

# CS445 / SE463 / ECE 451 / CS645

## Software requirements specification & analysis

### 7. UML state machine diagrams

Fall 2010 — Mike Godfrey and Dan Berry

# UML state machine diagrams

- Shows finely-grained behaviour *within* an object
- 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
  - Methods responses highly dependent on internal state
    - Only after all critical data fields filled in will system allow transition to “confirm payment” state
    - Certain events only relevant in certain situations  
e.g., Only first “walk” button press is significant

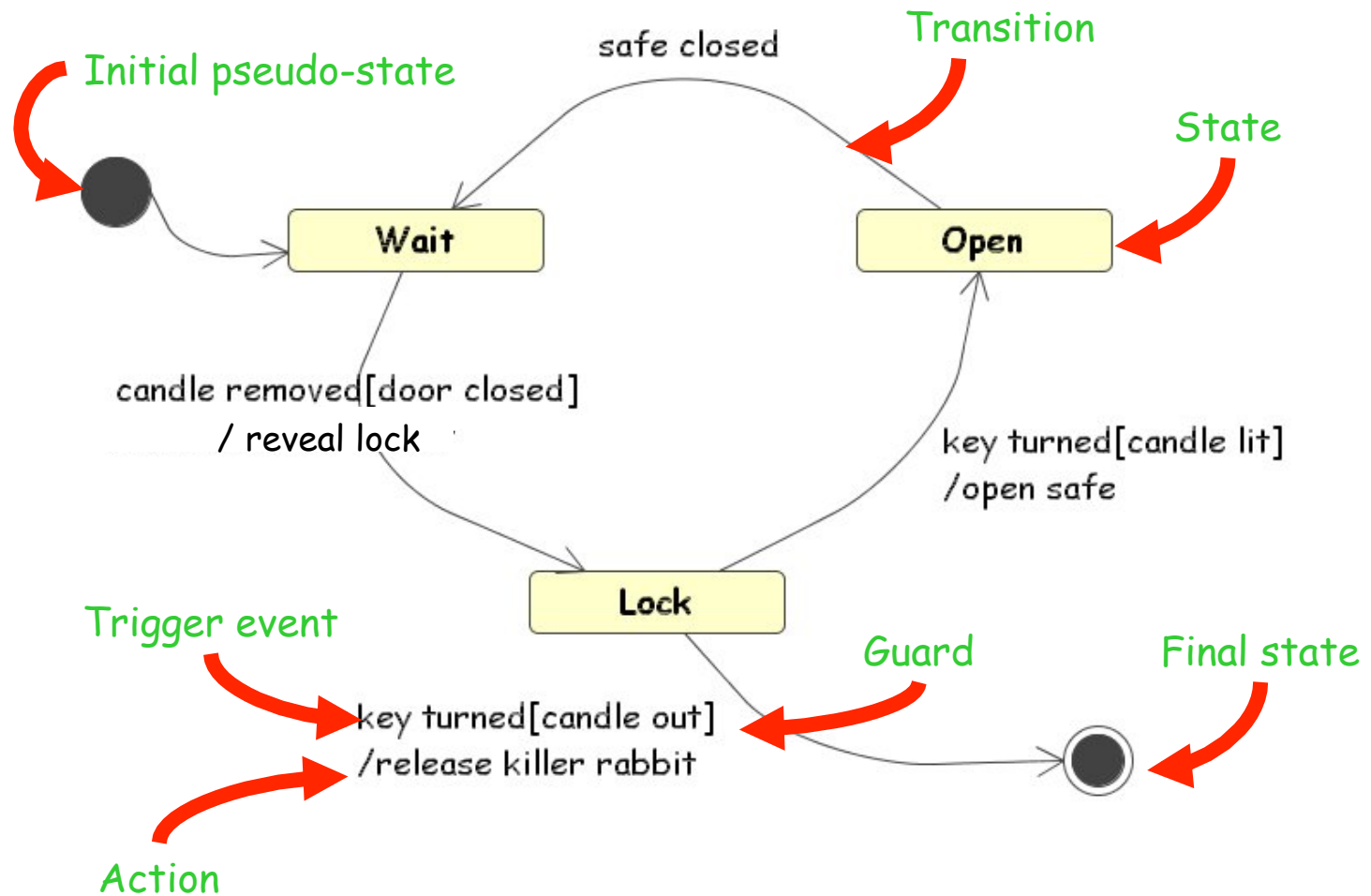
# UML state machine diagrams

- Normally, we use SM diagrams during design to describe a desired object's behaviour to guide implementers
  - Also used in RE to model UI specs (but it's really design *(unless you stay within INTF vocabulary)*)
- Possible RE use of SM diagrams:
  - Can use to specify each object's contribution to all scenarios of all use cases
  - Generally, tho, this is too detailed a model for RE *(unless you stay within INTF vocabulary)*

# The state-transition paradigm

- Most of the 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)
  - They respond more or less the same way to method invocations

# An example [Fowler p108]

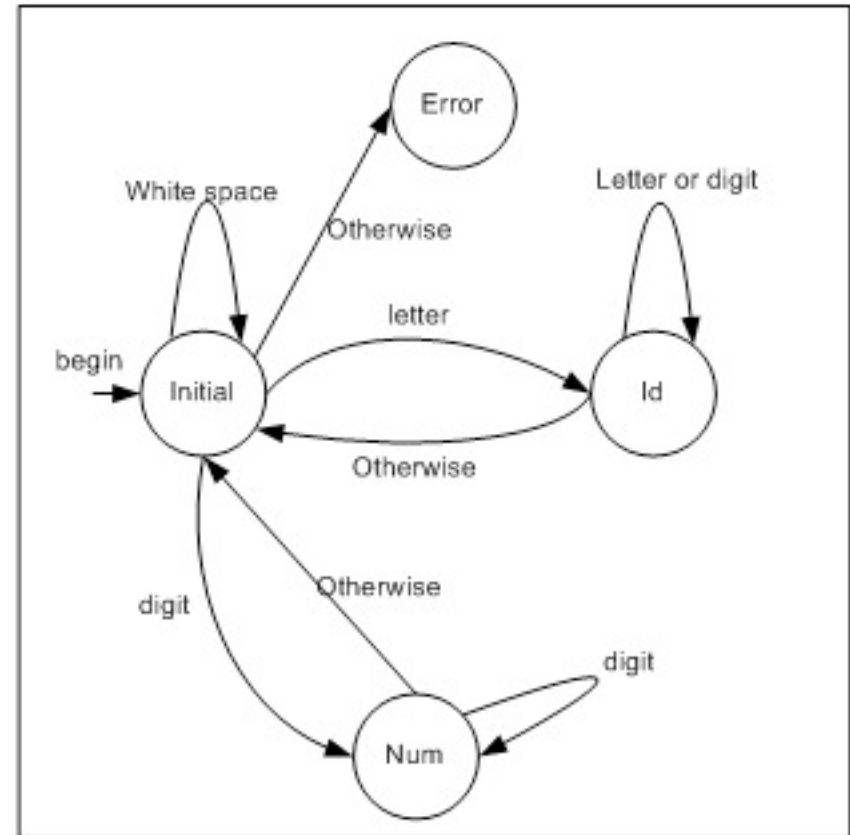


# A UML state machine

- ... is a *hierarchical, concurrent, extended finite state machine*:
  - It shows the lifecycle of an *instance* of the class
    - The objects starts in a given state, and transitions to others based on external “messages” (events detected) received and the values of its internal variables
  - It describes the behaviour of an object across *multiple* (perhaps all!) use cases.
  - It’s hierarchical because each state can in turn be broken down into sub-state machines
  - It supports concurrent regions
  - It’s extended in that it allows variables to augment the state descriptions

# Example FSM

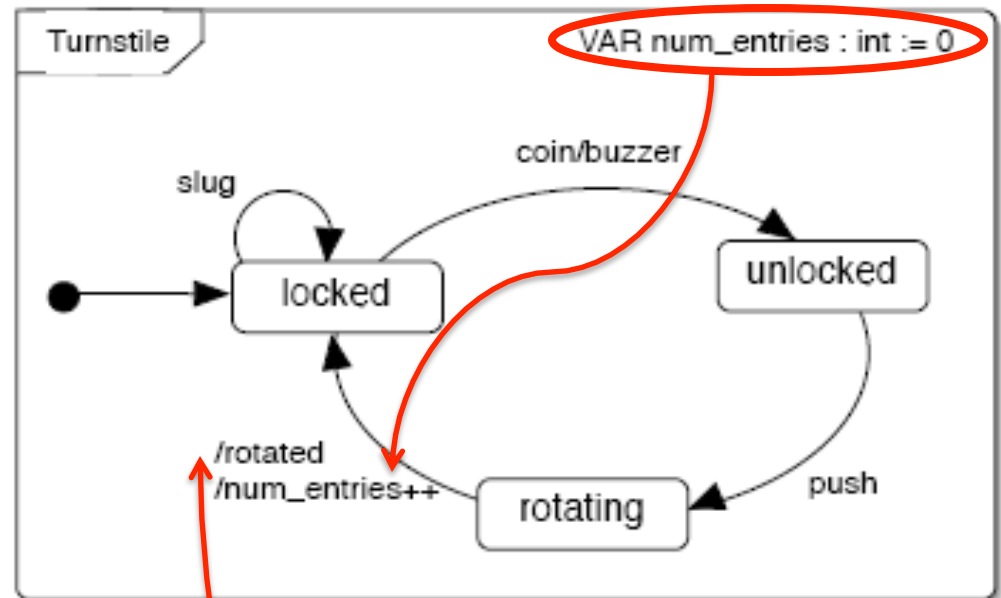
- You might have seen an FSM like this to represent a grammar in CS241
  - This FSM processes a character stream to produce a token stream with white space removed
- “Finite State Automaton” (FSA) is another term for FSM.



[http://www.codeproject.com/KB/recipes/Parser\\_Expression.aspx](http://www.codeproject.com/KB/recipes/Parser_Expression.aspx)

# 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.



Mistake: "rotated" should not have a leading "/"

# 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 vs. design model

- During *requirements engineering*, we may build a System-level State Machine Diagram (SSMD)
  - The SUD is the outermost “machine”
  - We decompose it into substates (hierarchical and concurrent) that respond to environmental events
  - We progressively merge the semantics of the system-level UCs into our model, showing how the SUD responds various inputs
- Note that the decomposition of the SSMD starts to resemble a preliminary design of the system
  - In particular, the SSMD may resemble the user interface structure if the SUD is an info system
  - This is normal; note, however, that the eventual design may differ a lot

# Requirements vs. design model

- In a *design* model, we may create state diagrams for the proposed design classes
  - Show how an object's behaviour changes over time receiving and sending messages (operation calls) from and to other objects
  - Describe how one object contributes to desired system behaviour
  - We often don't bother defining state diagrams for the classes with simple behaviour; we define state diagrams for only classes with complex, hard-to-describe behaviour.
- However, we won't address this in SE1.

# Requirements vs. design model: Validation

If you build such a *design or specification* model, you can subject it to a very useful kind of *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 the history of inputs so far.

# System-level state machine diagrams (SSMDs)

We are talking about specification-level state machine diagrams:

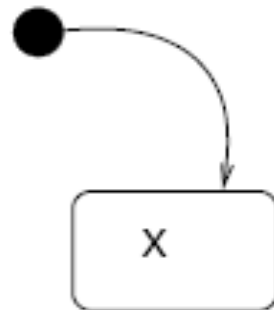
The vocabulary of the machine is limited to entities that appear in the Interface (intersection of Environment and System) part of the World diagram.

# System-level state machine diagrams (SSMDs)

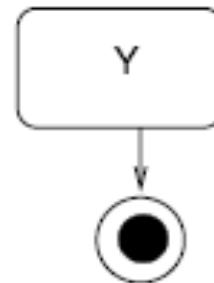
- For RE, the names of states should be meaningful: a state represent a “mode” of the system.
  - These names should make sense to the customer.
- The partitioning of behaviour provided by the states helps us to better understand the system.
- A state will sometimes represent an internal computation (e.g., a state of "Validating Customer") whose result is an event that triggers an outgoing transition.

# 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

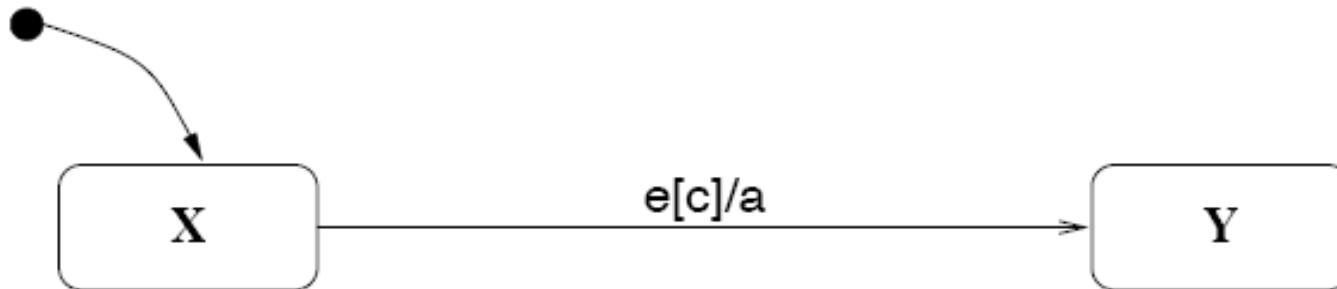


Bullseye is a final state

# 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

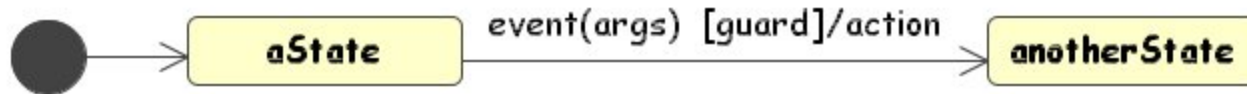
# Transitions



## 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)**

# Tool nit: Transition labels

- A transition might also have a label (or name), tho we won't use it much; the full format for a transition is thus actually:

*label: event [ guard ] / action*

e.g.,      L1: push [OKtoEnter] / startEntry

- In MagicDraw, if you name your transition by typing text onto the arrow, you are changing the transition's label, not naming the trigger event!
  - By default, labels aren't shown in MD (but you probably don't want to use transition labels anyway)
  - To edit the trigger event, bring up the transition specification, set *SignalEvent* as the trigger event type, and name the transition

Transition - <>

History : Transition:r[flattenedHierarchicalState:::A - flattene...

- Transition:r[flattenedHierarchicalState:::A - flattene...
- Documentation/Hyperlinks
- Inner Elements
- Relations
- Tags
- Constraints

Properties: Standard Customize

Transition	
Name	
Owner	[flattenedHierarchicalState]
Applied Stereotype	
Guard	
Target	B [flattenedHierarchicalState::]
Source	A [flattenedHierarchicalState::]
To Do	
Trigger	
Event Type	SignalEvent
Trigger	<input checked="" type="checkbox"/> Trigger:r [flattenedHierarchicalState::]
Event Element	<input checked="" type="checkbox"/> SignalEvent r [flattenedHierarchicalState::]
Name	r
Signal	
Effect	
Behavior Type	<UNSPECIFIED>
Behavior Element	

**Name**  
The name of the NamedElement.

Close Back Forward Help

Transition label;  
ignore this

Select "SignalEvent"

Fill in trigger event name

# 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.,
  - $x > 10$
  - *DoorsClosed*
- Conditions on transitions leaving the same state should be mutually exclusive
  - ... so that no two transitions can be simultaneously enabled

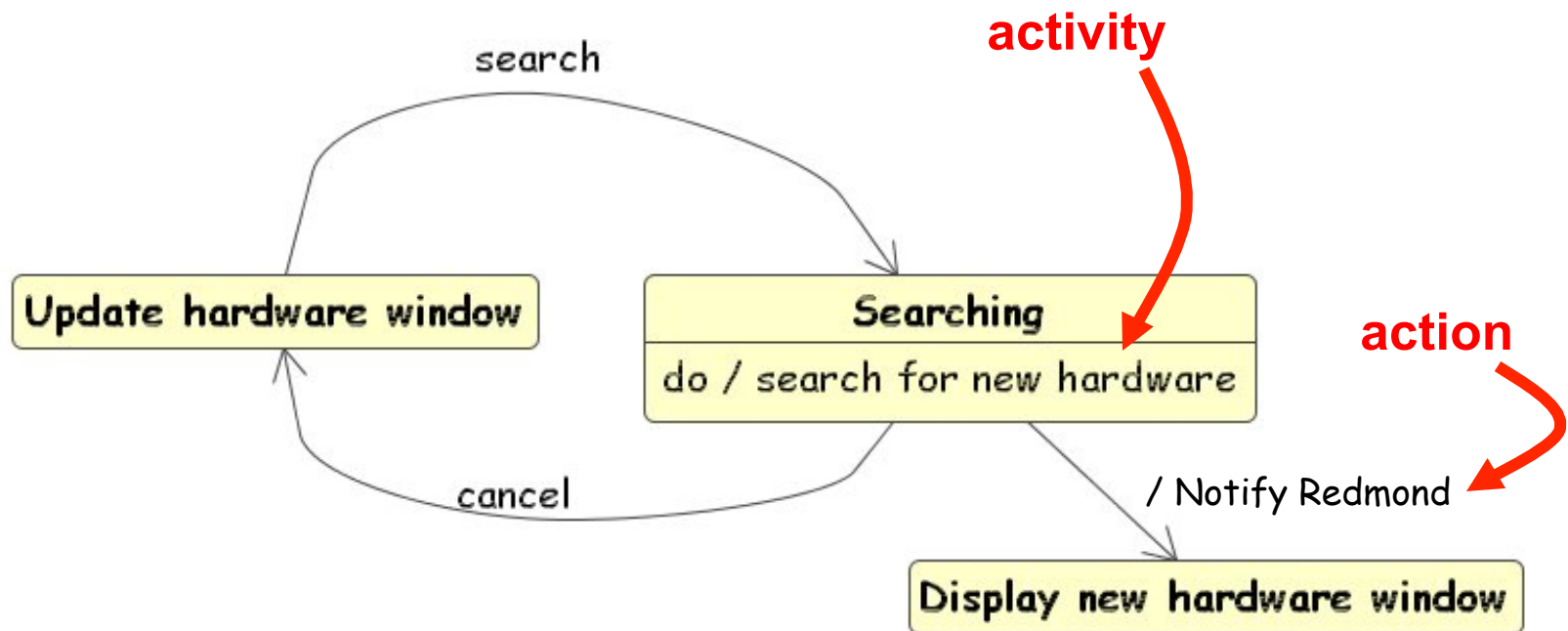
# State actions and activities

- A state can have *actions* and *activities* associated with it also.
  - 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

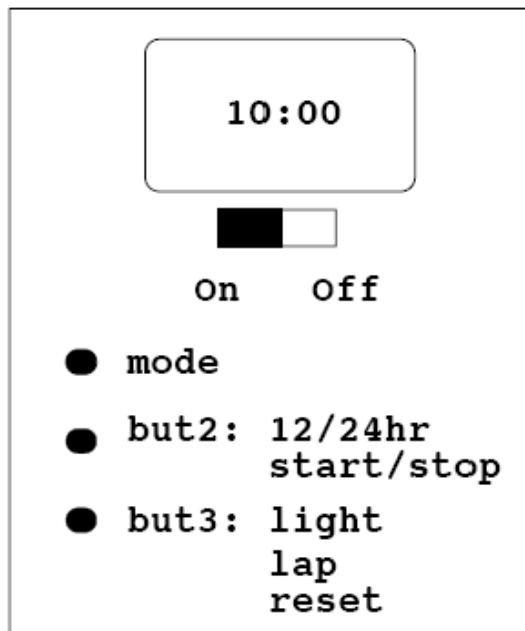


[Fowler p110]

# 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 lapttime; display lapttime; turn light on for 3 sec
but3 [stopwatch, timer stopped, display timer]	Reset timer; turn light on for 3 sec
but3 [stopwatch, display lapttime]	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

# 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  $X$   
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.

# State machine vs. sequence diagrams

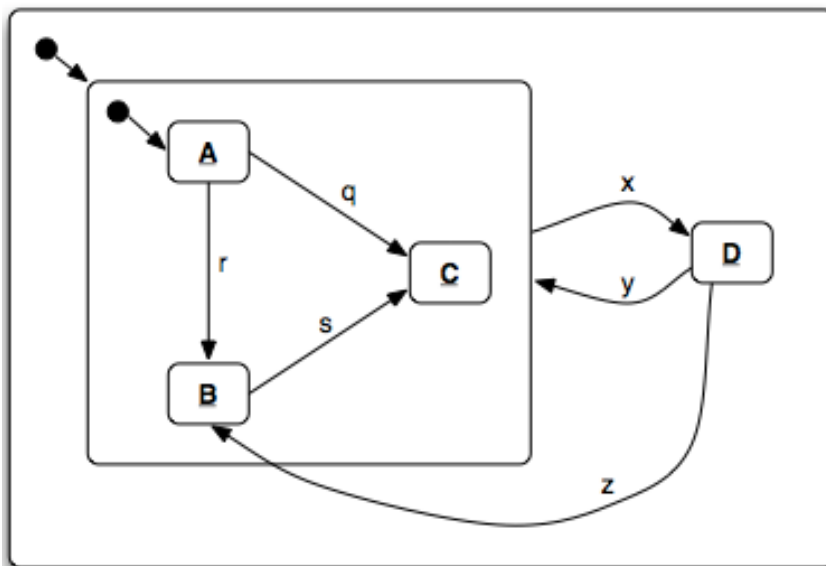
<b>State machine diagrams</b>	<b>Sequence diagrams</b>
<i>specifies</i> behaviour	<i>Illustrates</i> behaviour
all allowable scenarios	one allowable scenario, showing end-to-end behaviour (better feel for overall system behaviour)
models system inputs and outputs	shows the sources and sinks of system inputs and outputs
developer oriented	customer oriented
identifies system states, which represent equivalent input histories	
	can help developer validate state diagrams

# Composite states

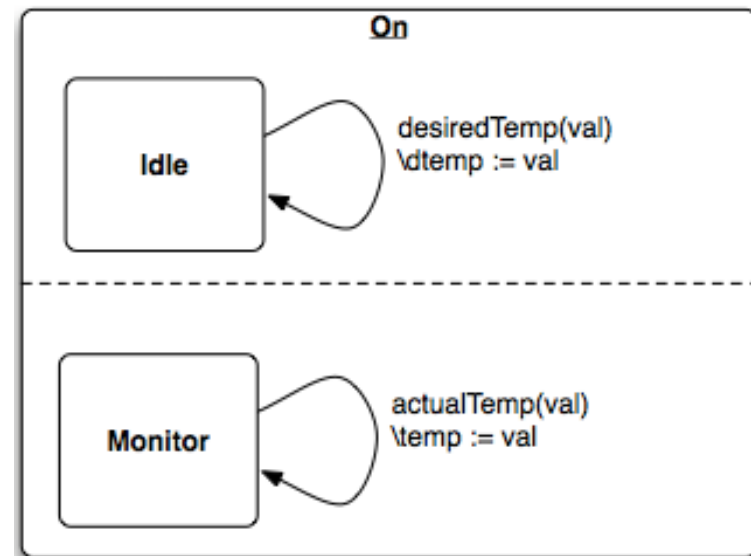
- A *composite state* combines states and transitions that work together towards a common goal. There are two kinds:
  1. Hierarchical (aka “simple” / “OR-states”)
  2. ~~Concurrent (aka “orthogonal” / “AND-states”)~~
- A state that does not contain other states is called a *basic state*

We will ignore these

Hierarchical

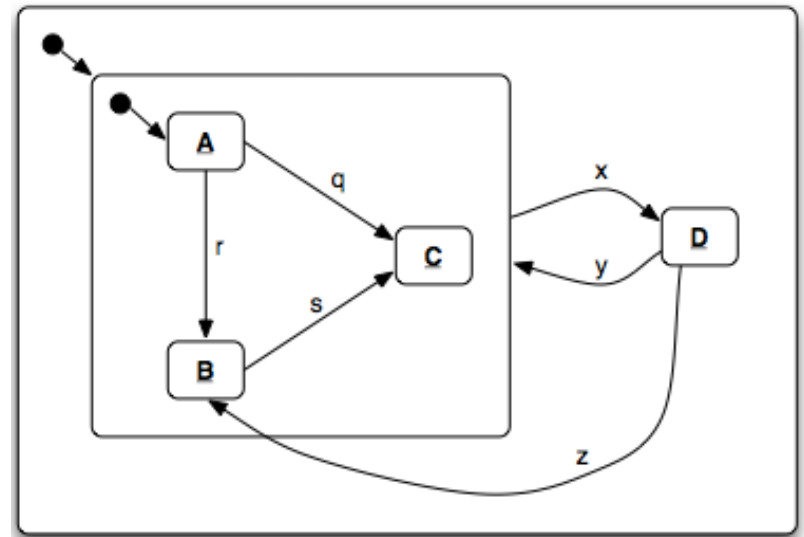


Concurrent

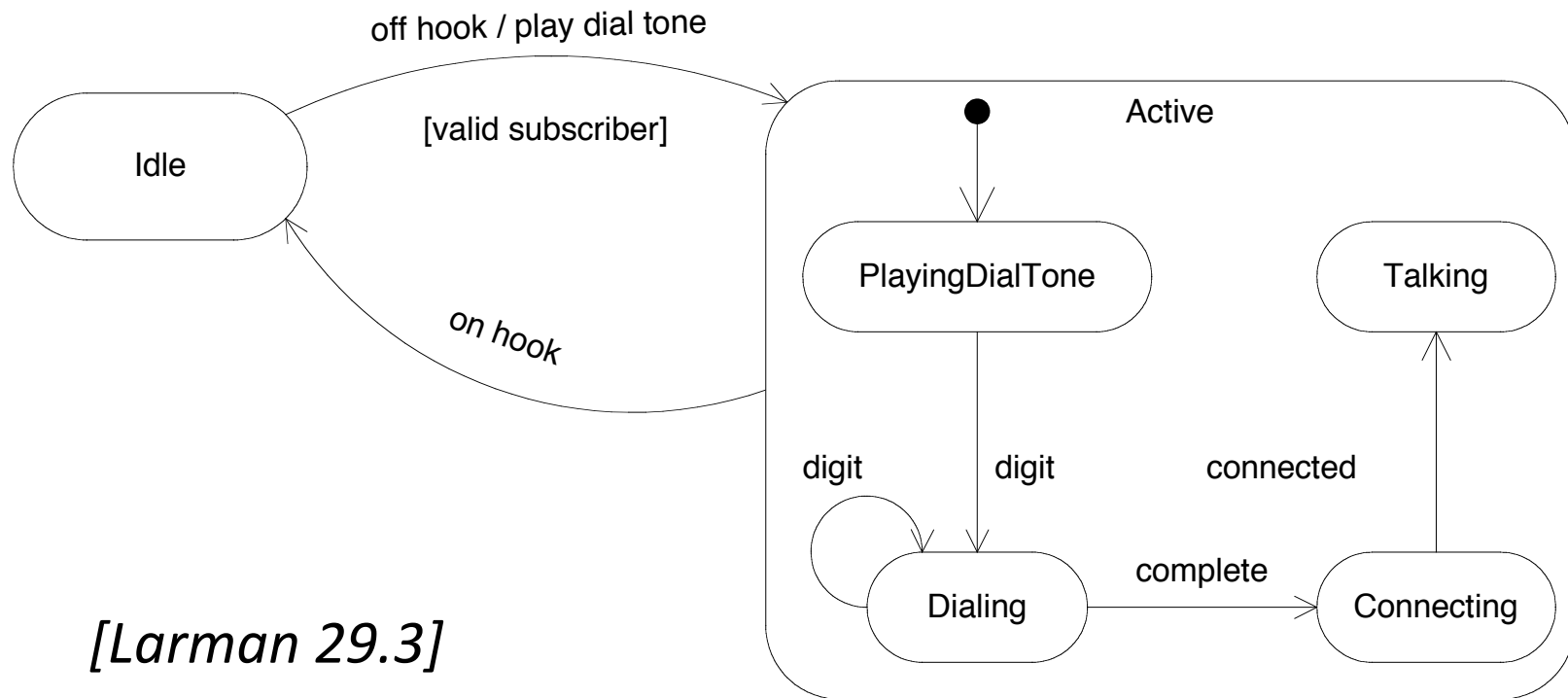


# Hierarchical states

- 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.
  - So there should *usually* be a default initial state at every level in the hierarchy.

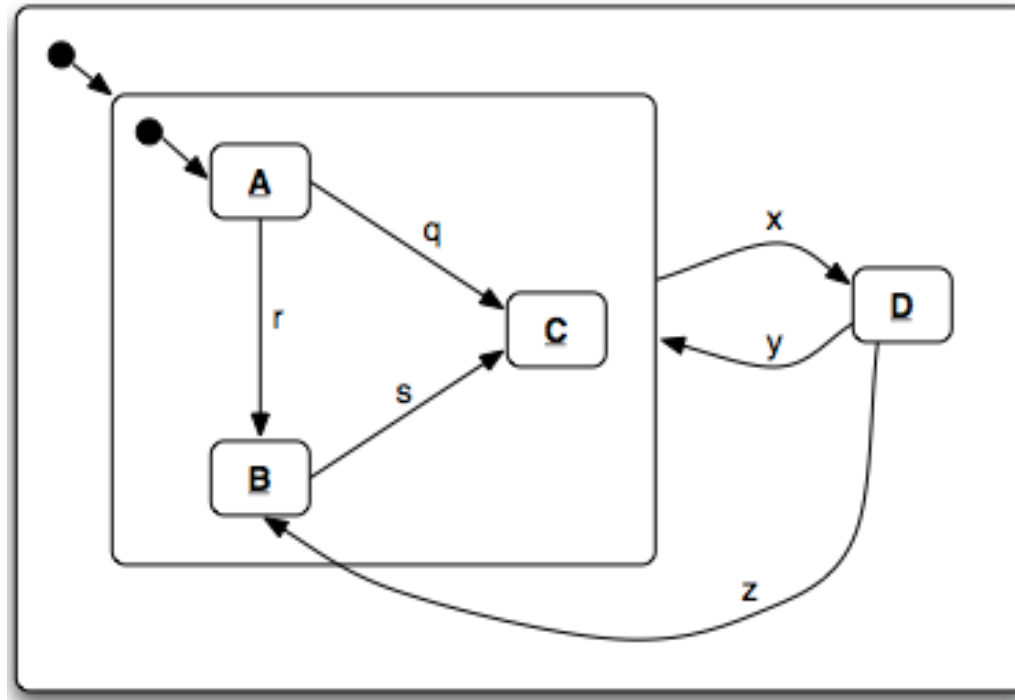


# Another example



[Larman 29.3]

# Hierarchical states



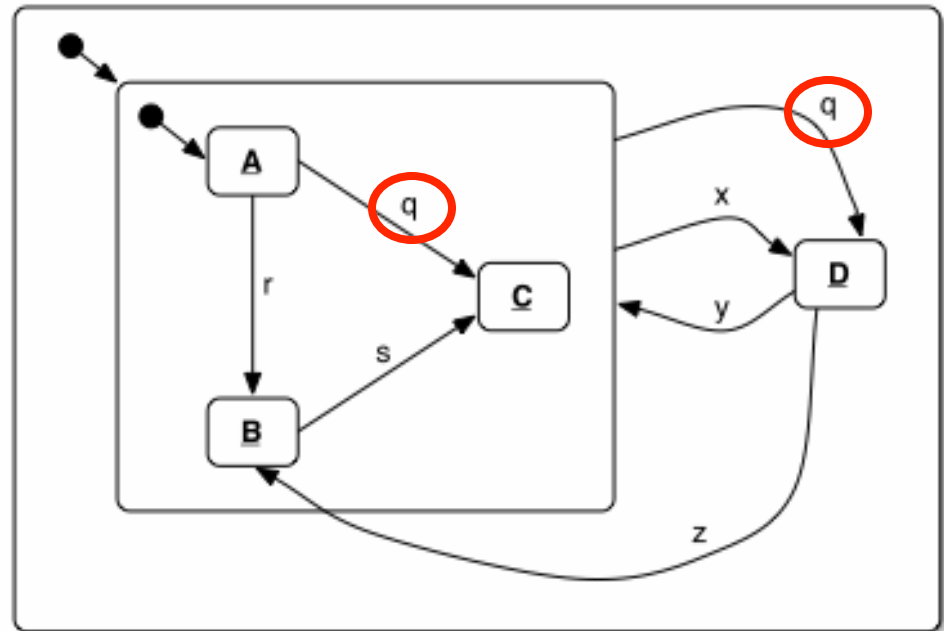
- Hierarchy can be used to abbreviate a “flat” state machine.
  - One transition leaving a superstate can represent many transitions in a flat state machine.

# 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

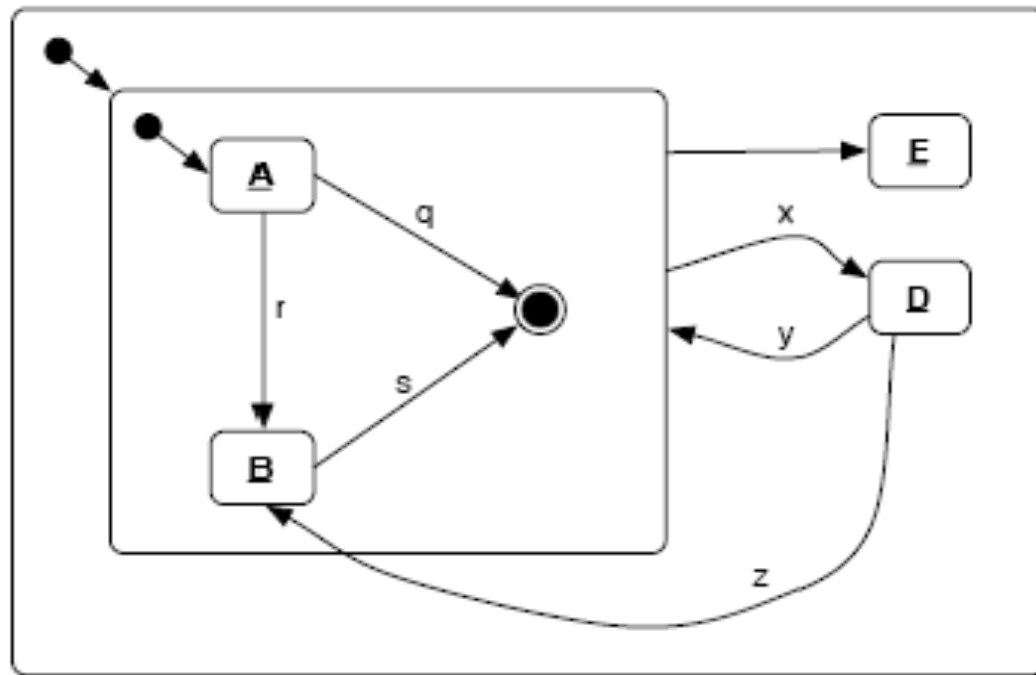


# More to worry about

- 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.

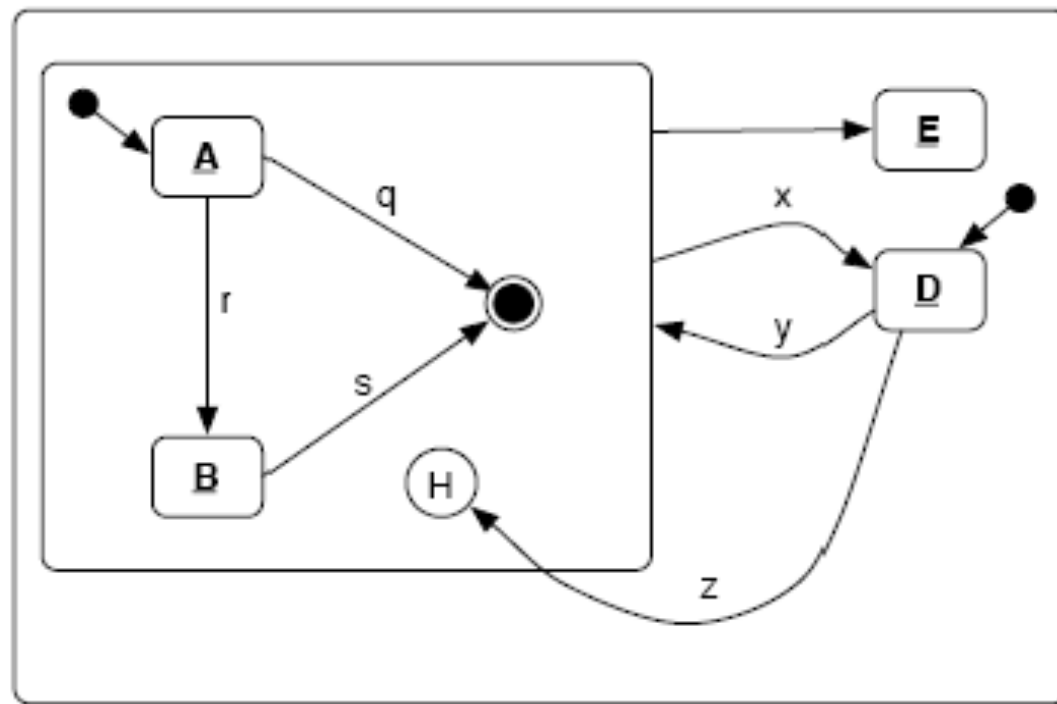


# Final state

- 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.



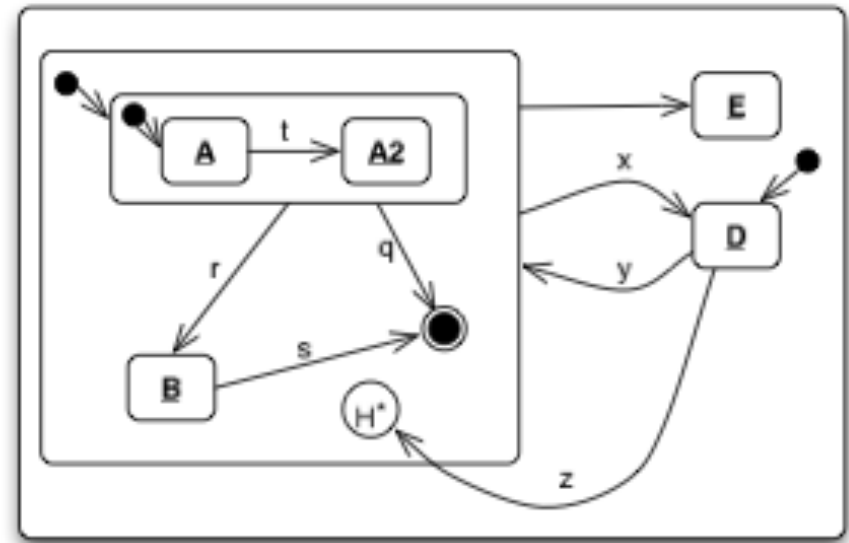
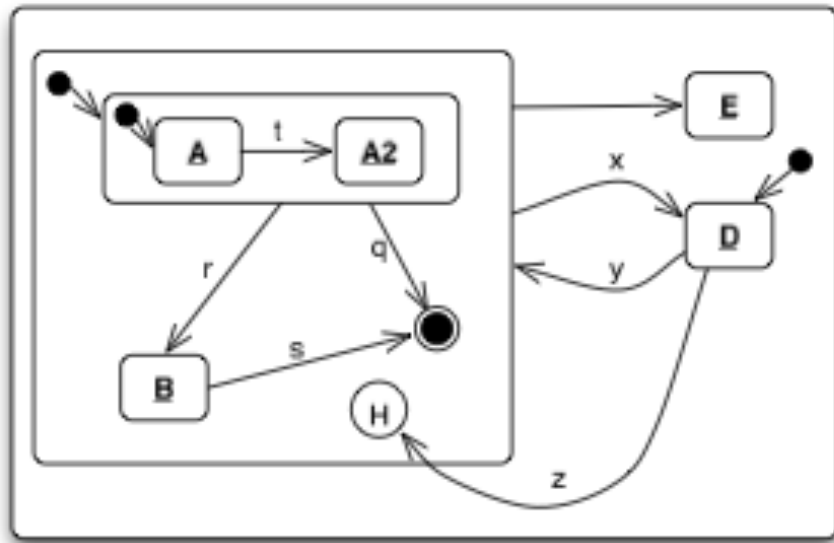
# History

- 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: $H^*$

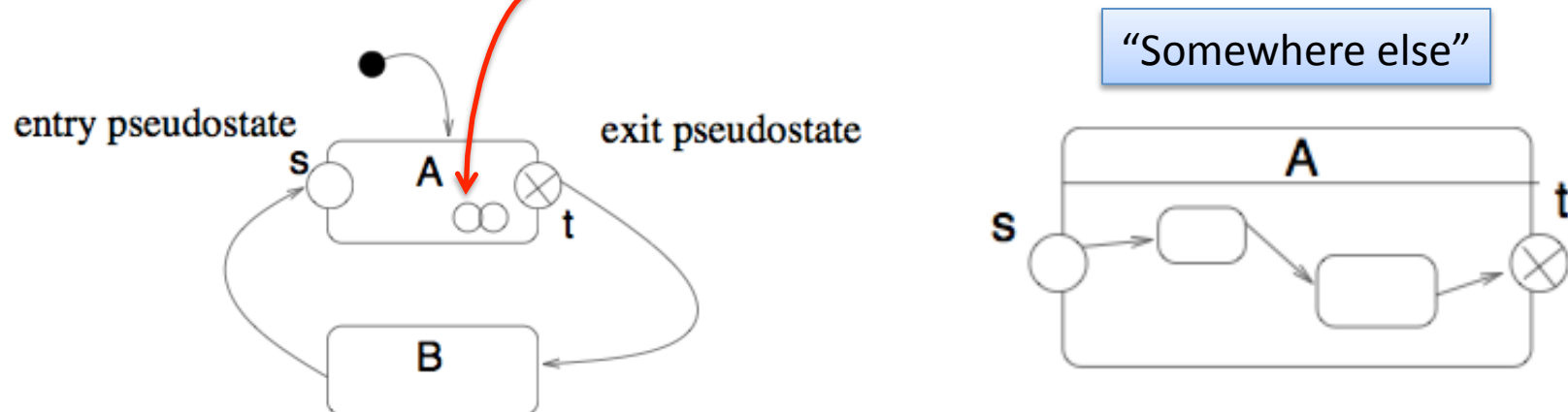
- If a *deep history* pseudo-state is the destination of a transition or a default arrow, then at all levels in the hierarchy below this one the system should enter the substate that it was last in when that state was exited
  - i.e., apply history at all levels in the hierarchy below this one
    -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



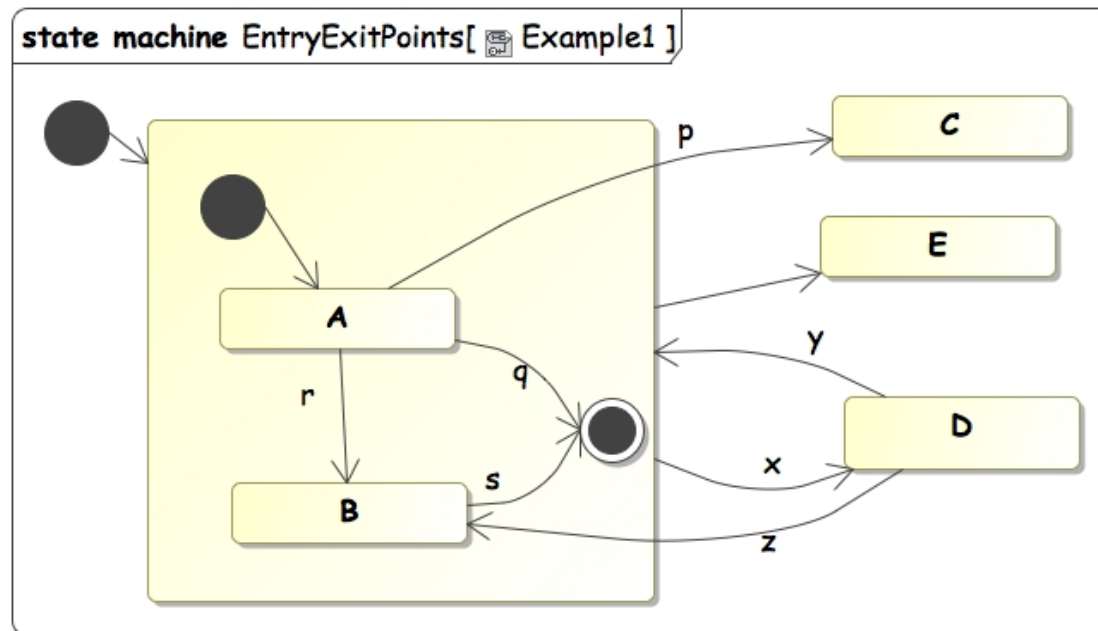
# Composition issues

- The details of a composite state can be shown in a separate diagram.
  - Can use the *composition icon* in the state, which means “has real content but is defined somewhere else”



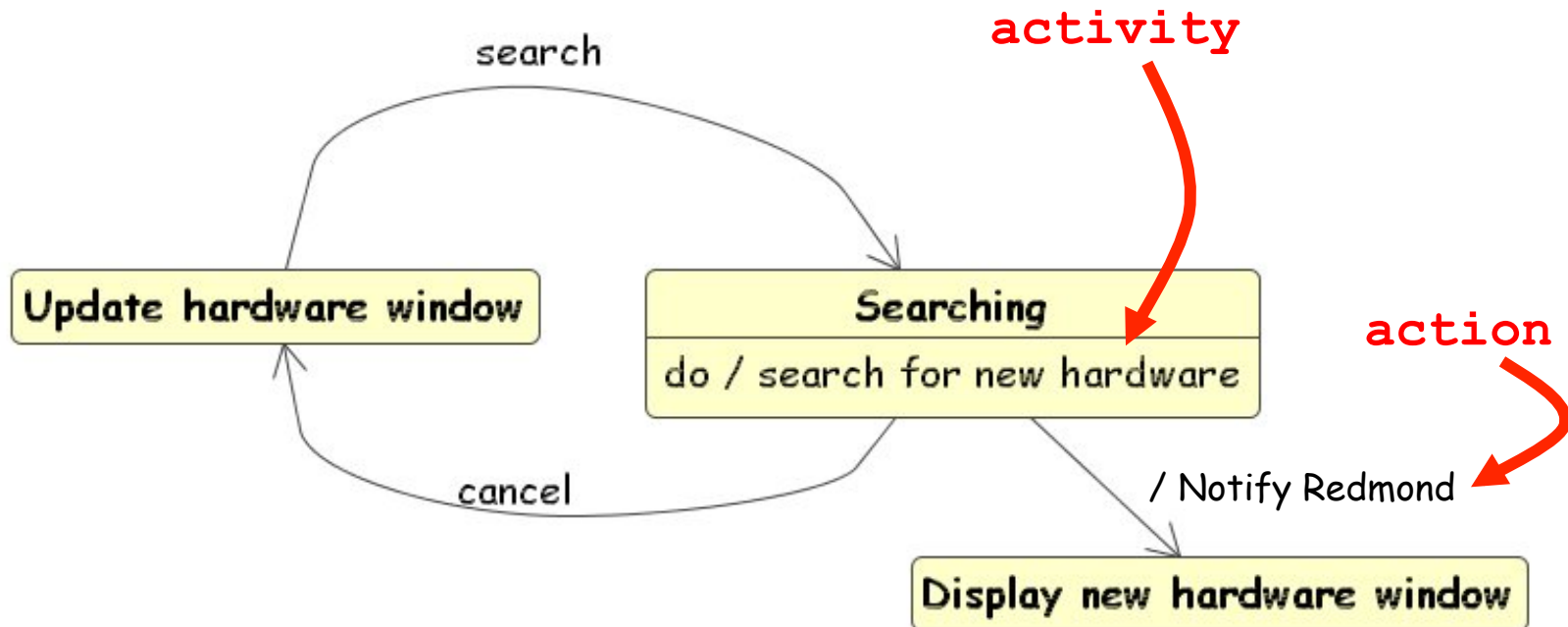
# Composition issues

- A composite state whose internal details are defined elsewhere can't have transitions going to/from its internal states arbitrarily ...
  - So we use *entry* and *exit points* (which are pseudo states) to make non-standard entry/exit into submachine easier to specify



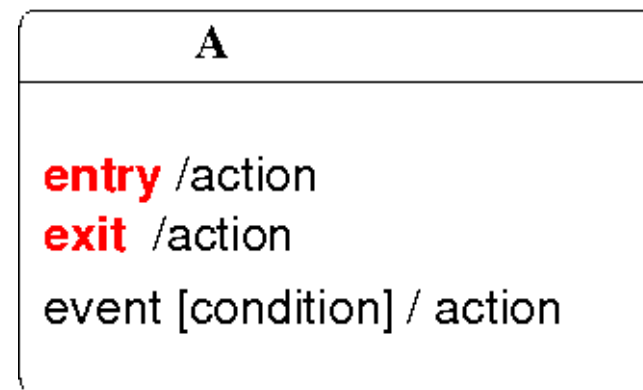
# Review: Actions and activities

- *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*



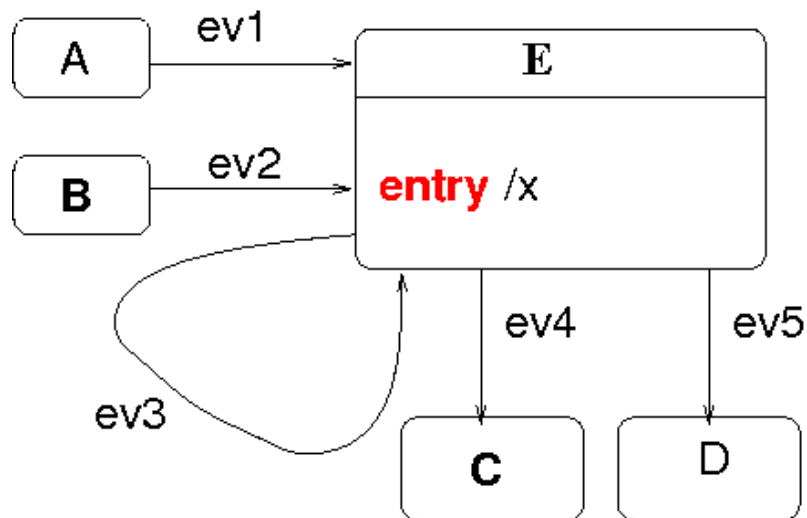
# State actions

- 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



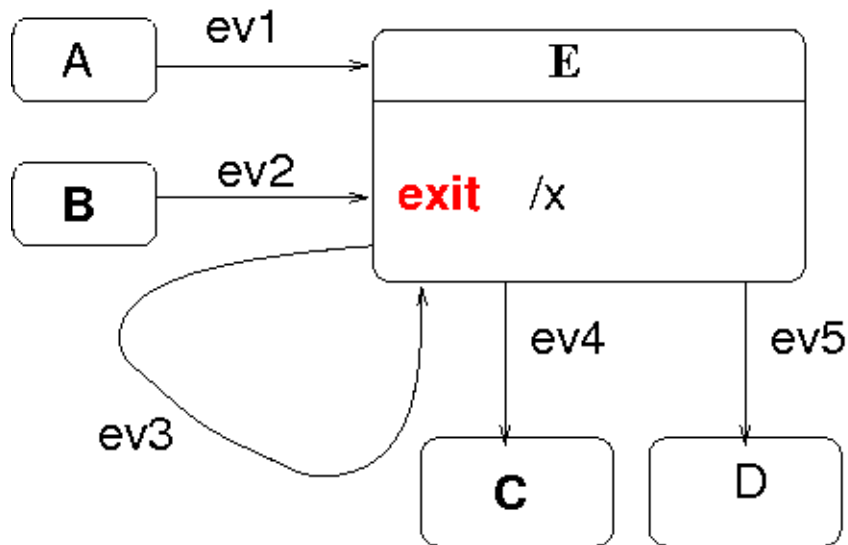
# Entry actions

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



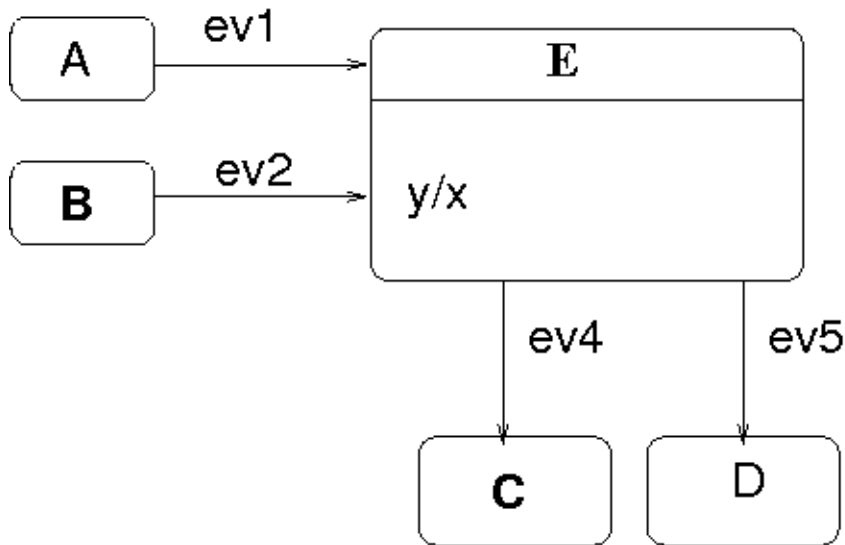
# Exit actions

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



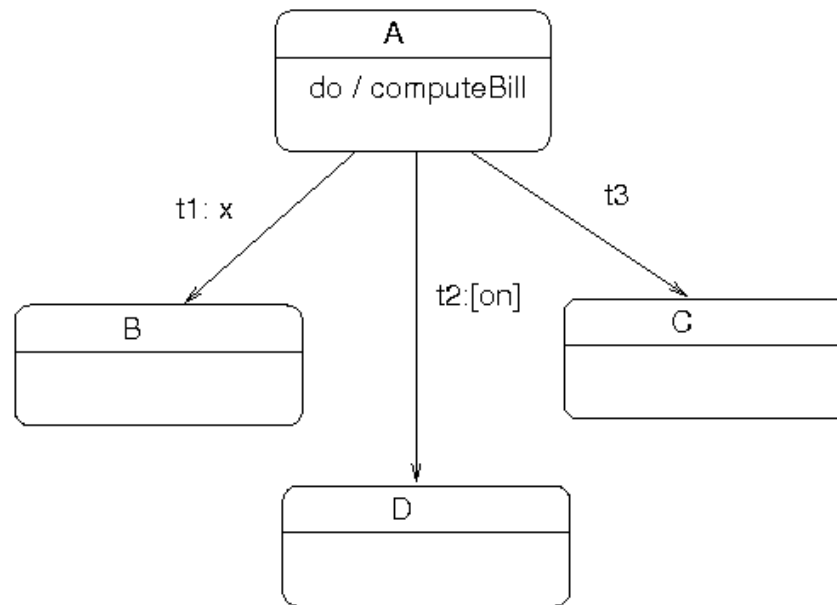
# Internal actions

- 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.

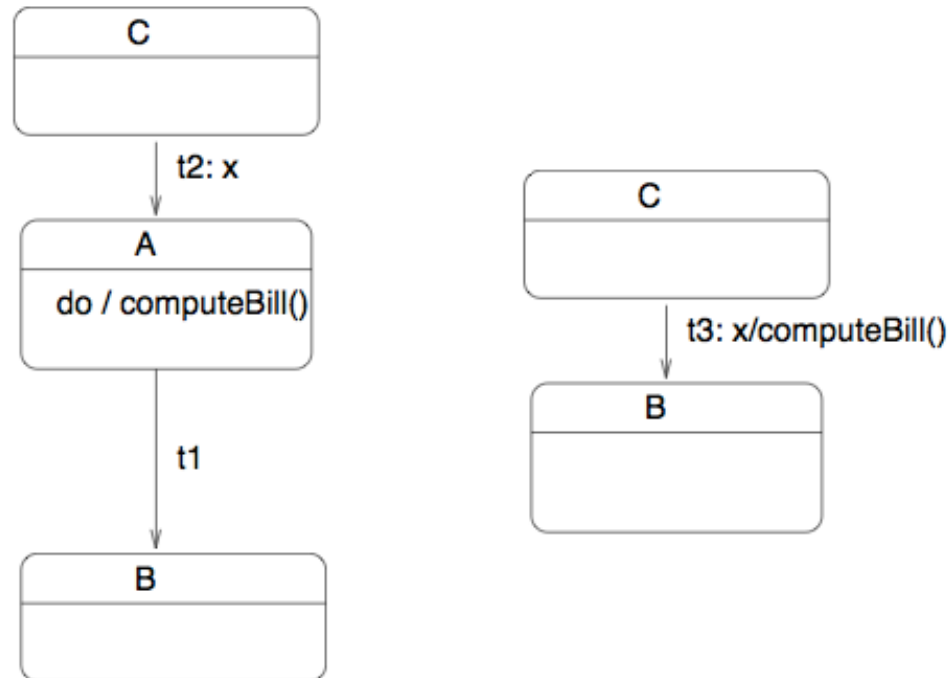
# Synchrony hypothesis

- The *synchrony hypothesis* is the assumption that the system can respond to an input faster than another input can be provided.
  - It's usually a reasonable and valid assumption during requirements modelling.
  - It simplifies state machine models because actions can be used rather than activities.

[Due to Gerard Berry, creator of the specification notation Esterel]

# Actions vs. activities

- If there are no other transitions leaving A, these two models are equivalent, and the second is shorter!



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

# Actions vs. activities

- In an actual implementation, every task will take time
  - Just because we consider actions to occur instantaneously in a state machine *model* does not mean that their implementation will be instantaneous.
- In thinking about whether to make something an action or an activity, consider whether it is *interruptible* by another input, rather than whether it takes time.
  - e.g., “*Creating a user account*” can probably be cancelled in the middle, therefore it is a good candidate to make into an activity rather than an action.

# 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.

# State actions and activities

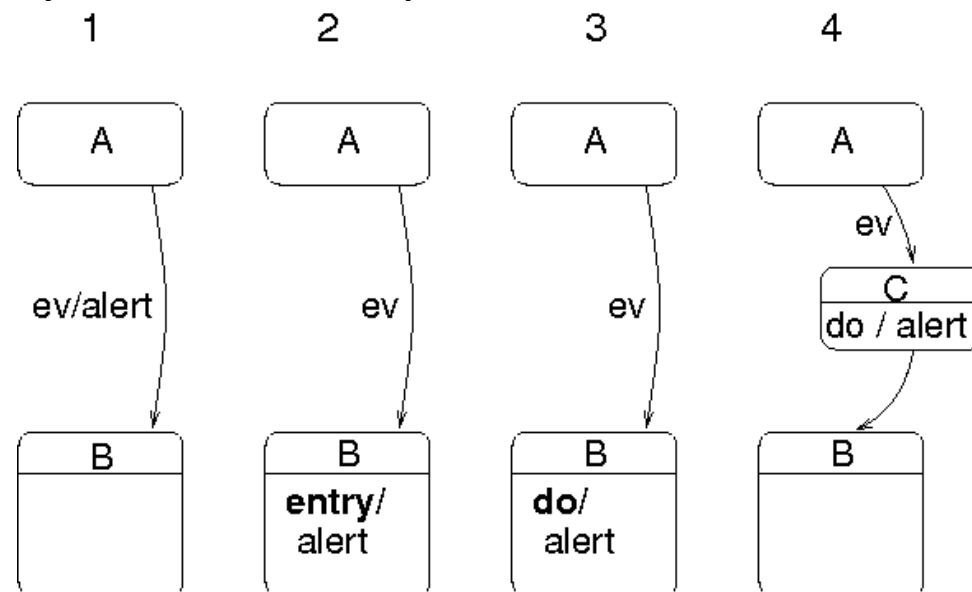
- States can be annotated with entry or exit actions, internal actions, and activities:
  - *entry / action* [red means “keyword”]
  - *event / action*
  - *exit / action*
  - *do / activity*
- A “naked” transition exiting a state (i.e., having no event or condition associated with it) fires as soon 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

# Modelling alternatives

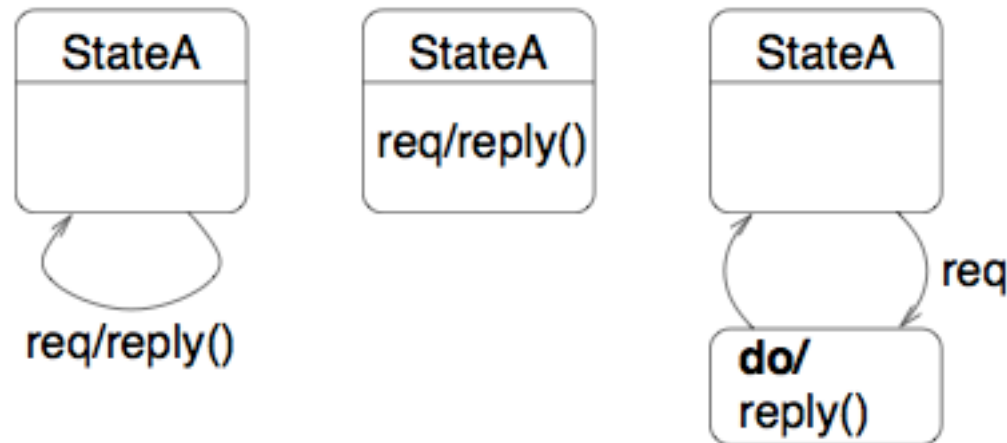
- A system response can be modeled as:
  - an action in state
  - an activity in state
  - an activity in another, special state



**Q:** Which of the above are equivalent? Under what conditions?

# Modelling alternatives

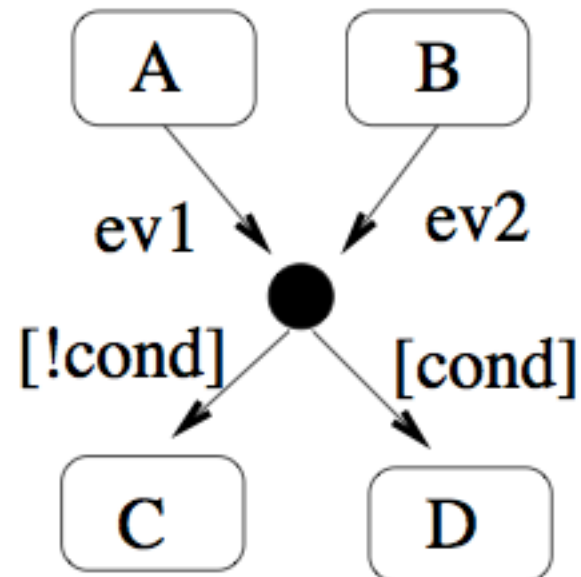
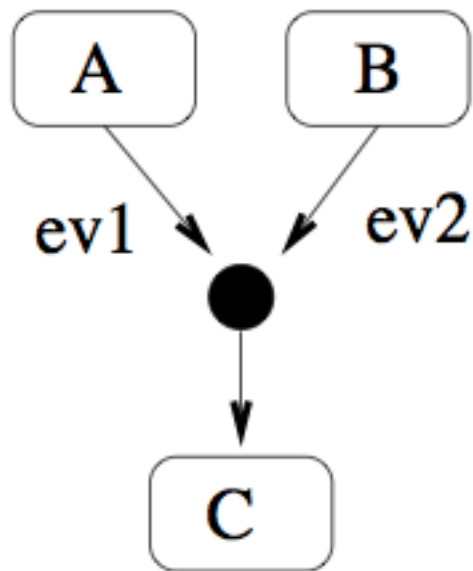
- A system response that does not cause a change in state can be modelled as:
  - an action on a self-transition
  - an action on an internal transition
  - an activity in an intermediate state



Q: Which of the above are equivalent? Under what conditions?

# Junction points

- A junction point is a pseudo-state that reduces clutter by combining common transition segments.
  - Conditions are evaluated at the start of the compound 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 you're 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.

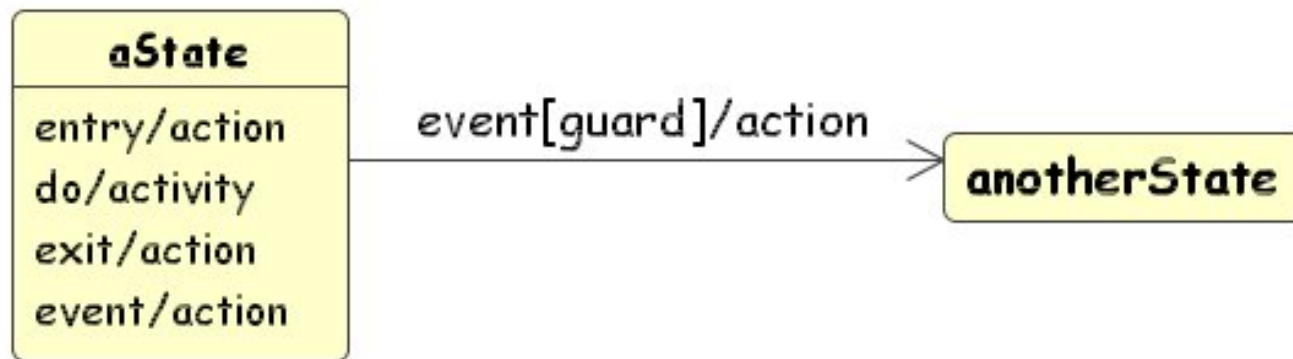
# Time event

- 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.

# Creating an SSMD

- Inputs to this process:
  - Use cases
  - System sequence diagrams
  - Input events
  - Output events
- Output:
  - System state machine diagram (SSMD)
    - Possibly with concurrent components

# Creating an SSMD: Process

1. Identify input and output events
2. Think of a natural partitioning into states activities waiting on an event
  - *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 [not in Fall 2010]
  - Use hierarchy, and entry/exit actions, to abbreviate common behaviour

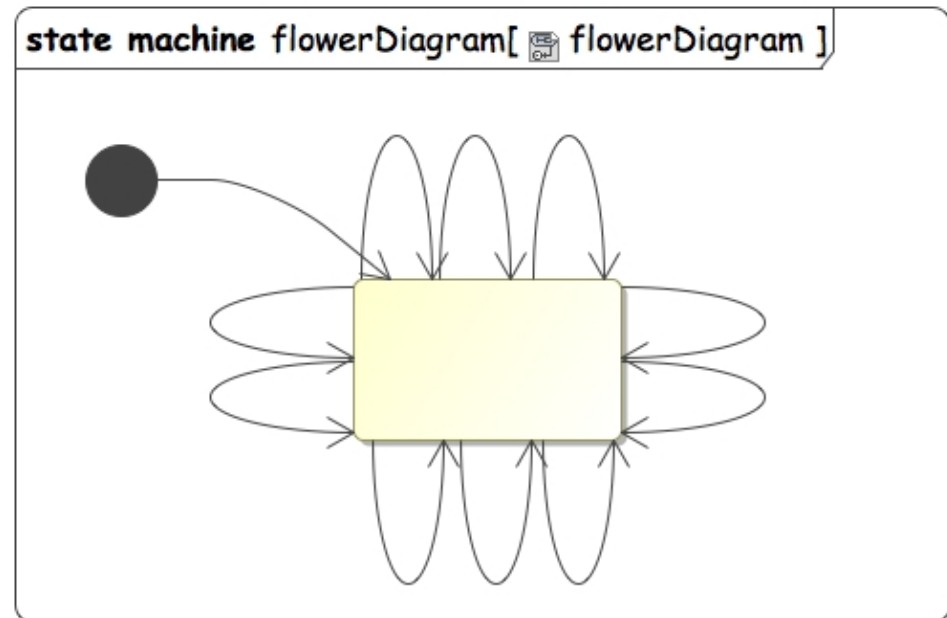
# Creating an SSMD

- Two states are equivalent if they wait on the same event and they have outgoing transitions to equivalent states
- Every scenario of the use cases (and system sequence diagrams) must be a possible behaviour of the system state diagram.
- This can also be done for major subsystems rather than the whole system.

# Modelling decisions

- Choose a natural set of basic states
  - Think of the modes of the system; i.e., when the system waits for input from the environment (or the passage of time) before it can change

- Avoid flower diagrams!



# Validating state machine diagrams

- *Avoid inconsistency*: don't have multiple transitions leave the same state under the same event.
- *Ensure completeness*: a reaction is specified for every possible input at a state.
  - This is an issue particularly when transitions are conditional.
  - 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 sequence diagrams.
  - All paths through the sequence diagrams should be paths in the collection of state machines.

# Good style

- The best state machine model is usually the one that is the clearest. What does this mean?
  - Fewer transitions are better
    - i.e., Use hierarchy to reduce the number of transitions
  - Don't overspecify! If an event is not relevant leaving a state, don't have a transition based on that event.
    - Occasionally this is OK because of the clarity provided by hierarchy.
  - Use history + deep history
  - Use concurrency to recognize orthogonal aspects of the problem.
  - Use variables to model state information that does not reflect flow of control

# CS445 / SE463 / ECE 451 / CS645

## Software requirements specification & analysis

### 7. UML state machine diagrams

Fall 2010 — Mike Godfrey and Dan Berry