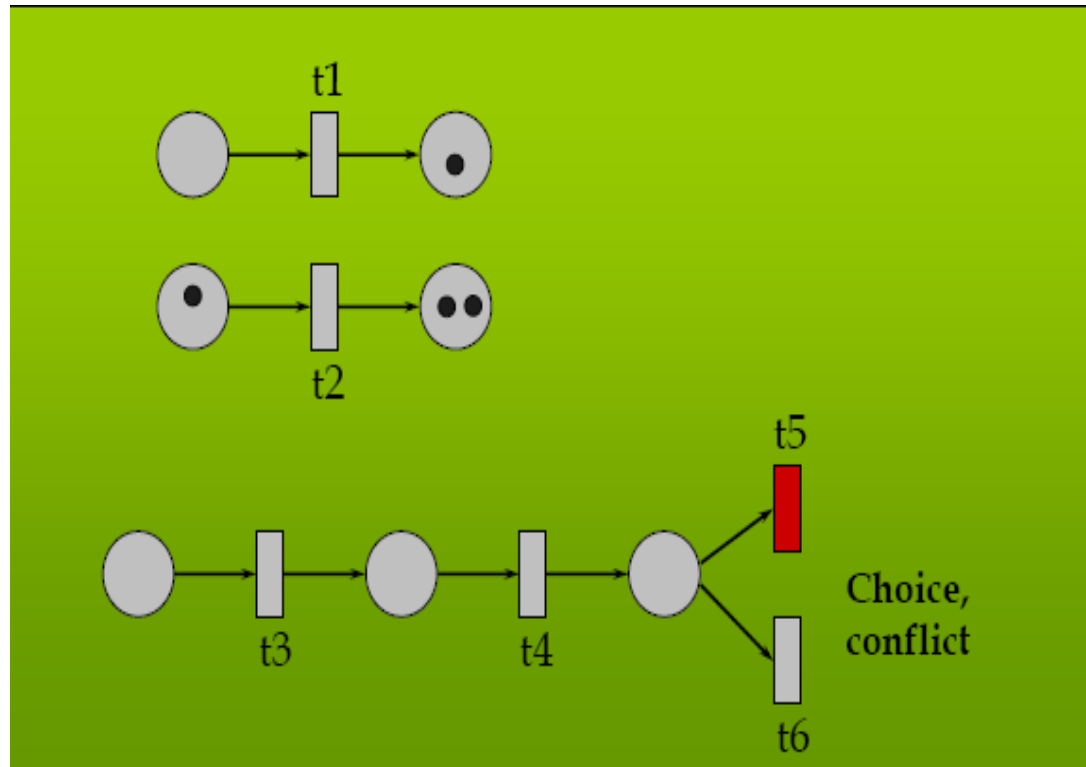
Concurrency, Causality, Choice

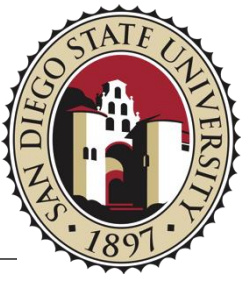


Concurrency, Causality, Choice



Concurrency, Causality, Choice

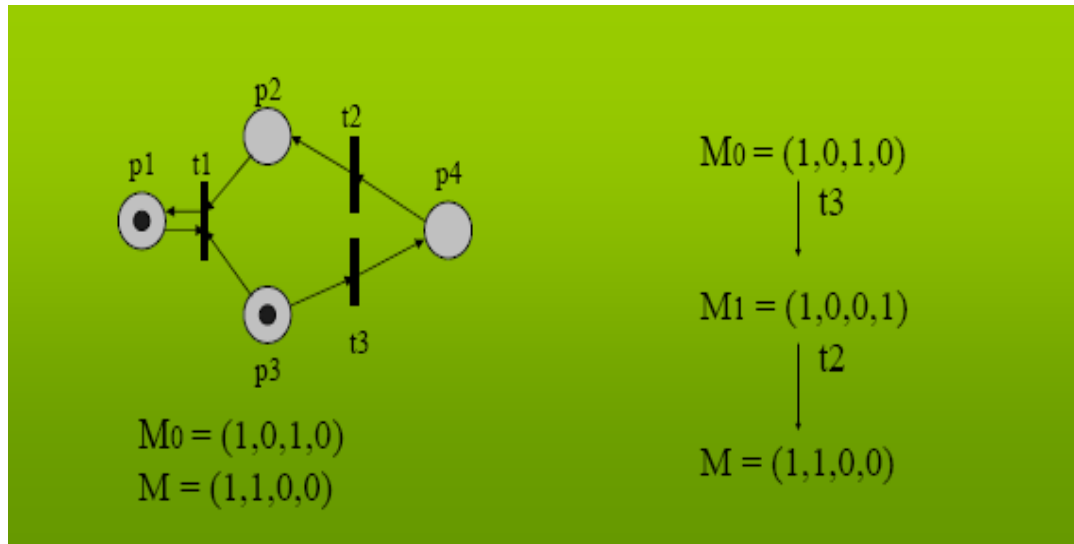




Petri Net Properties

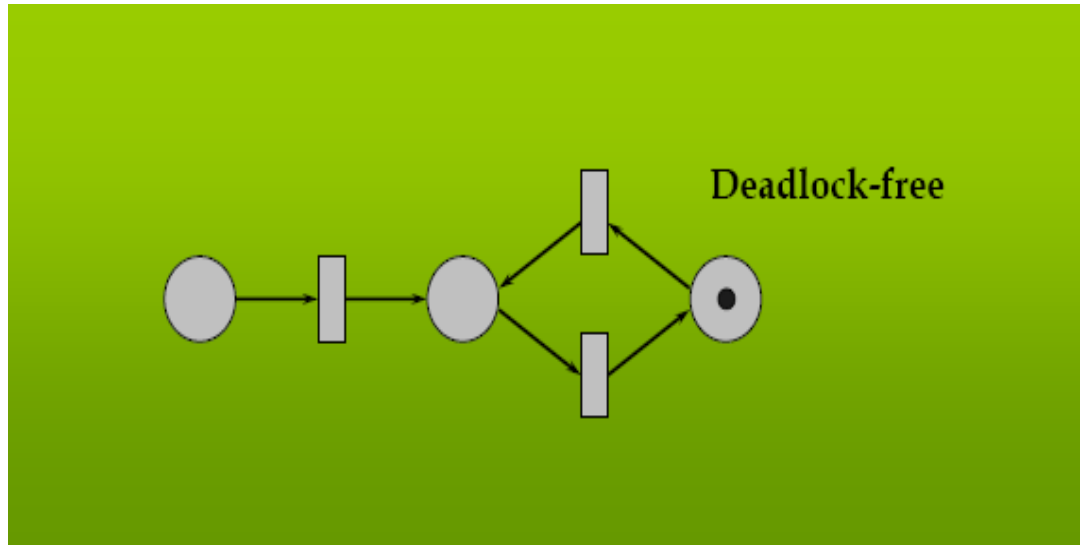
- Behavioral
 - Reachability
 - Marking M reachable from marking M_0
 - k - Boundedness
 - Number of tokens in each place does not exceed finite number k
 - Safe if 1-bounded
 - Liveness
 - A transition is live if it can never deadlock.
 - A transition is deadlocked if it can never fire.
- Structural
 - Controllability
 - Any marking can be reached from any other marking
 - Structural boundedness
 - Conservativeness – weighted sum of tokens constant

PN Properties - Reachability



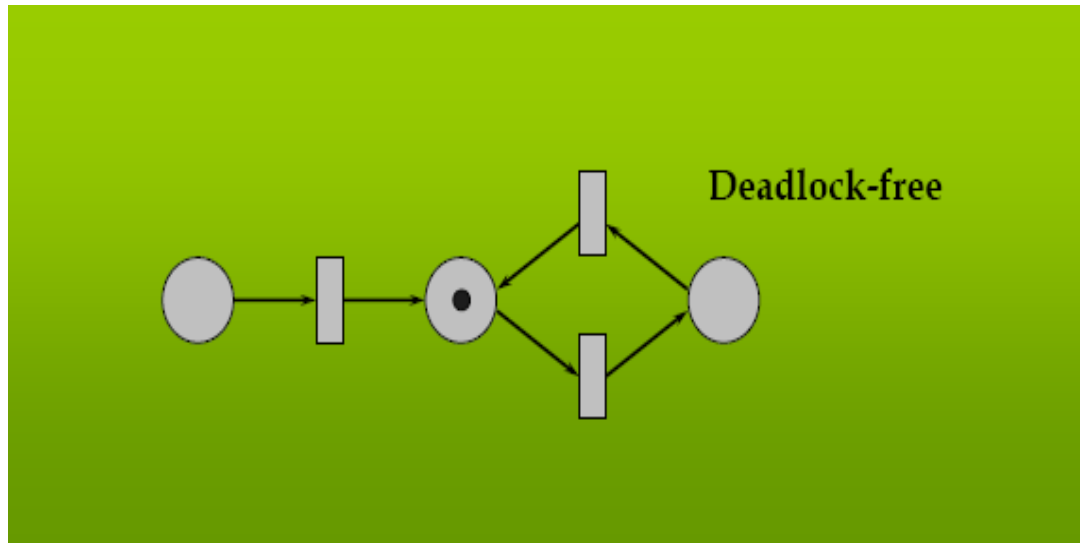
- Marking M is reachable from marking M_0 if there exists a sequence of firings $f = M_0$ to M_1 to ... that transforms M_0 into M

PN Properties – Deadlock-free



- A transition is live if it can never deadlock.
- A transition is deadlocked if it can never fire.

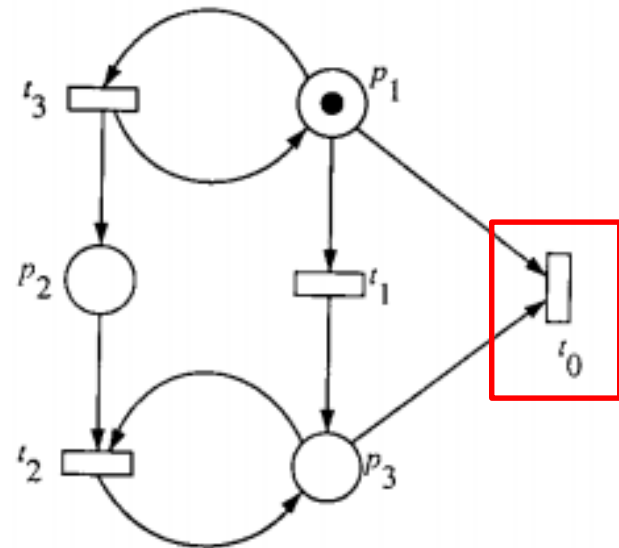
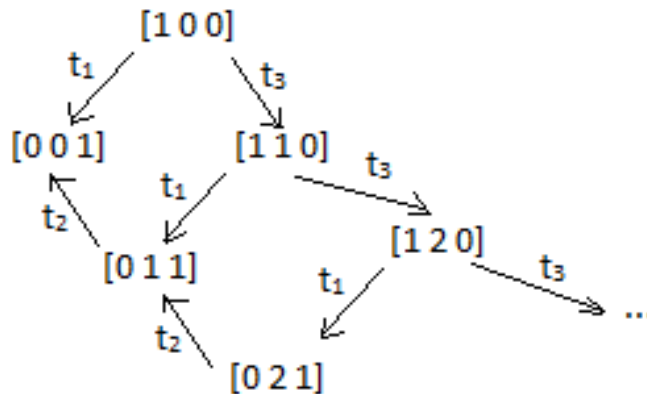
PN Properties – Deadlock-free



- A transition is live if it can never deadlock.
- A transition is deadlocked if it can never fire.

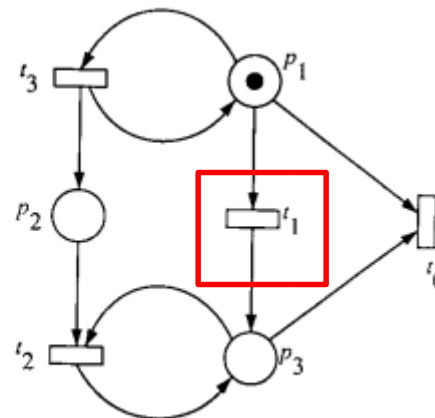
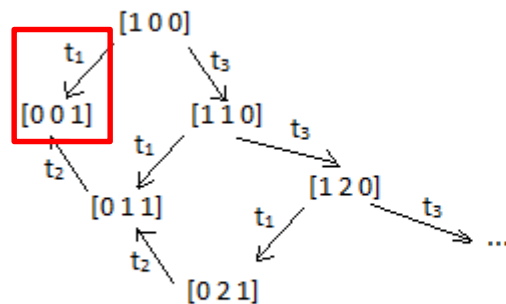
Petri Net Liveness

- L_0 -live (dead): a particular transition can never fire
 - In the figure below, t_0 can never fire
 - see reachability graph



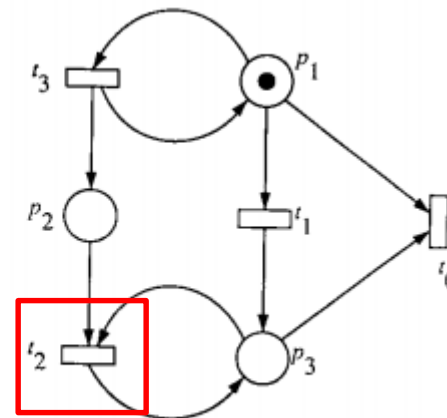
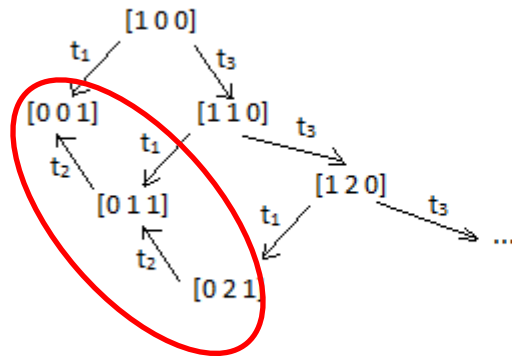
Petri Net Liveness

- **L₁-live:** a particular transition can fire at least once for some firing sequence
 - In the figure below, t_1 can only fire once, for some firing sequence.



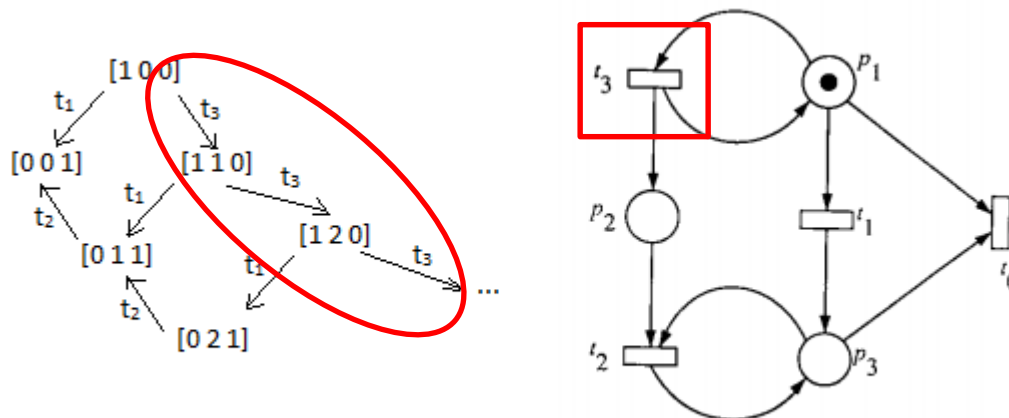
Petri Net Liveness

- **L2-live:** a particular transition can fire k times for a particular firing sequence, for any k .
 - In the figure below, t_2 can only fire once, twice, etc., for different firing sequences.

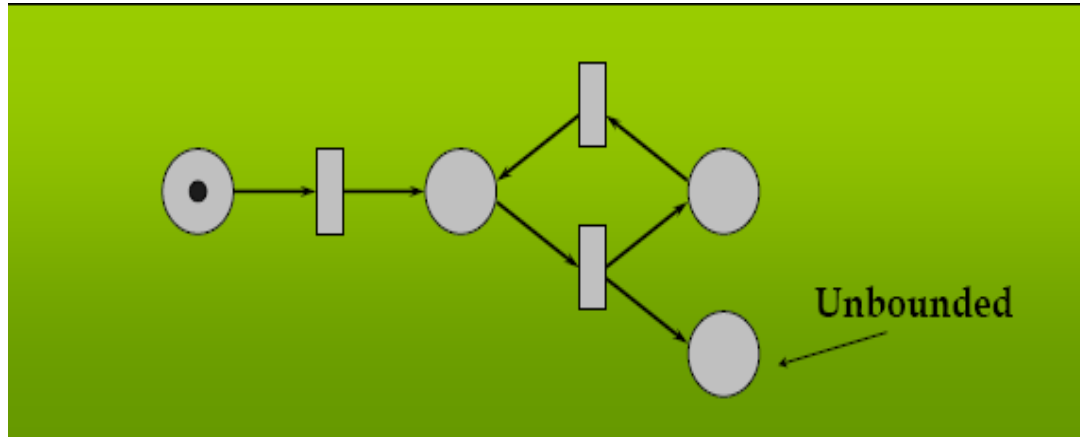


Petri Net Liveness

- **L₃-live:** a particular transition can fire infinitely in a particular firing sequence.
 - In the figure below, t_3 can fire infinitely for the firing sequence $t_3, t_3, t_3, t_3, \dots$
 - Note that the number of times t_1 and t_2 fire is finite for any firing sequence.



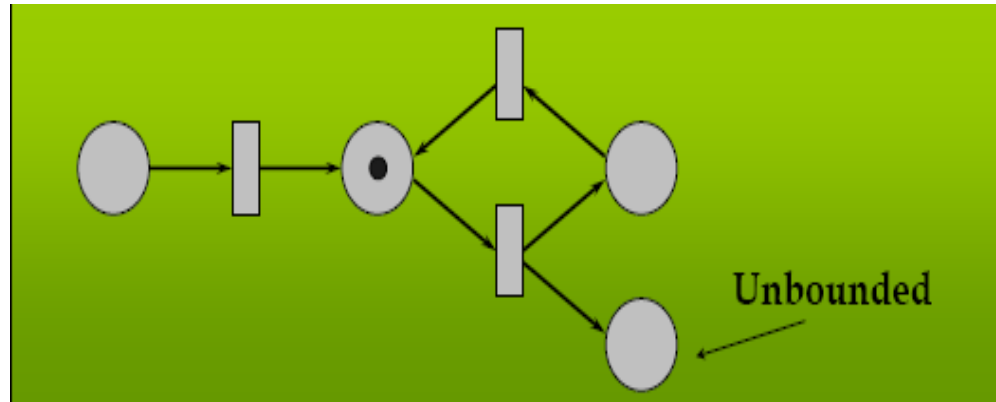
PN Properties - Boundedness



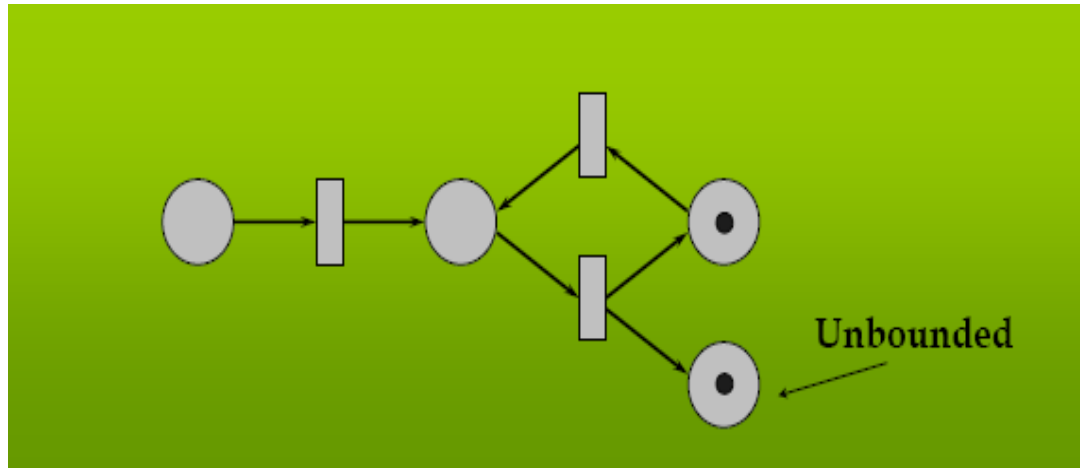
The number of tokens in any place can't grow indefinitely

- 1 – bounded is also called safe
- Places represent buffers and registers so we can check for overflow

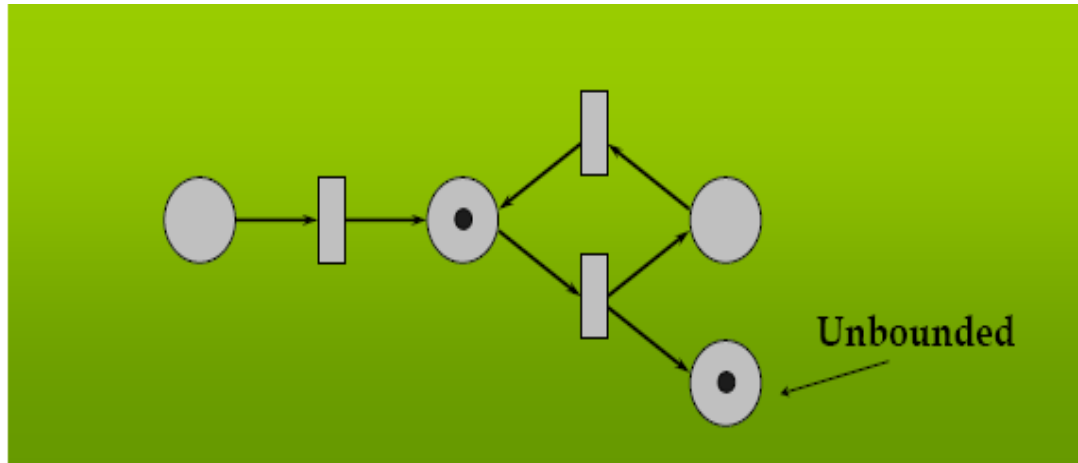
PN Properties - Boundedness



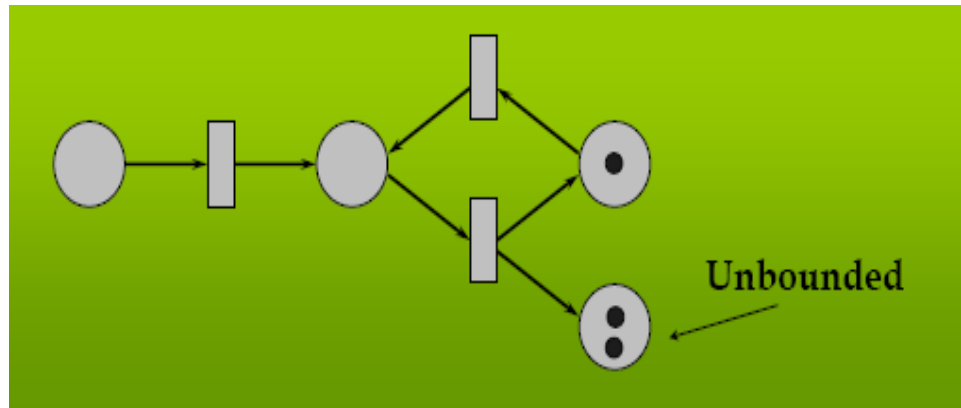
PN Properties - Boundedness



PN Properties - Boundedness



PN Properties - Boundedness



PN Properties - Purenness

- A PN is called pure if it does not have any self loops, i.e., there exists no such **place** in the net, which is simultaneously an input place and an output place to a transition.

Example 2.3:

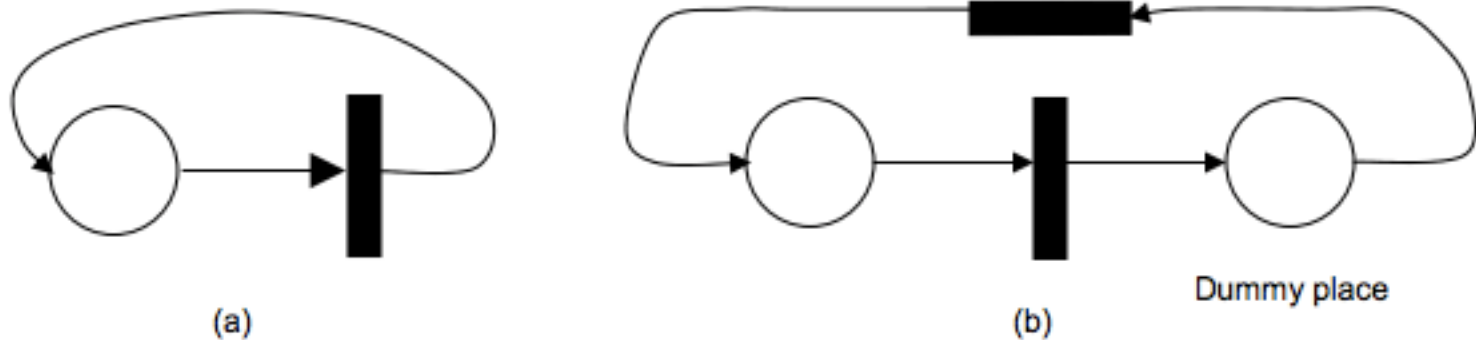


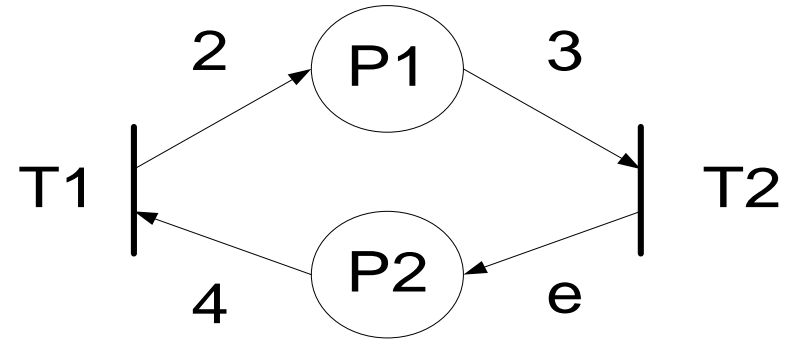
Figure 2.3: (a) Impure Petri net, (b) Pure Petri net

Figure from: Venkateswarlu et al. A Study of Petri Nets



Example

- Assume
 - $e=6$
 - $M_0 = [0 \ 12]$
- Can we reach $M=[6 \ 0]$ from M_0 ?
- What size buffers are needed at P_1 & P_2 ?

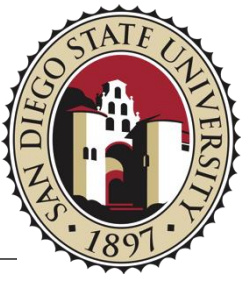




Petri net problem

Draw a Petri net that describes a game in which there is an urn with black and red balls. In each round the player takes two balls:

- If both are black, then the player returns one black
- If both are red, then the player returns one red
- If one is red and one is black, then the player returns one red.
- How to make this design a conservative net (i.e. total # of tokens is constant)?



Petri nets: Applications

- Model, simulate and analyze networking protocols (e.g. TCP, Ethernet, etc)
- Model, simulate and analyze complex network elements (e.g. router, switch, optical mux); check their logical behavior
- Design and analyze network performance and AoS with logical models of traffic generators, protocols and network elements
- Study network behavior characteristics (e.g. throughput, blocking probability etc)

Petri nets in practice



- Example apps:
 - TCP performance
 - Security system design and automated code generation
 - MAC design
 -





Petri Nets - Summary

- PN Graph
 - places (buffers), transitions (action), tokens (data)
- Firing rule
 - Transition enabled if enough tokens in a place
- Properties
 - Structural (consistency, structural boundedness)
 - Behavioral (reachability, boundedness, etc.)
- Applications
 - Modeling of resources, mutual exclusion, synchronization



Models of Computation

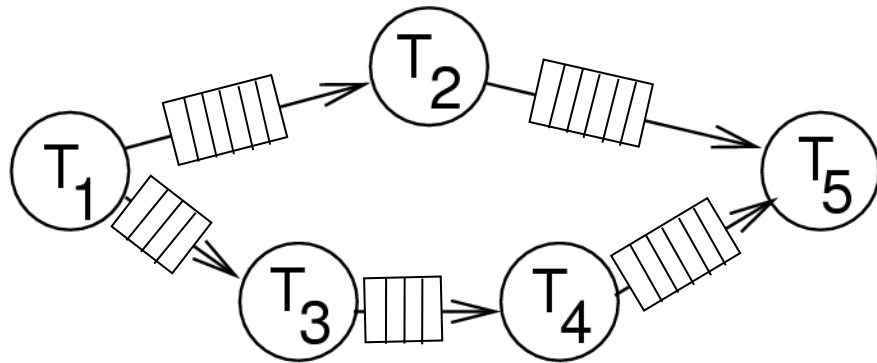
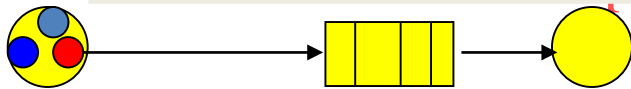
- State Machine Models
 - FSM, StateCharts, SDL, CFSM
- Petri nets
- **Communicating Processes**
 - **Kahn processes**, Communicating Sequential Processes
- Ada
- Dataflow models
 - DFG, SDFG
- Discrete Event Systems
 - VHDL, Verilog, SystemC, SpecC

Kahn process network (KPN)

KPN - distributed model of computation.

A group of deterministic sequential processes are communicating through unbounded FIFO channels

Non-blocking write, blocking read => **DETERMINE**



Important properties of KPN:

Continuous

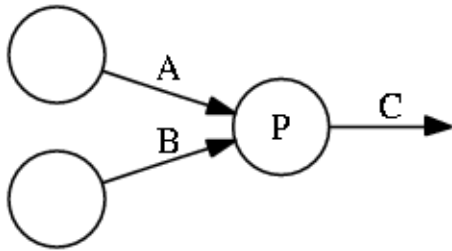
Output signals can be gradually produced; never have to consume all input to produce some output

Monotonic

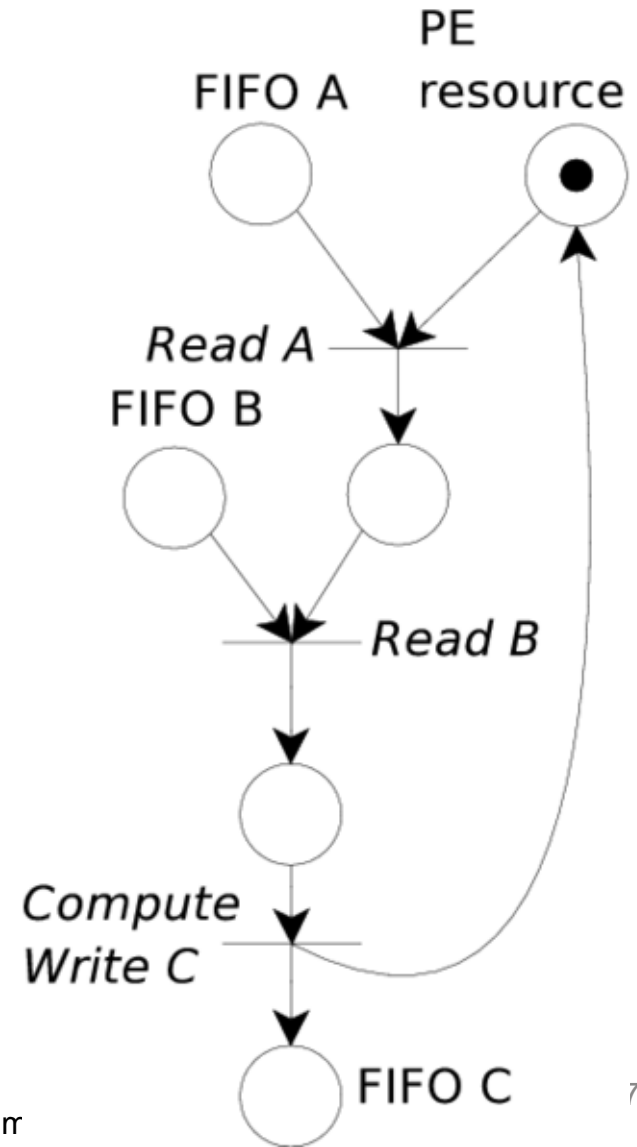
Output depends on input but doesn't change previous output

Continuous monotonic processes => **ITERATIVE**

Petri net model of a Kahn process



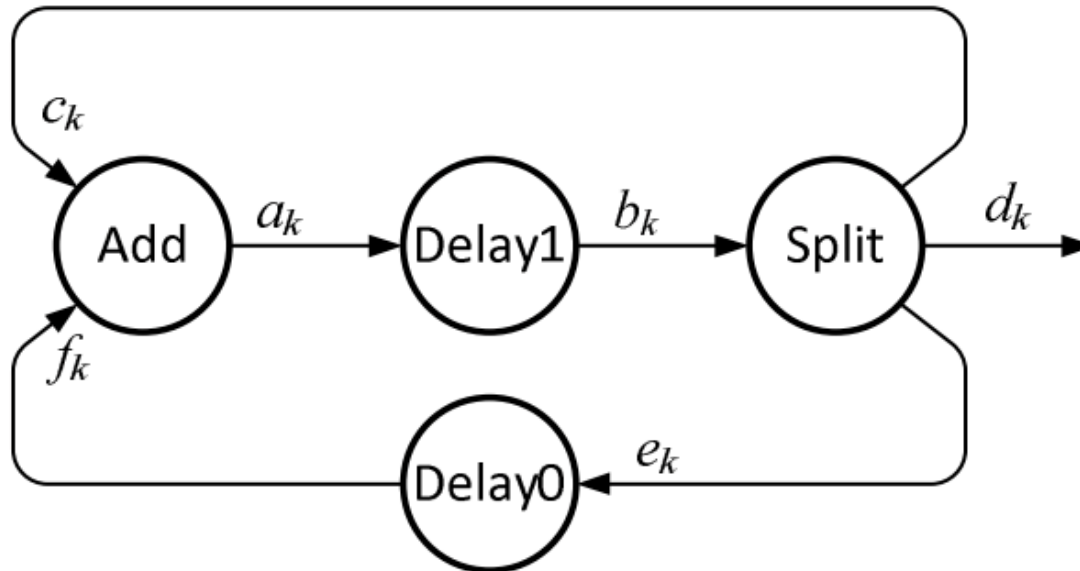
- KPNs are deterministic:
 - Output determined by
 - Process, network, initial tokens

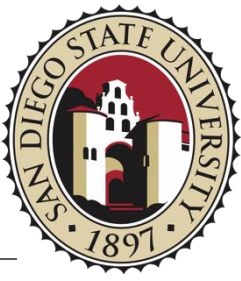




KPN Example: Fibonacci Numbers

- Fibonacci numbers: $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 1$ and $F_1 = 1$
- Initially, Delay0 and Delay1 write 0 and 1 to f_k and b_k respectively
- What makes it possible for this to work?





KPN Properties

- KPNs have to be deterministic
 - For a certain sequence of inputs, there is only one possible sequence of outputs (regardless, for example, how long time it takes for a certain computation or communication to finish)
 - Looking only at the specification (and not knowing anything about implementation) you can exactly derive the output sequence corresponding to a certain input sequence



Scheduling KPNs

- Kahn process networks are dynamic dataflow models: their behavior is data dependent; depending on the input data one or the other process is activated
- Kahn process networks cannot be scheduled statically
It is not possible to derive, at compile time, a sequence of process activations such that the system does not block under any circumstances
 - Kahn process networks have to be scheduled dynamically
which process to activate at a certain moment has to be decided, during execution time, based on the current situation
 - There is an overhead in implementing Kahn process networks



Other Problems

- Another problem: memory overhead with buffers. Potentially, it is possible that the memory need for buffers grows unlimited
- Kahn process networks are strong in their expressive power but sometimes cannot be implemented efficiently
 - Introduce limitations so that you can get efficient implementations

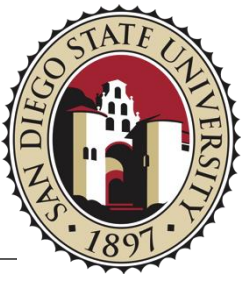


Models and Languages Comparison

Language	Behavioral Hierarchy	Structural Hierarchy	Programming Language Elements	Exceptions Supported	Dynamic Process Creation
StateCharts	+	-	-	+	-
VHDL	+	+	+	-	-
SpecCharts	+	-	+	+	-
SDL	+ -	+ -	+ -	-	+
Petri nets	-	-	-	-	+
Java	+	-	+	+	+
SpecC	+	+	+	+	+
SystemC	+	+	+	+	+
ADA	+	-	+	+	+

- A single Model of Computation cannot provide every functionality
 - There are other requirements than the ones listed above

Models and Languages Summary



- Multiple models and languages are essential for high-level design
 - Managing complexity by abstraction
 - Formality ensures refinement correctness
 - Model choice depends on
 - Class of applications
 - Required operations (synthesis, scheduling, ...)
- Multiple MOCs can co-exist during all phases of design
 - Specification
 - Architectural mapping and simulation
 - Synthesis, code generation, scheduling
 - Detailed design and implementation
 - Co-simulation