McGill

ECSE 421 Lecture 3: Communicating Finite State Machines

ESD Chapter 2

Last Time

- Requirements of Specification Languages
- Introduction to Modeling
- Introduction to Models of Computation
  - Why not von Neumann?
  - Representing execution
  - Communication
  - Organizing execution
- Supporting early design phases
  - (Message) Sequence Charts

Where Are We?

W	D	Date		Topic	ESD	PES	Out	In	Notes
1	T	12-Jan-2016	L01	Introduction to Embedded System Design	1.1-1.4				
	R	14-Jan-2016	L02	Specifying Requirements / MoCs / MSC	2.1-2.3				
2	T	19-Jan-2016	L03	CFSMs	2.4				
	R	21-Jan-2016	L04	Data Flow Modeling	2.5	3.1-5,7			
3	T	26-Jan-2016	L05	Petri Nets	2.6				
	R	28-Jan-2016	L06	Discrete Event Models	2.7	4			
4	T	2-Feb-2016	L07	DES / Von Neumann Model of Computation	2.8-2.10	5			Guest lecturer
	R	4-Feb-2016	L08	Sensors	3.1-3.2	7.3,12.1-6			
5	T	9-Feb-2016	L09	Processing Elements	3.3	12.6-12			
	R	11-Feb-2016		Processing Elements	3.3				
6	T	16-Feb-2016	L10	More Processing Elements / FPGAs					
	R	18-Feb-2016	L11	Memories, Communication, Output	3.4-3.6				
7	T	23-Feb-2016	L12	Embedded Operating Systems	4.1				
	R	25-Feb-2016		***Midterm exam: in-class, closed book***					Chapters 1-3
8	T	1-Mar-2016		***No class***					Winter break
	R	3-Mar-2016		***No class***					Winter break
	T	8-Mar-2016	L13	Middleware	4.4-4.5				
	R	10-Mar-2016	L14	Performance Evaluation	5.1-5.2				
9	T	15-Mar-2016	L15	More Evaluation and Validation	5.3-5.8				
	R	17-Mar-2016	L16	Introduction to Scheduling	6.1-6.2.2				
10	T	22-Mar-2016	L17	Scheduling Aperiodic Tasks	6.2.3-6.2.4				
	R	24-Mar-2016	L18	Scheduling Periodic Tasks	6.2.5-6.2.6				

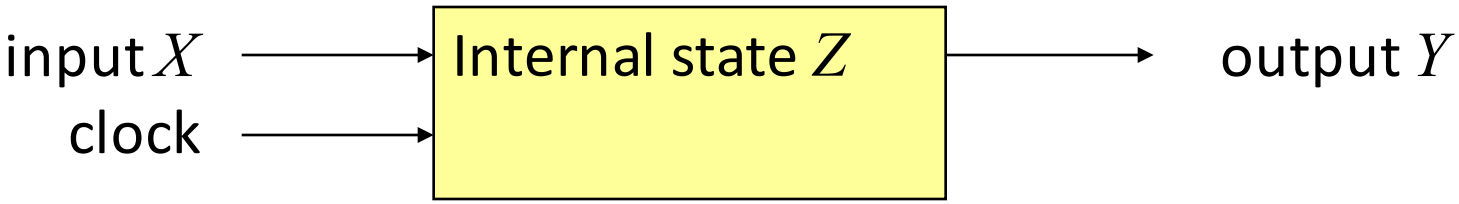
Today

- Communicating Finite State Machines
  - StateCharts
  - Synchronous languages
  - SDL

MoCs Considered in 421

Communication/ local computations	Shared memory	Message passing Synchronous	Asynchronous
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	**StateCharts**		SDL
Data flow	(Not useful)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

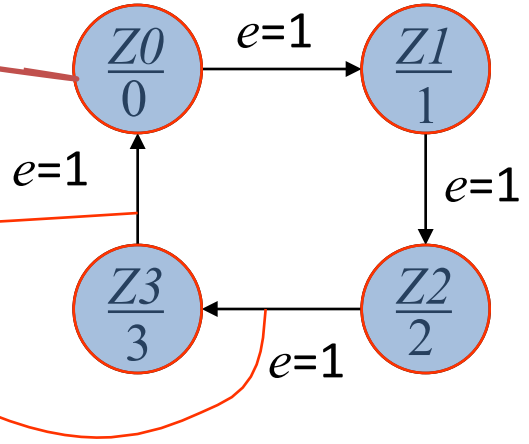
Review of Classical Automata



Next state Z^+ computed by function δ
Output computed by function λ

Moore- + Mealy-automata = finite state machines (FSMs)

- Moore-automata
  $Y = \lambda\ (Z);\quad Z^+ = \delta\ (X, Z)$
- Mealy-automata
  $Y = \lambda\ (X, Z);\quad Z^+ = \delta\ (X, Z)$

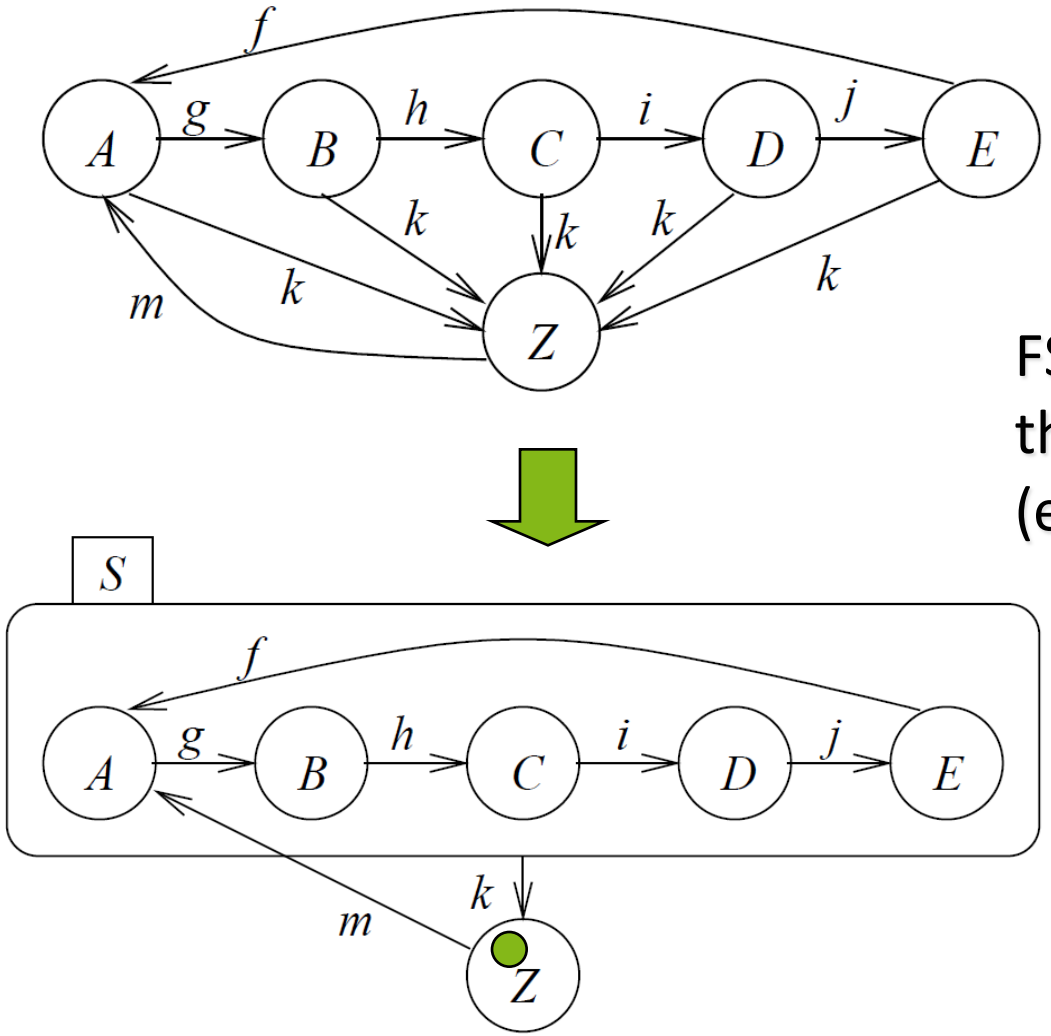


StateCharts

- Classical automata are not useful for complex systems
  - Complex graphs cannot be understood by humans
- StateCharts [Harel, 1987]
  - The only unused combination of “flow” or “state” with “diagram” or “chart”
  - Introduction of hierarchy

Used here as a (prominent) example of a model of computation based on shared memory communication. Appropriate only for local (non-distributed) systems!

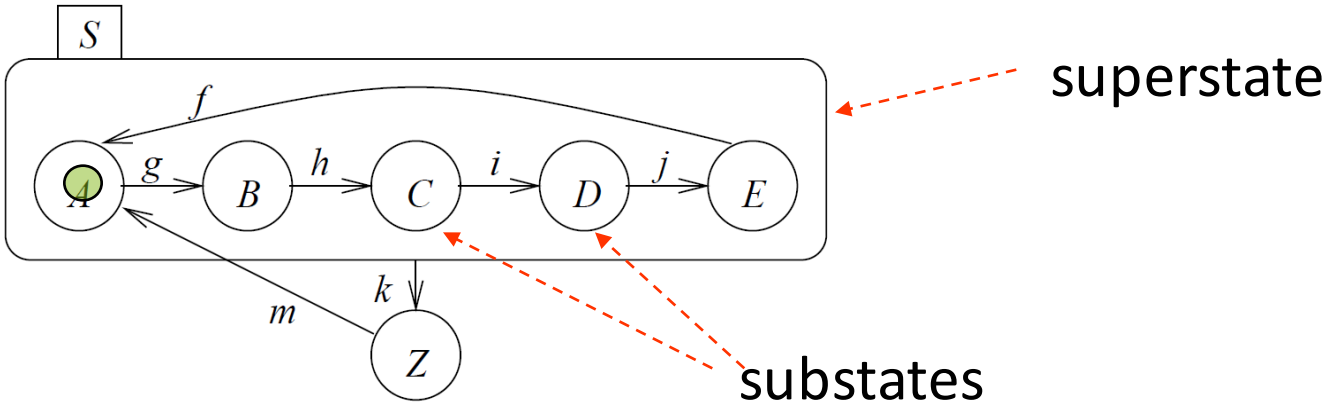
Introducing Hierarchy



FSM will be **in** exactly one of the substates of S if S is **active** (either in A or in B or ..)

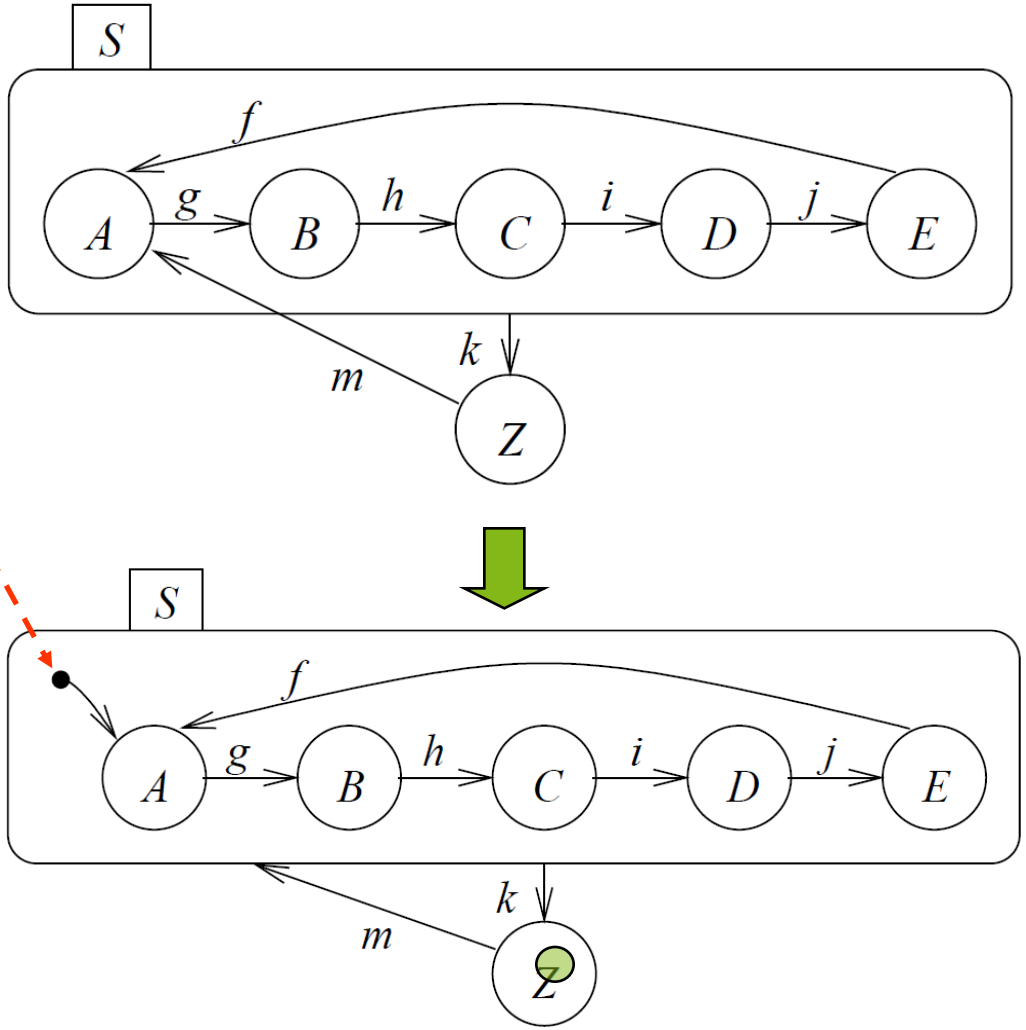
Definitions

- *Active states*: current states of FSMs
- *Basic states*: states not composed of other states
- *Super-states*: states containing other states
  - A super-state *S* is an *OR-super-state* if exactly one of the sub-states of *S* is active whenever *S* is active

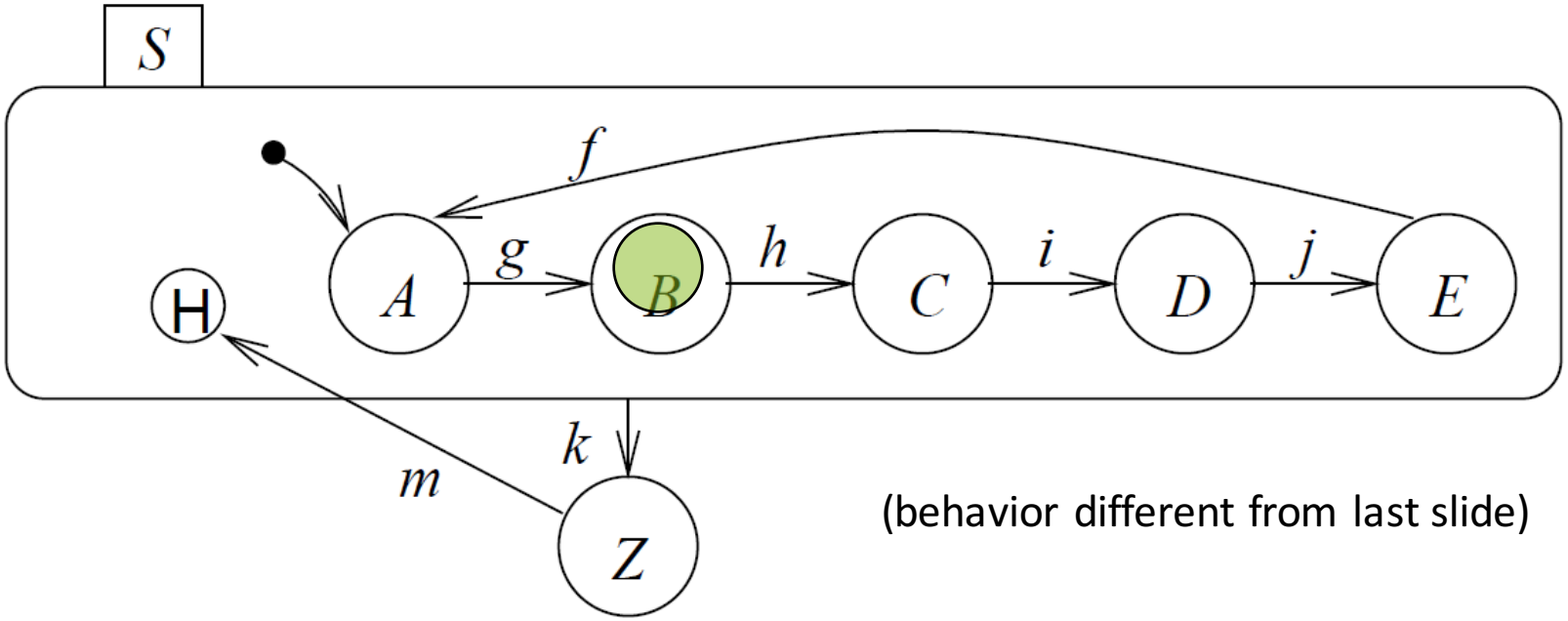


Default State Mechanism

- Default state hides internal structure from outside world
  - Indicated with a filled circle
  - Marks the sub-state entered whenever super-state is entered
- Default state is not a state itself!



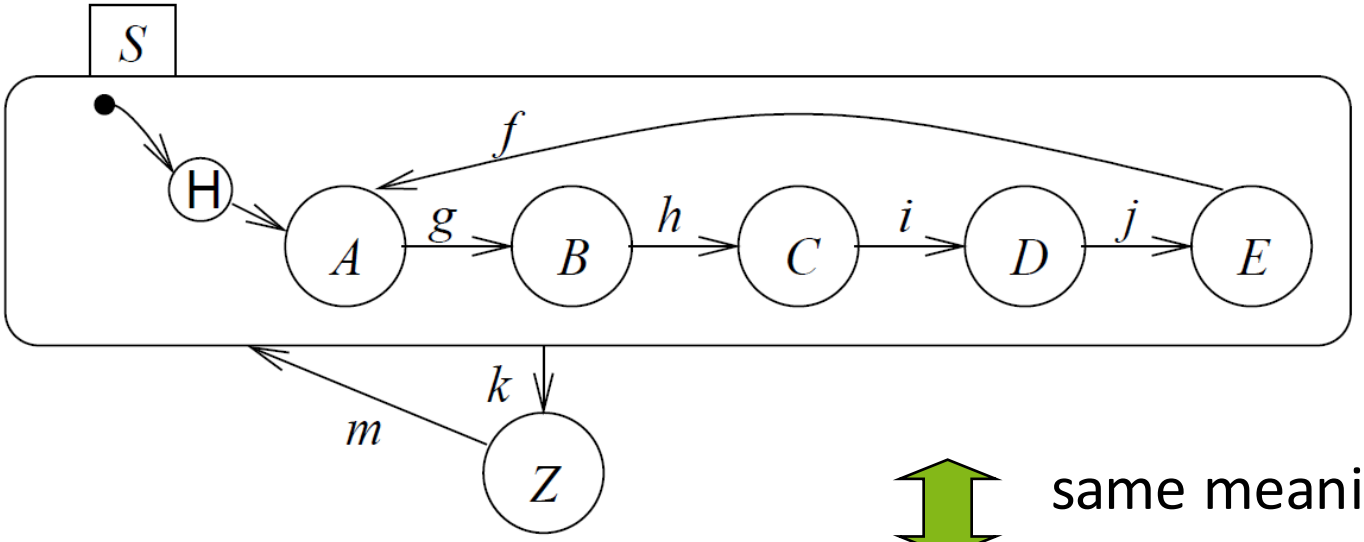
History Mechanism



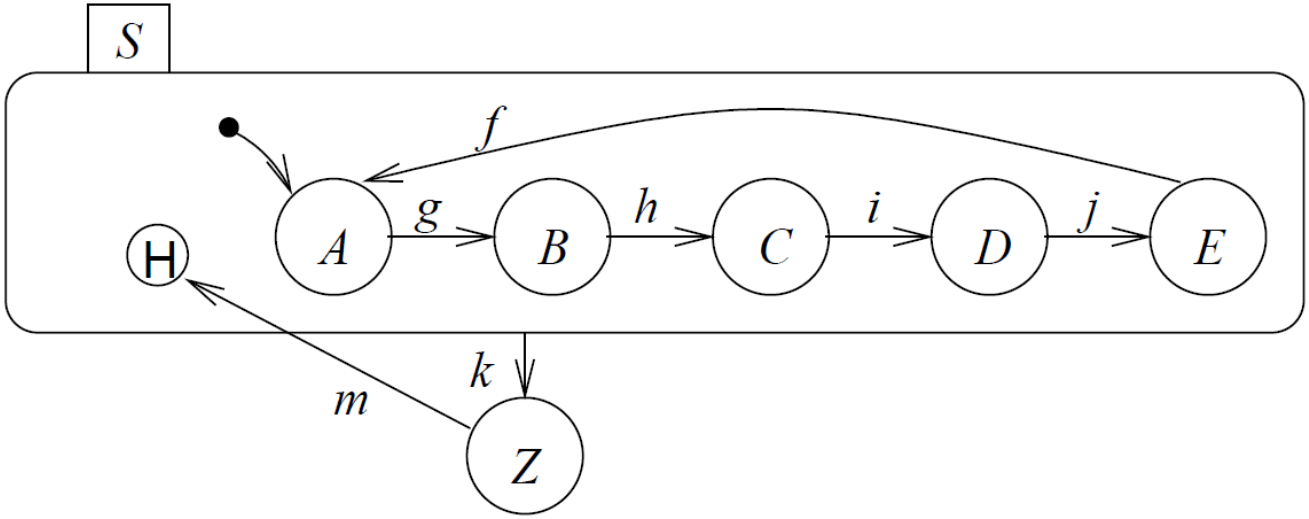
(behavior different from last slide)

- 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 first time, the default mechanism applies
- History and default mechanisms can be used hierarchically

History and Default State Mechanisms

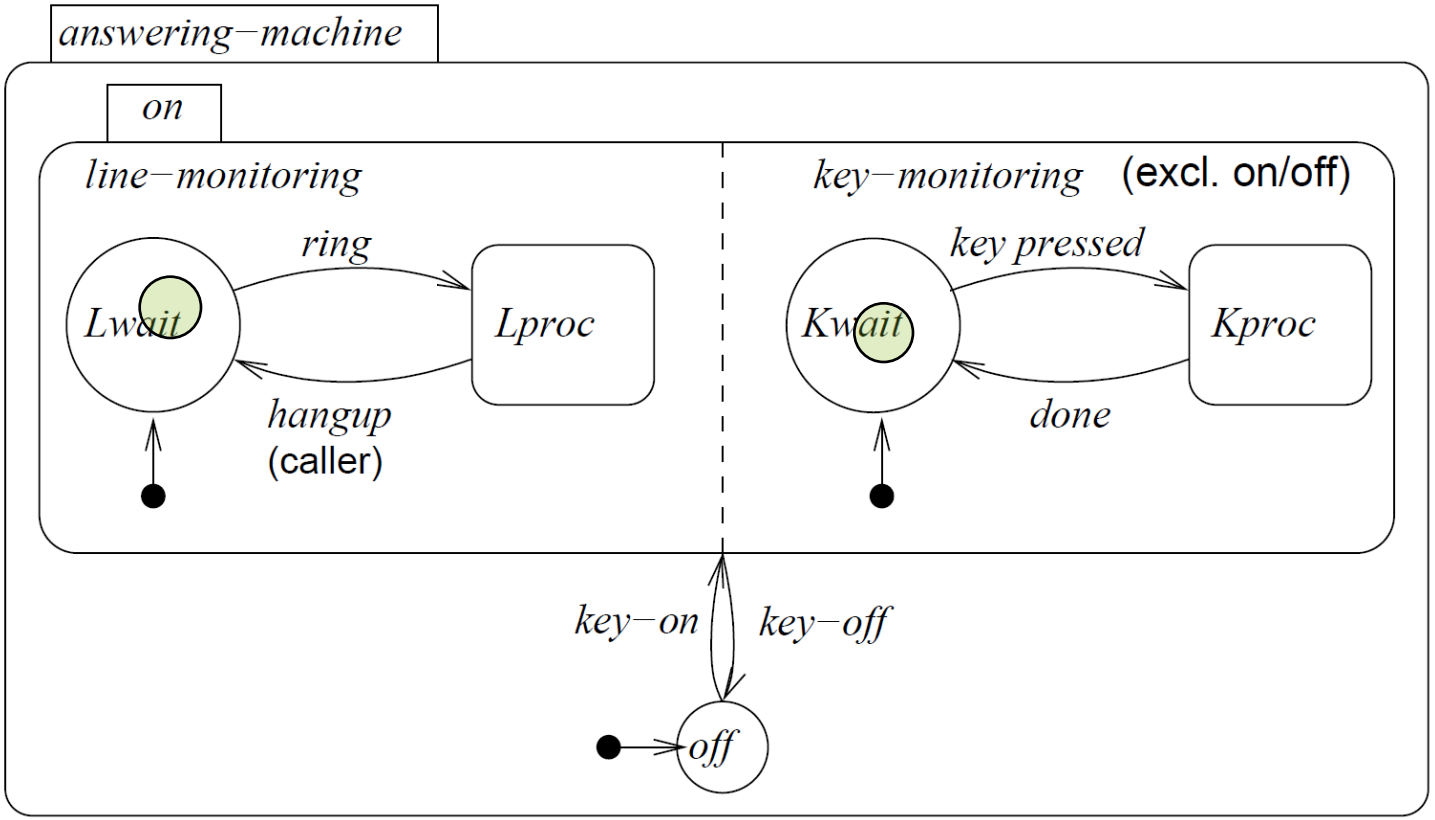


same meaning



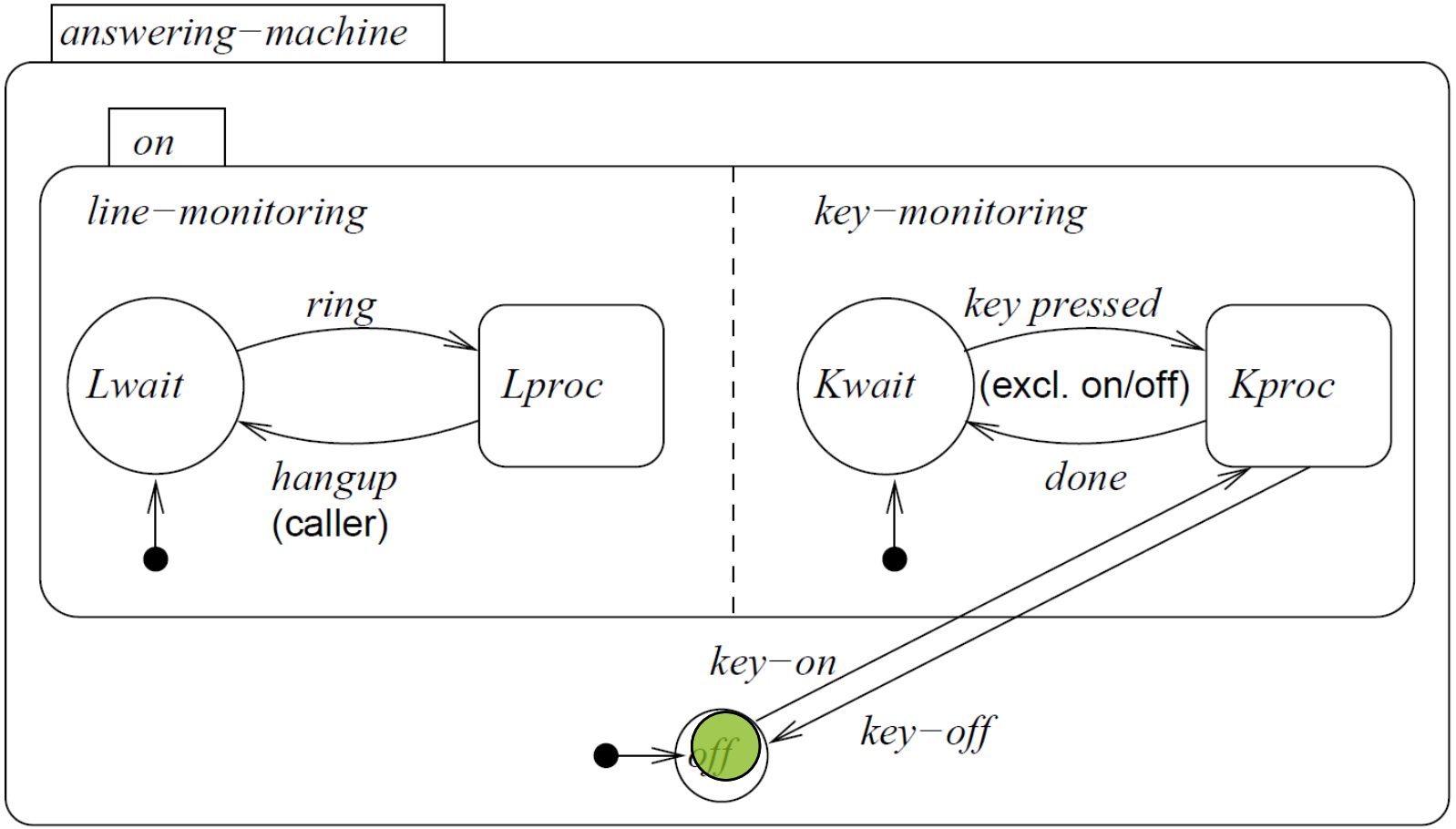
Concurrency

- *AND-super-states*: convenient ways of describing concurrency requirements
  - FSM is in all (immediate) sub-states of a super-state;
- Example:

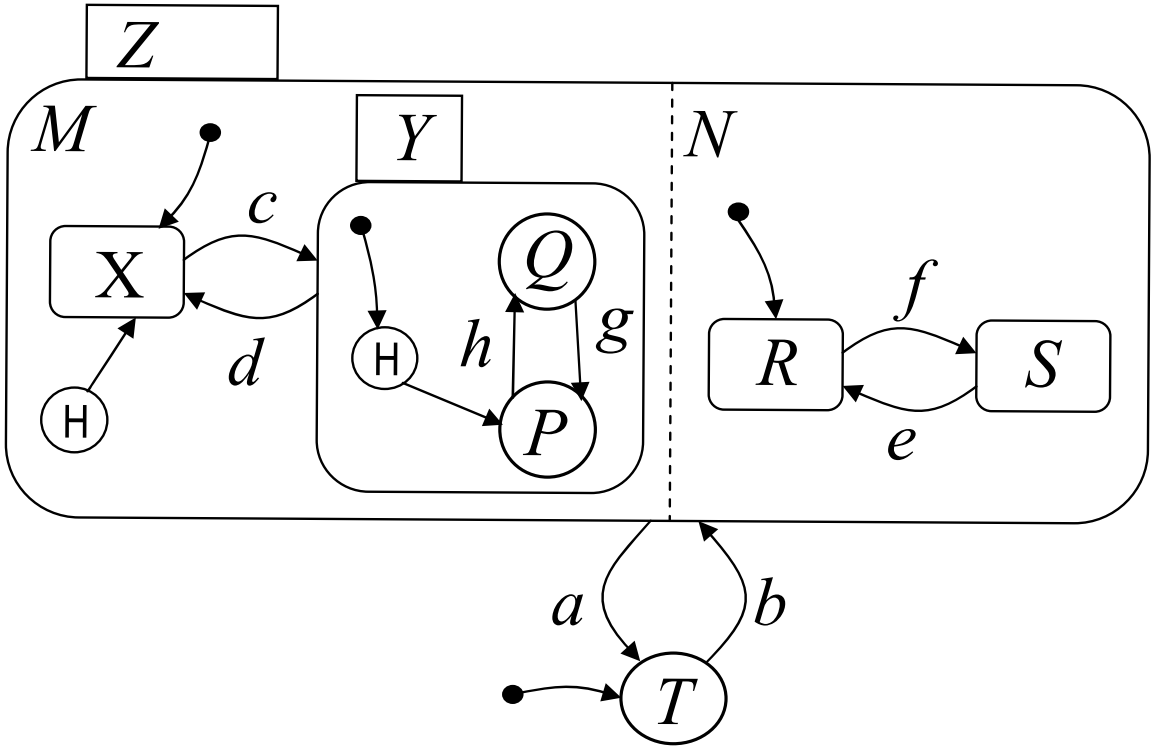


Entering and leaving AND-super-states

- Line-monitoring and key-monitoring are entered and left, when service switch is operated



StateCharts Example

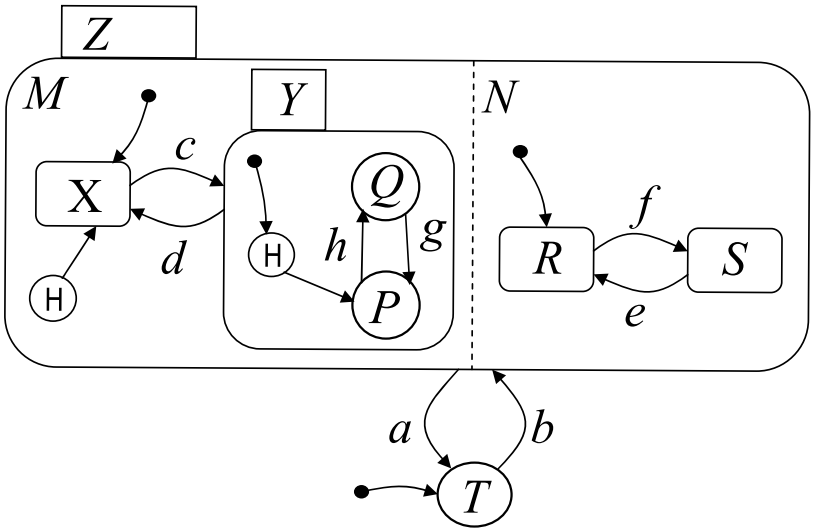


- Given the sequence: *RESET b c f h g h*
- What states are occupied after each event?

StateCharts Example, Cont'd

Event	M	N	P	Q	R	S	T	X	Y	Z
RESET							X			
b										
c										
f										
h										
g										
h										

- Multiple states can be active at once
- A subset of exercise 2.3

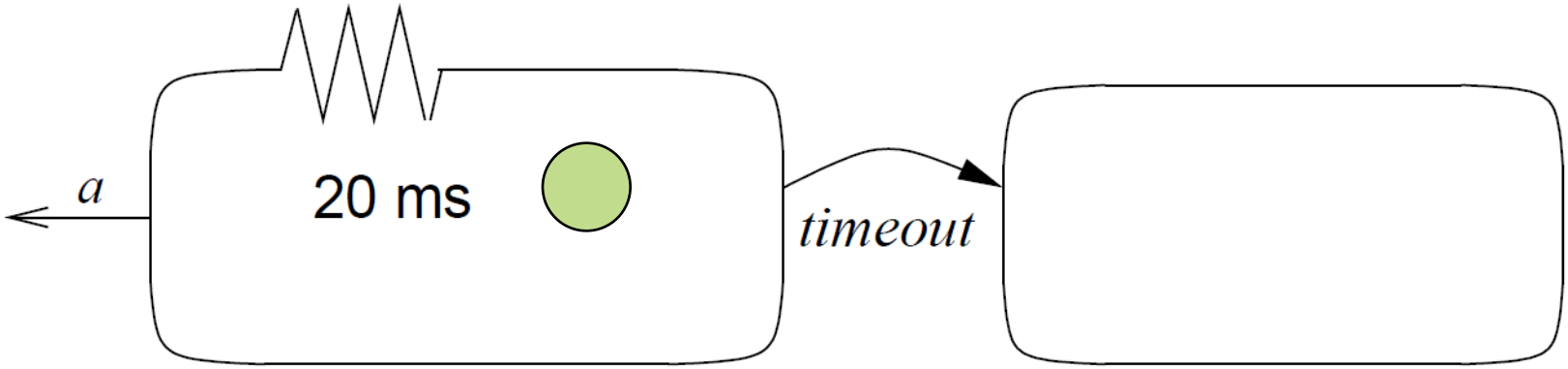


StateCharts So Far

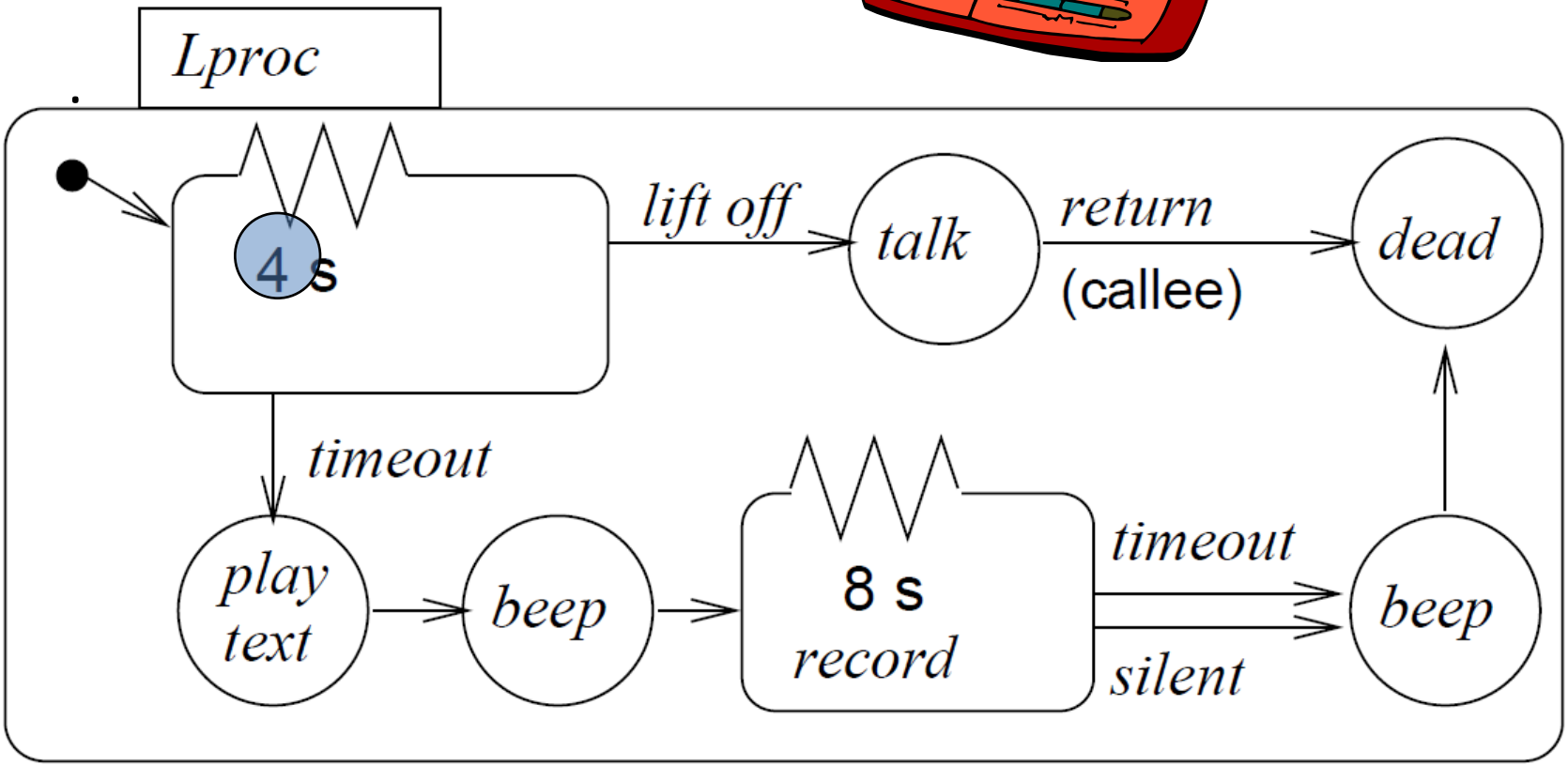
- States are either
  - basic states, or
  - AND-super-states, or
  - OR-super-states
- Hierarchy makes complex, concurrent state machines easier to understand
- But what about modeling time?

Timers

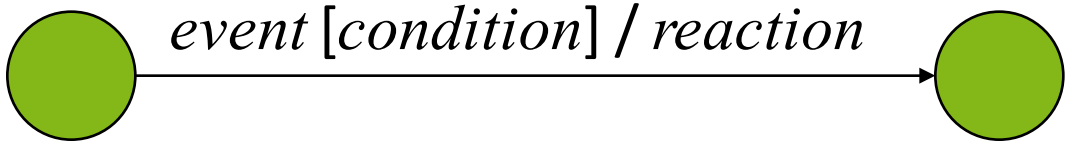
- Special edges can be used for timeouts
- Example: 20 ms timer
  - If after 20 ms in the left state an event (*a*) does not happen, move to the right state



Timer Example: Answering Machine



Edge Labels

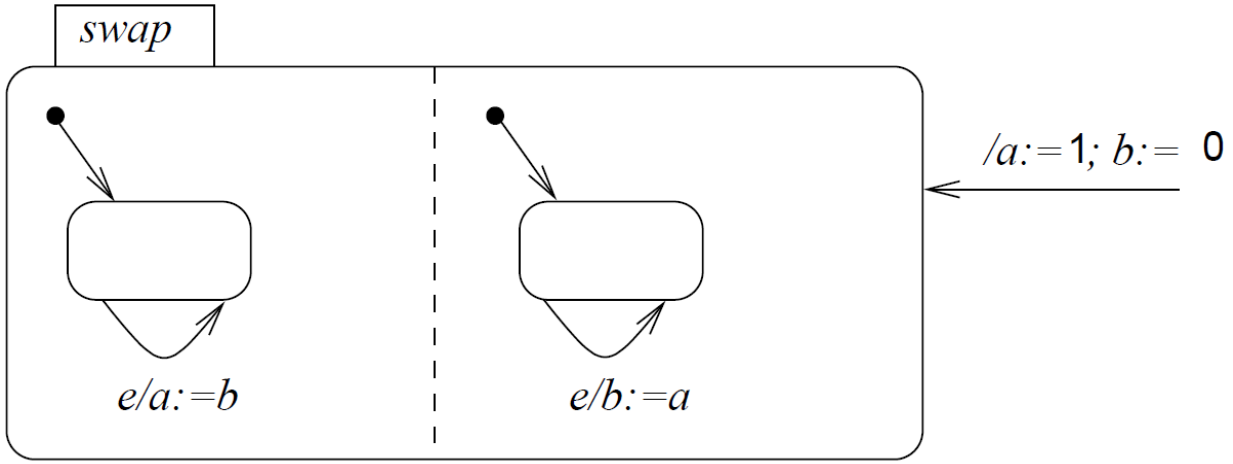


- Events:
  - Exist only until the next evaluation of the model
  - Can be either internally or externally generated
- Conditions:
  - Variables that keep their value until they are reassigned
- Reactions:
  - Can either be assignments for variables
  - or creation of events
- Example:
  - Service-off [not in Lproc] / service:=0

Simulating StateCharts with StateMate

- How are edge labels evaluated?
- Three phases:
  - Evaluate the effect of external changes
    - On events and conditions
  - Compute
    - The set of transitions to be made in the current step, and
    - The right hand sides of assignments
  - Transitions become effective, variables are assigned
- Separation of phases 2 and 3 enables a resulting unique ("determinate") behavior

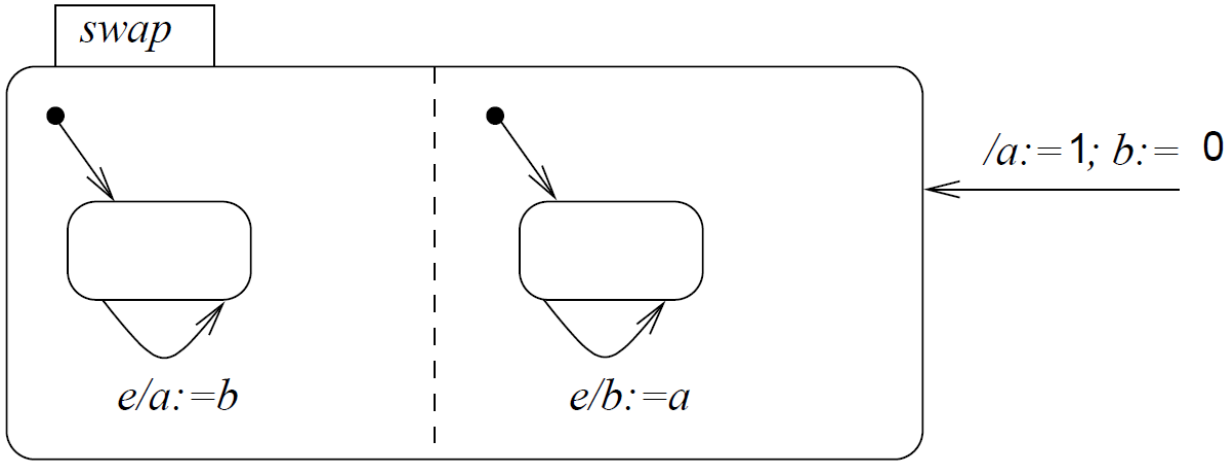
Example: Swapping Variable Values



- Assume multiple evaluation/assignment phases
- In phase 2, variables *a* and *b* are assigned to temporary variables
- In phase 3, these are assigned to *a* and *b*

  ⇒ Variables *a* and *b* are swapped!

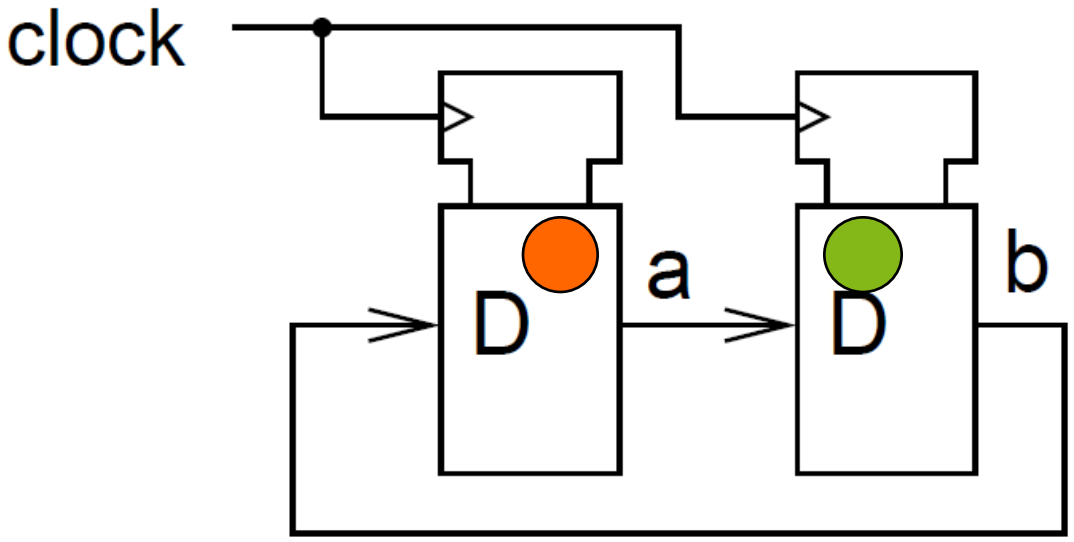
Example: When Swapping Fails



- Assume single evaluation/assignment phase
- Executing the left state first
  - Assigns the old value of b (=0) to a and b
- Executing the right state first
  - Assigns the old value of a (=1) to a and b
- Neither result matches the intention of the chart
  - And the result depends on the execution order!

Multi-phase Model Reflects Clocked HW

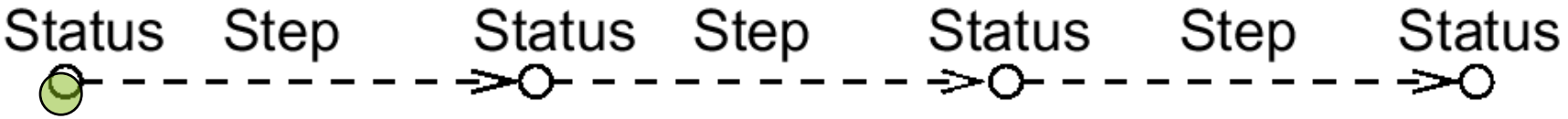
- In a clocked (synchronous) hardware system, both registers would be swapped as well



The same separation into phases is found in other languages as well, especially those that are intended to model hardware.

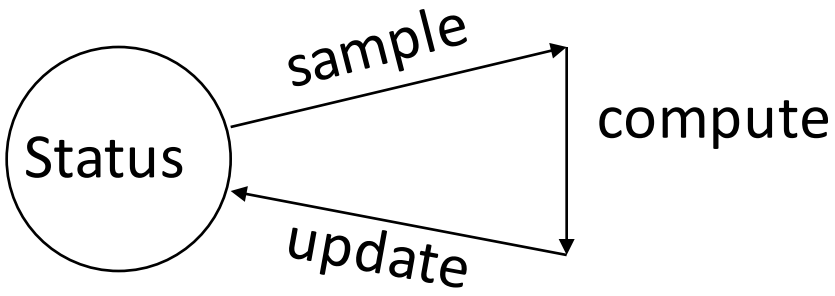
StateMate Execution

- Execution of a StateMate model consists of a sequence of (status, step) pairs



Status = values of all variables + set of events + current time

Step = execution of the three phases (StateMate semantics)



Other implementations of StateCharts do not have these 3 phases (and hence could lead to different results)!

Other Semantics

- Some languages lack three simulation phases
  - UML, dave, …
  - Support hierarchical state machines
  - But correspond to a SW view of modeling:
    - No synchronous clocks!
- Other systems allow the enabling and disabling of multi-phased simulation

Broadcast Mechanism in StateCharts

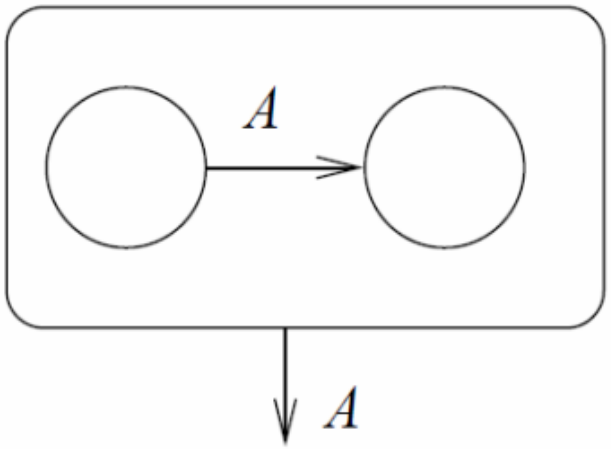
- Variables are visible to all parts of the model
  - New values are assigned in phase 3 of the current step
  - These values are observed by all parts of the model in the following step
  - StateCharts implicitly assume variables are **broadcast**
- Broadcast communication implies ***shared memory communication***
  - Other implementations would be very inefficient
- StateCharts is appropriate for local control systems
- But not for distributed applications for which updating variables might take some time
  - E.g., distributed sensing and control applications

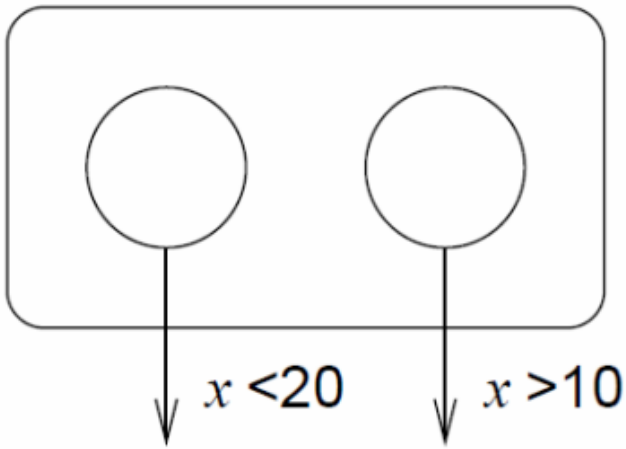
Lifetime of Events in StateCharts

- Events live until the step following the one in which they are generated ("one shot-events")

Conflicts

- It is often important that state machines behave in a predictable way
- What's the problem here?



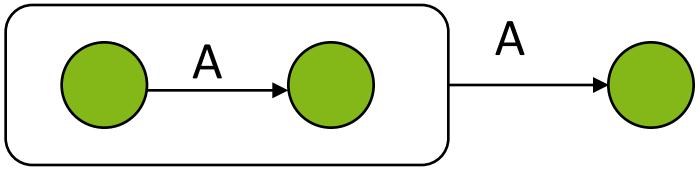


Determinate vs. Deterministic

- A system is *determinate* if it always obtains the same result for a fixed set (and timing) of inputs
  - Kahn (1974)
- Others call this property *deterministic*; however, this term has several meanings:
  - Non-deterministic finite state machines
  - Non-deterministic operators (*e.g.* + with non-deterministic result in low order bits)
  - Behavior not known before run-time (unknown input results in non-determinism)
  - In the sense of determinate as used by Kahn
- To avoid confusion, we use the term "determinate"

Are StateCharts Determinate?

- Must all simulators return the same result for a given input?
- Three-phase simulation is required
  - Otherwise evaluation order affects result
- No transition conflicts
  - Which is the correct transition?



Tools typically issue a warning if such a situation could exist

⇒ StateMate is **determinate** if transition conflicts are resolved and no other sources of undefined behavior exist

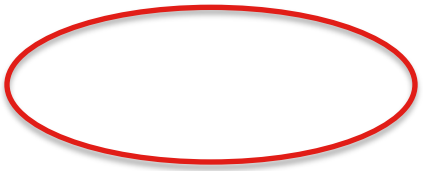
Evaluation of StateCharts: Pros

- Hierarchy allows arbitrary nesting of AND- and OR-super states
- Many commercial simulation tools are available
  - StateMate, StateFlow, BetterState, …
- "Back-ends" translate StateCharts into C or VHDL
  - Enabling software or hardware implementations!

Evaluation of StateCharts: Cons

- Cons:
  - Not useful for distributed applications
  - No program constructs
  - No description of non-functional behavior
  - No object-orientation
  - No description of structural hierarchy
  - Generated C programs may be inefficient
- Extensions:
  - Module charts for description of structural hierarchy

MoCs Considered in 421

Communication/ local computations	Shared memory	Message passing Synchronous	Asynchronous
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	(Not useful)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

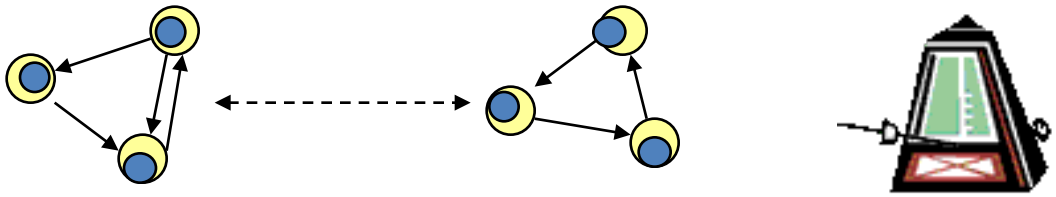
Synchronous (Programming) Languages

- Descriptions of concurrent processes is non-determinate in many languages:
  - Task execution order is not specified
  - This may affect results!
- Synchronous languages explicitly prevent this behavior by assuming a global clock

  "… when automata are composed in parallel, a transition of the product is made of the 'simultaneous' transitions of all of them."

  [Nicolas Halbwachs]

Synchronous Languages Basics



- Global clock synchronizes all state machines
- Each clock tick:
  - All inputs are considered,
  - New outputs and states are calculated, and then
  - All state transitions are made
- Communication is “instantaneous”

Abstraction of Delay

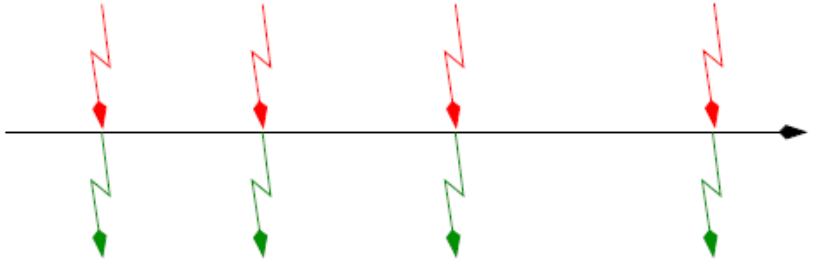
- Let
  - $f(x)$: some function computed from input x,
  - $\Delta(f(x))$: the delay for this computation
  - δ: some abstraction of the real delay (*e.g.* a safe upper bound)
- Consider the composition: $f(x) = g(h(x))$
- Then, the sum of the delays of g and h would be a safe upper bound on the delay of f
- Two solutions:
  - $\delta = 0$, always $\Rightarrow$ synchrony
  - δ = ? (hopefully bounded) $\Rightarrow$ asynchrony
- *Asynchronous languages don't work* [Halbwachs]
  - *E.g.* what exactly does a wait (10 ns) in a programming language do?

Based on slide 15 of N. Halbwachs: Synchronous Programming of Reactive Systems, *ARTIST2 Summer School on Embedded Systems*, Florianopolis, 2008

Composition

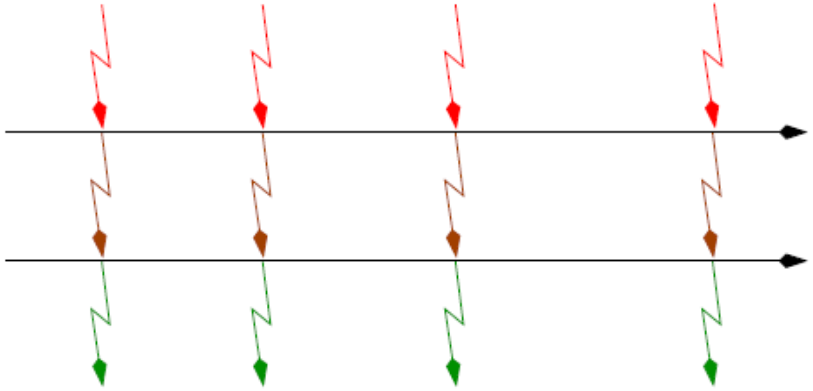
Abstract synchronous behavior

sequence of reactions to input events, to which all processes take part:



At the abstract level, a single FSM reacts ***immediately***

Composition of behaviors:



At the abstract level, reaction of connected other automata is ***immediate***

Based on slide 16 of N. Halbwachs: Synchronous Programming of Reactive Systems, *ARTIST2 Summer School on Embedded Systems*, Florianopolis, 2008

Concrete Behavior

The abstraction of synchronous languages is valid as long as real delays are always shorter than the clock period.

Reference: slide 17 of N. Halbwachs: Synchronous Programming of Reactive Systems, *ARTIST2 Summer School on Embedded Systems*, Florianopolis, 2008

Synchronous Languages

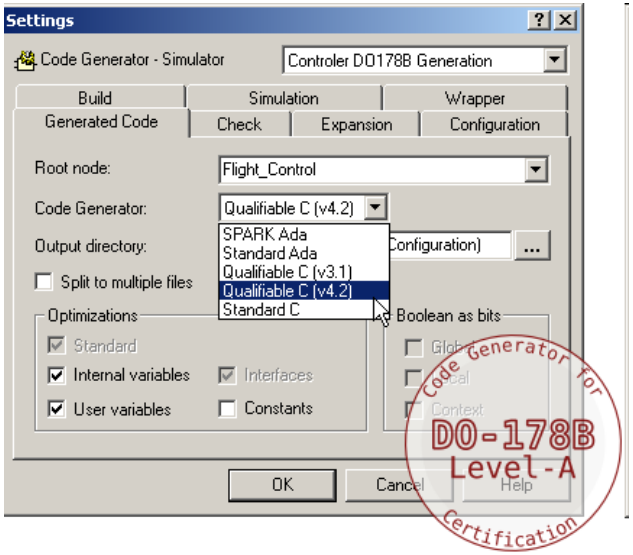
- Require a broadcast mechanism
- Take an idealistic view of concurrency
- Guarantee determinate behavior
- StateCharts (using StateMate semantics)
  - "Almost" a synchronous language [Halbwachs]
  - Immediate communication ($\delta = 0$) is missing

Implementation and Specification Model

- Implementation model: finite state machines (FSMs)
- Different notational styles for specification
  - "Imperative": Esterel (textual)
  - SyncCharts: graphical version of Esterel
  - "Data-flow": Lustre (textual)
  - SCADE (graphical) is a mix
- Specifications always include a close link to the generated FSMs
  - "Imperative" does not have semantics close to von-Neumann languages

Applications of Sync Languages

- *SCADE Suite and SCADE KCG Qualified Code Generator*
  - *Used by AIRBUS and many of its main suppliers*
  - *Used for most of A380 and A400M critical on board software*
  - *Used for the A340-500/600 Secondary Flying Command System*



Instance of "model-based design"

Source: http://www.esterel-technologies.com/products/scade-suite/

MoCs Considered in 421

Communication/ local computations	Shared memory	Message passing Synchronous	Asynchronous
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		**SDL**
Data flow	(Not useful)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

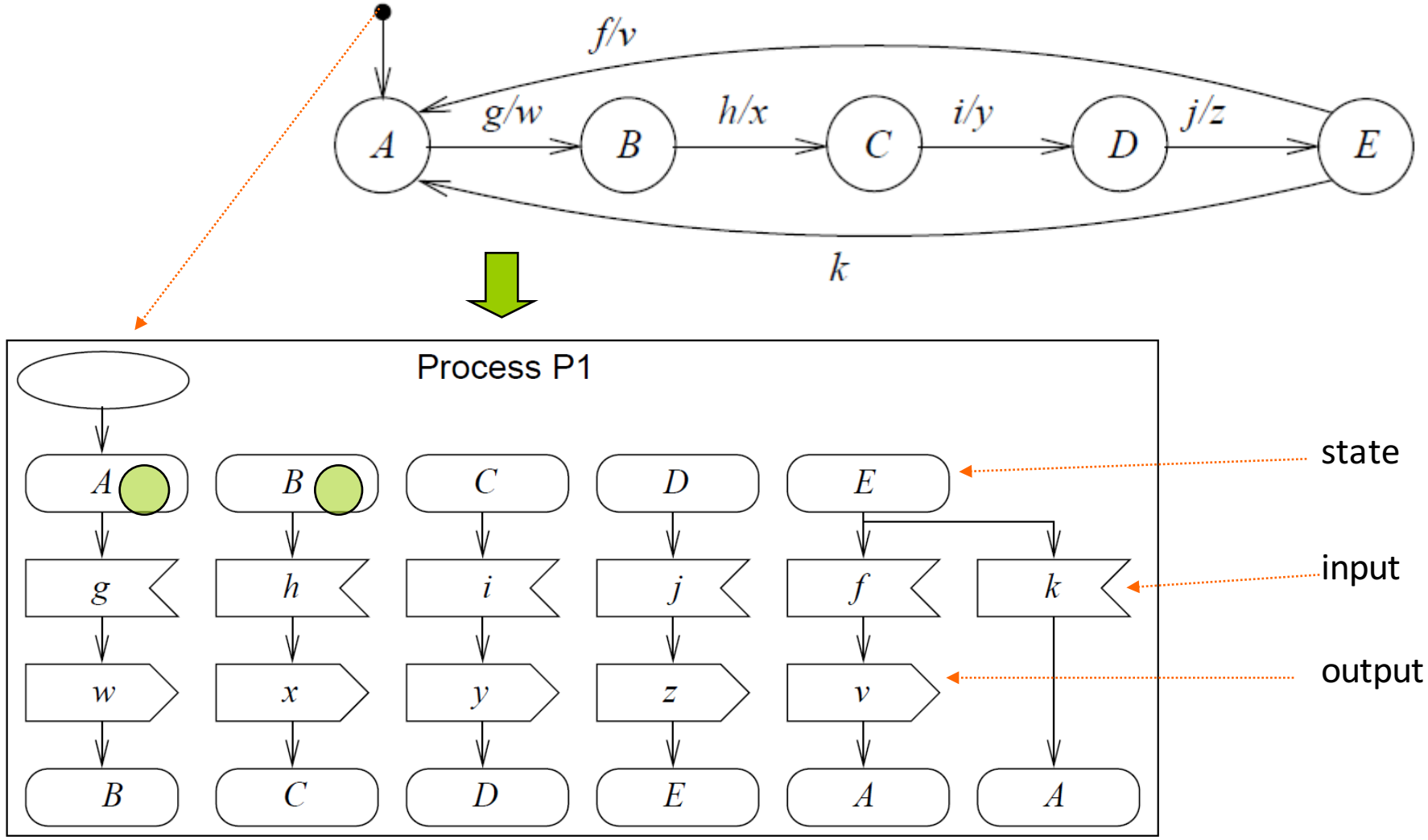
SDL

- Specification and Description Language
  - A prominent example of MoC using **asynchronous message passing communication**
- Designed for specification of distributed systems
  - Unlike StateCharts
  - Dates back to early 70s
  - Formal semantics defined in the late 80s
  - Defined by ITU (International Telecommunication Union): Z.100 recommendation in 1980
    Updates in 1984, 1988, 1992, 1996 and 1999

SDL Basics

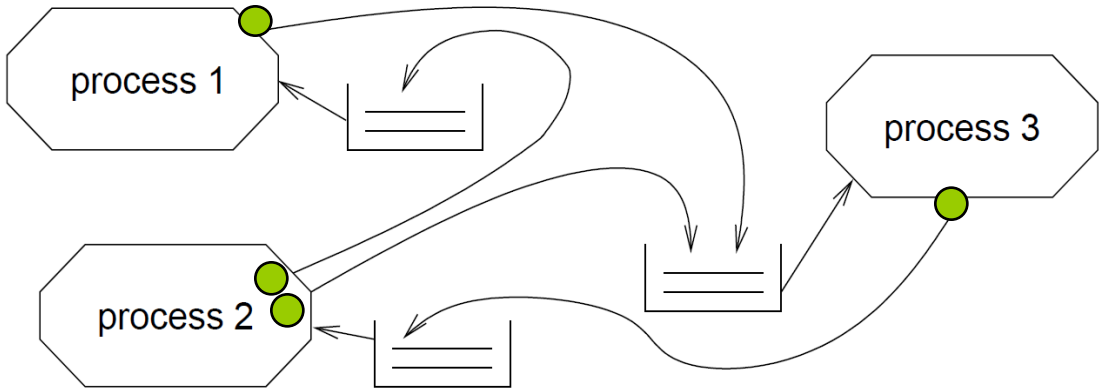
- Provides textual and graphical formats
- Like StateCharts
  - CFSM model of computation
  - Each FSM is called a process
- Unlike StateCharts
  - Message passing instead of shared memory
- SDL supports operations on data

Representation of SDL-FSMs



Communication among SDL-FSMs

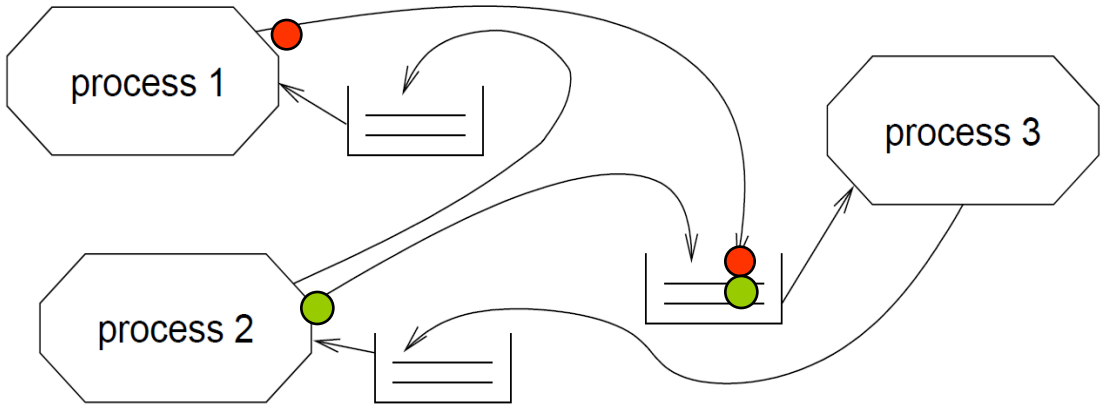
- Communication between FSMs (or "processes")
  - Based on message-passing
  - Assumes a potentially infinite FIFO queue



- Each process fetches next entry from the FIFO, and
- checks if input enables transition;
- if yes: makes transition;
- if no: input is ignored (exception: SAVE-mechanism)

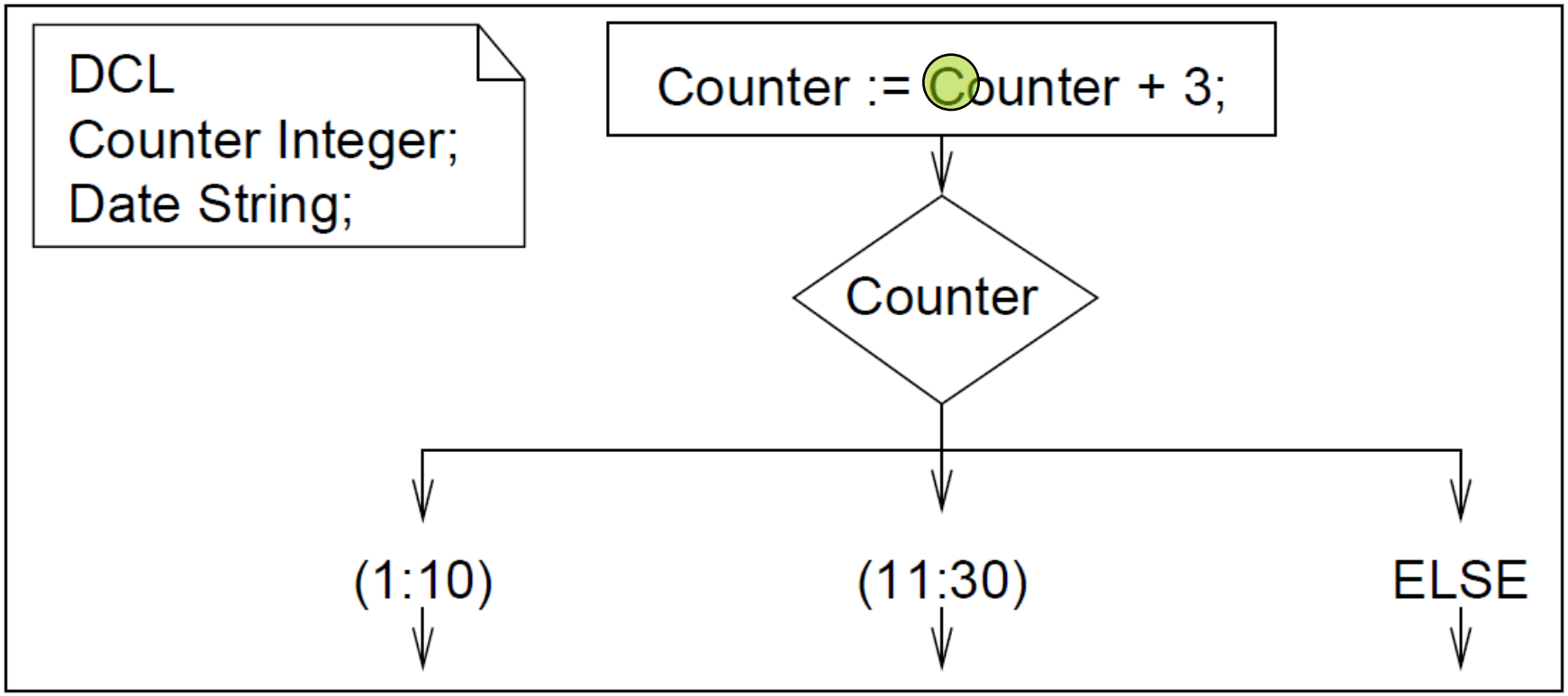
Is SDL Determinate?

- Let tokens arrive at FIFO at the same time
- All orders are legal, but
  - The order in which they are stored is unknown
- Simulators can show different behaviors for the same input, all of which are correct



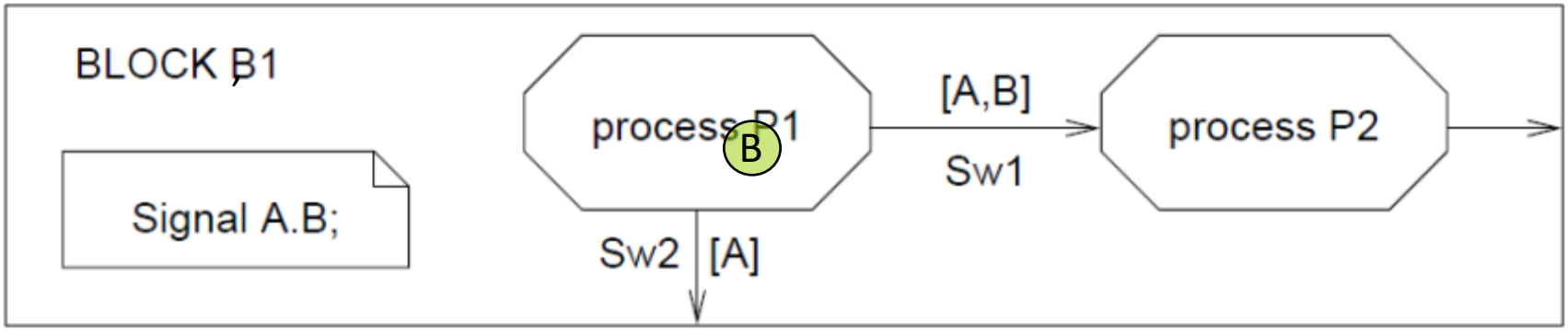
Operations on Data

- Variables can be declared locally for processes
- Their type can be predefined or defined in SDL itself
- SDL supports abstract data types (ADTs)
- Examples:



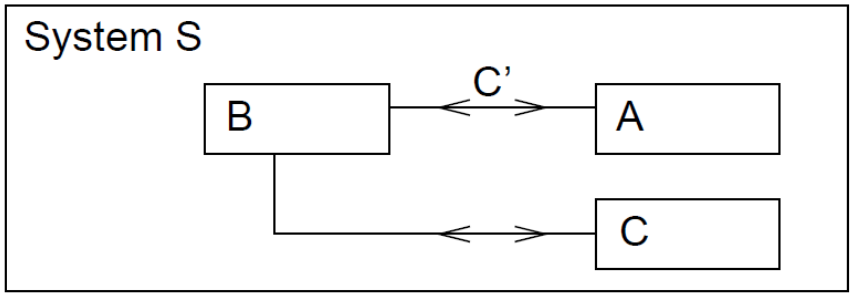
Process Interaction Diagrams

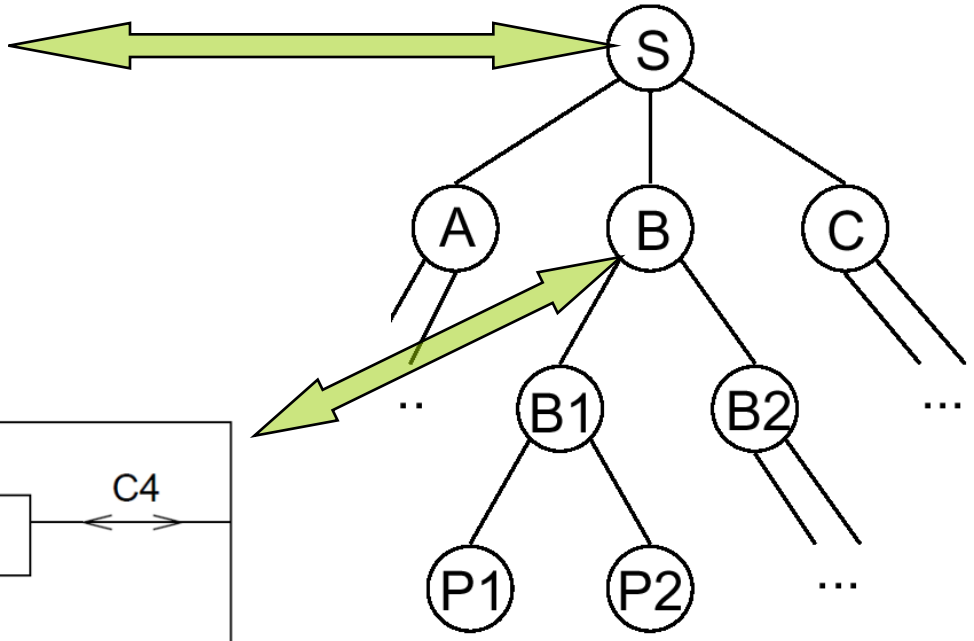
- Describe interaction of processes
  - Special case of block diagrams
- Process interaction diagrams contain
  - Processes (P1, P2 below)
  - Signal declarations (A, B below)
  - Channels (Sw1, Sw2 below)
- Example:

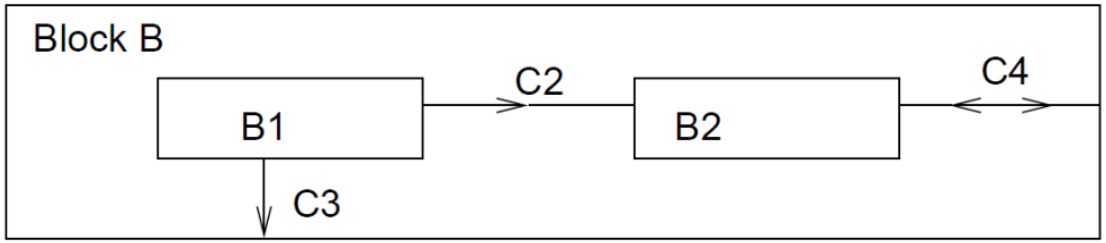


Hierarchy in SDL

- Process interaction diagrams can be included in blocks
- The root block is called “system”
- Processes cannot contain other processes
  - Unlike in StateCharts

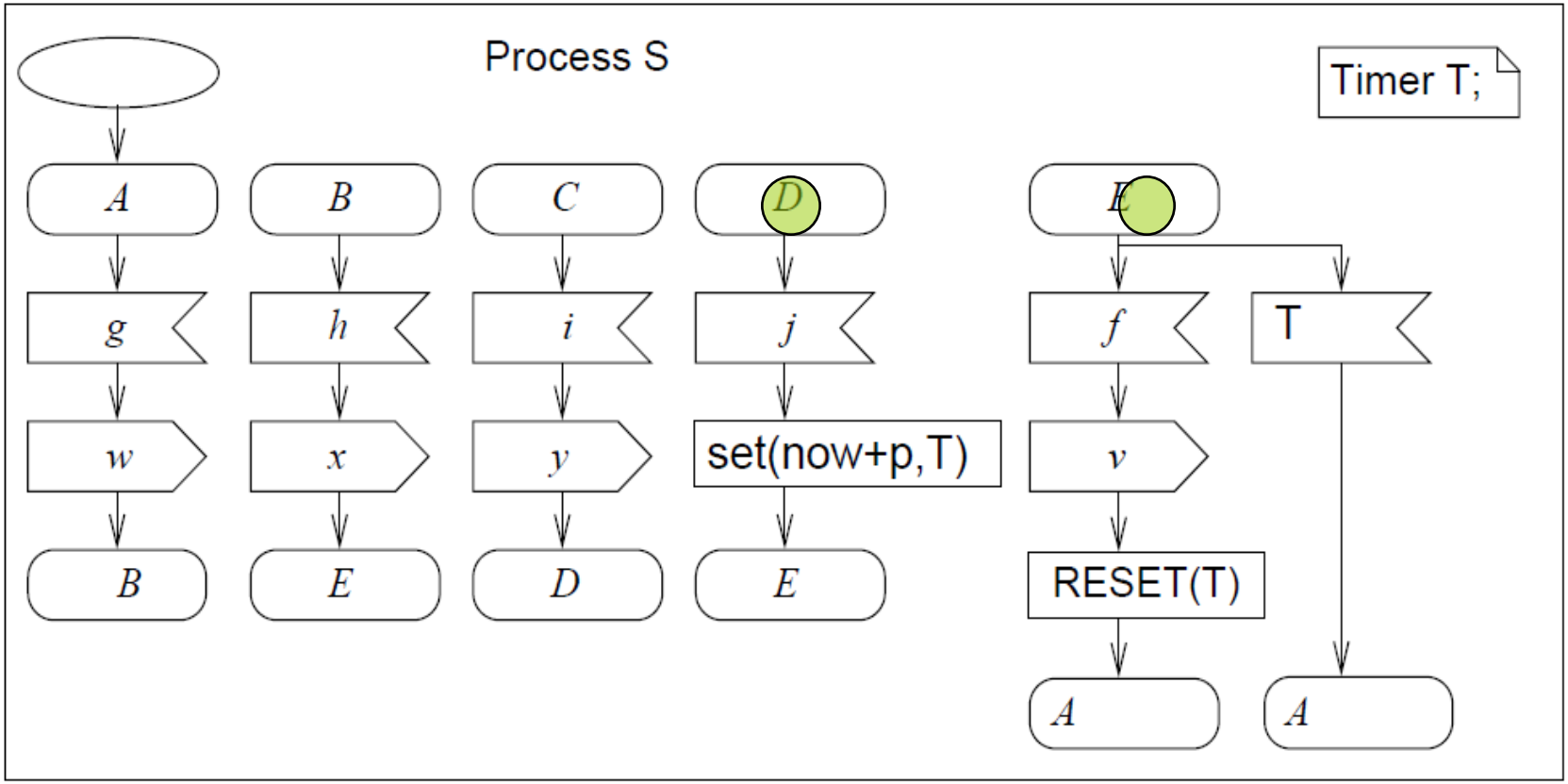






Timers

- Timers can be declared locally
- Elapsed timers put signals into queue
  - Not necessarily processed immediately: other signals may precede
  - RESET removes timer (also from FIFO-queue)

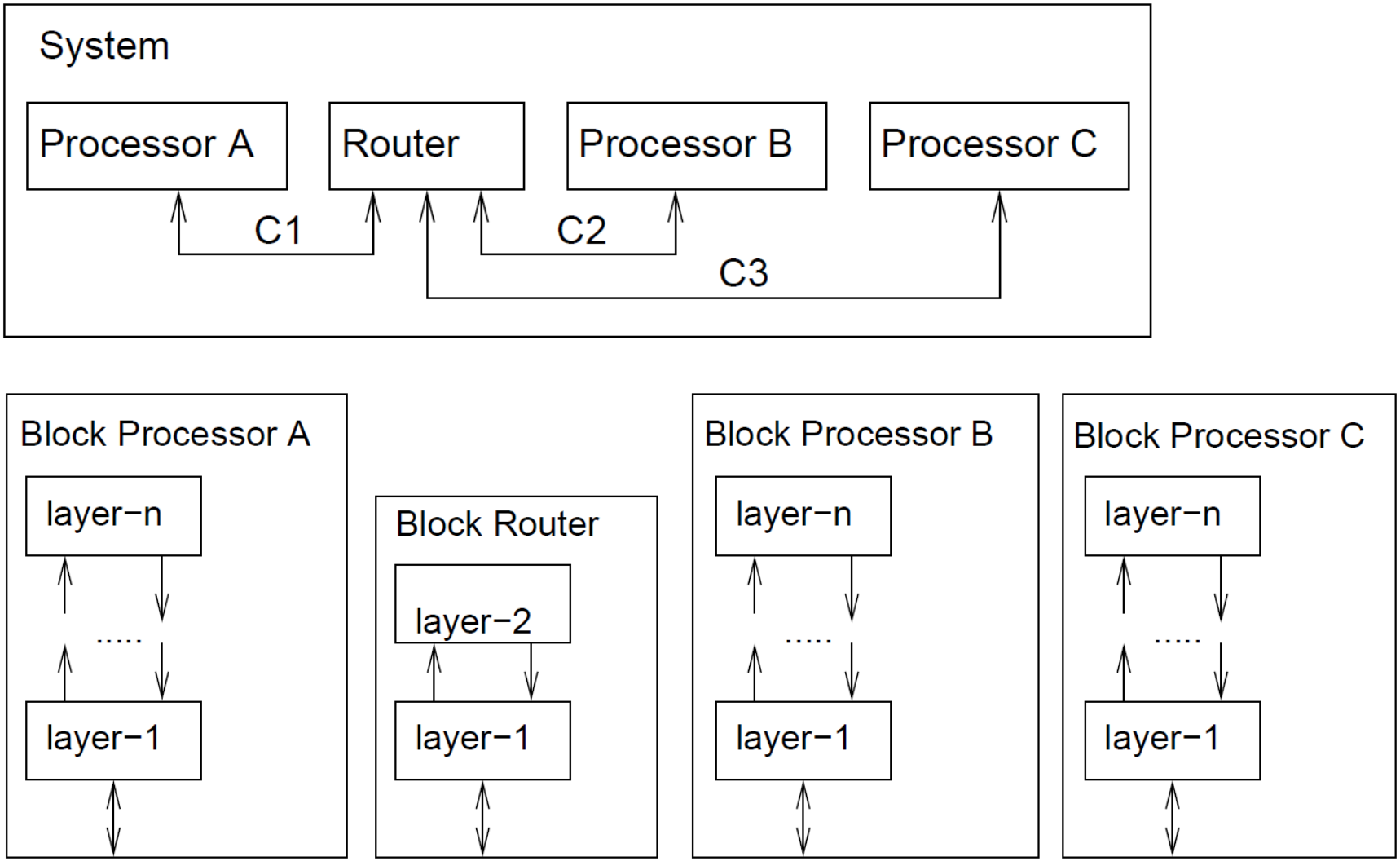


Additional Language Elements

- SDL includes a number of additional elements
  - Procedures
  - Creation and termination of processes
  - Advanced description of data
  - More features added for SDL-2000

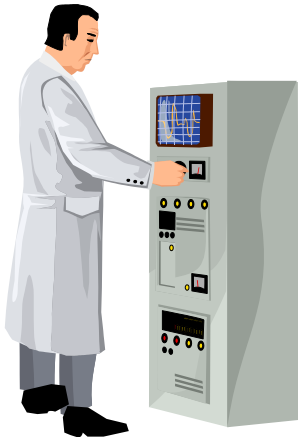
    (not well accepted)

Application: Network Protocols



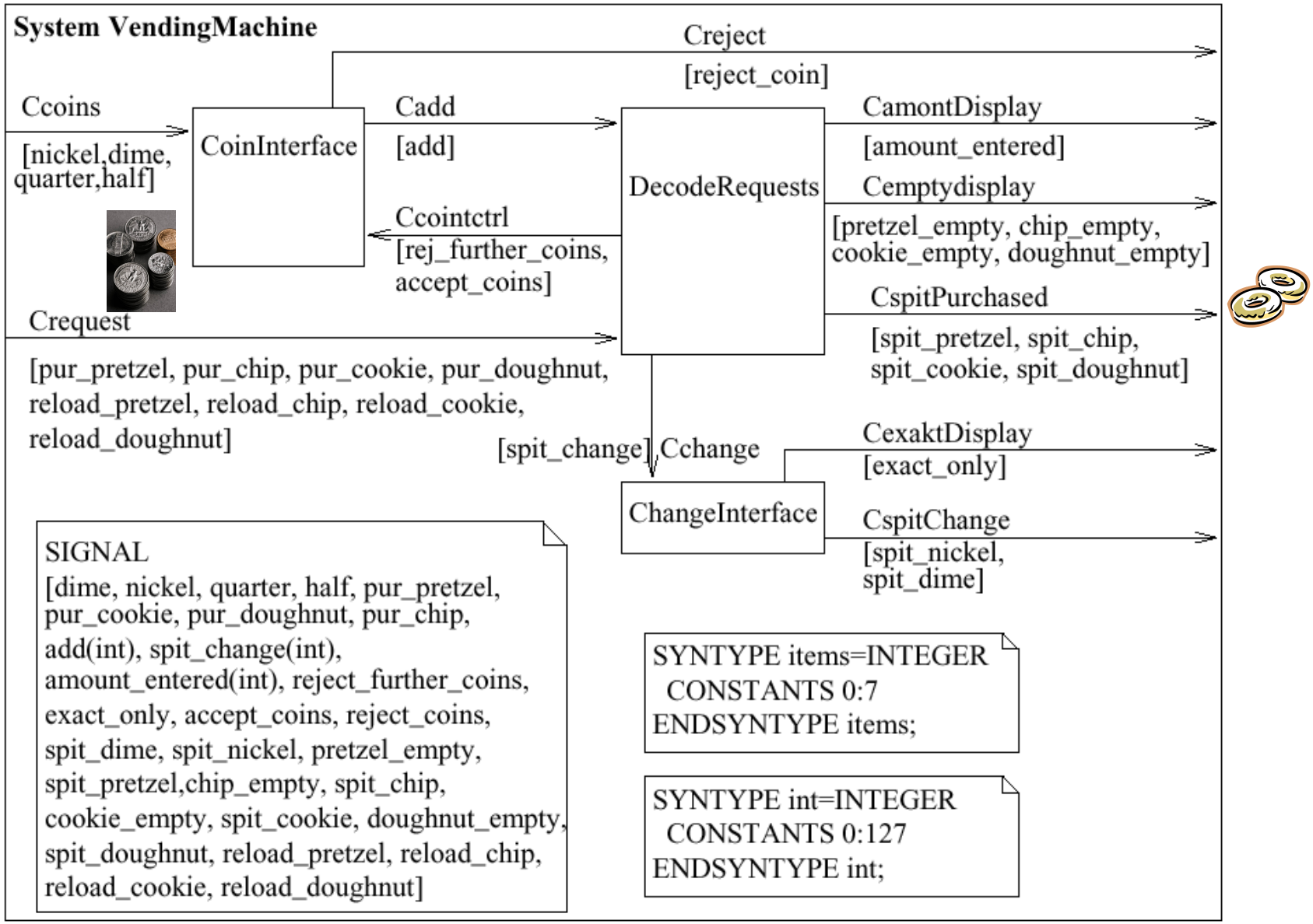
Larger Example: Vending Machine

- Machine[1] selling pretzels, (potato) chips, cookies, and doughnuts:
- Accepts nickels, dime, quarters, and half-dollar coins
- Not a distributed application

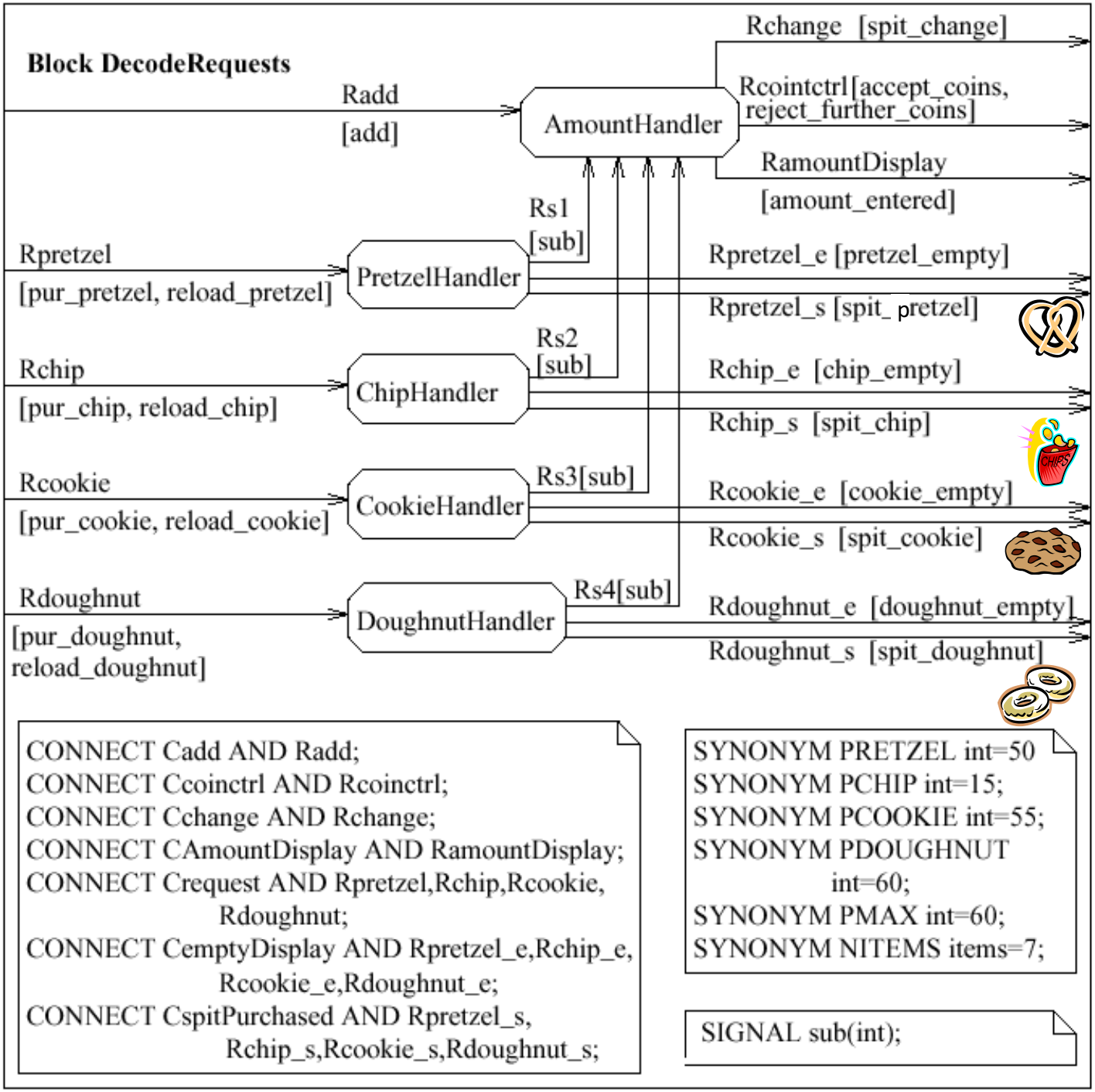


[1] [J.M. Bergé, O. Levia, J. Roullard: High-Level System Modeling, Kluwer Academic Publishers, 1995]

Vending Machine Overview

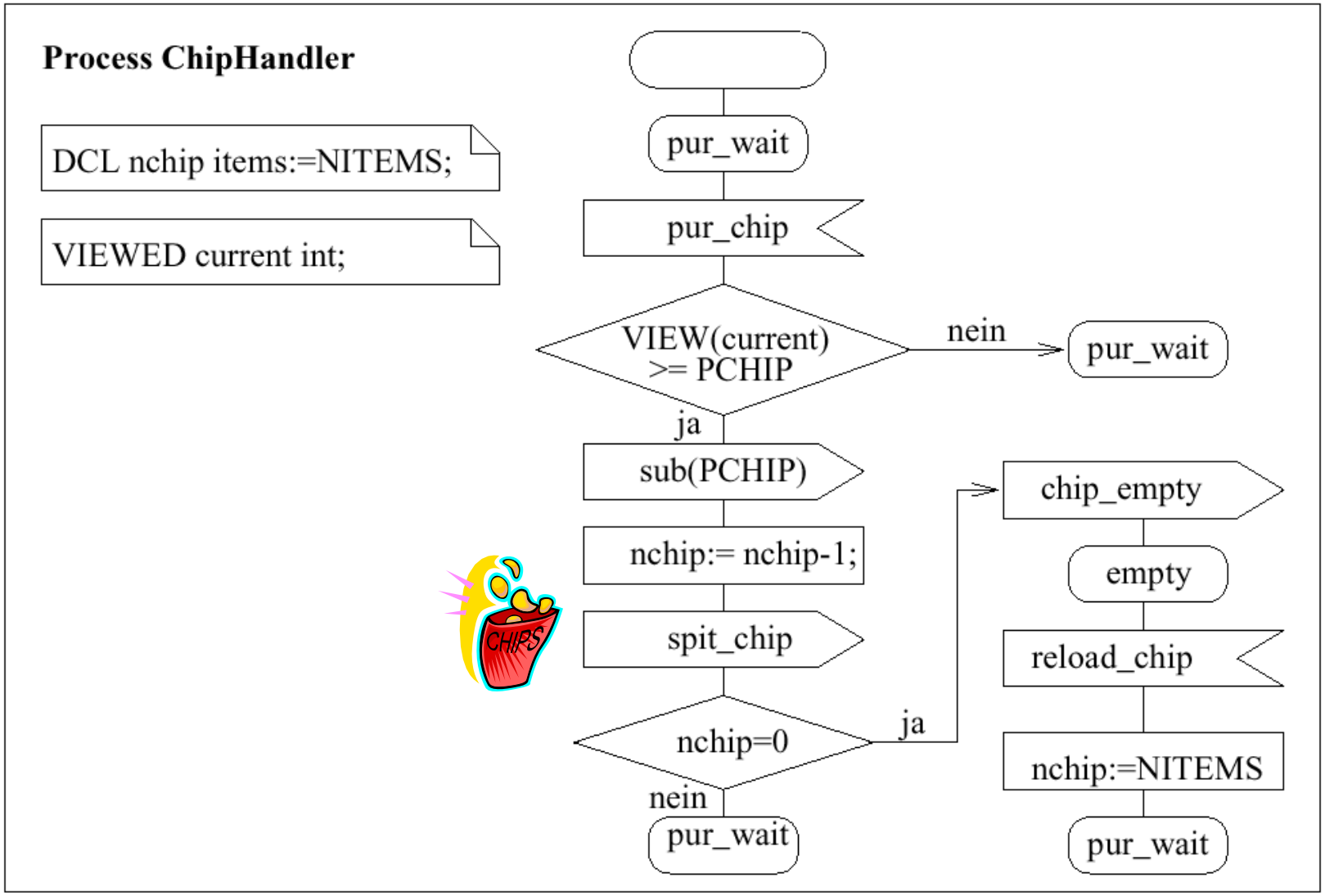


Decode Requests



McGill

ChipHandler



SDL: Summary

- Modeling
  - Computation: FSM model for the components
  - Communication: Non-blocking message passing
- Pros
  - Excellent for distributed applications (used for ISDN)
  - Commercial tools available

    (see http://www.sdl-forum.org)

SDL: Summary, Cont'd

- Cons
  - Implementation requires upper bound on FIFO length: may be difficult!
  - 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

Summary

- StateCharts
  - Hierarchical states: OR-States and AND-States
  - A notion of time: timers
  - Communication: variable updates are broadcast
    - Shared memory model
  - Determinate when conflicts aren't present

Summary, Cont'd

- Synchronous (programming) languages
  - Based on clocked finite state machine view
  - Based on assumption of instantaneous communication
    - Real delays must be small
  - Guaranteed to be determinate
- Specification and Description Language
  - Built for distributed systems; not determinate

Next Time

- Data Flow Modeling
  - Chapter 2.5