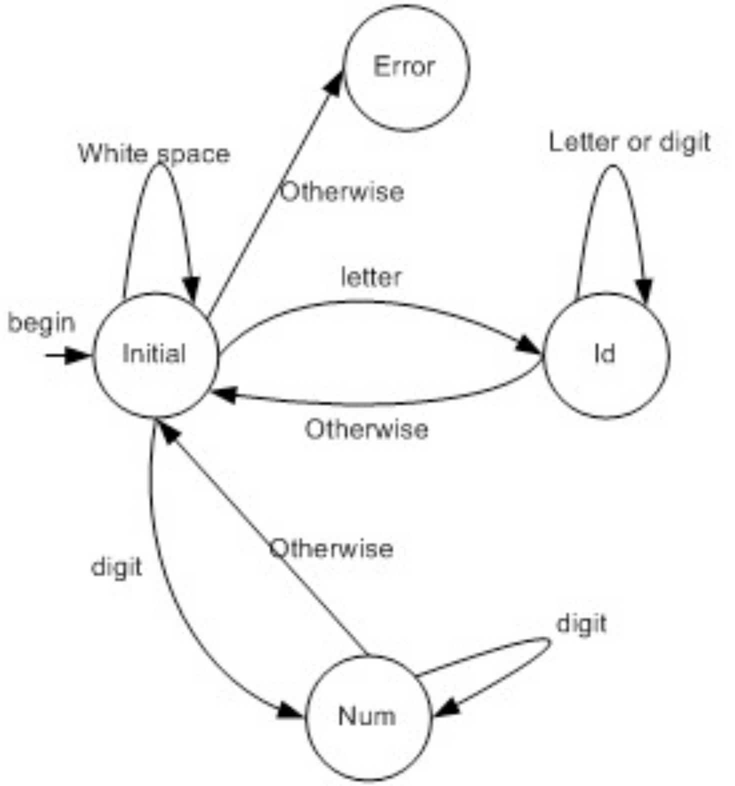
SE463
Software Requirements Specification & Analysis

Behaviour Modelling

Finite State Machines

- You have probably seen an FSM like this to represent a grammar
  - This FSM recognizes a token stream of identifiers and numbers

- “Finite State Automaton” (FSA) is another term for FSM.

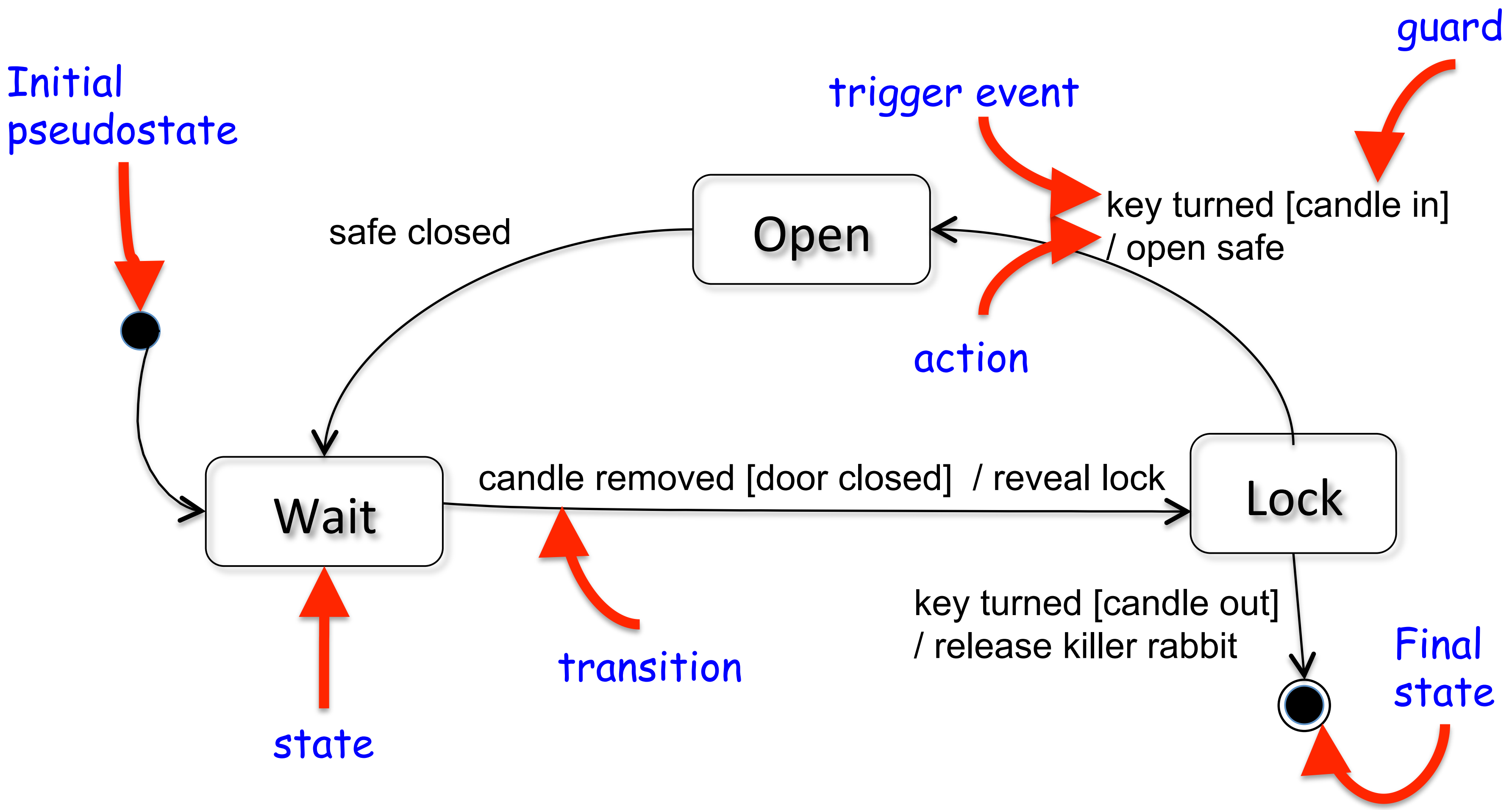


http://www.codeproject.com/KB/recipes/Parser_Expression.aspx

State Machines

Fowler, UML Distilled,3ed, p.76

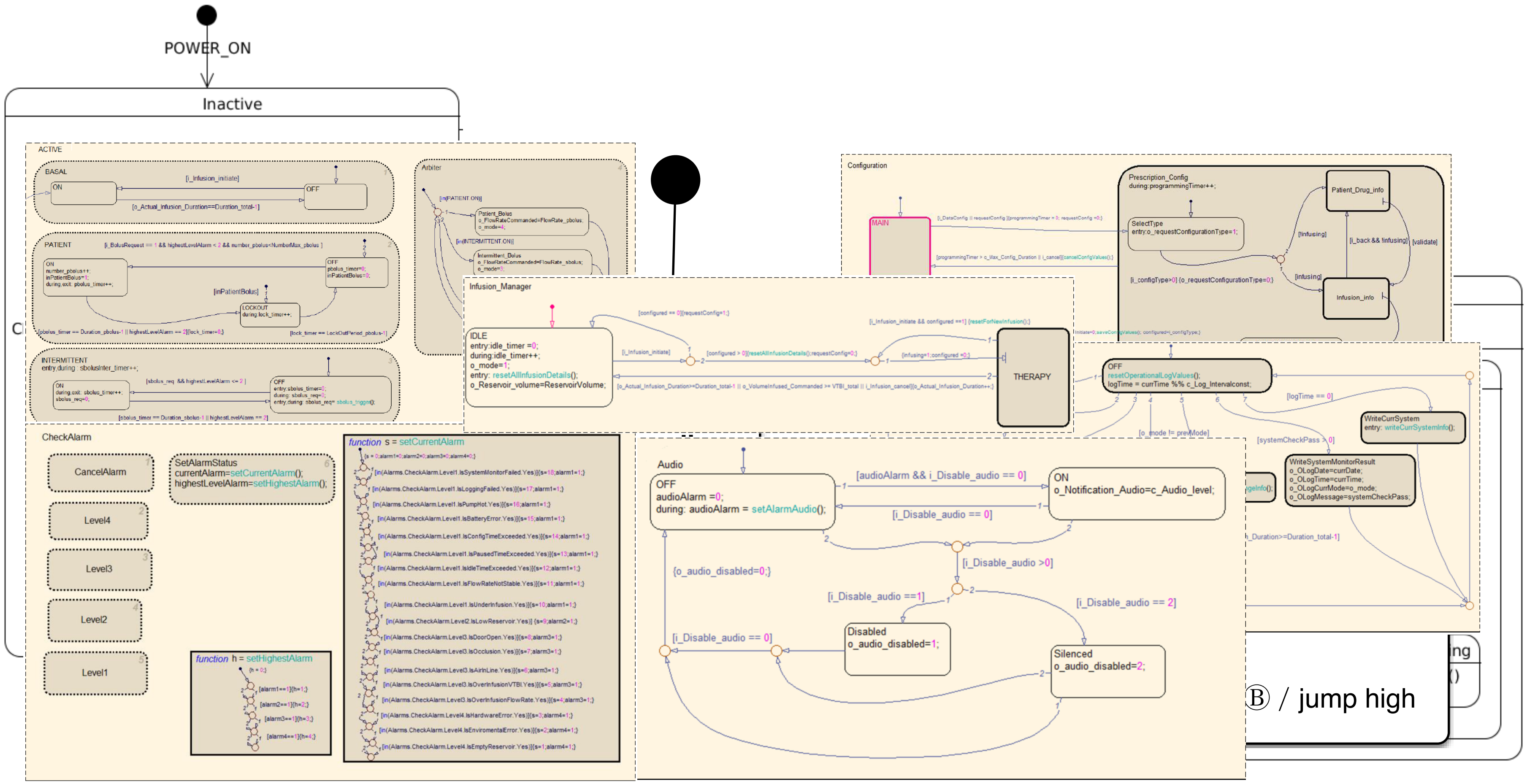
A state machine models behaviour that depends on the history of inputs received so far:



State Machines in Practice

Used to understand complex behaviour

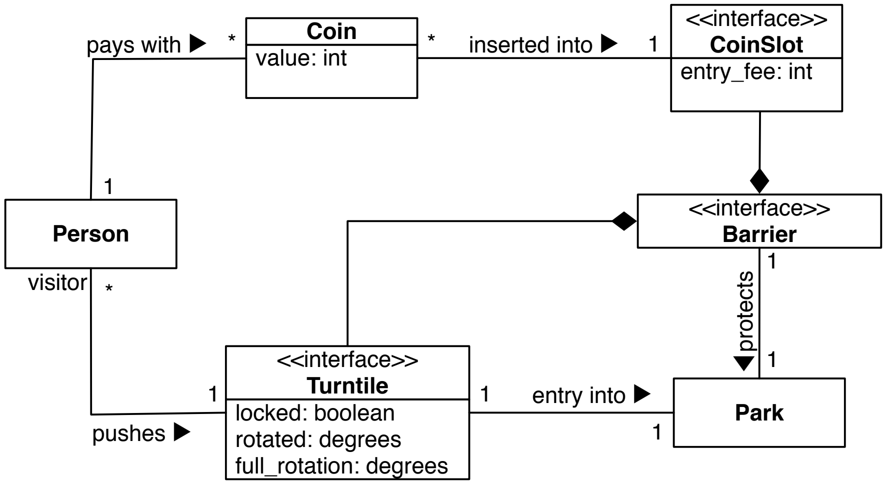
- multiple sequences, cases, conditions

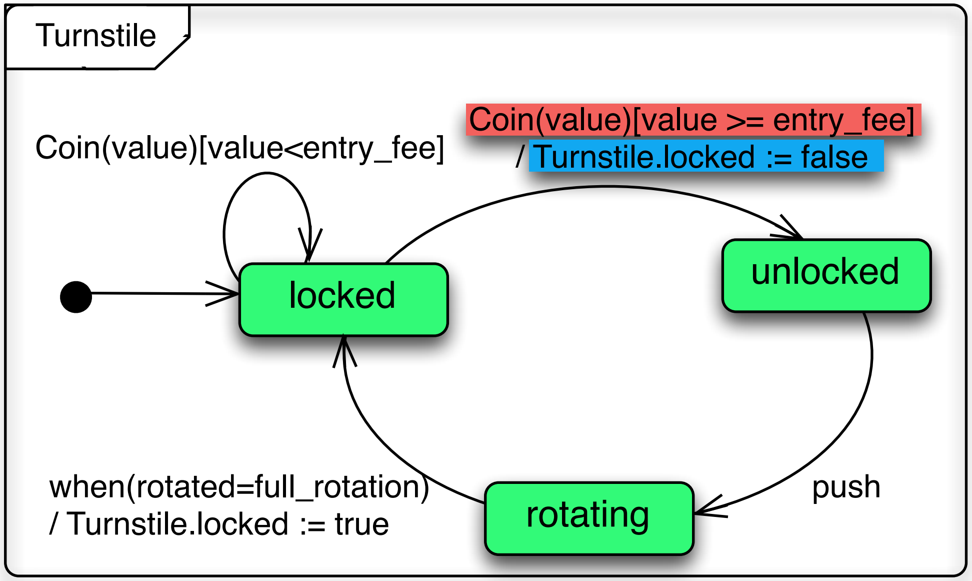


Copyright © Atlee, Berry, Day, Godfrey 1995-2016

UML State Machines

- States represent equivalence classes of input traces
- Inputs are events and conditions on ≪interface≫ phenomena
- Outputs are actions on ≪interface≫ phenomena
- State machines can have internal variables (not shown)




Turnstile
Coin(value)[value<entry_fee]
Coin(value)[value >= entry_fee]
/ Turnstile.locked := false
locked
unlocked
rotating
push
when(rotated=full_rotation)
/ Turnstile.locked := true

States

- States partition the behaviour of the system

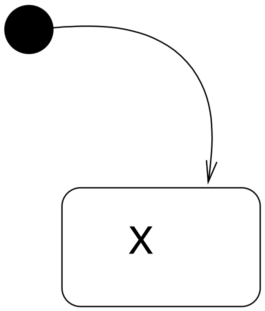
- Each state represents a possible mode of operation, such as
  - equivalence set of history of inputs
  - equivalence set of future behaviours
    - distinct set of input events of interest
    - distinct reactions to input events
  - internal processing of input (delimited by transitions)

States and pseudo-states

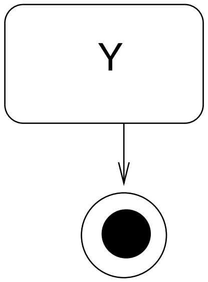
- There always needs to be a designated starting/initial state.

- The designator of an initial state is a *pseudo-state.*
  - A pseudo-state is NOT a real state (no time is spent there)
  - Later, we will see the History pseudo-state

- Often there is a designated final state. This is a real state.



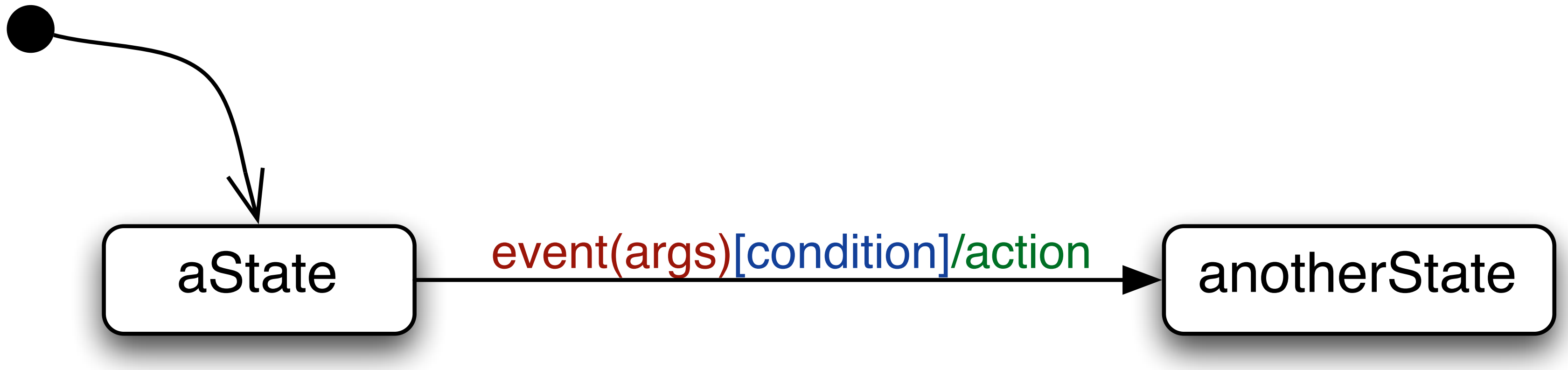
X is the initial State



Bullseye is a final state

Transitions

Transitions represent observable execution steps



Each of these parts of the transition is optional.

- event(args) — input event / message that triggers the transition
- [condition] — (boolean) guard condition; the transition cannot fire unless the guard condition is *true*
- /action — a simple, fast, non-interruptible action,

  e.g., variable assignment, output event, start complex operation

Events

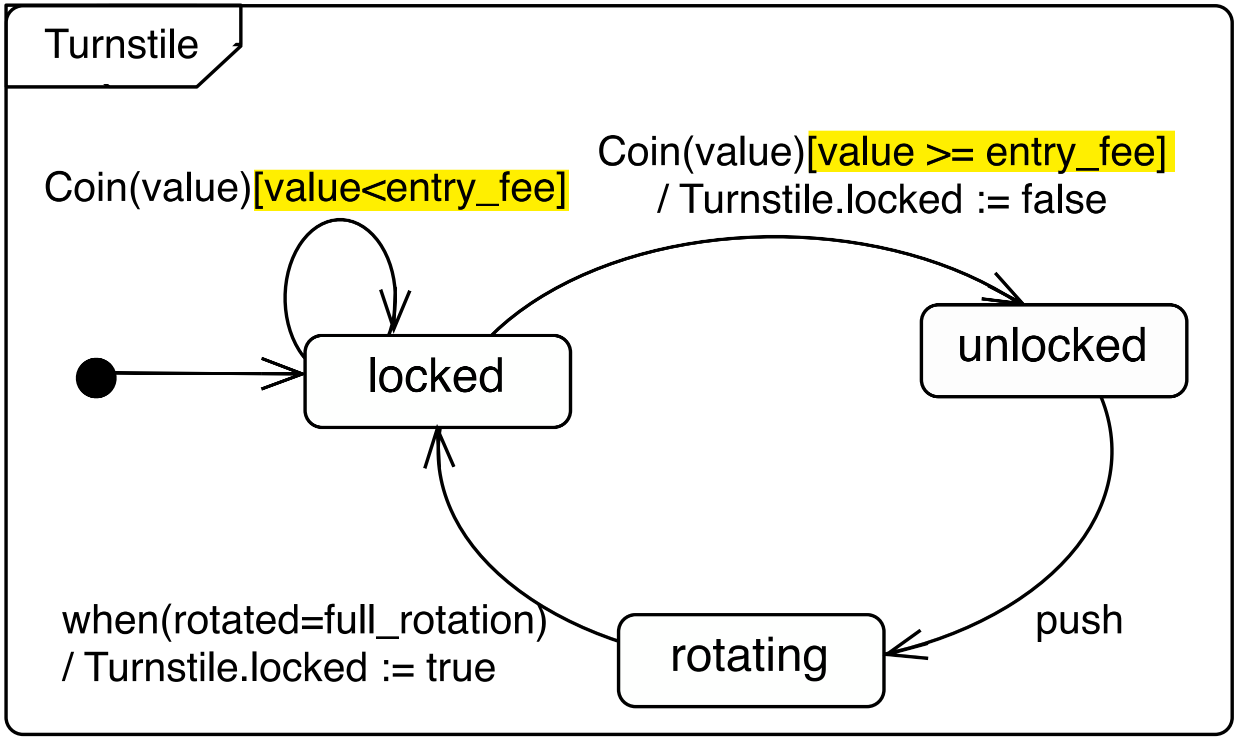
An event is a significant or noteworthy change in the environment

- input message from the environment
  - login request
- some change to ≪interface≫ phenomena
  - e.g., coin inserted, turnstile pushed, elevator button pressed
- passage of time
- multiple events on a transition label are alternative triggers

Conditions

A condition is a Boolean expression

- over ≪interface≫ phenomena (e.g., [value >= fee])
- over state-machine variables


Turnstile
Coin(value)[value<entry_fee]
Coin(value)[value >= entry_fee]
/ Turnstile.locked := false
locked
unlocked
rotating
push
when(rotated=full_rotation)
/ Turnstile.locked := true

Actions

Actions are the system's response to an event

- output message
- change to ≪interface≫ phenomenon
  - e.g., Turnstile.locked := true
  - e.g., AddLoan(m:LibraryMember, p:Publication, today:Date)

An action is non-interruptible (i.e., atomic)

Multiple actions on a transition are separated by ";" and execute sequentially.

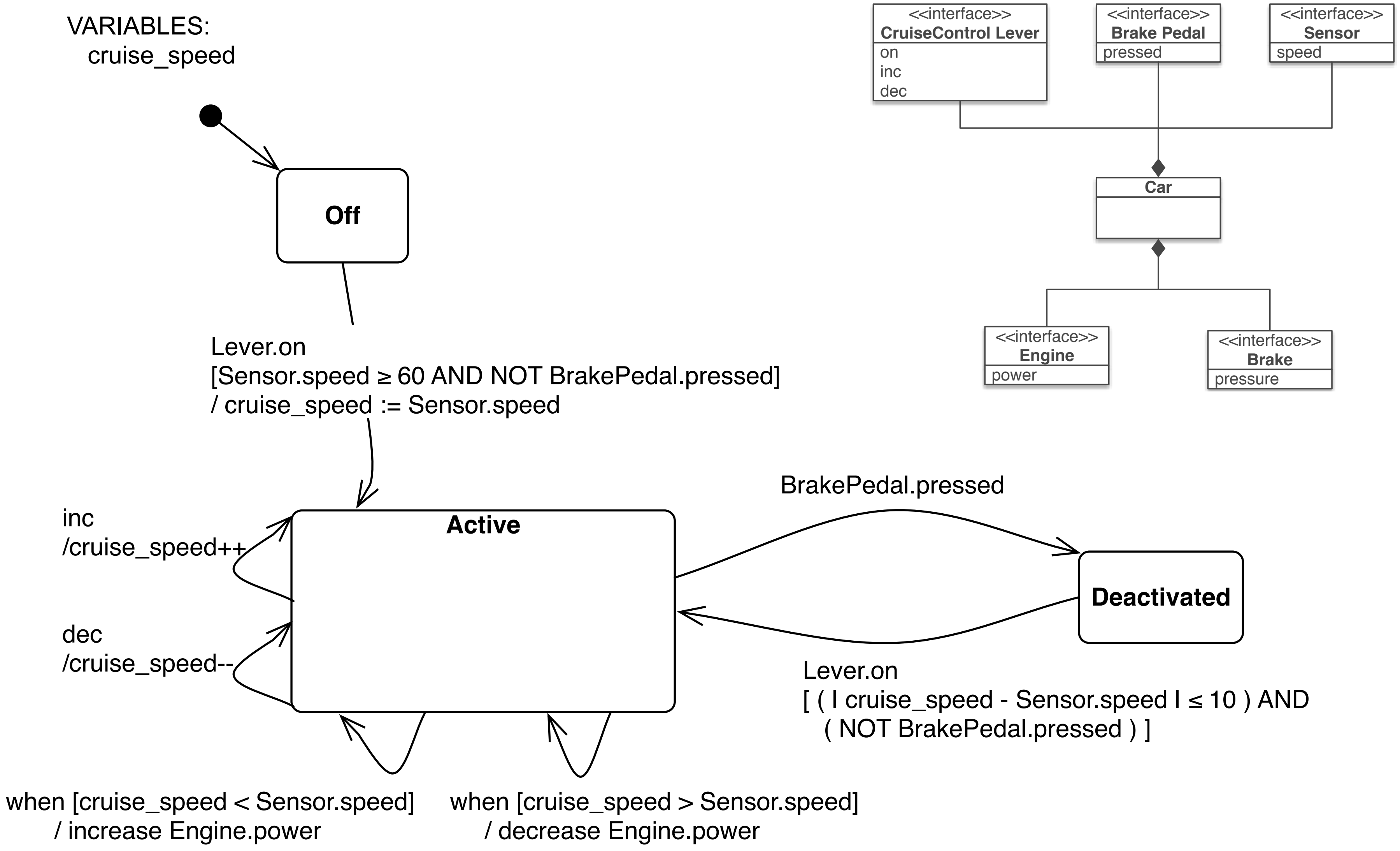
Creating a Behaviour Model

Process:

1. Identify input and output events
   - Changes to interface phenomena in the Domain Model
2. Think of a natural partitioning into states:
   - Activity states - system performs activity or operation
   - Idle states - system waits for input
   - System modes - use different states to distinguish between different reactions to an event
3. Consider the behaviour of the system for each input at each state.

Cruise Control Example

VARIABLES:
cruise_speed



<<interface>>
CruiseControl Lever
on
inc
dec

<<interface>>
Brake Pedal
pressed

<<interface>>
Sensor
speed

Car

<<interface>>
Engine
power

<<interface>>
Brake
pressure

Off

Lever.on
[Sensor.speed ≥ 60 AND NOT BrakePedal.pressed]
/ cruise_speed := Sensor.speed

Active

inc
/cruise_speed++

dec
/cruise_speed--

BrakePedal.pressed

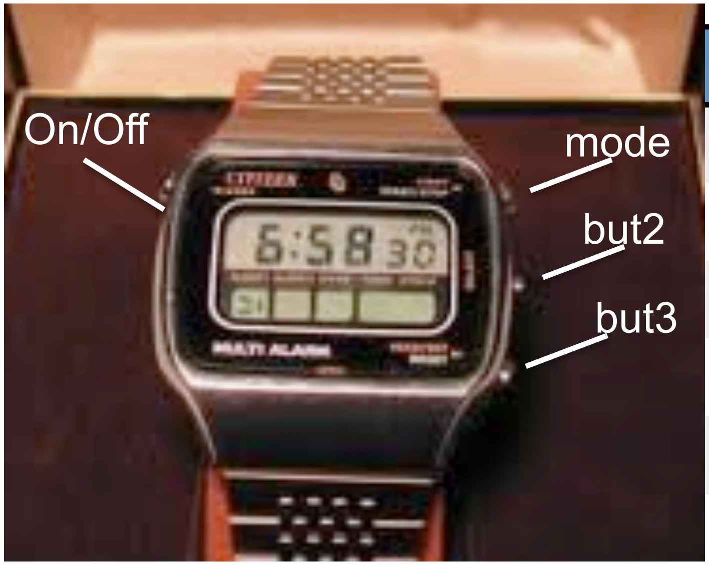
Deactivated

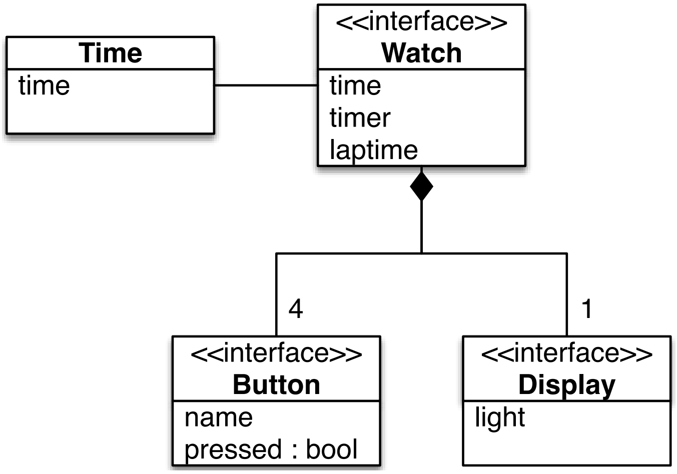
Lever.on
[(I cruise_speed - Sensor.speed I ≤ 10) AND
(NOT BrakePedal.pressed)]

when [cruise_speed < Sensor.speed]
/ increase Engine.power

when [cruise_speed > Sensor.speed]
/ decrease Engine.power

Stopwatch Example





Time
time

<<interface>>
Watch
time
timer
laptime

4

1

<<interface>>
Button
name
pressed : bool

<<interface>>
Display
light

Input	Meaning of Input Command
on	Turn watch on
off	Turn watch off
mode	Toggle between watch and stopwatch
but2 [watch]	Toggle between 12h and 24h display
but2 [stopwatch]	Pause/ resume timer
but3 [watch]	Turn light on for 3 sec
but3 [stopwatch, timer running, display timer]	Record laptime; display laptime; turn light on for 3 sec
but3 [stopwatch, timer stopped, display timer]	Reset timer; turn light on for 3 sec
but3 [stopwatch, timer running, display laptime]	Display timer; turn light on for 3 sec

Check for Completeness

For each state X, consider the system's response to each event e:

- There is a transition on e from state X
- Event e cannot physically occur in state X
  - e.g., doorOpened cannot occur when the door is already open
  - no transition on e is needed from X
- Event e is possible but the system should ignore it
  - e.g., multiple "door close" button presses; only first one is significant
  - no transition on e is needed from X
- Event e is possible in state X, but the system should report an error
  - a transition is needed to report error

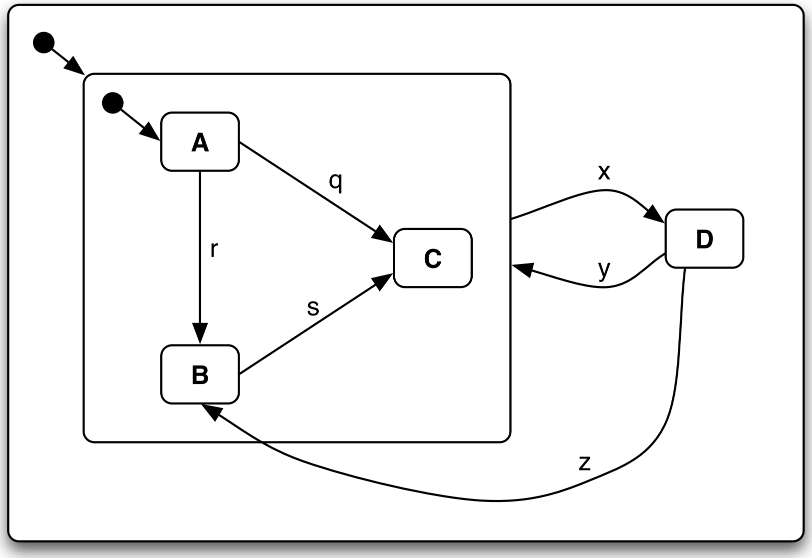
State machine vs. sequence diagrams

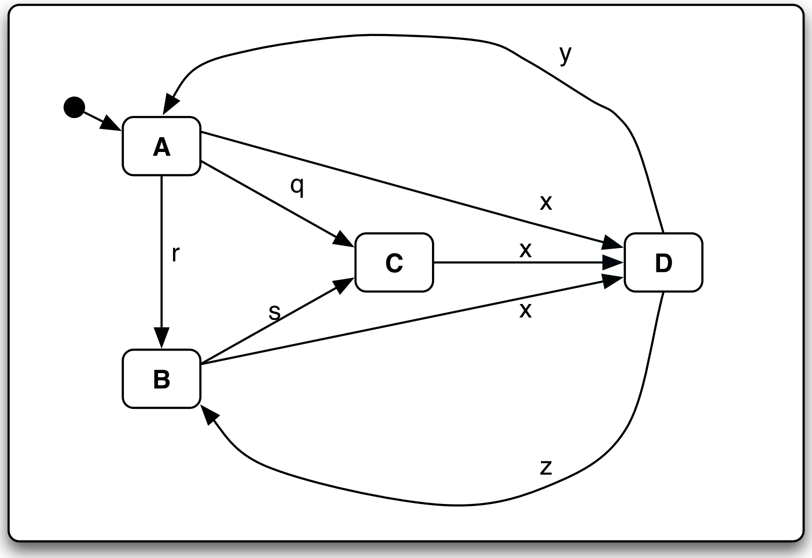
State machine diagrams	**Sequence diagrams**
specifies behaviour	*Illustrates* behaviour
all allowable scenarios	one allowable scenario, showing end-to-end behaviour (better feel for overall system behaviour)
developer oriented	customer oriented
identifies system states, which represent equivalent input histories	
	can help developer validate state diagrams

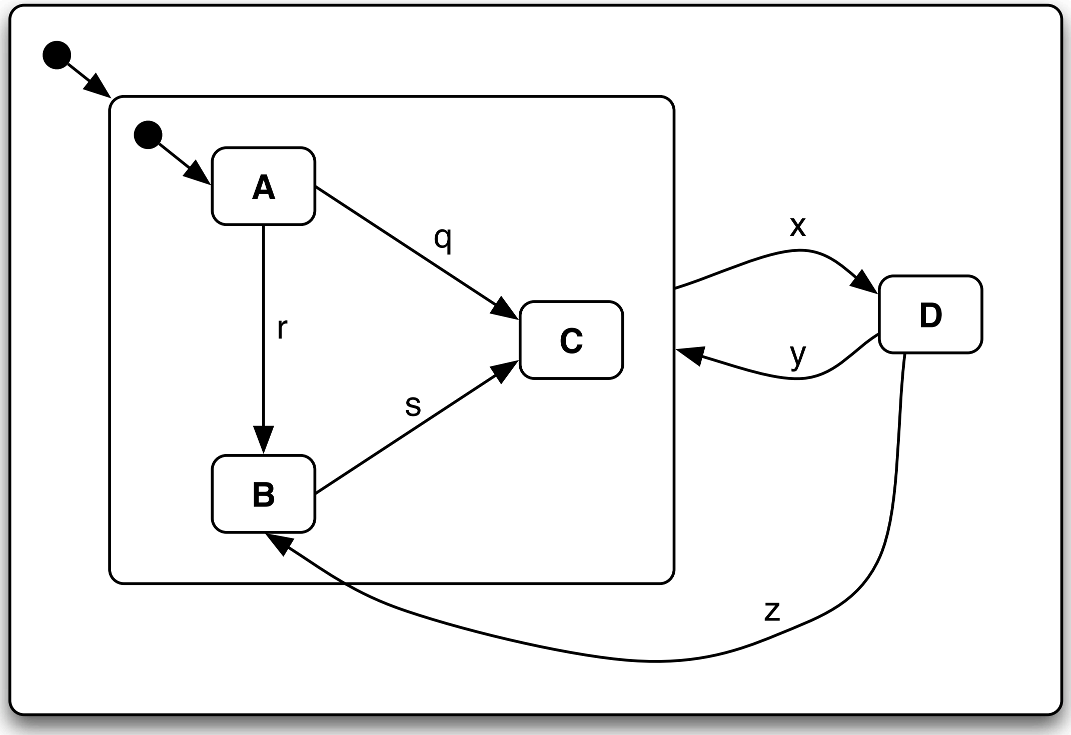
Hierarchical states

Hierarchy is used to cluster states that have some similar behaviours / exiting transitions.

- One transition leaving a superstate represents a transition from each of the superstate's descendent states.

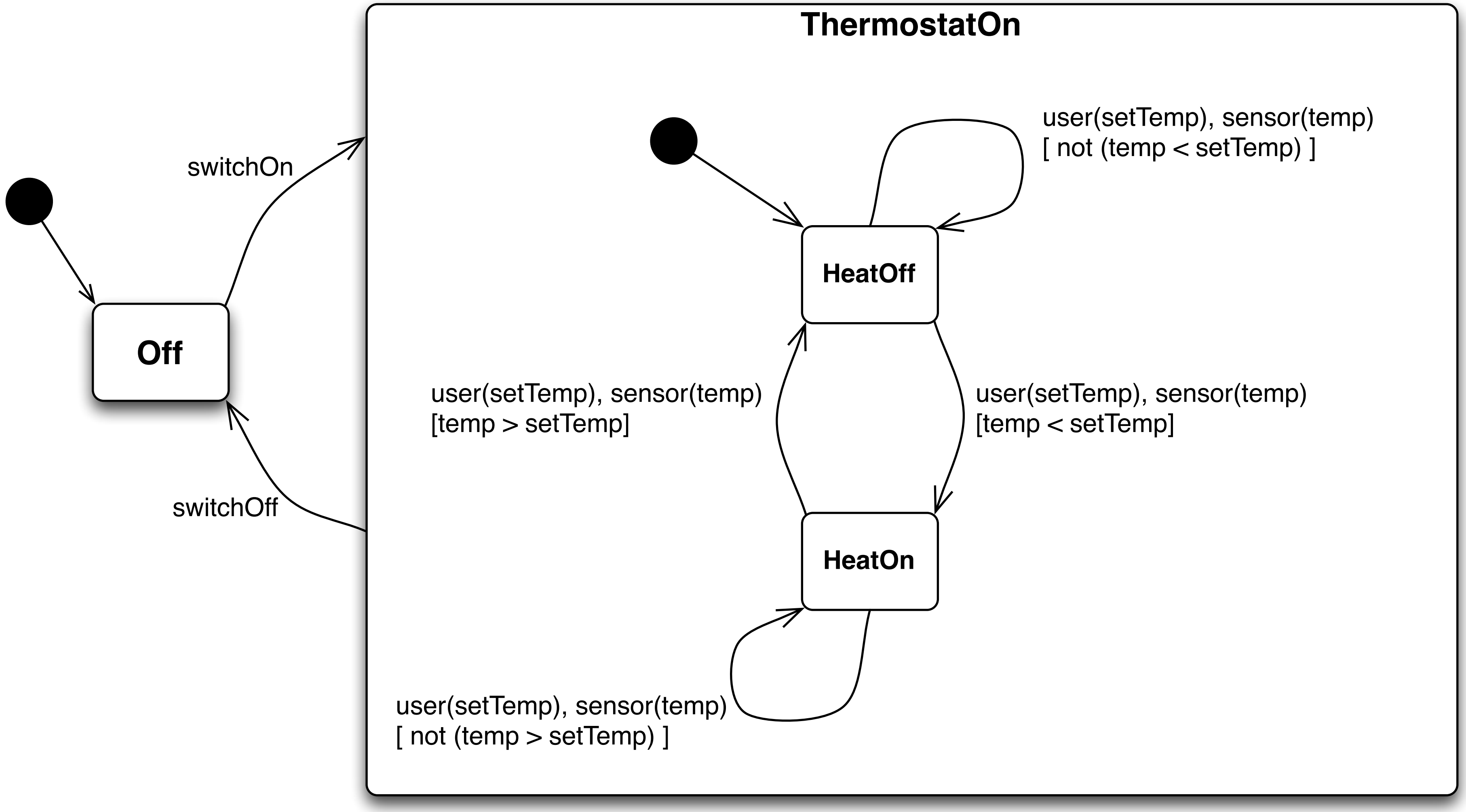






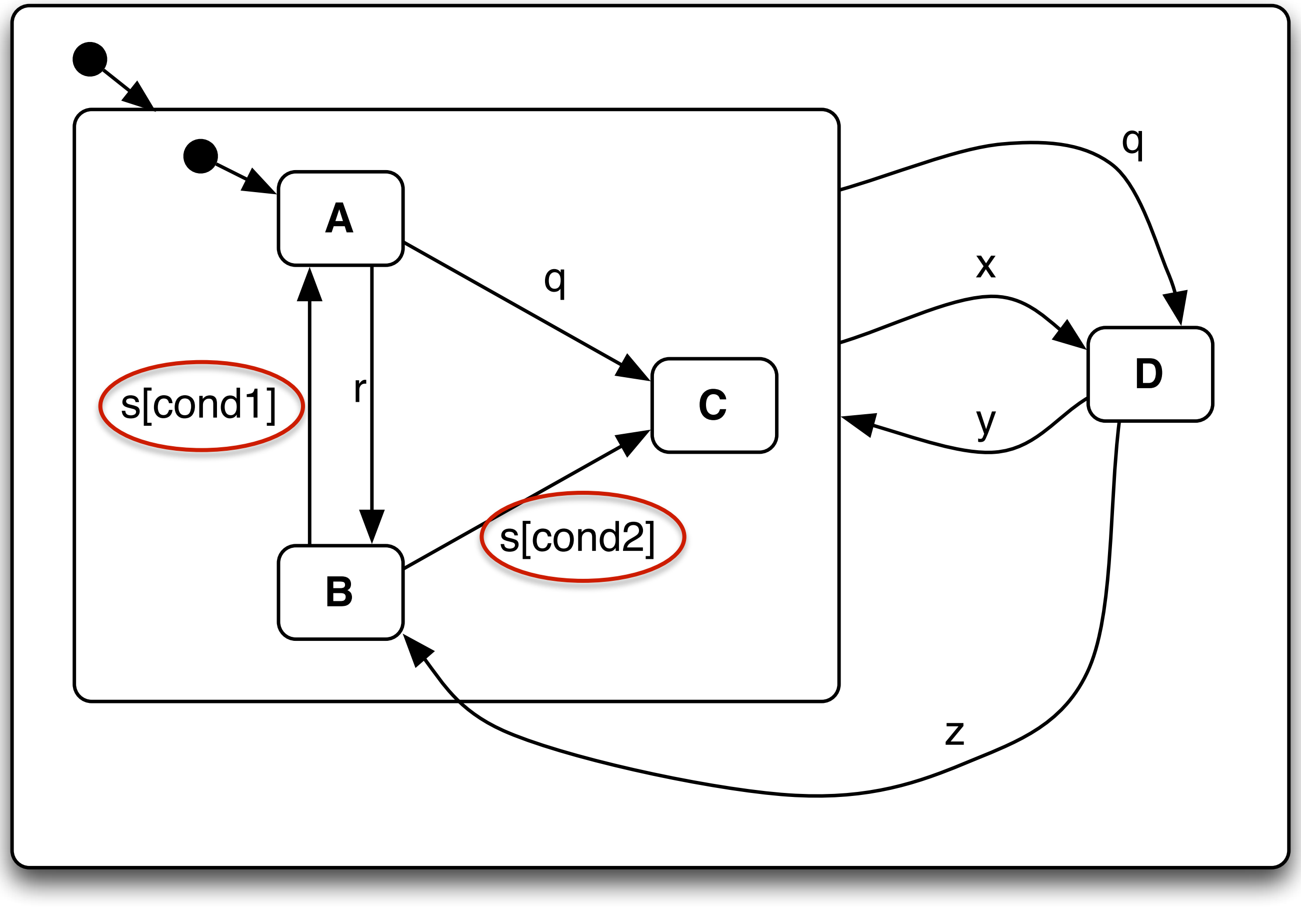
- what happens if event z occurs when in state D?
- what happens if event y occurs when in state D?
- can the execution ever leave state C?

Thermostat example



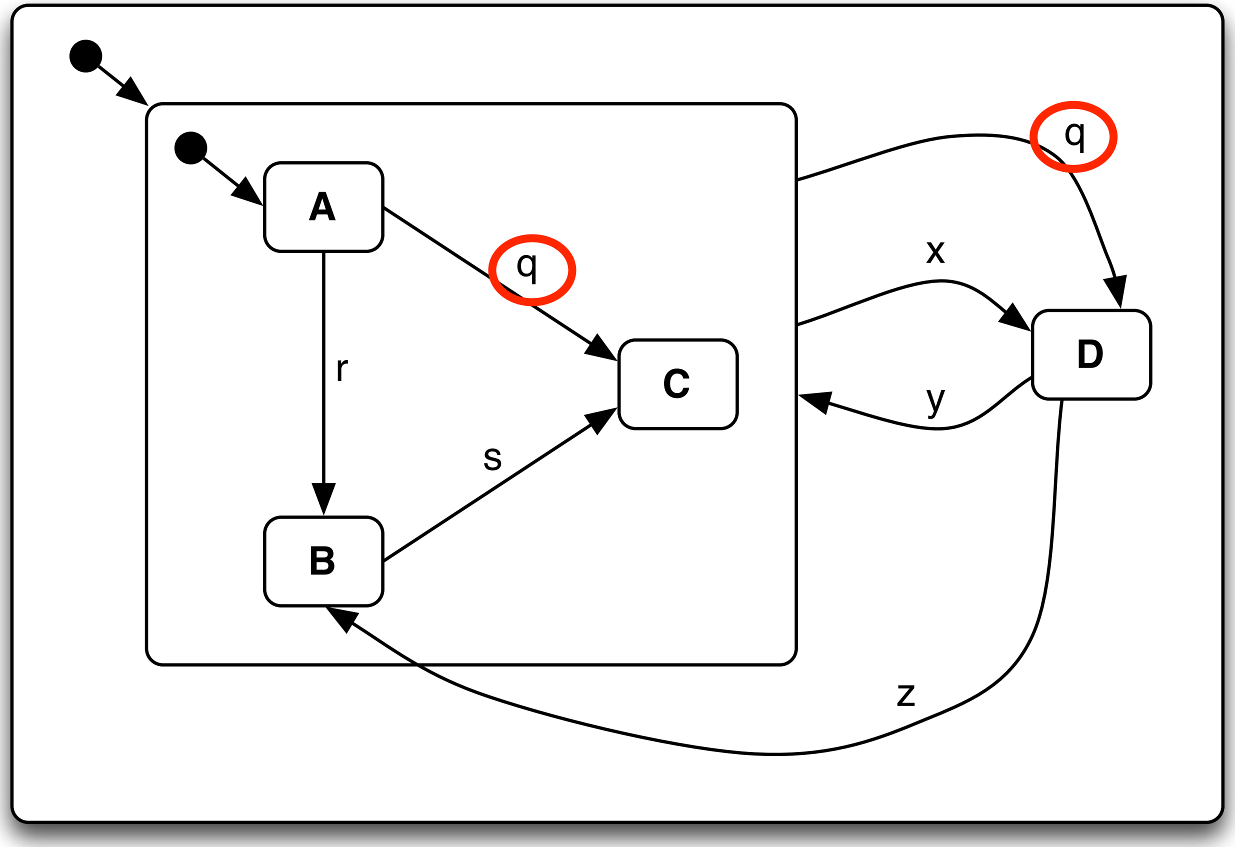
Determinism

What if the machine is in state B and event s occurs?



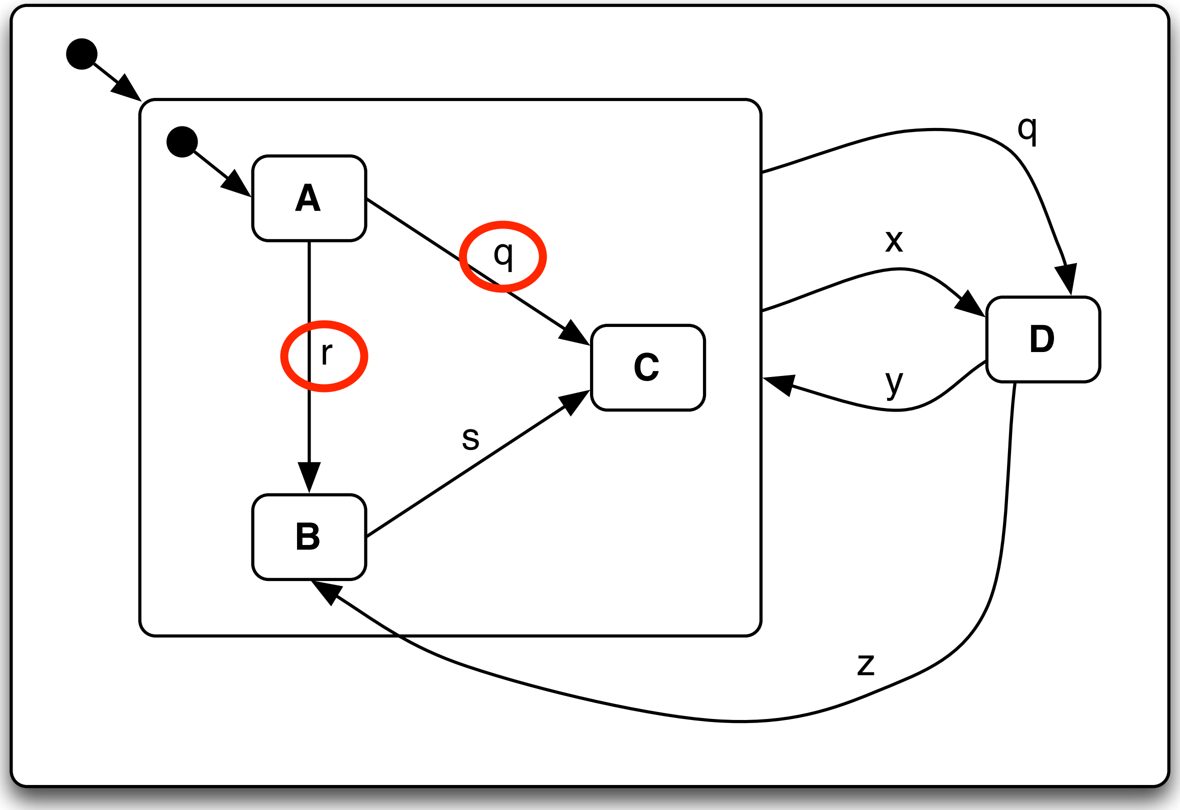
Priority

What if the machine is in state A and event q occurs?

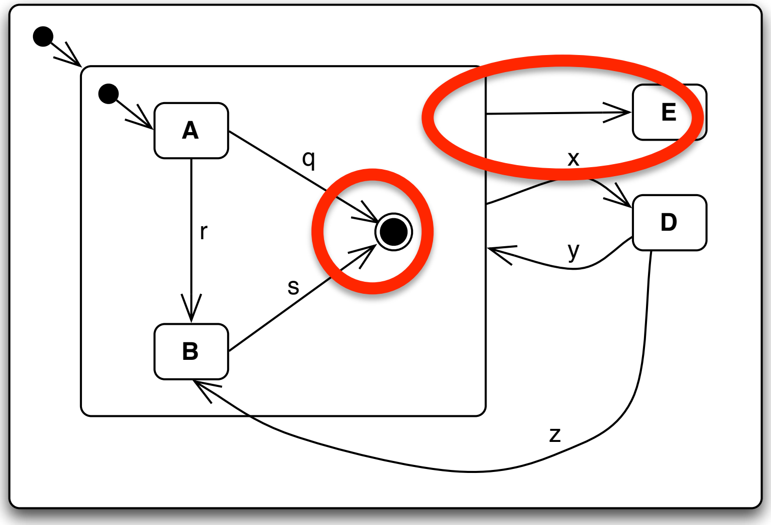


Priority

What if the machine is in state A and events q and r occur simultaneously?



Final state



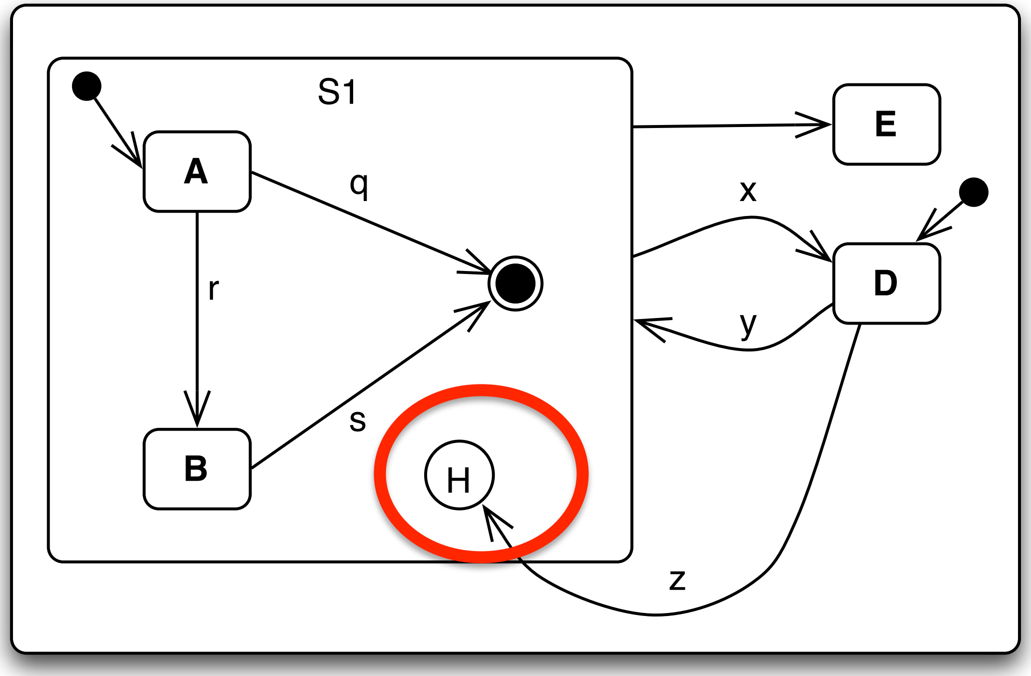
A final state represents the end of computation within a superstate.

A transition that has no event or condition in its label is enabled when its source state is idle.

- source state is a basic state
- source superstate entered its final state
- source basic state has finished internal activity

History

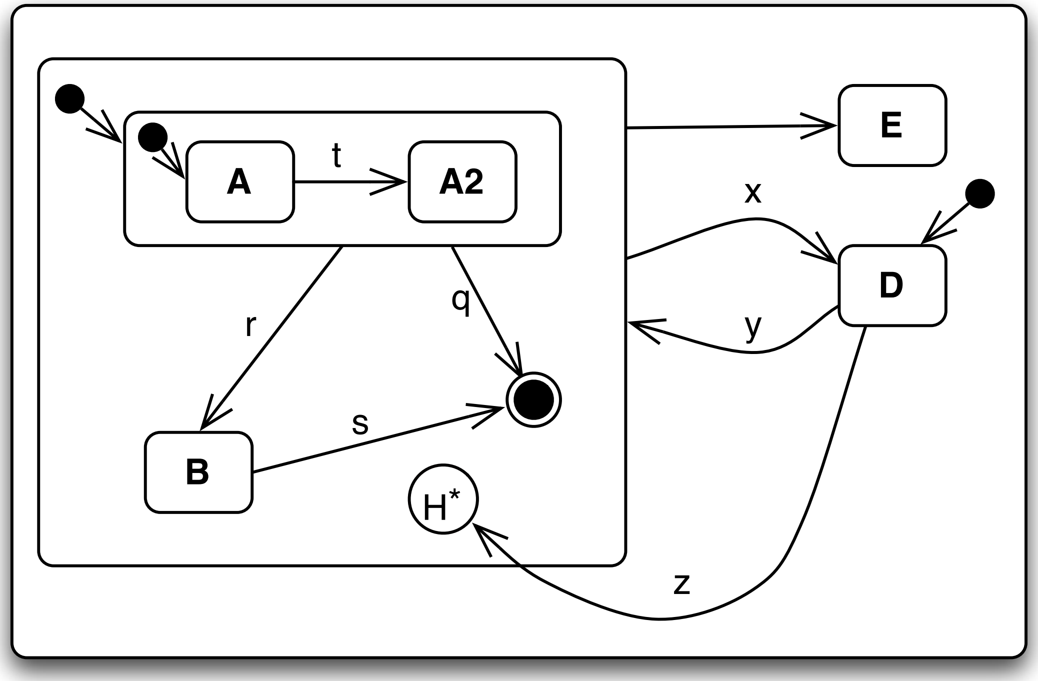
History Ⓗ is a pseudo-state that designates the child state of Ⓗ’s parent state (i.e., child state of S1) that the execution was in when the parent state (S1) was last exited.



Deep history: H*

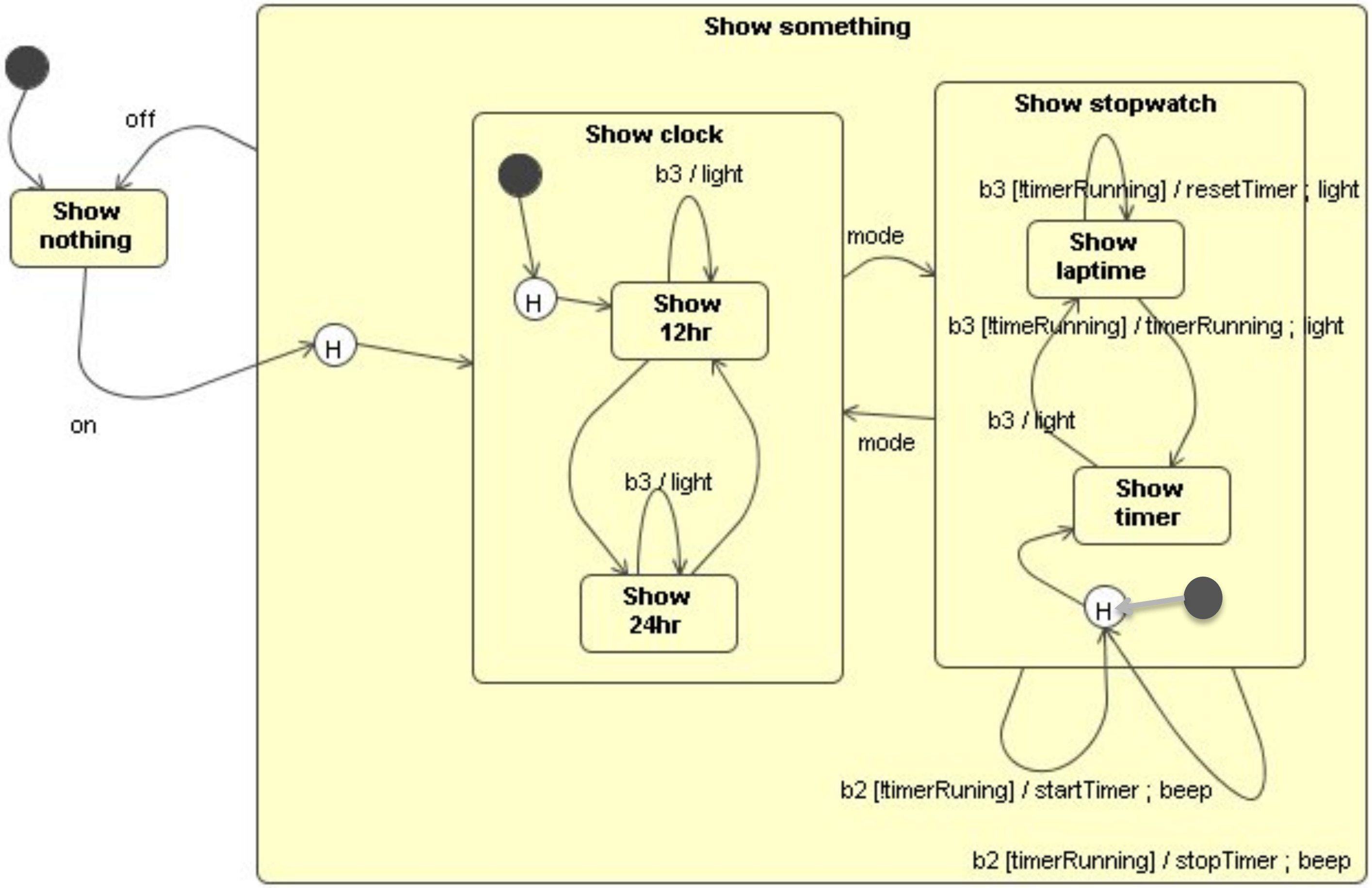
Deep history H* is a pseudo-state that designates the descendant state of H*'s parent state (i.e., descendent state of S1) that the execution was in the when the parent state (S1) was last existed.

s.



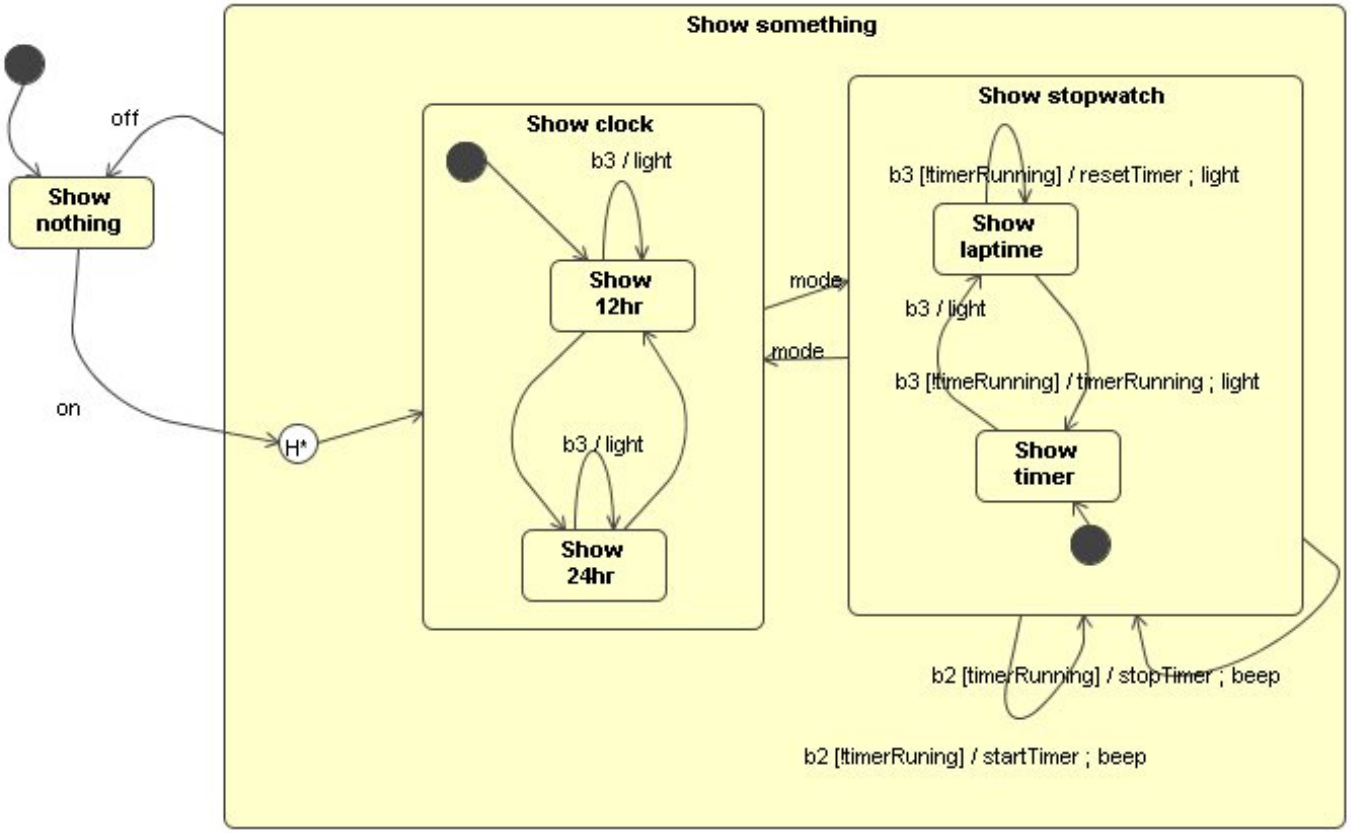
Stopwatch Example

What state is the watch in when it is turned on?


Show something
off
Show nothing
on
H
Show clock
b3 / light
H
Show 12hr
b3 / light
Show 24hr
mode
mode
Show stopwatch
b3 [!timerRunning] / resetTimer ; light
Show laptime
b3 [!timeRunning] / timerRunning ; light
b3 / light
Show timer
H
b2 [!timerRuning] / startTimer ; beep
b2 [timerRunning] / stopTimer ; beep

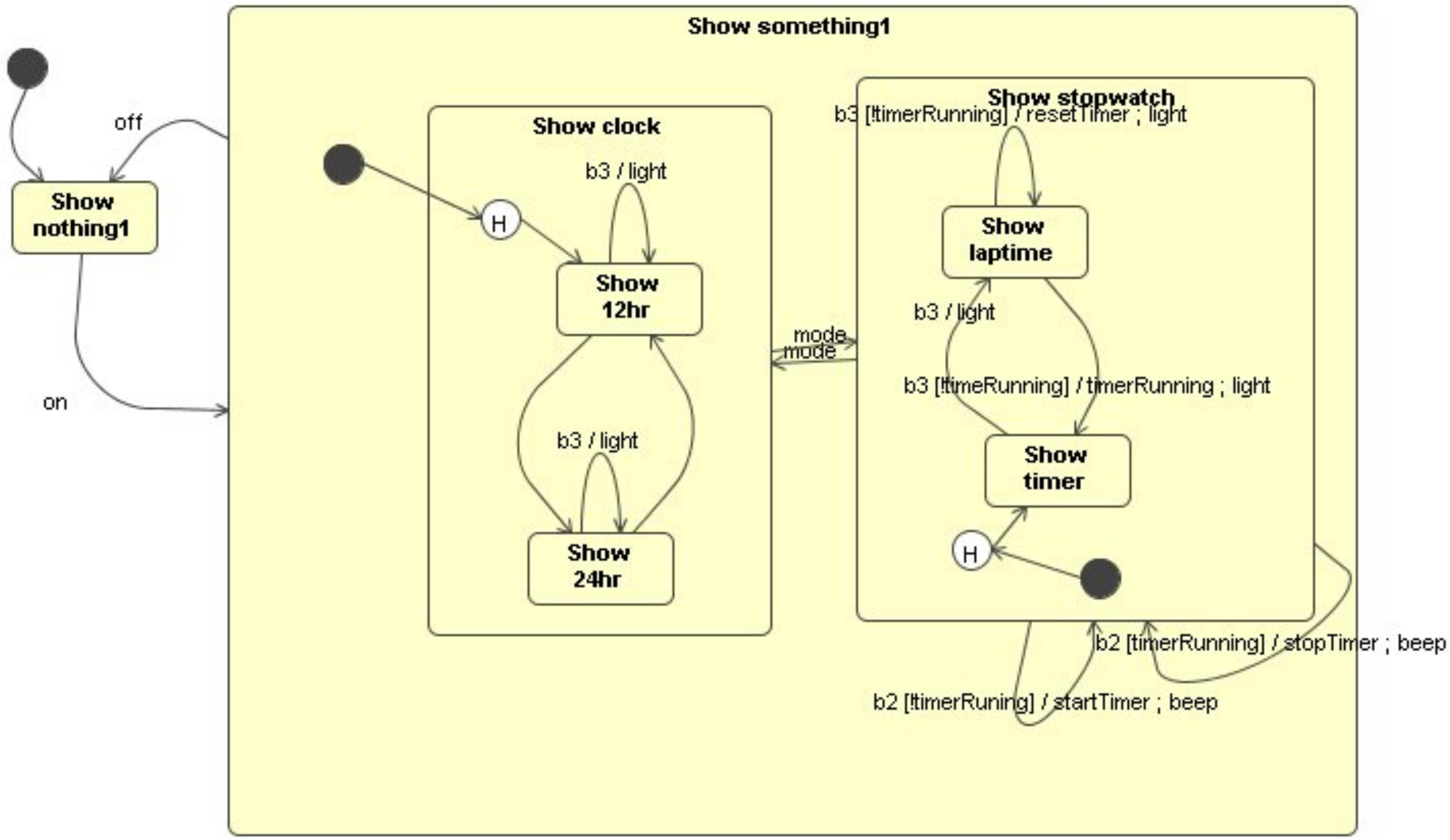
Stopwatch Example

Is this model equivalent to the previous model (i.e., do they represent the same set of execution traces)?


Show something
off
Show nothing
on
H*
Show clock
b3 / light
Show 12hr
b3 / light
Show 24hr
mode
mode
Show stopwatch
b3 [!timerRunning] / resetTimer ; light
Show laptime
b3 / light
b3 [!timeRunning] / timerRunning ; light
Show timer
b2 [timerRunning] / stopTimer ; beep
b2 [!timerRuning] / startTimer ; beep

Stopwatch Example

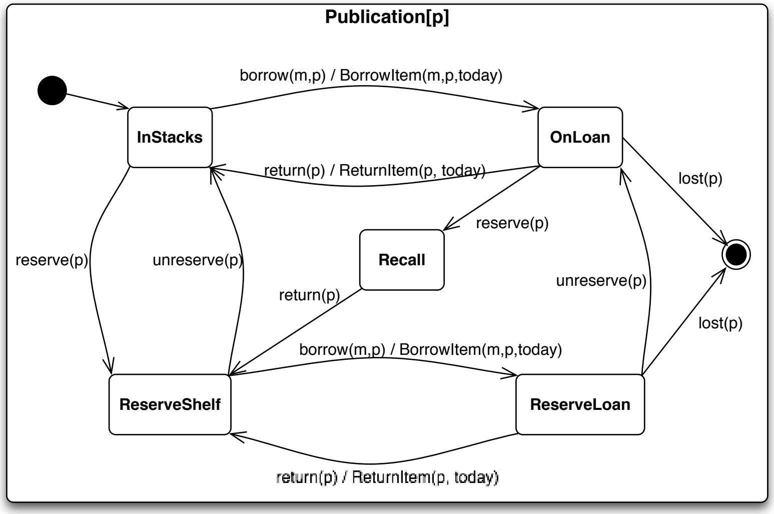
What state is the watch in when it is turned on?

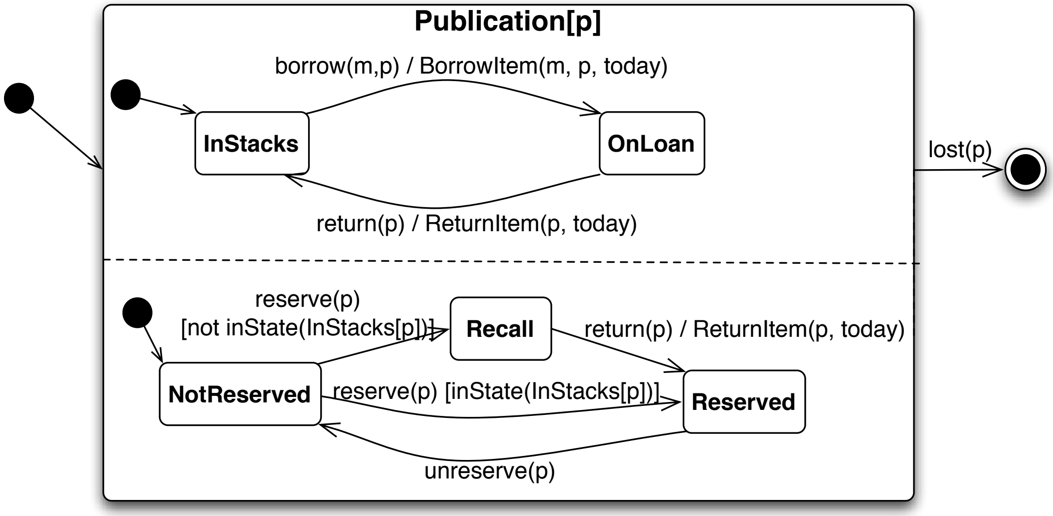

Show something1
off
Show nothing1
on
Show clock
b3 / light
H
Show 12hr
b3 / light
Show 24hr
mode
mode
Show stopwatch
b3 [!timerRunning] / resetTimer ; light
Show laptime
b3 / light
b3 [!timeRunning] / timerRunning ; light
Show timer
H
b2 [timerRunning] / stopTimer ; beep
b2 [!timerRuning] / startTimer ; beep

Concurrent regions

Some systems have orthogonal behaviours that are best modelled as concurrent state machines

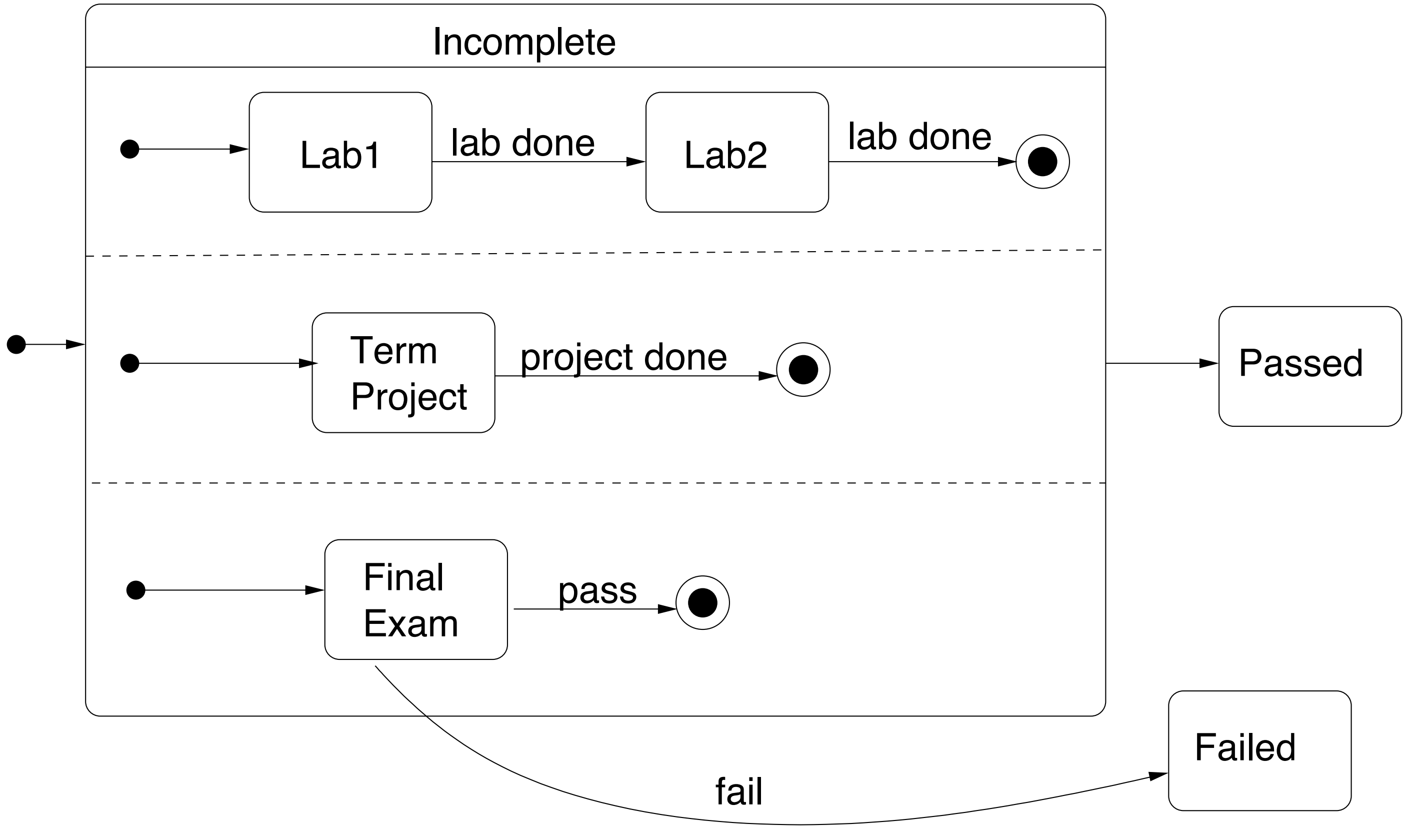
- *Regions* within a concurrent state execute in parallel.
- Each has its own thread of control.
- Each can “see” and react to events /conditions in the world

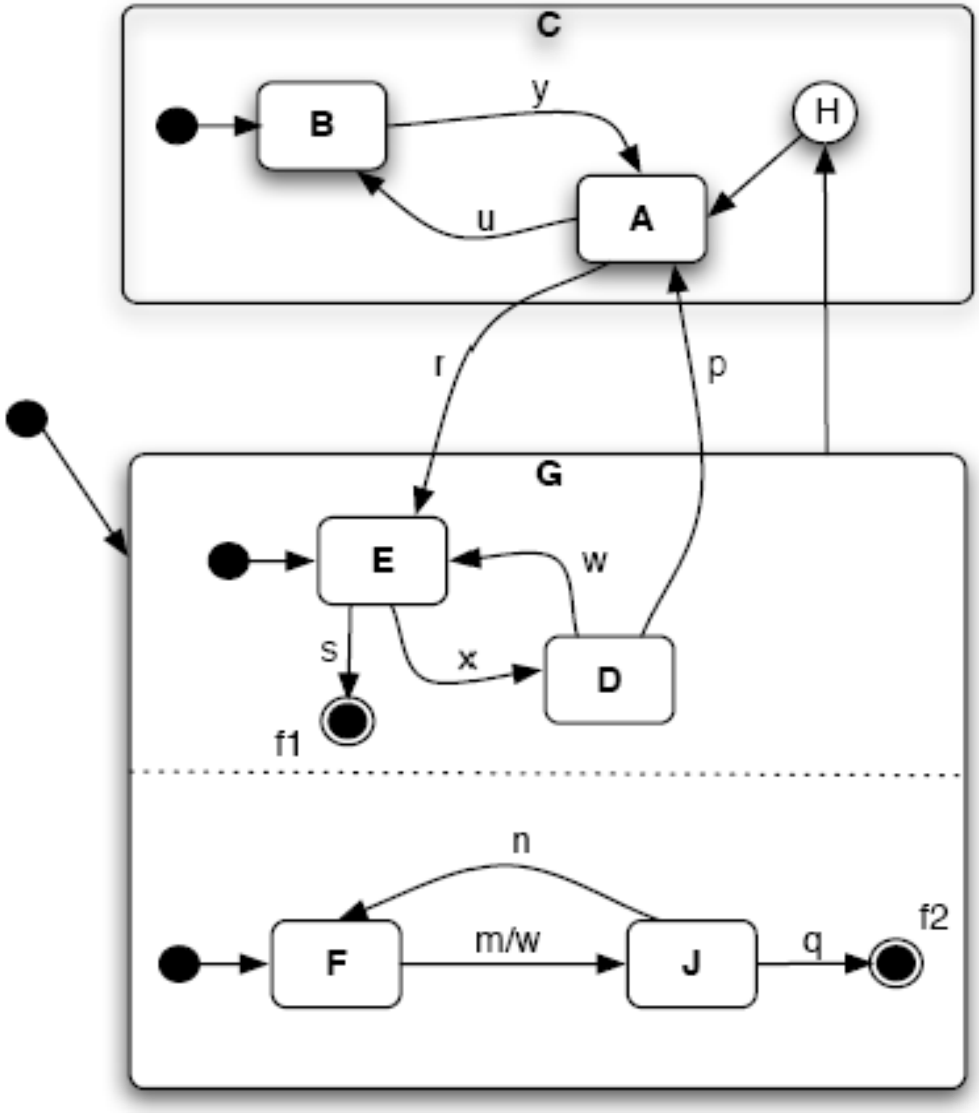

Publication[p]
borrow(m,p) / BorrowItem(m,p,today)
InStacks
OnLoan
return(p) / ReturnItem(p, today)
lost(p)
reserve(p)
Recall
reserve(p)
unreserve(p)
return(p)
unreserve(p)
lost(p)
borrow(m,p) / BorrowItem(m,p,today)
ReserveShelf
ReserveLoan
return(p) / ReturnItem(p, today)


Publication[p]
borrow(m,p) / BorrowItem(m, p, today)
InStacks
OnLoan
lost(p)
return(p) / ReturnItem(p, today)
reserve(p)
[not inState(InStacks[p])]
Recall
return(p) / ReturnItem(p, today)
NotReserved
reserve(p) [inState(InStacks[p])]
Reserved
unreserve(p)

Concurrency and Final States

Execution of a concurrent state is considered finished when all its regions are in their final states.

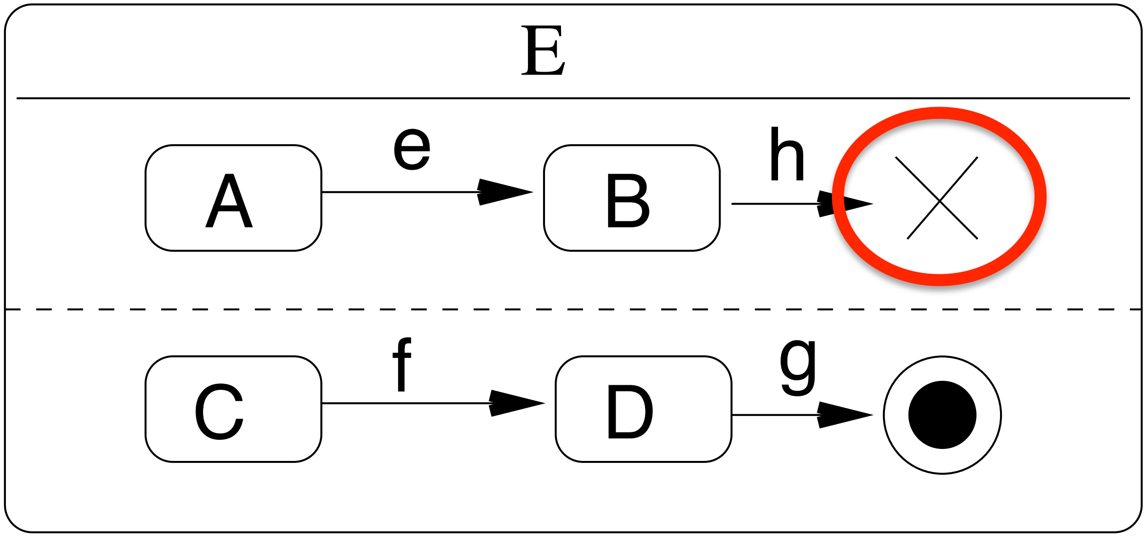



C
B
y
H
u
A
r
p
G
E
w
s
x
D
f1
n
f2
F
m/w
J
q

Termination

To represent the end of system execution, use the termination state.

- It's like a poison pill for the machine
- Not the same as a final state in a region, which waits patiently for the other regions to finish



Coordination of Regions

Ways to coordinate regions with each other

- Regions react to the same input event
- Regions react to the value of the same ≪interface≫ phenomenon
- One region generates an event that other regions react to
- A region's transition has guard [inState(x)] that evaluates to *true* iff another region is in state *x*,
  - [*NOT part of UML, just a useful convention!*]

Time event

A time event is the occurrence of a specific date/ time or the passage of time.

- Absolute time:
  - at (12:12 pm, 12 Dec 2012)

- Relative time:
  - after (10 seconds since exit from state A)
  - after (10 seconds since x)
  - after (20 minutes) // since the transition's source state was entered

Change events

A change event is the event of a condition becoming true.

- The event "occurs" when the condition changes value from *false* to *true*.
  - when (temperature > 100 degrees)
  - when (on)

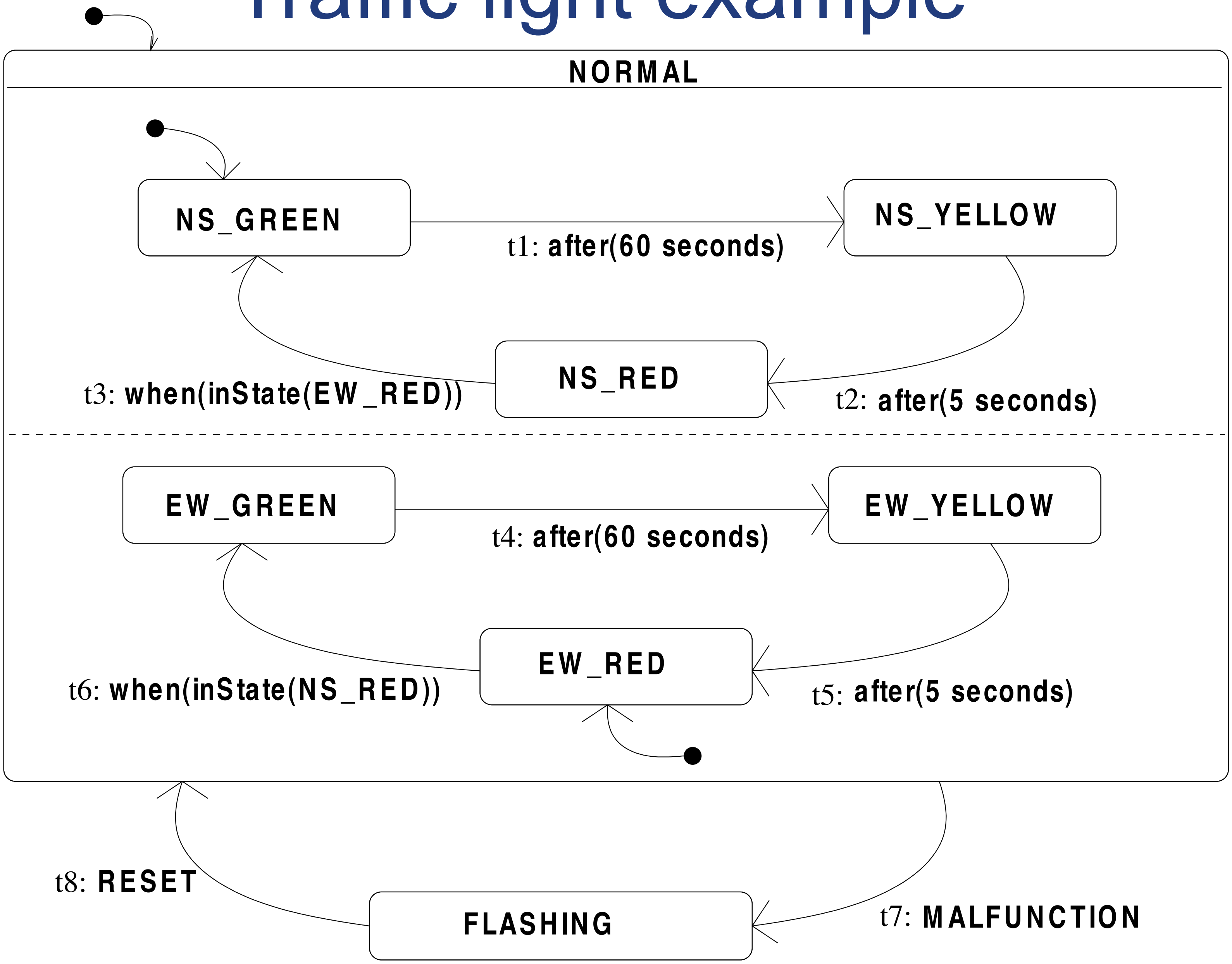
- The event does not reoccur unless the value of the condition becomes *false* and then returns to *true*.

when(X) vs. [X]

- A change event is what you want if you're sitting in a state waiting for a condition to become true

- A transition with a guard but no triggering event
  - called a "completion transition" in UML
  - semantics:
    - guard is checked once (when the source state becomes idle)
    - transition fires if the guard is true
  - guard is not checked again
  - not likely what you want to say

Traffic light example

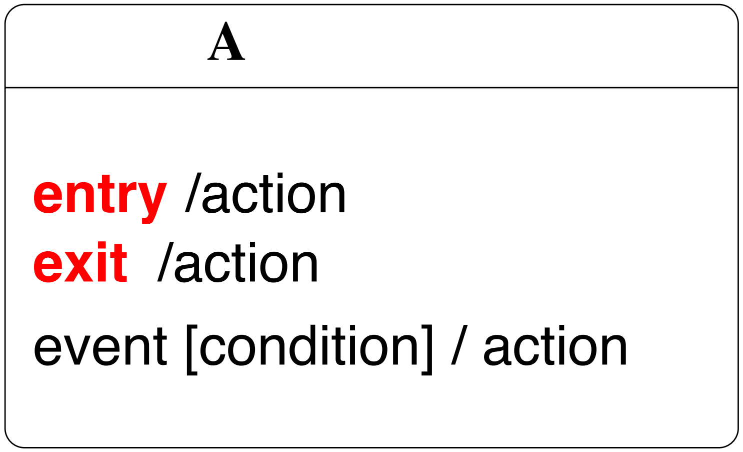


State Actions

Recall: An action is uninterruptible.

States can also be annotated with entry or exit actions, and with internal actions.

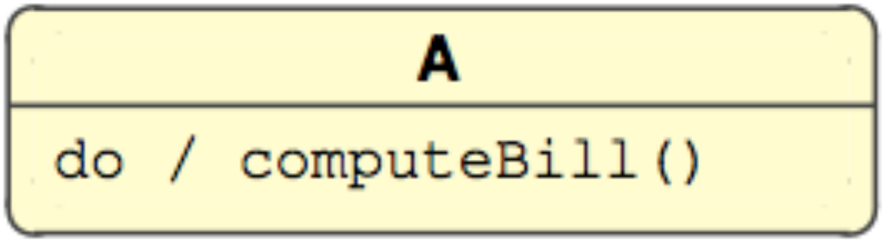
- Entry actions: – actions that occur every time the state is entered by an explicit transition.
- Exit actions: – actions that occur every time the state is exited by an explicit transition.
- Internal actions: on events



State Activities

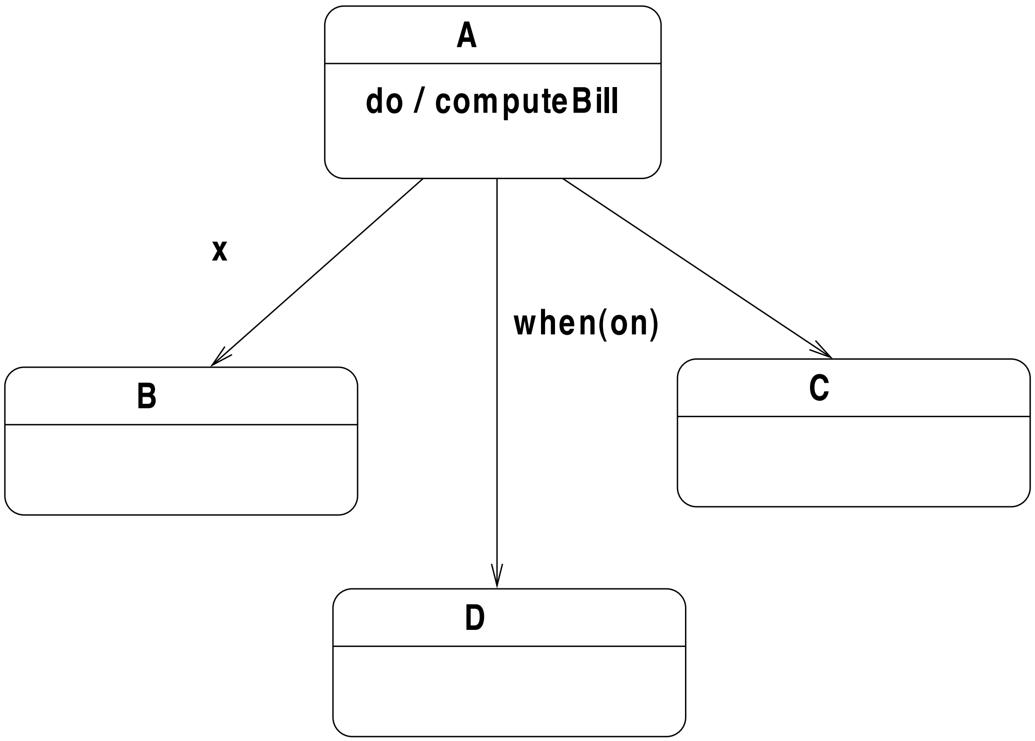
An activity is computation of the system that takes time, and can be interrupted.

- c.f., an action, which is uninterruptible
- An activity may be associated with a state.
- States with activities are called *activity states*.



State Activities

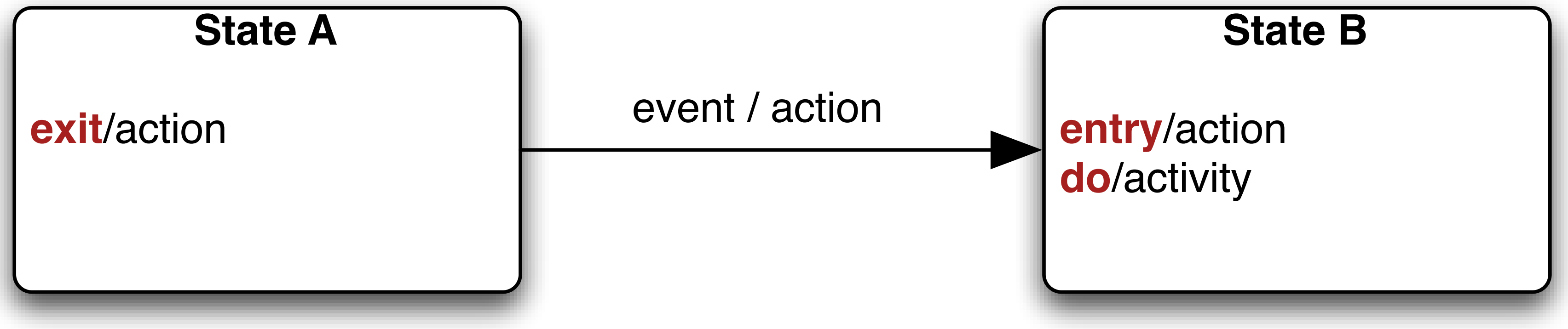
Because activities take time, they can be interrupted by transitions with triggers and/or conditions



Actions vs. Activities

- UML 2.0 has dropped actions ☹
  - This is too bad because the synchrony hypothesis is a very useful, valid, and reasonable simplifying assumption for requirements modelling.
  - Therefore, we will use actions in our state machine models anyway. So there.

Execution Order



In an explicit transition (including self-looping transitions!), the order of effects is:

1. exit actions of source state, then
2. transition actions (in listed order), then
3. entry actions of destination state, then
4. state activities.

Creating a Behaviour Model

1. Identify input and output events

2. Think of a natural partitioning into states
   - *Activity states* – system performs activity or operation
   - *Idle states* – system waits for input
   - *System modes* – use different states to distinguish between different reactions to an event

3. Consider the behaviour of the system for each input at each state.

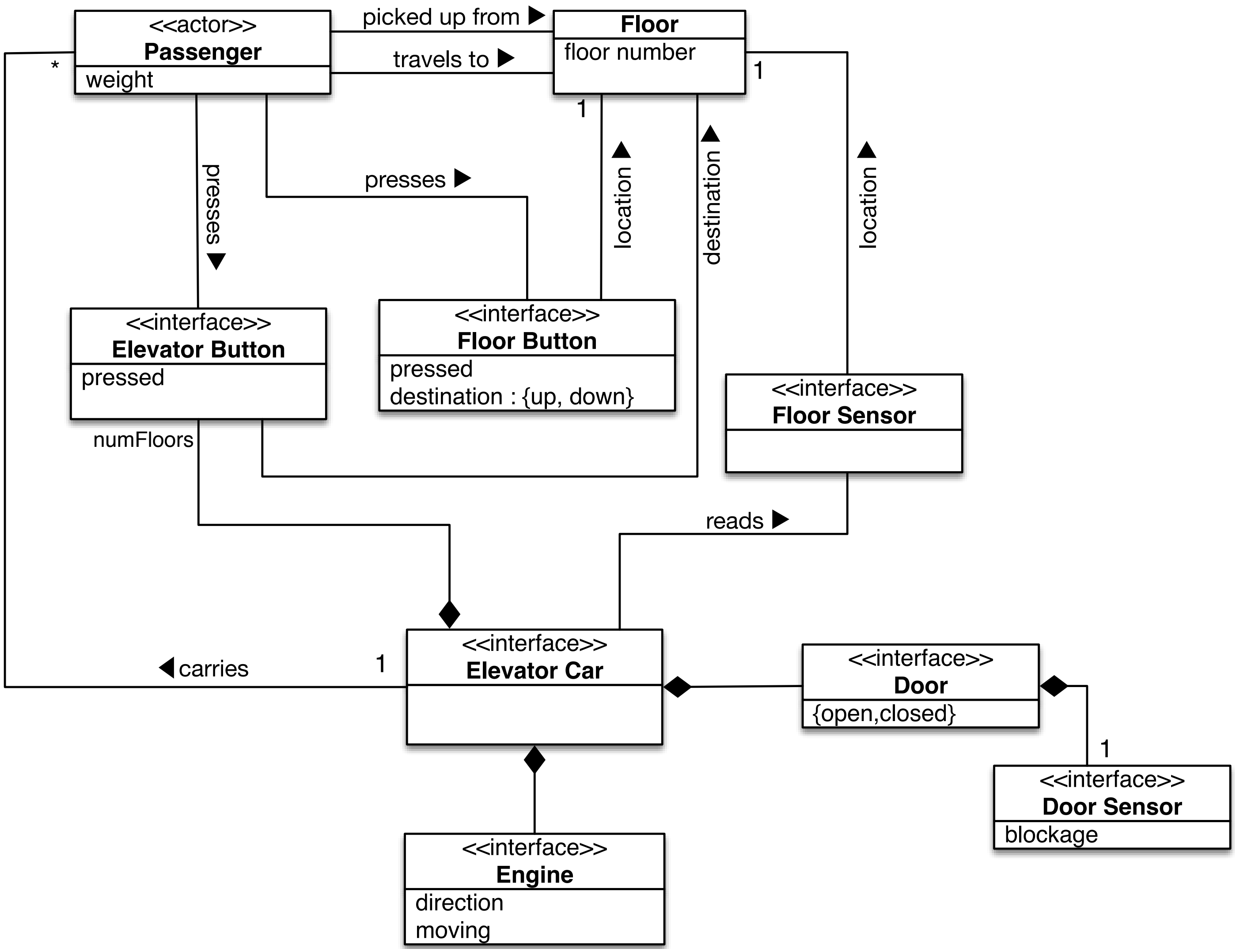
4. Revise (using hierarchy, concurrency, state events)
   - Use concurrency to separate orthogonal behaviour
   - Use hierarchy, and entry/exit actions, to abbreviate common behaviour

Example: Elevator

- An elevator passenger who wants to travel from one floor to another higher (lower) floor presses the Up (Down button at his current floor.
- The light beside the button must then be lit, if it was not lit before.
- The elevator must arrive reasonably soon, travelling in an upwards direction.
- The direction of travel is indicated by an arrow illuminated when the elevator arrives.
- The doors must open, and stay open long enough for the passenger to enter the elevator.
- The doors must never be open except when the elevator is stationary at a floor.

Michael Jackson, *Software Requirements and Specifications, Addison-Wesley, 1995.*

Elevator



<<actor>>
Passenger
weight
picked up from ▶
travels to ▶
Floor
floor number
*
1
1
presses
presses ▶
location
destination
location
<<interface>>
Elevator Button
pressed
<<interface>>
Floor Button
pressed
destination : {up, down}
<<interface>>
Floor Sensor
numFloors
reads ▶
◀ carries
1
<<interface>>
Elevator Car
<<interface>>
Door
{open,closed}
1
<<interface>>
Door Sensor
blockage
<<interface>>
Engine
direction
moving

Good style

The best state machine model is usually the one that is the clearest.

- Use states to model modes of operation, and use variables to model other information that affect flow of control
- Fewer transitions are better
  - i.e., Use hierarchy to reduce the number of transitions
- Use history + deep history
- Use concurrency to recognize orthogonal aspects of the problem

Common problems

- Over-specification:
  - Don't build an executable model
    - Model the possible flows of control, not the possible computations
    - e.g., don't keep track of the actual time in stopwatch model
- Under-specification:
  - Do specify a response to every event that is relevant at a state.

Validating Behaviour Models

- Avoid inconsistency: multiple transitions that leave the same state under the same event/conditions.

- Ensure completeness: specify a reaction for every possible input at a state.
    - If there are transitions triggered by an event conditioned on some guard, what happens if the guard is false?

- Walkthrough: compare the behaviour of your state diagrams with the use-case scenarios.
    - All paths through the scenarios should be paths in the state machines.

Summary

State Machine Models

- applied to a world state (of the domain model)
- hierarchy
- concurrency
- state actions / activities