CS445 / SE463 / ECE 451 / CS645

Software requirements specification & analysis

UML state machine diagrams

Fall 2013 — Mike Godfrey, Dan Berry, and Richard Trefler

UML state machine diagrams

- Shows finely-grained component/program behaviour
- Useful for describing the inner behaviour of a class that conforms well to the state-transition paradigm:
  - Finitely many discernable inner states
    - Waiting for input, mid-transaction, idle
  - Responses highly dependent on internal state
    - Only after all critical data fields filled in will system allow transition to "confirm payment" state
    - Events only relevant in certain situations

    e.g., Only first "walk" button press is significant

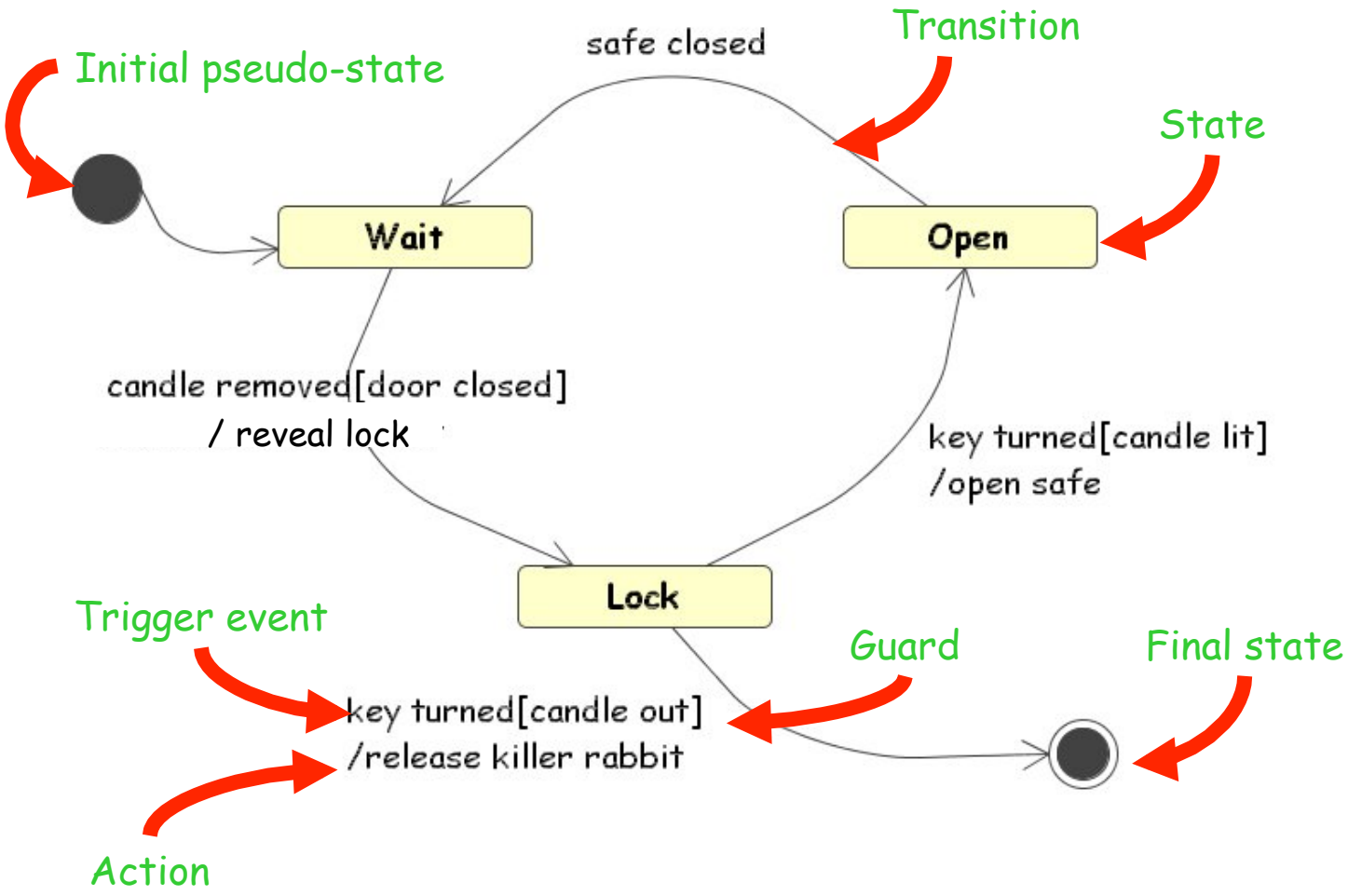
UML state machine diagrams

- SM diagrams commonly used in design to describe object's behaviour as a guide to implementation
- Used in RE to model interface specs (e.g. UI)
- Other RE use of SM diagrams:
  - Specify each object's contribution to all scenarios of all use cases
  - May be too detailed a model for RE

The state-transition paradigm

- Many programs you have written in previous programming courses do not conform well to this paradigm
  - They may have infinitely many possible abstract inner states (defined implicitly by their instance variables)

An example *[Fowler p108]*

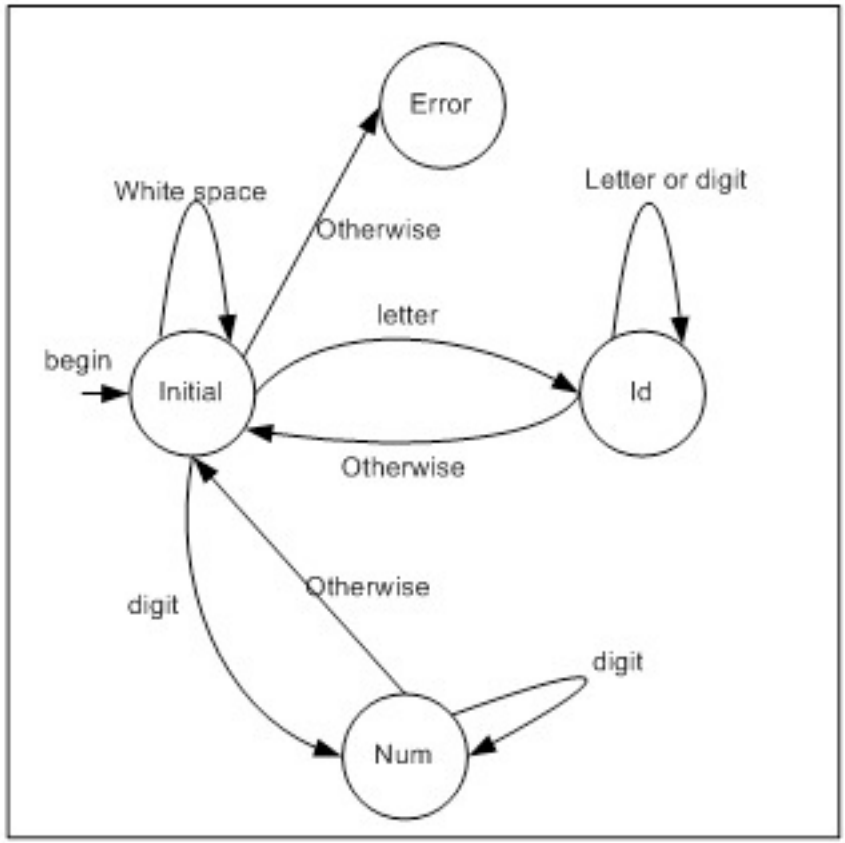


A UML state machine

- … is a *hierarchical, concurrent, extended finite state machine*:
  - Shows the lifecycle of an *instance* of an object
    - The object starts in a given state, and transitions to other states based on external "messages" (events detected) received and the values of its internal variables
  - Describes the behaviour of an object across *multiple* (perhaps all!) use cases.
  - Hierarchical — each state may be broken down into sub-state machines
  - Supports concurrent regions
  - Extended — allows variables that augment the state descriptions

Example FSM

- You might have seen an FSM like this to represent a grammar in CS241
  - This FSM processes a character stream to produce a token stream with white space removed
- "Finite State Automaton" (FSA) is another term for FSM.



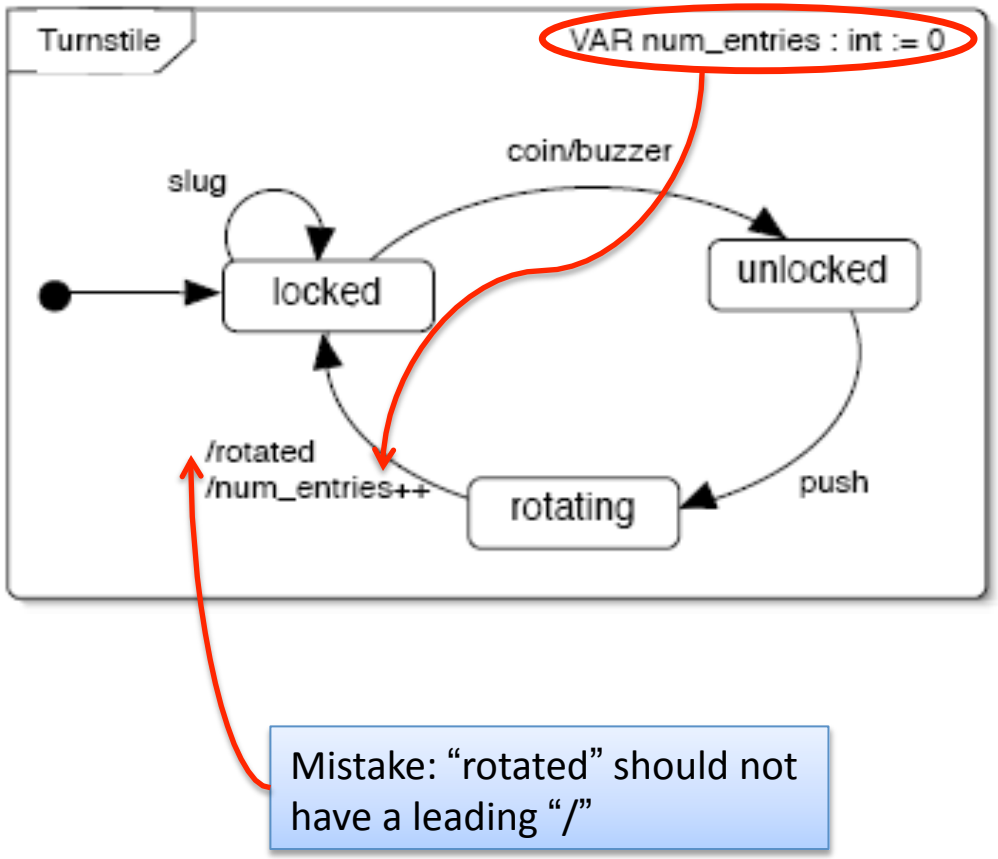
http://www.codeproject.com/KB/recipes/Parser_Expression.aspx

FSM — Review

- $A = (Q, \Sigma, \Delta, q_0, F)$ where
  - $Q = \{q_0, q_1, \ldots, q_n\}$ is a finite set of states.
  - $\Sigma = \{a_1, \ldots, a_m\}$ is a finite set of letters.
  - Δ is a transition function mapping pairs in $Q \times \Sigma$ into Q, i.e., $\Delta: Q \times \Sigma \rightarrow Q$,
  - q_0 is the start state for A, and
  - $F \subseteq Q$, is the set of accepting (i.e., final) states.

Extended FSMs

- An *extended finite state machine* (EFSM) is one that includes variables.
- Transitions can depend on the value of conditions (expressions on variables).
- Outputs can be sent messages or assignments of values to variables.



ESFMs and variables

- Variables are used to reduce the number of states in the model.
  - In the example, without variables, we'd need a distinct state to represent different values of the number of entries recorded.
- The resulting model may no longer be finite, strictly speaking
  - So sometimes we just say “extended state machine”
- UML state machines are EFSMs
  - i.e., you can use variables within states

Requirements model

The way to keep the model at the requirements level is to restrict it to the vocabulary of the Interface part of your Domain Model.

You would have one state machine diagram for each Interface class, describing the use cases it offers to the actors.

Requirements model: Validation

If you build such a model, you can subject it to a very useful kinds of verification or validation:

Walk thru each scenario, and make sure that the system's response for each user input is specified and agrees with what happens in the scenario

States

- A *state* normally represents a moment in time when the system does not change and is waiting for another input before the system changes.
  - In response to events and conditions, the system follows transitions to change states.
- States partition the behaviour of the system:
  - In different states, the system reacts differently (or not at all) to events.
    e.g., not being able to check out a borrowed book
  - The state an object is in affects what input the object will react to
    e.g., ignoring most input in the state OFF
- A state (incl. the values of its variables) represents a partial history of inputs/outputs so far.

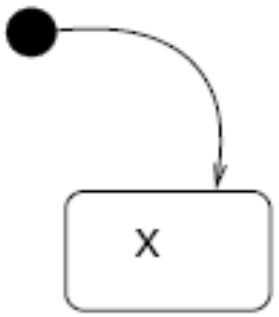
What kinds of systems are amenable to SM modeling?

Systems that are state-machiny, e.g.
VCRs
traffic lights,
etc.

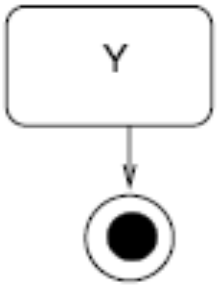
But NOT systems that model
continuous functions, e.g.,
weather forcasting,
driving a vehicle,
etc.

States and pseudo-states

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



X is the initial State



Bullseye is a final state

Events and transitions

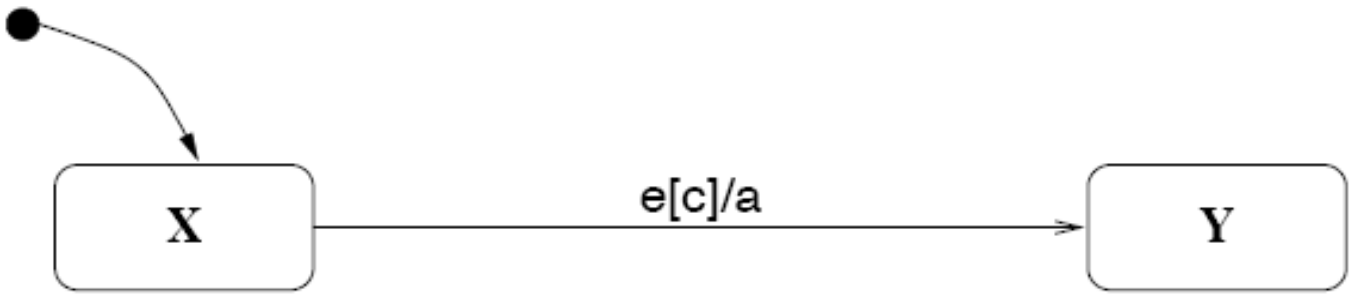
- An *event* is *"a significant or noteworthy occurrence"* [Larman]
  - An event may make an object *transition* to a different state
  - An event may cause the object / system to perform an action
  - An event is considered to occur instantaneously — it doesn't persist.
  - Multiple events on a transition label are alternative triggers. That is, any of the listed events can trigger the transition.

- In a requirements model, an event is often a message from the environment that something of interest has occurred

  e.g., "off-hook", "coin", user enters info through UI, timer goes off, API call from external software system

- In a design model, an event can be a message/method call from another object within the system

Transitions

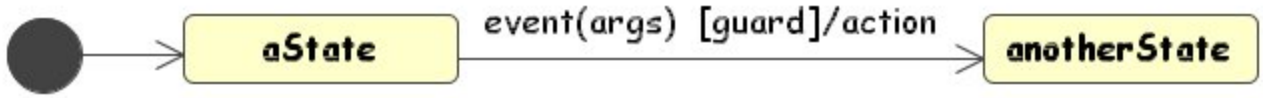


Semantics:

- When in state X, if event e occurs and condition c is true, carry out and complete action a and move to state Y.

- If in a state and there is not an outgoing transition triggered by a received event, the event is ignored.

Transitions



- Each of these parts of the transition is optional.
  - *event(args)* — event / message that triggers the transition
  - *[condition]* — (boolean) guard condition; the transition cannot fire unless the guard condition is *true* (can use *args* in guard expression*)*
  - */action* — a simple, fast, non-interruptible action (can use *args* in action body),

    e.g., variable assignment,

    send a message to an object: **`Object.event(args)`**

Conditions

- A *condition* is a Boolean expression whose value depends on the value of variables.

- The value of a condition persists until the variables involved in the condition change their values, e.g.,
  - *x > 10*
  - *DoorIsClosed*

- Conditions on transitions leaving the same state should be mutually exclusive
  - … so that no two transitions can be simultaneously enabled

State actions and activities

- A state can have *actions* and *activities* associated with it.
  - State actions and activities can manipulate object attributes or other variables.

- *Action*: instantaneous, non-interruptible, simple. It can be:
  - associated with a transition, or
  - performed on state entry or exit.

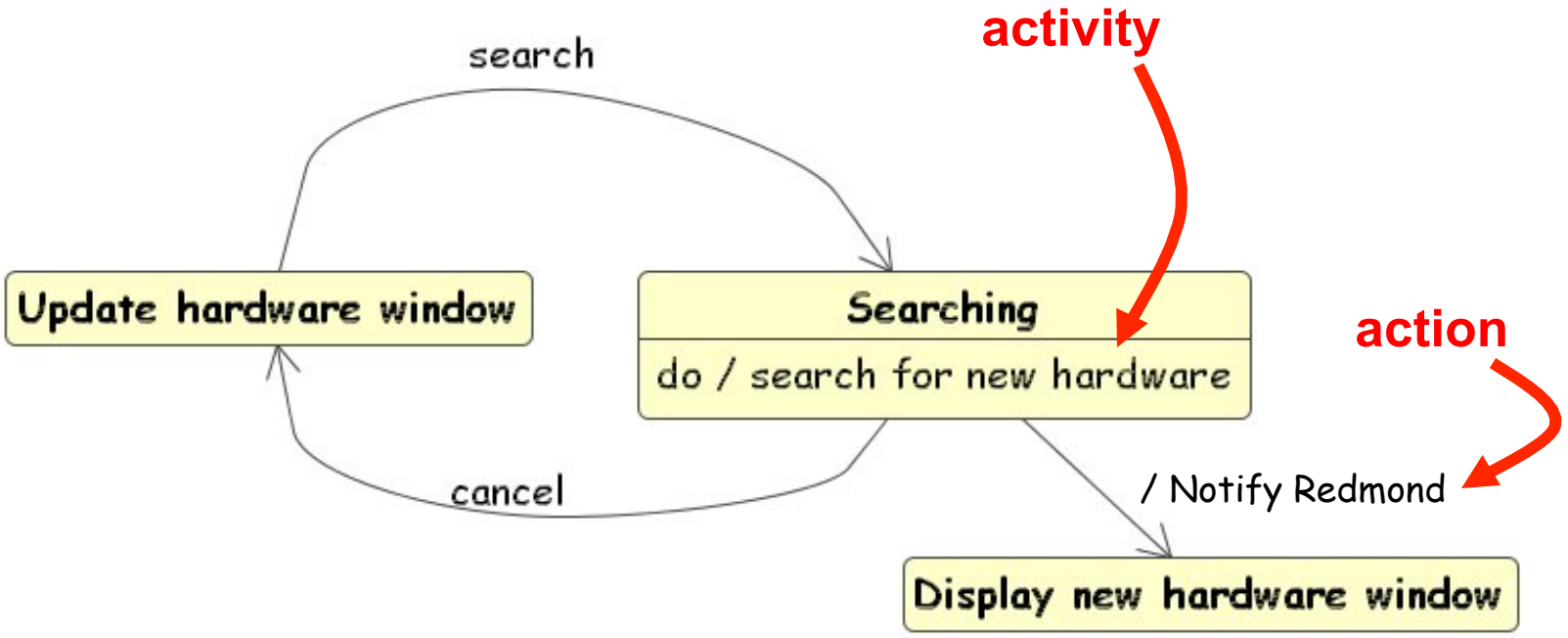
- *Activity*: takes time, interruptible, may require computation. It can be:
  - associated with a state, and
  - can be interrupted by a transition.

- In UML 2.0, the terminology is different. As defined above:
  - Actions are known as "regular activities"
  - Activities are known as "do-activities"

Actions

- Actions are what the system does in response to events
  - … in addition to changing state
- Most common actions:
  - Send a message/event to the environment
    e.g., *setTone(. . .)*
  - Change the value of a variable
    e.g., *x := 5*
- An action is non-interruptible (i.e., atomic)
  - It completes before the destination state of the transition is entered.
- Multiple actions on a transition are separated by “;” and executed sequentially.

An example



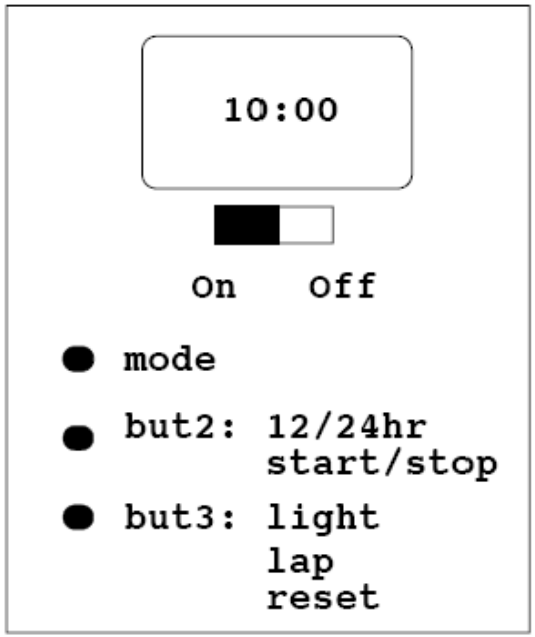
[Fowler p110]

States (again)

- States make the requirements easier to understand by partitioning the behaviour of the system into *modes*:
  - The reaction of the system to the same event may be different in different states.
  - In some states, there may be no response to certain events.

- Modes you may already know and love:
  - Setting up a clock radio or DVR
  - Moded editors like vi/vim (versus modeless like emacs and most IDE editors)
  - Navigating through UIs (what happens if you hit return?)

Recall stopwatch example



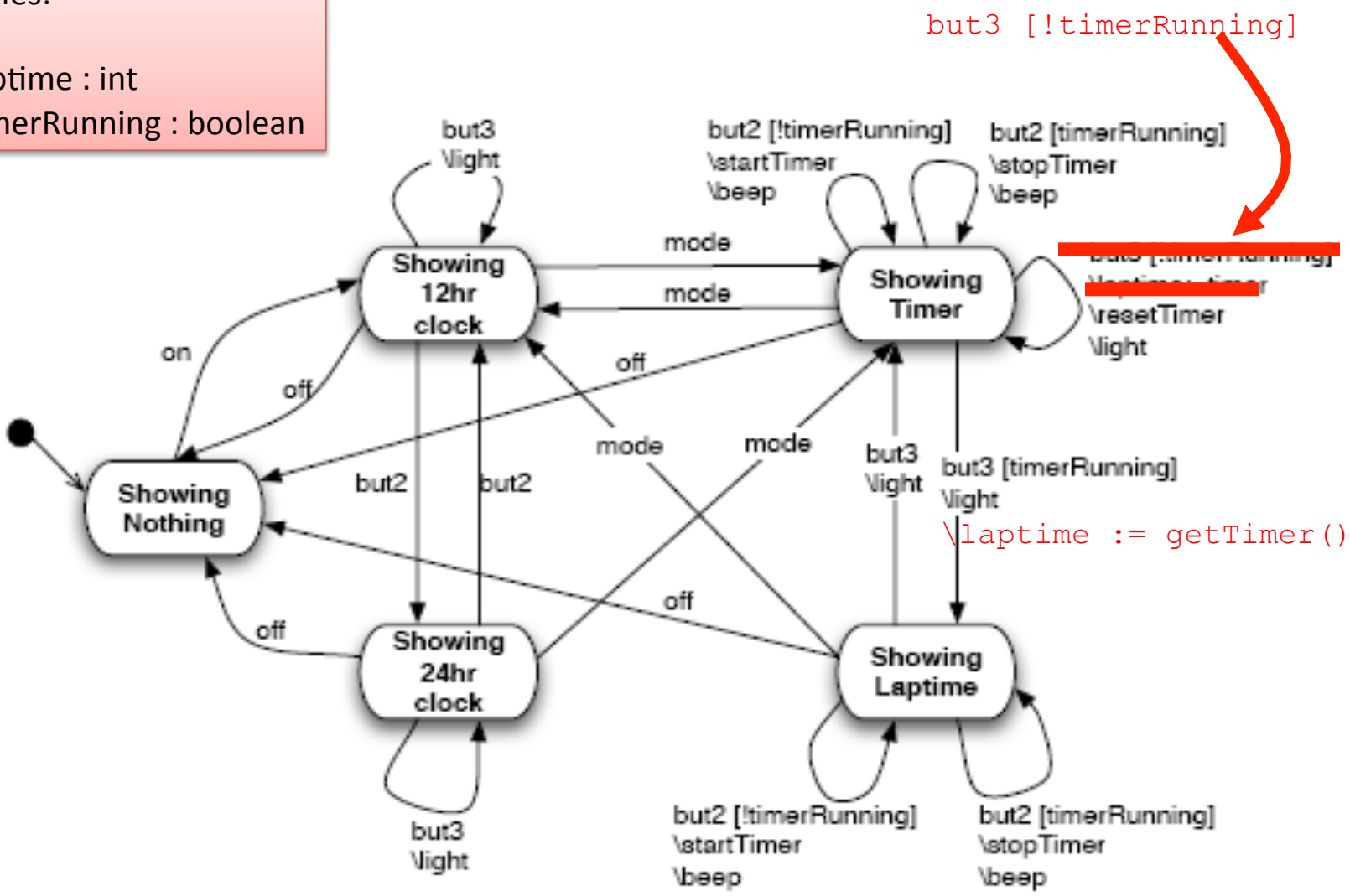
Action	Meaning
on	Turn watch on
off	Turn watch off
mode	Toggle between time and stopwatch
but2 [time]	Toggle between 12h and 24h display
but2 [stopwatch]	Start / stop timer; beep for 0.25 sec
but3 [time]	Turn light on for 3 sec
but3 [stopwatch, timer running, display timer]	Record laptime; display laptime; turn light on for 3 sec
but3 [stopwatch, timer stopped, display timer]	Reset timer; turn light on for 3 sec
but3 [stopwatch, display laptime]	Display timer; turn light on for 3 sec

Recall stopwatch example

- Stopwatch starts in off state
- When "off",
  - Display is turned off
  - Battery continues to power an internal "wall clock"
  - Last value of timer is kept in memory, but timer is turned off (if it was running)
- When powered on,
  - Display is turned on
  - The timer is off (but not reset)
  - Default initial display is 12 hour wall clock time
- Hardware has built-in timer mechanism
  - Can start/stop/reset/get value
- Starting/stopping the timer should cause an audible beep for 0.25s
  - Hw supports "beep", but is not tied to start/stop by default

Variables:

var laptime : int
var timerRunning : boolean



Validation

- Given the list of possible events, for each state X, consider whether each event e is possible. It could be the case:

1. There is a transition on e from state X

2. Event e cannot physically occur in state X
   - no transition on e is needed from X

   e.g., doorOpened cannot occur when the door is already open

3. Event e is possible but the system should ignore it
   - no transition on e is needed from X

   Self loop on X…i.e., the system does not change if event e occurs in state X

   e.g., multiple “door close” button presses; only first one is significant

4. Event e is possible in state X, but the system should report as error
   - a transition is needed to report error

Common problems

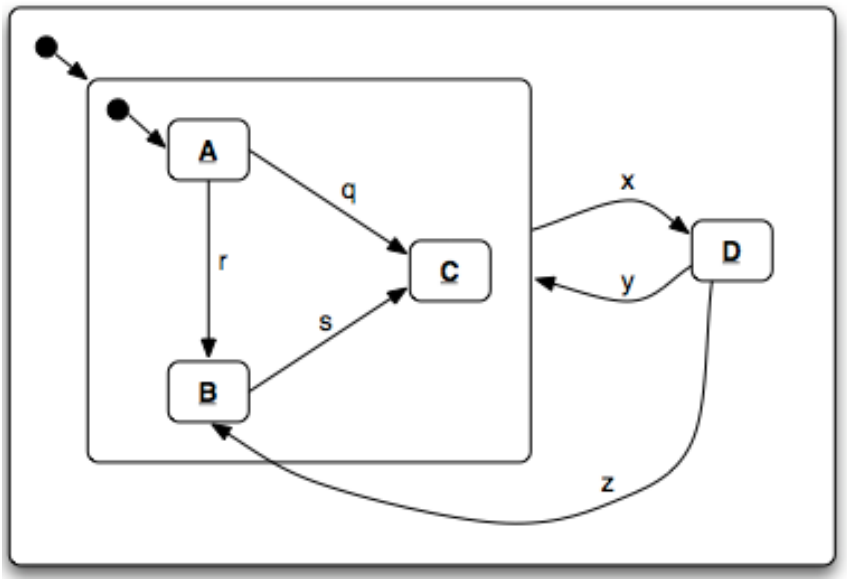
- *Over-specification:*
  - Specifying a response to an event that can't occur in the state
    - … in an attempt to ensure that the specification is complete
  - Trying to maintain enough state information (e.g., by variables) to always know the system's exact response to an input.
    - Keeping track of the number of active phone calls, so that the state machine model can detect when a set limit has been reached

- *Under-specification:*
  - Not specifying a response to an event that is relevant at a state, thereby leaving out requirements of the system.

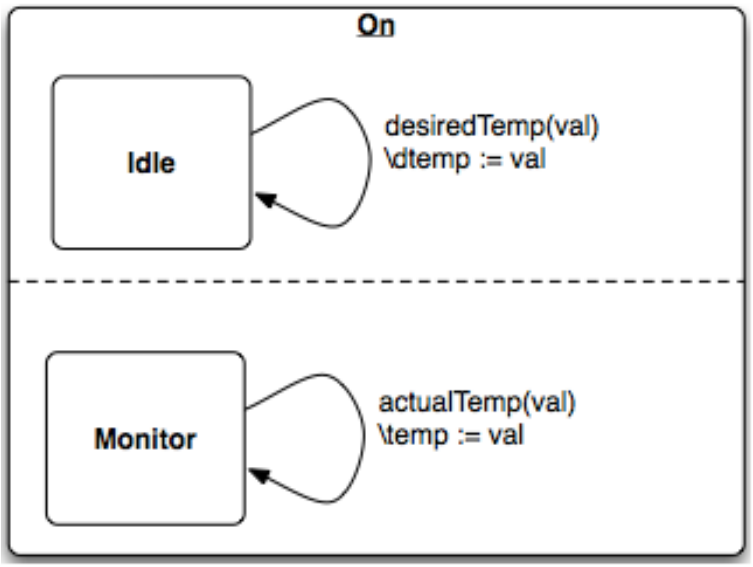
Composite states

- A *composite state* combines states and transitions that work together towards a common goal. There are two kinds:
  1. Hierarchical (aka "simple" / "OR-states")
  2. Concurrent (aka "orthogonal" / "AND-states")
- A state that does not contain other states is called a *basic state*

Hierarchical

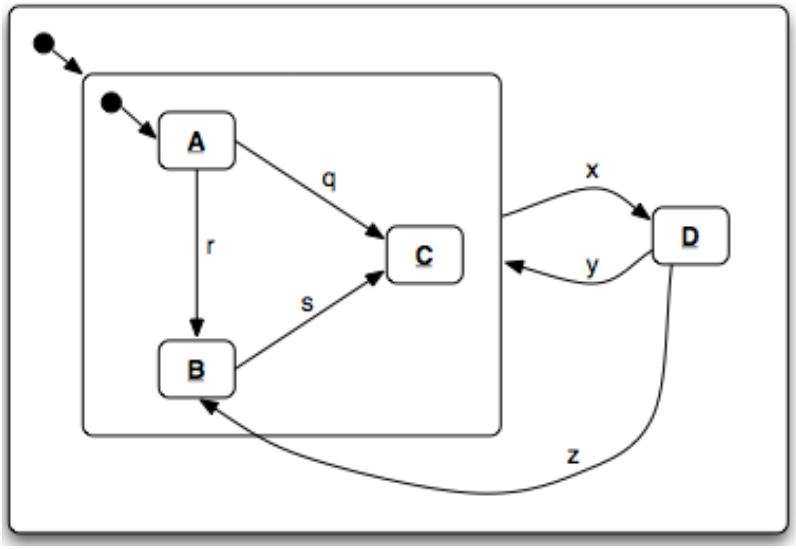


Concurrent

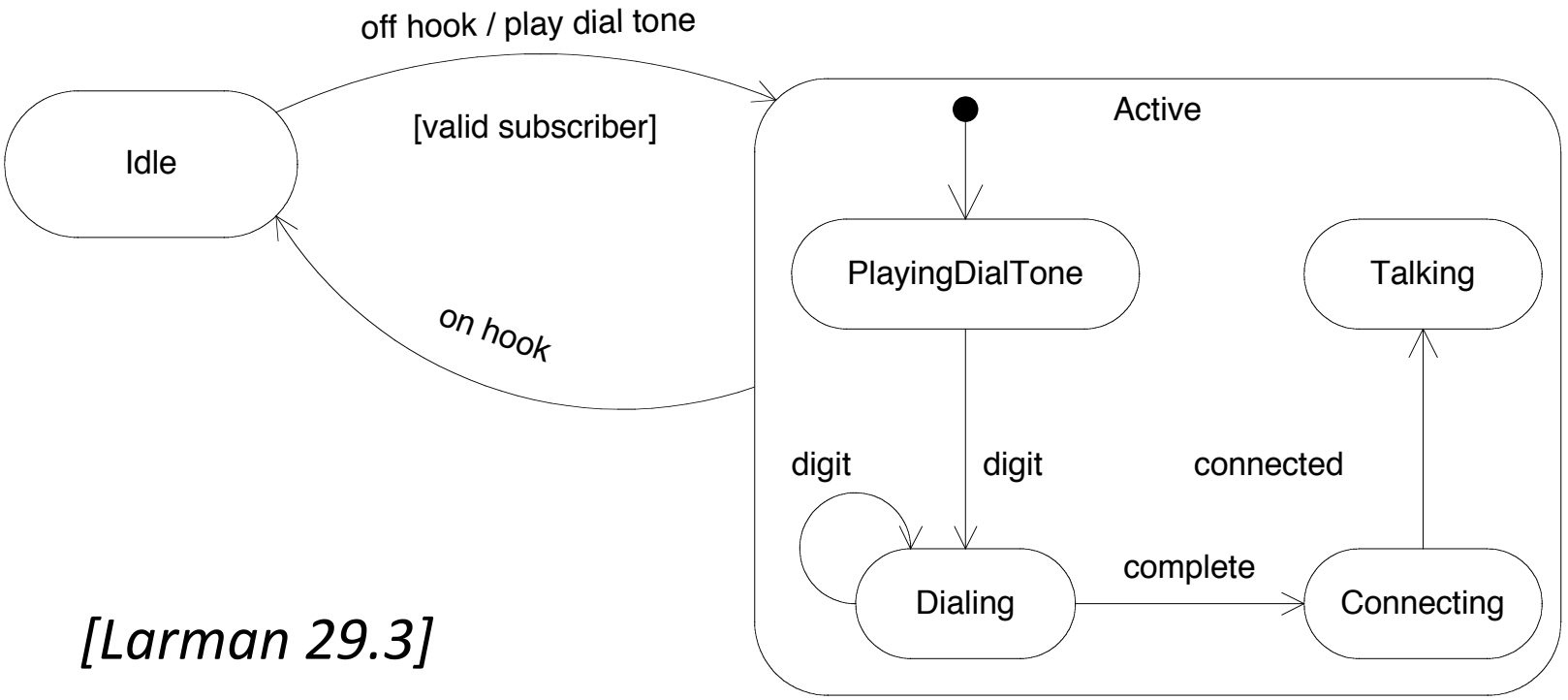


Hierarchical states

- If a transition leaves a composite state (aka "submachine"), the transition applies to all substates.
  - The substates “inherit” the transitions of the superstate.
- If a transition ends at a composite state, the transition is continued by the default initial state in the submachine.
  - *Usually* have a default initial state at every level in the hierarchy.

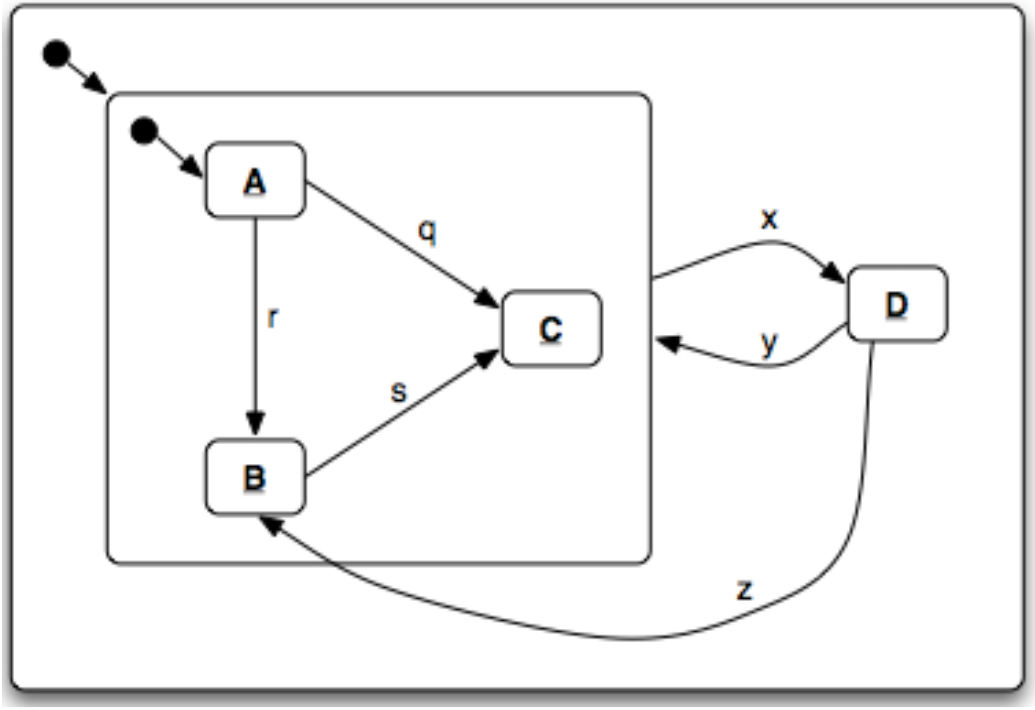


Another example



[Larman 29.3]

Hierarchical states



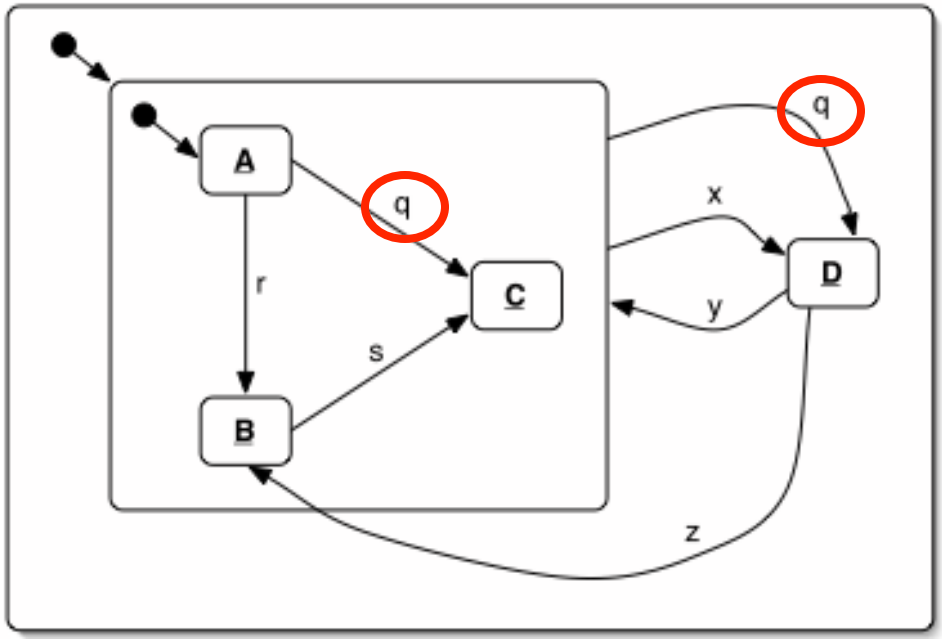
- Hierarchy can be used to abbreviate a "flat" state machine.
  - One transition leaving a superstate can represent many transitions in a flat state machine.

Priority

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

A: UML gives priority to transitions leaving a state *lower* in the hierarchy

i.e., sub-machines can override the behaviour of their ancestor states.



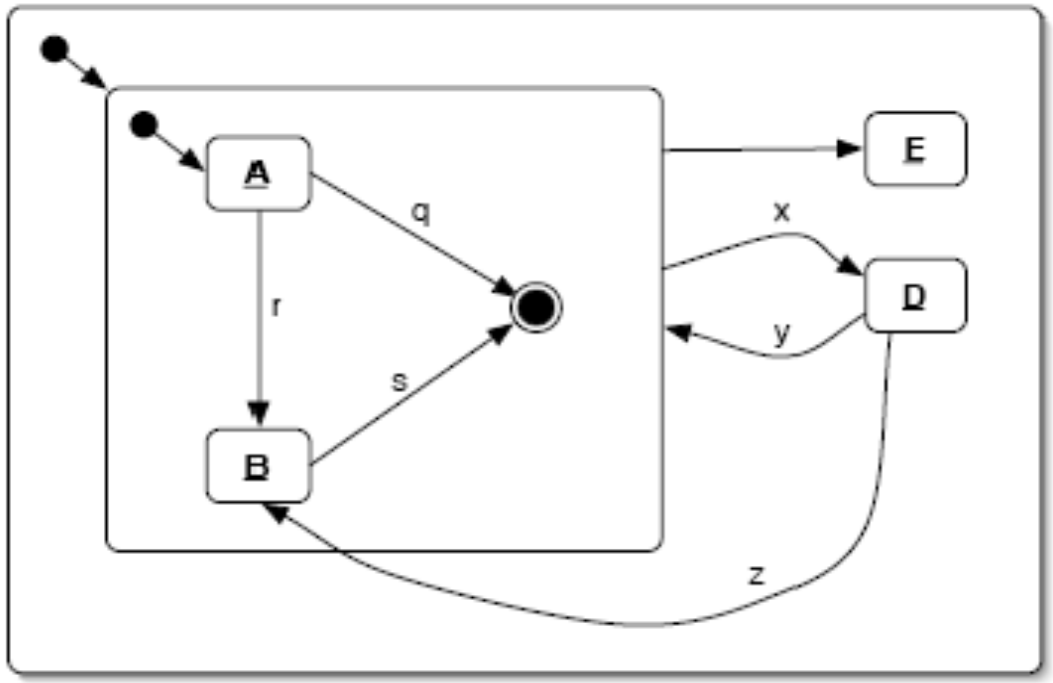
- Conceptually, can think of this like an inheritance child overriding its parent's default behaviour in OOP

More to worry about

- What if two things (e.g., events) happen at the same time?
- What if one scenario happens while another part of the system is in a particular state?
- What if the callee picks up the headset just as a connection is being completed to that callee?
- What if automated maintenance tests are activated while the phone is being used?
- What if a caller picks up the headset while the phone is undergoing automated maintenance?

Final state

- A final state represents the end of computation within a composite state.
  - Recall that a final state is a real state.



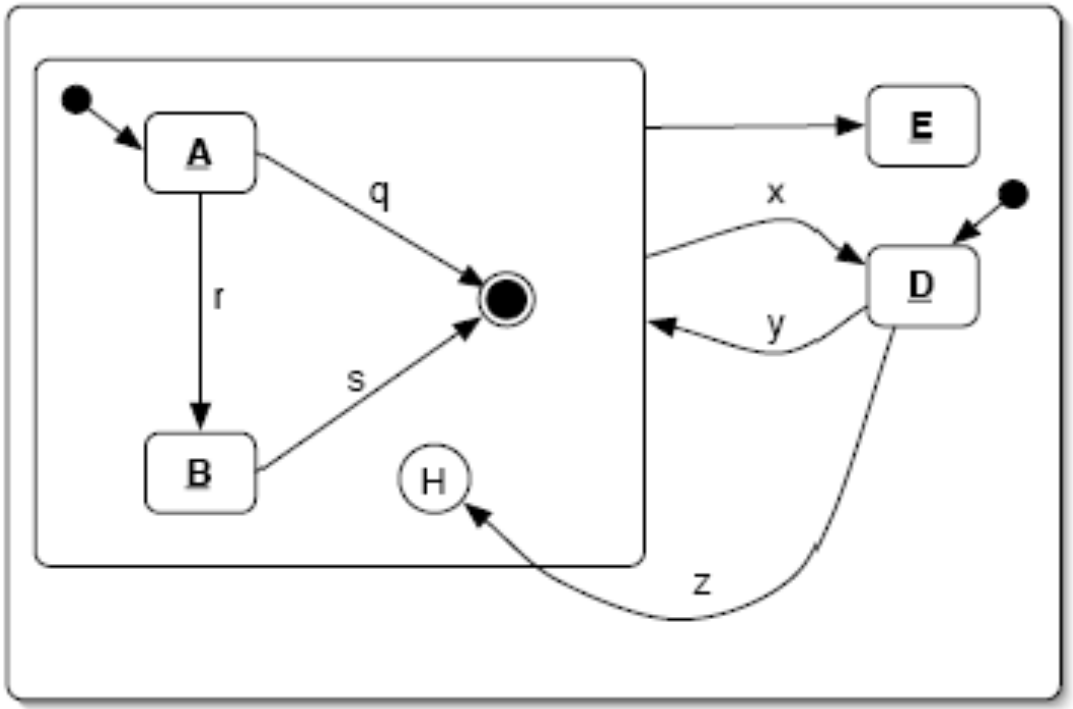
Final state

- A transition leaving a basic state that has no event or condition in its label is always enabled.
  - If a composite state has a final state, a transition leaving a composite state that has no event or condition in its label is enabled when the state is in its final state.

- Transitions based on events and/or conditions are enabled from any state within a composite state.

History

- *History* is a pseudo-state that designates the immediate sub-state at this level in the hierarchy that the system was in when the parent state was last exited.



History

- A history pseudo-state can be the destination state of a transition or a default arrow.
- A transition leaving a history state indicates what state to enter if the system has never been in this superstate before.
  - If no transition is provided, then the default initial state is used.
- Usually transitions entering a history state and leaving a history state are not labelled.

Deep history: H*

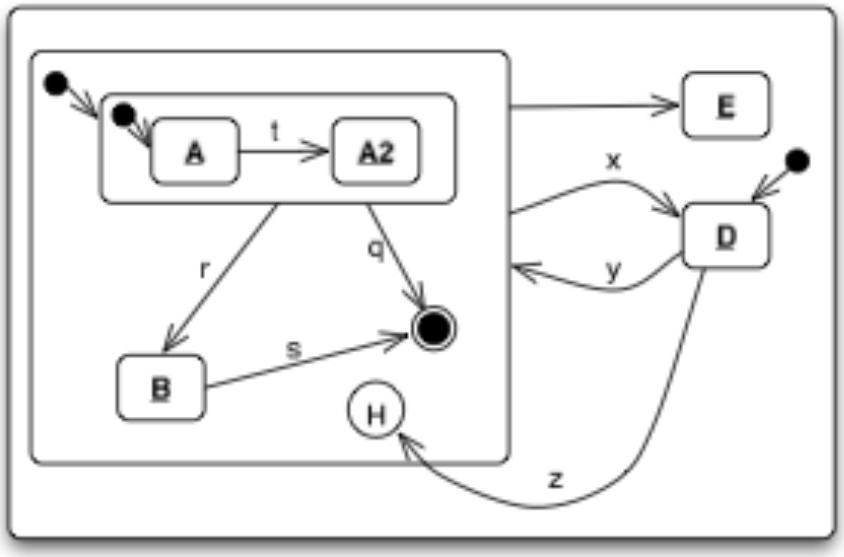
- If a *deep history* pseudo-state is the destination of a transition or a default arrow, then at all levels in the hierarchy below this one the system should enter the substate that it was last in when that state was exited

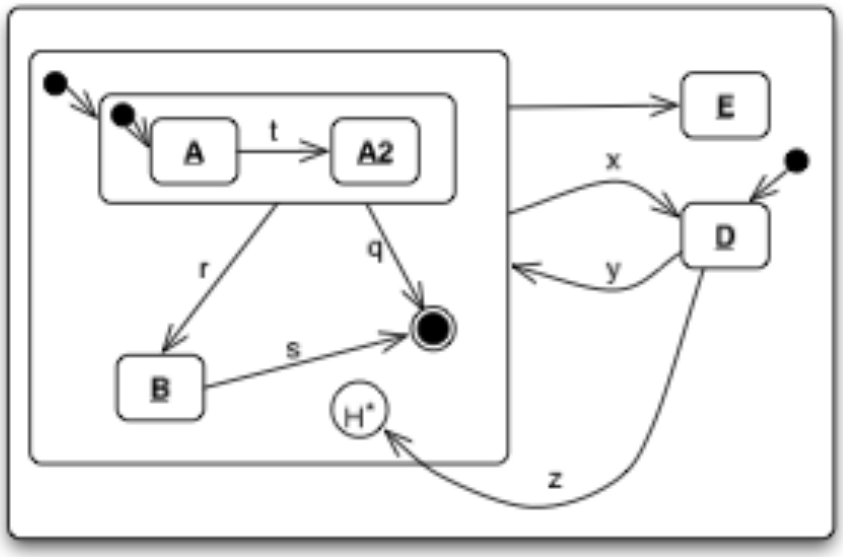
  i.e., apply history at all levels in the hierarchy below this one

  – In other words, deep history recursively applies the history construct until a basic state is reached.

- Notes:
  – History and deep history states are pseudo-states – no time is spent in them; they are just the continuation of a transition.
  – Don't use "H" as a state name yourself!

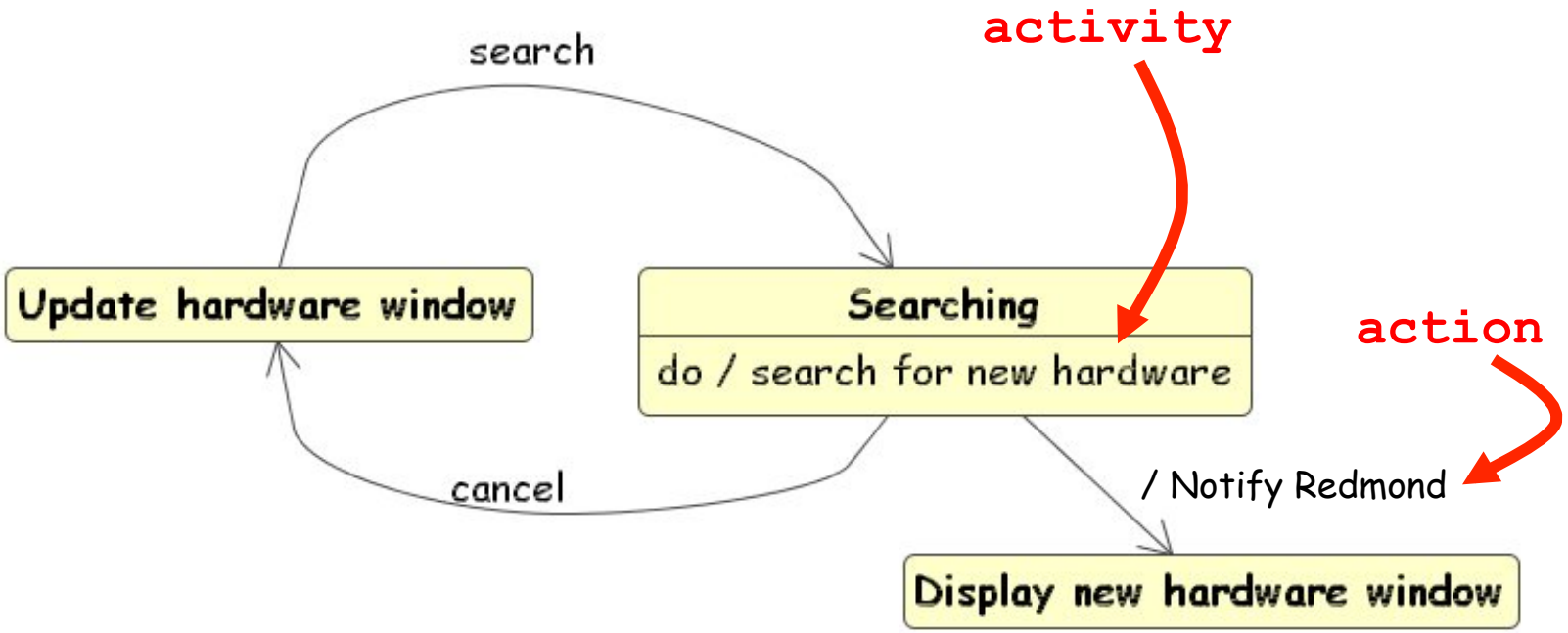
Deep history





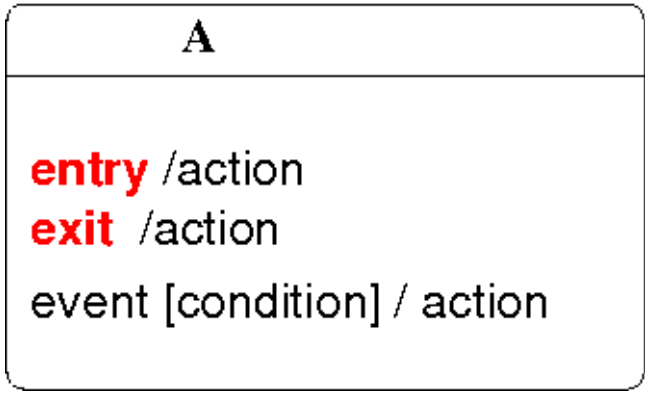
Review: Actions and activities

- *Actions* are considered to be instantaneous (non-interruptible)
- *Activities* occur inside states (usually)
  - Activities are computations that "take time" and can be interrupted
  - States with activities are called … *activity states*



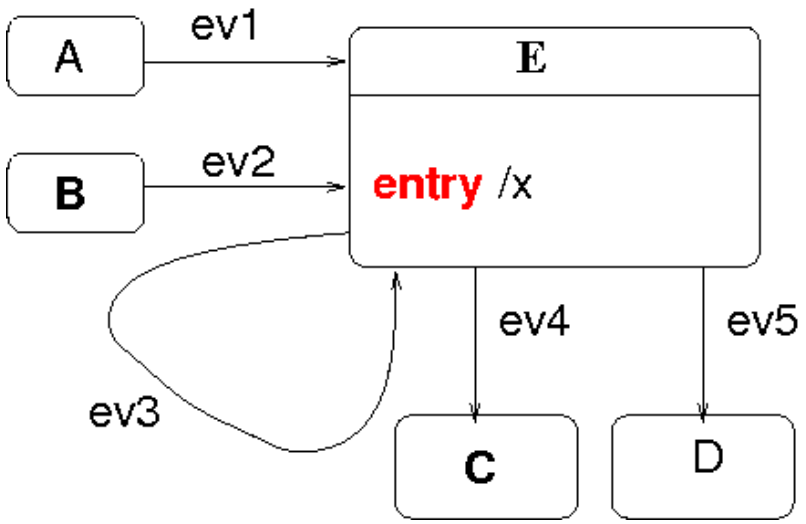
State actions

- 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



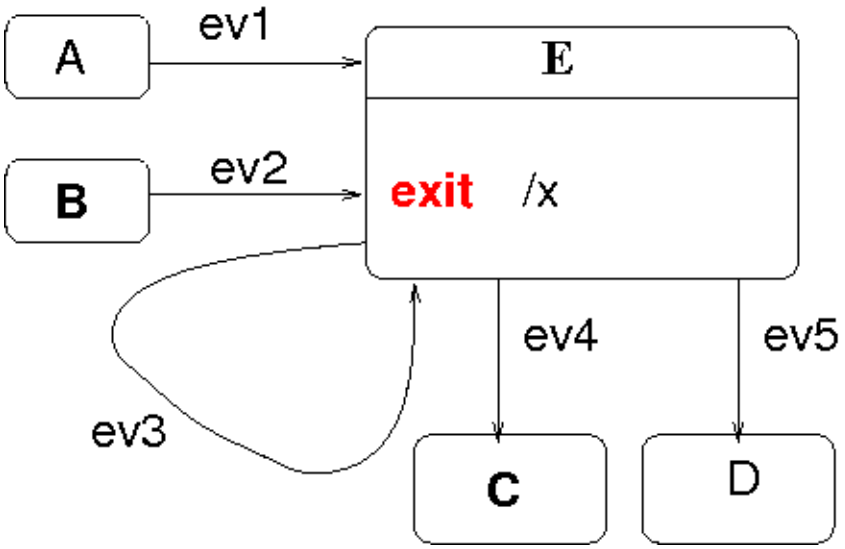
Entry actions

- *entry /x* is equivalent to adding action x onto all *incoming* transitions
  - incl. self-transitions and the initial pseudo-state



Exit actions

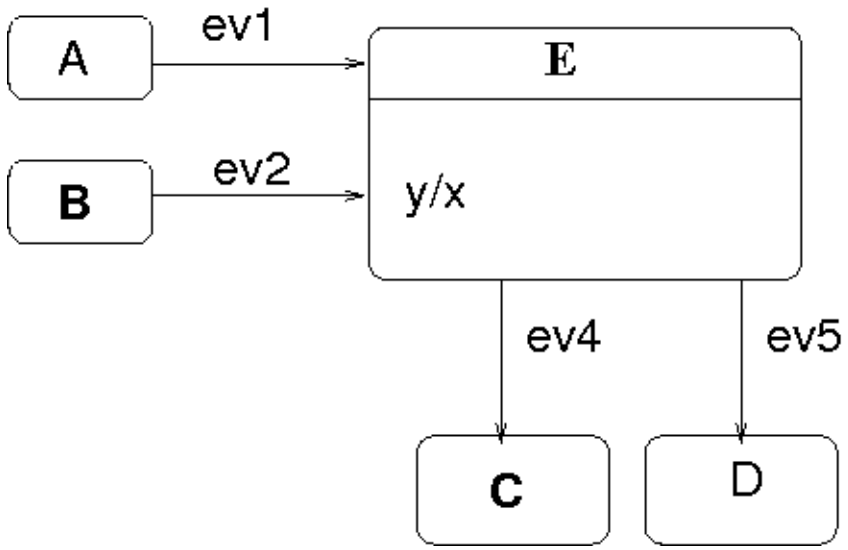
- *exit/x* is equivalent to adding action x onto all *outgoing* transitions (incl. self-transitions)



Internal actions

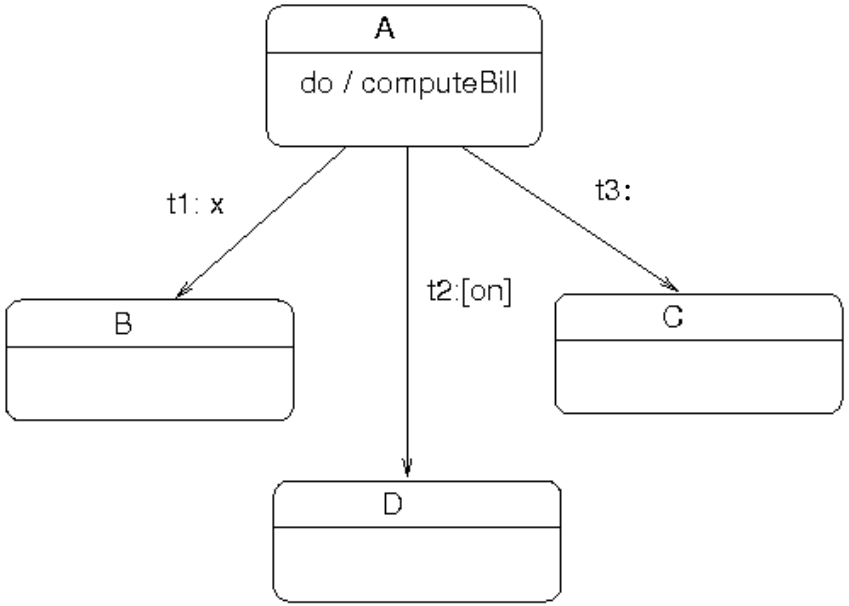
- Internal action *y/x* is equivalent to a single self-transition, if we ignore entry / exit actions

  i.e., entry / exit actions are NOT performed as part of an internal action.



State activities

- Because activities take time, they can be interrupted by transitions with triggers and/or conditions
  - If there are no interruptions, then the outgoing transition from the activity state is likely to have *no* trigger or condition ("naked")



Note that t1, t2, and t3 are simply *transition labels*; they are *not* events, conditions, or actions.

Actions vs. activities

- UML 2.0 has dropped actions
  - This is too bad because making more explicit the distinction is very useful.
  - Therefore, we will use actions in our state machine models anyway.

State actions and activities

- States can be annotated with entry or exit actions, internal actions, and activities:
  - *entry / action* *[red means "keyword"]*
  - *event / action*
  - *exit / action*
  - *do / activity*

- A "naked" transition exiting a state (i.e., having no event or condition associated with it) fires as soon as any activity associated with the state is complete.
  - If there's no internal activity, it fires immediately
  - Naked transitions are commonly used to exit from activity states and concurrent states

State actions and activities

- 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.

- If you want a self-looping transition that does not activate exit and entry events, use an internal action instead of a transition

Change events

- A *change event* is the event of a condition becoming true.
  - Think of like a hardware sensor

- The event "occurs" when the condition goes from false to true because the values of some variables used in the condition change their values. For example:
  - *when (temperature > 100 degrees)* *[red means "keyword"]*
  - *when (on)*

- The event does not reoccur unless the condition turns to false and then returns to true.

when(X) vs. [X]

- A change event is what you want if the process is sitting in a state waiting for a condition to become true
  - A change event implicitly polls the condition regularly for a status change
- Don't use a guard on a naked transition (called a "completion transition" in UML)
  - The semantics of that is, check the condition once (e.g., when the activity is done), and fire if the guard is true
  - The guard is never checked again after that initial check.

Time event

- A *time event* is the occurrence of a specific date/ time or the passage of time.
  - Absolute time:
    - *at (9:00 am, 9 Oct 2010)* *[red means "keyword"]*
  - Relative time:
    - *after (10 seconds since exit from state A)*
    - *after (10 seconds since x)*
    - *after (20 minutes)*

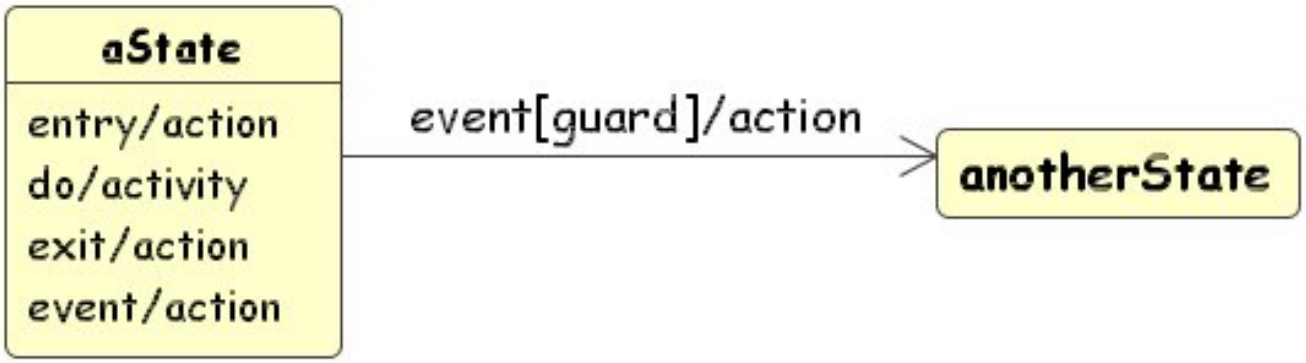
      [Since execution entered the transition's source state]

Event summary

- An event is instantaneous.

- Kinds of events:
  - *external* – change in the environment (external signal)
    e.g., “off-hook”
  - *external/internal* – change events, occurrence of a condition becoming true
  - *internal* – a message from a concurrent region
  - *time events* – occurrence of relative or absolute passage of time

Summary



Each event, guard, action, and activity may have arguments.

Can also combine multiple events, actions, etc. into one "slot" using a semi-colon.

CS445 / SE463 / ECE 451 / CS645

Software requirements specification & analysis

UML state machine diagrams

Fall 2013 — Mike Godfrey, Dan Berry, and Richard Trefler