Ch. 5.2: Part-A Advanced Behavioral Specification in UML

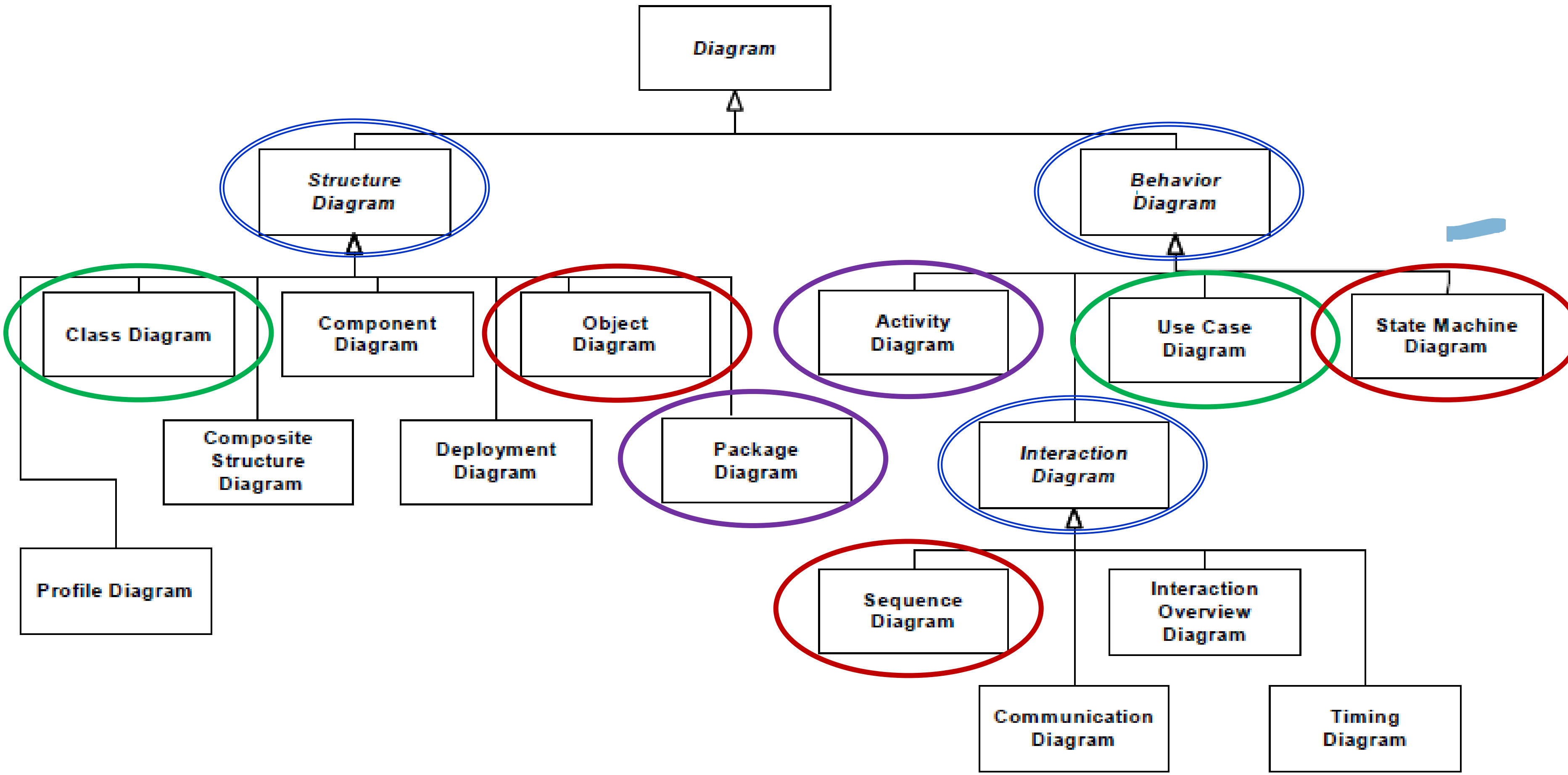
H.S. Sarjoughian

CSE 460: Software Analysis and Design

School of Computing, Informatics and Decision Systems Engineering
Fulton Schools of Engineering

Arizona State University, Tempe, AZ, USA

UML Languages



Element Abstract Syntax

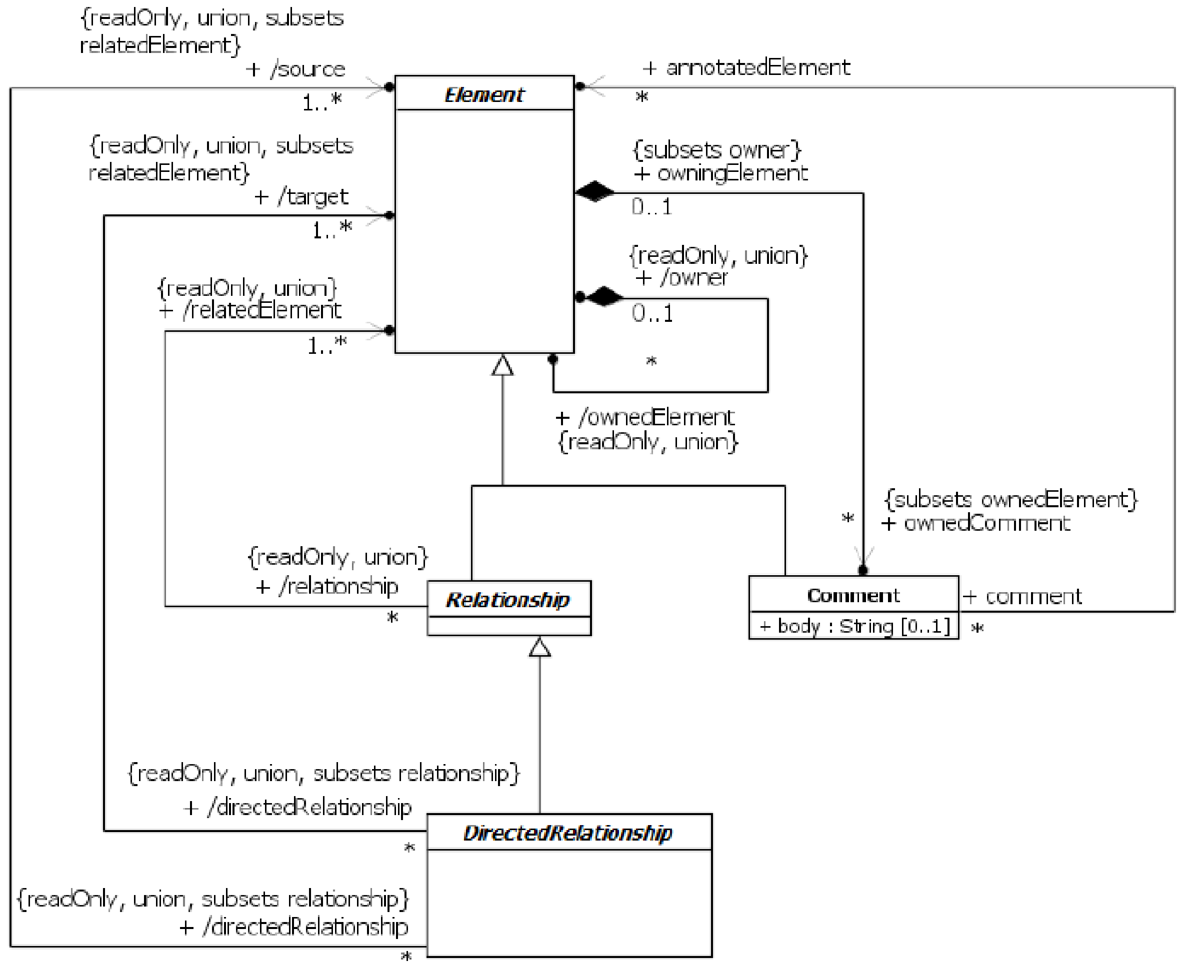
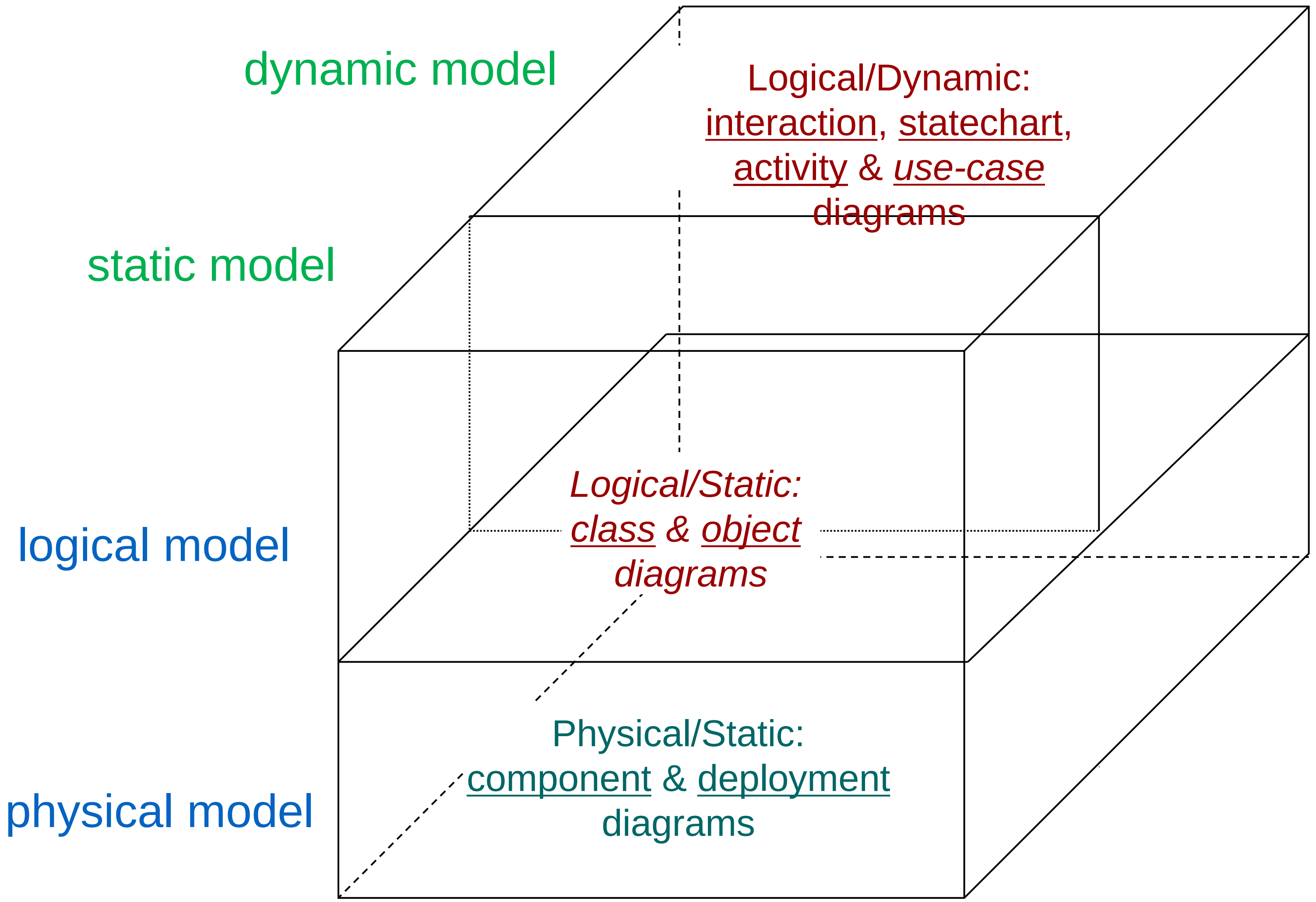
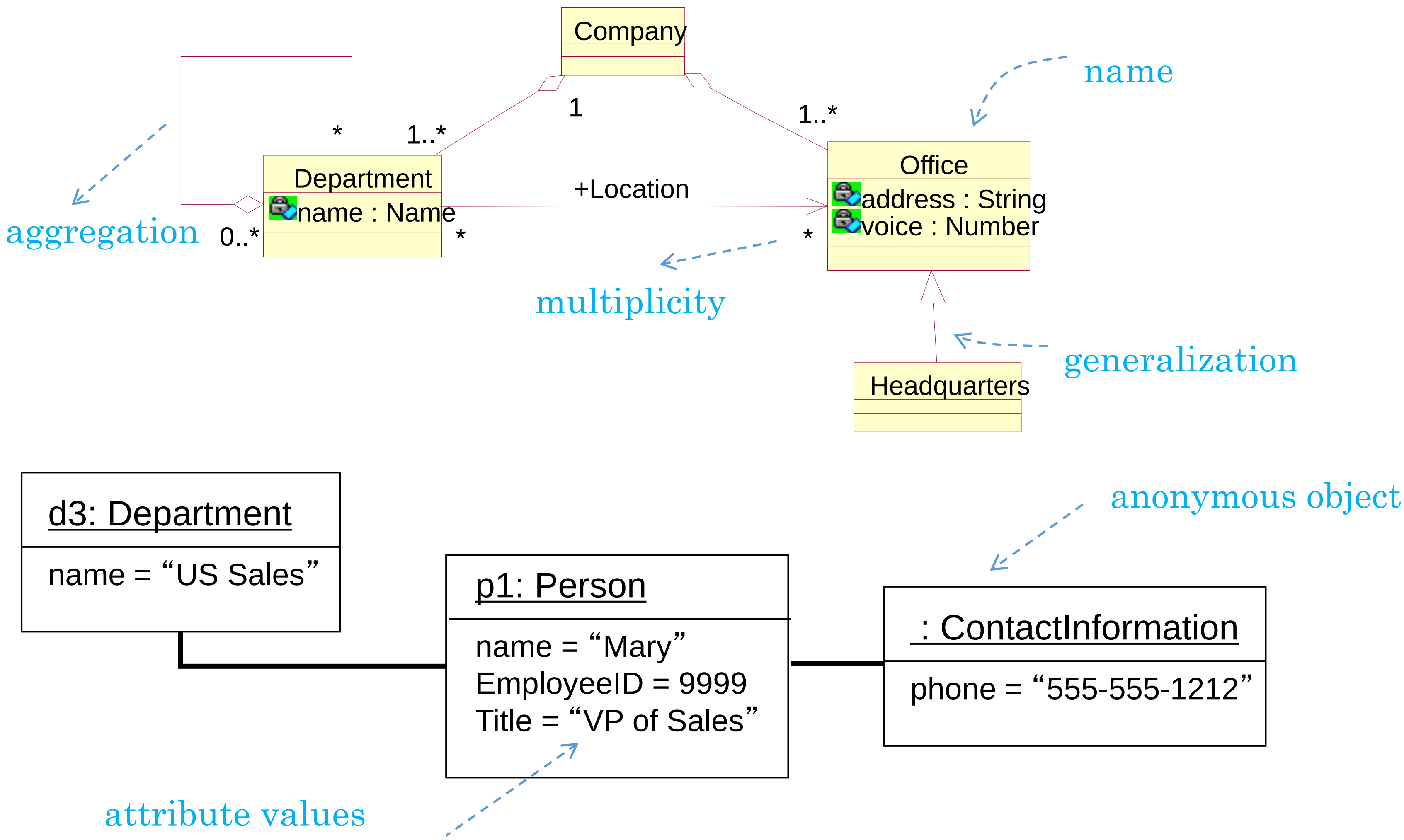


Figure 7.1 Root

Models and Views of Object-Oriented Development



Class and Object Snippet Diagrams



State Machines and Statechart Diagrams

State Machines and Statechart diagrams capture *dynamic* aspects of a software intensive system

- *Behavioral modeling*

Basic elements of a dynamic system can be represented with *states, events, transitions,* and *actions*. These model lifetime of an object in response to external and internal stimuli

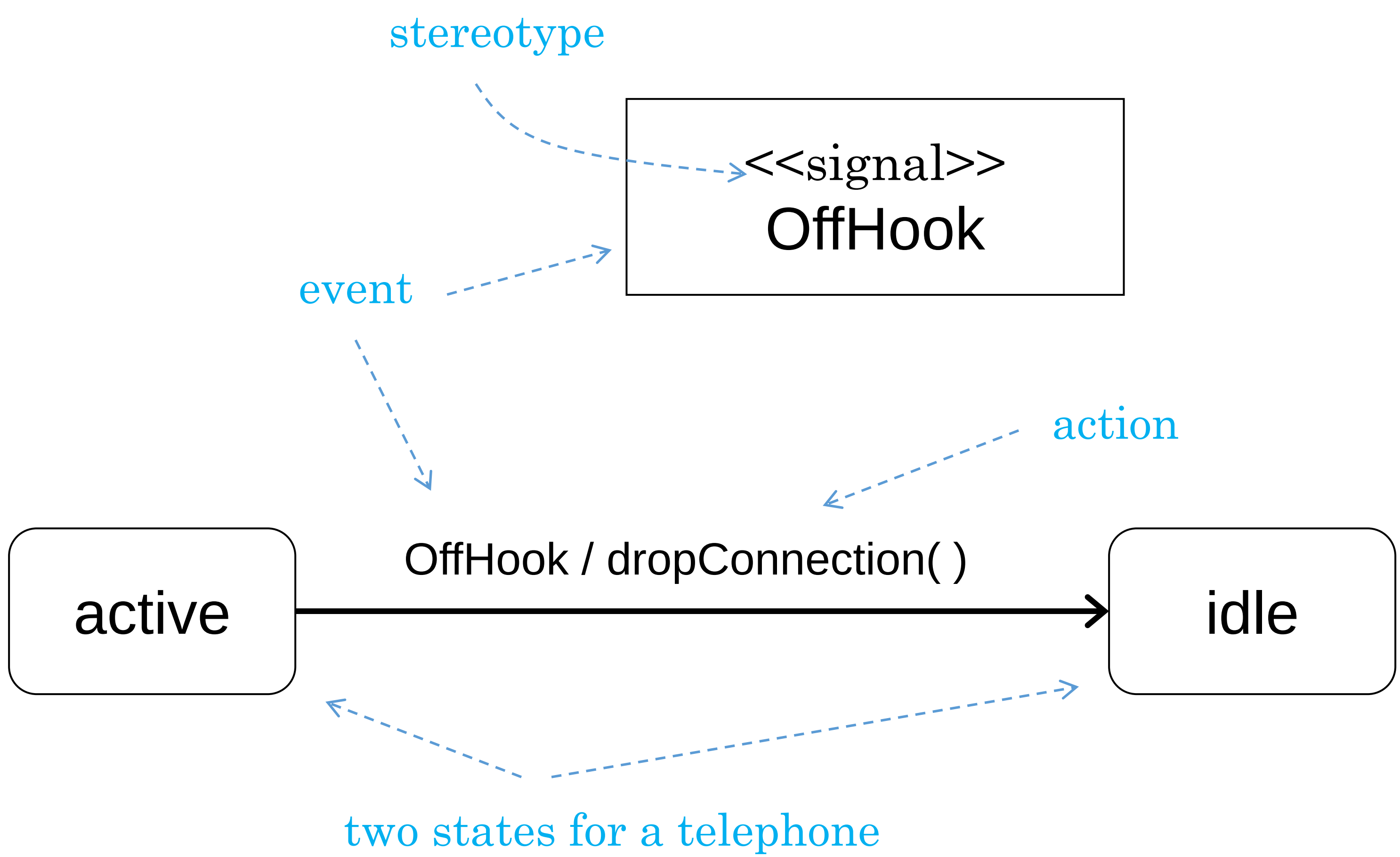
Events

Things that happen (e.g., OffHook) are modeled as events

- Event – is an occurrence of a stimulus that can trigger a state transition or invoke an operation
- Events are either external or internal
  - External: events that are created externally – events that pass between the system and its actors
    - pressing a car’s cruise control button
  - Internal: events that are created internally – events that pass among objects living inside the system
    - overflow of a buffer
- An event has a location in time and space

Events (cont.)

UML notation



Events (cont.)

There are four kinds of events:

- Signals
- Calls
- Change
- Time

- Signals, Time, and Change events are *asynchronous*
- Call events are generally *synchronous*

Signal Events

- **Signals: a signal represents a named object that can be dispatched by one object and received by another in an asynchronous manner**

- *Signals are similar to classes* – they may

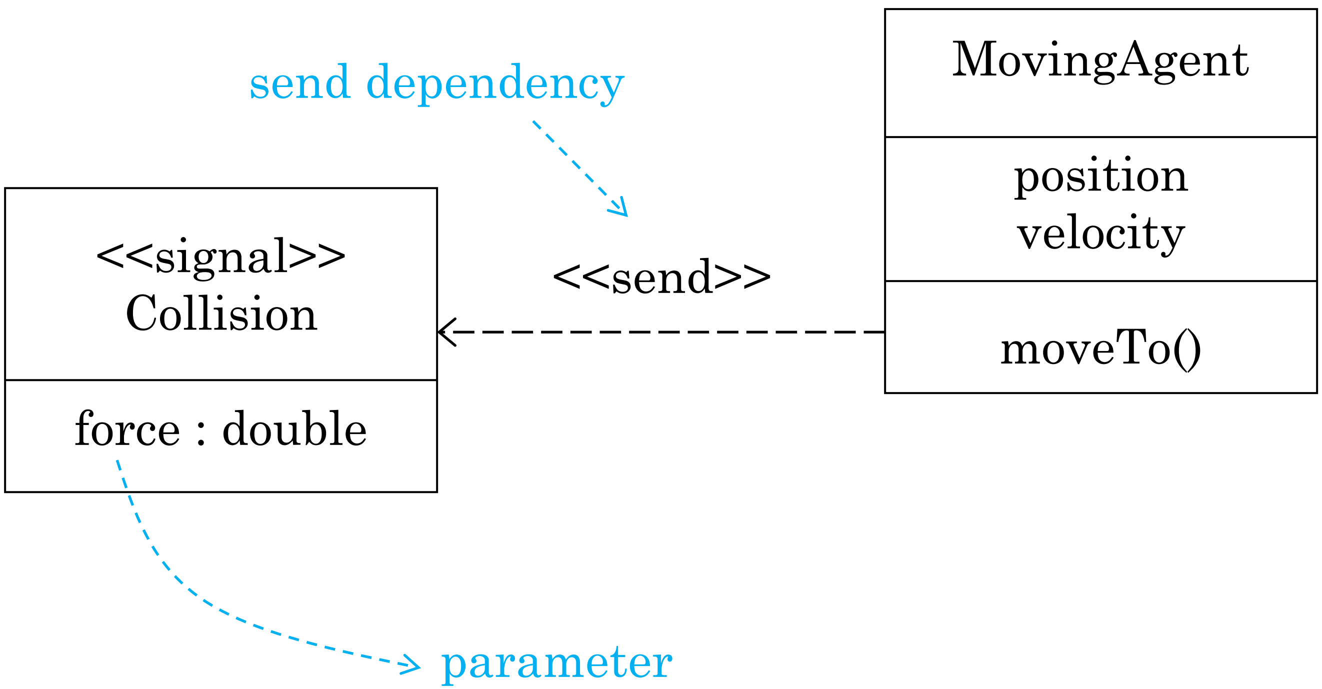
  - have attributes which serve as parameters – e.g., Collision(5.3)
  - have operations
  - have instances
  - participate in generalization relationships – e.g., HardwareFault is generalization of BatteryFault

- Exceptions are a common kind of internal signals

- Signals are sources of events for state transitions and interactions

Signal Events (cont.)

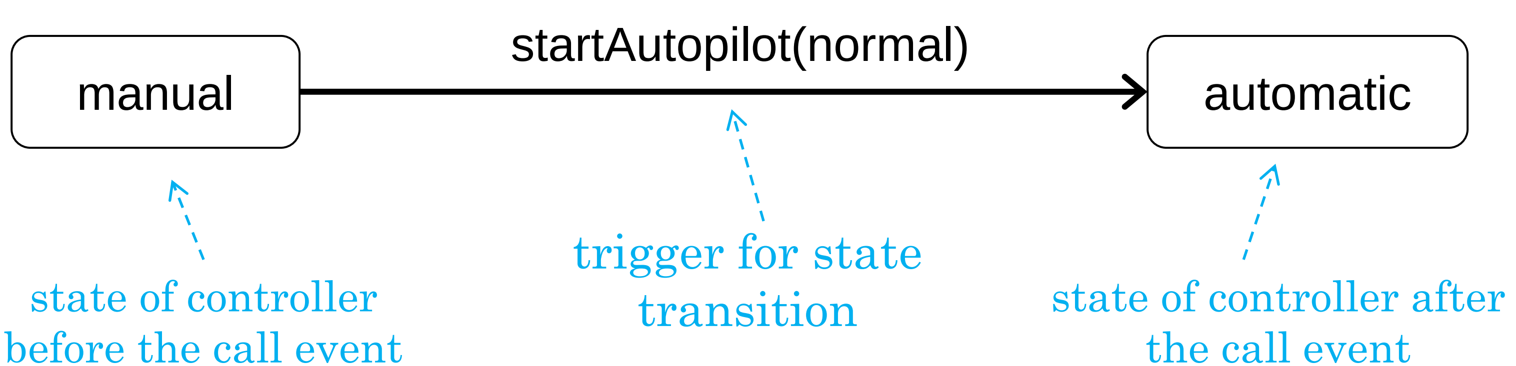
A class behavior can be specified in terms of the signals its operations can send



Call Events

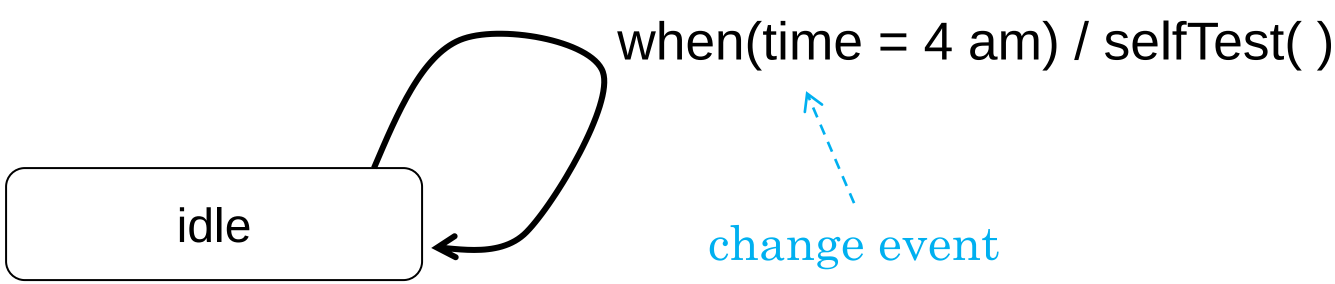
Call event: represents dispatch of an operation

- A call can trigger a state transition in a state machine
- A call is generally synchronous
  - An object (sender) invokes an operation of another object (receiver) which results in the receiver to take control
  - Upon completion of the operation by the receiver, the receiver transitions to a new state and control is returned to the sender



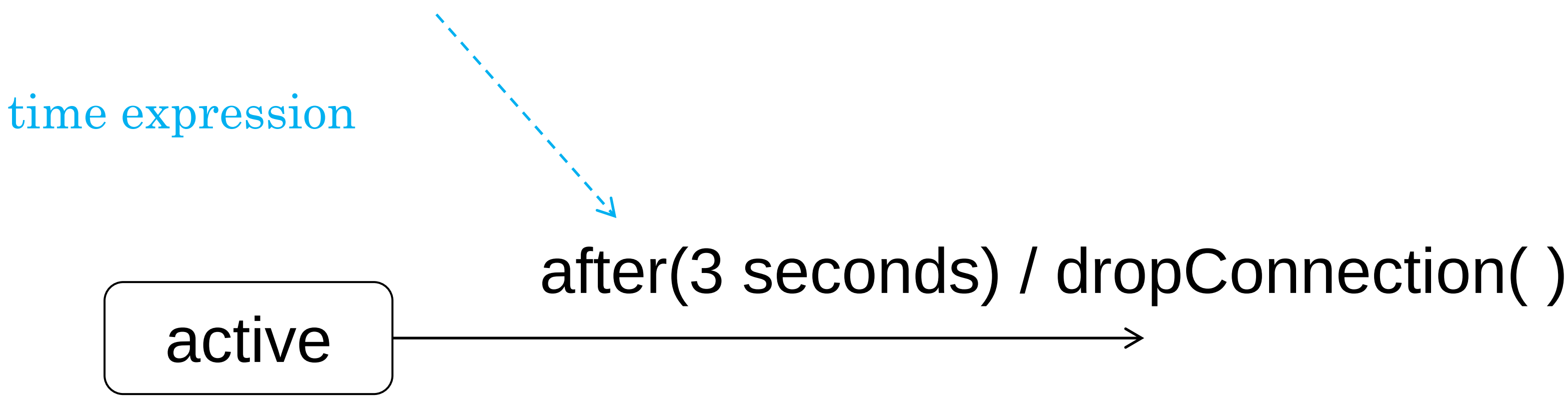
Change Events

- Change event: represents an occurrence of an event when a Boolean expression becomes true as a result of state change or the satisfaction of some condition
  - keyword ***when*** signifies the change event
  - a Boolean expression is used with the *when* keyword to invoke the change event
  - when(Boolean expression)/change event
  - change event ideally needs to be evaluated continuously, instead usually evaluation takes place at some discrete time points



Time Events

- Time event: represents the passage of time – i.e., expiration of a deadline
  - Keyword ***after*** signifies the time event
  - An expression which evaluates to some period of time is used with the *after* keyword to invoke the time event
  - after(expression)/time event



Signal and Call Events: Send and Receive

- **Signals**
  - Signals do not cause change of control from sender to receiver – sender continues along its flow of control after it dispatches a signal
  - Sender can send a signal as broadcast (any object which is listening can get the signal) or as multicast (all objects designated/targeted to receive the signal)

- **Calls**
  - Calls can be synchronous. There exists a rendezvous for the duration of the operation – i.e., the flow of control of the sender is in lock-step with the control of the receiver
  - Calls can be asynchronous. This is similar to signal events in that the sender and receiver are not bound together – the sender issues the call, but it does not wait for a response from the receiver for continuing with its next operation

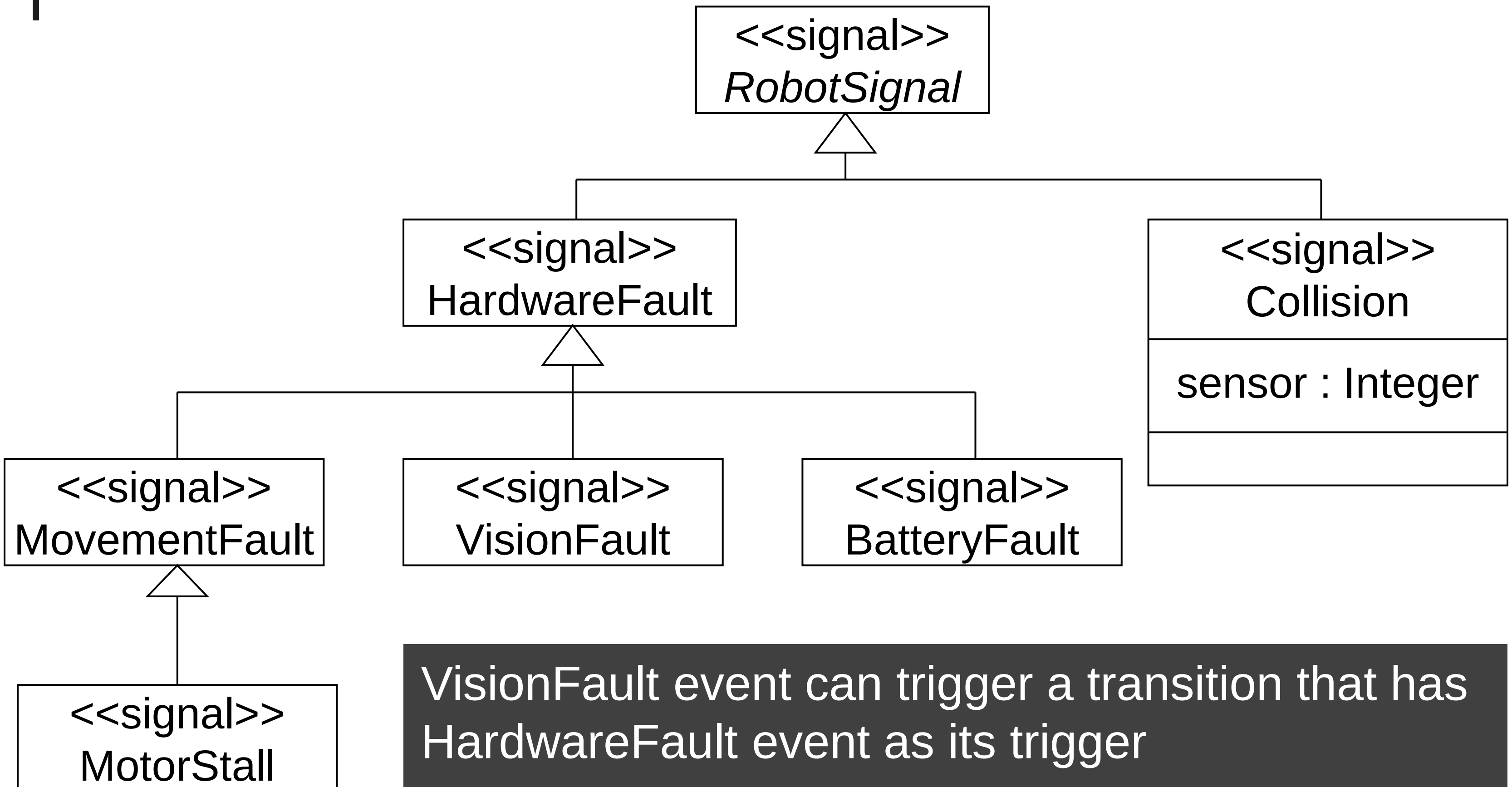
Modeling Signals

In event-driven systems signal events may have hierarchical relationship to one another.

Modeling Procedure

- Identify all different kinds of signal events to which one or more objects may respond
- Determine common kinds of signals and place them in **generalization/specialization** hierarchy. Use inheritance to elevate more general signals and to lower more specialized signals
- Determine where polymorphism (polymorphic event) can be effectively used in the state machines of the objects which can respond to the signals
- Revise signals hierarchy in view of state machines polymorphism

Modeling Signals: An Example



VisionFault event can trigger a transition that has HardwareFault event as its trigger

important for *software safety* analysis & design
(see Structured Analysis and Design chapter)

State Machines

- State machine is a behavior that specifies the sequence of states an object goes through during its lifetime

- A state machine represents an object's behavior in response to
  - **external stimulus** (e.g., an object's operation invoked by another object)
  - **internal changes** (e.g., state transition)
  - **passage of time**

State Machines (cont.)

A state machine models the dynamic behavior of an instance of a class, a (sub-) system, or a use-case

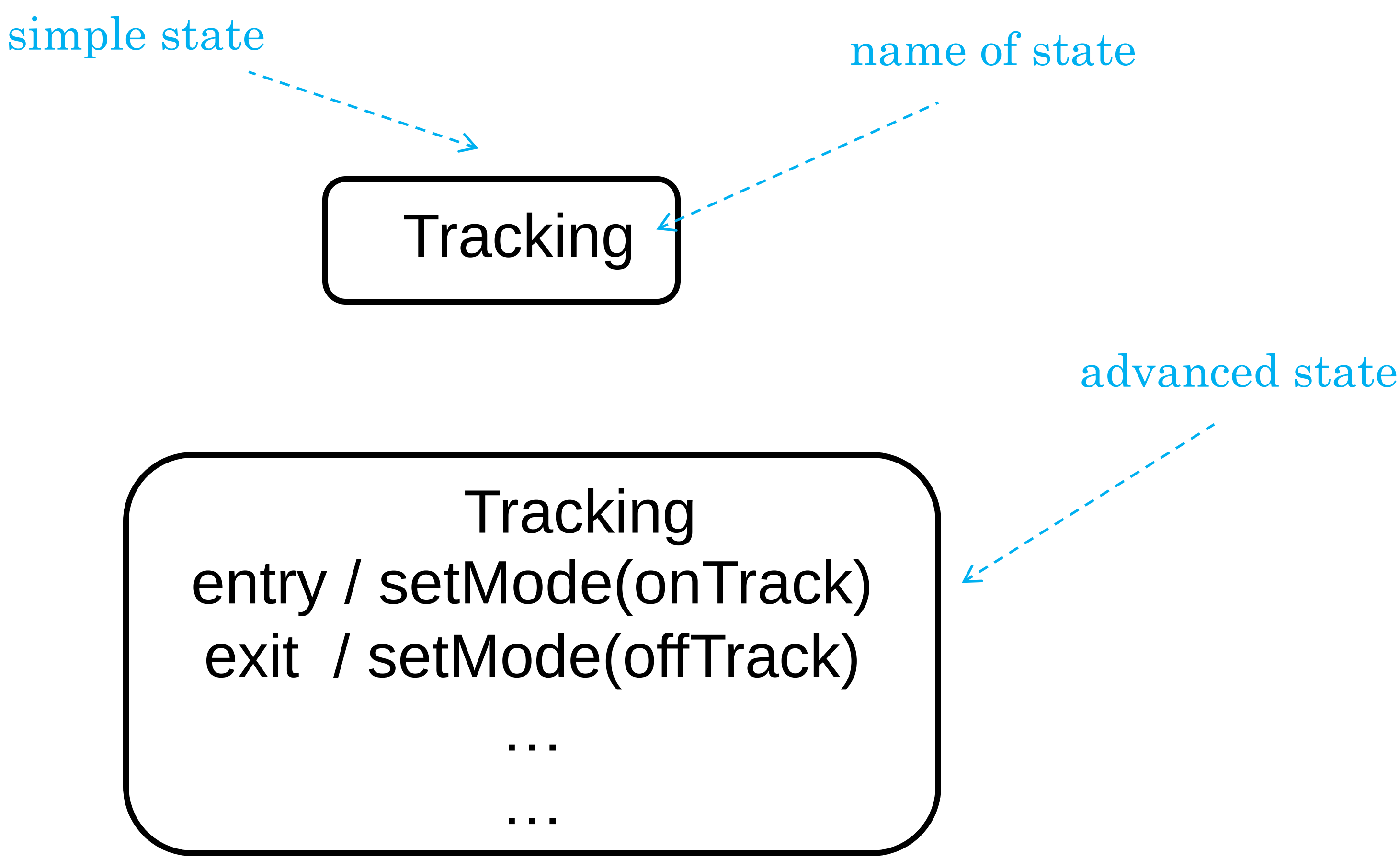
- **A state machine has**
  - set of states
  - set of events
  - sequence of state changes in response to received events (transitions)
  - responses due to received events (actions)
- **A state machine emphasizes**
  - lifetime of an object
  - potential states and transitions from one state to another
  - flow of control from activity to activity (Activity diagram)
- Well-structured state machines are simple, efficient, and adaptable.

State Machines Artifacts

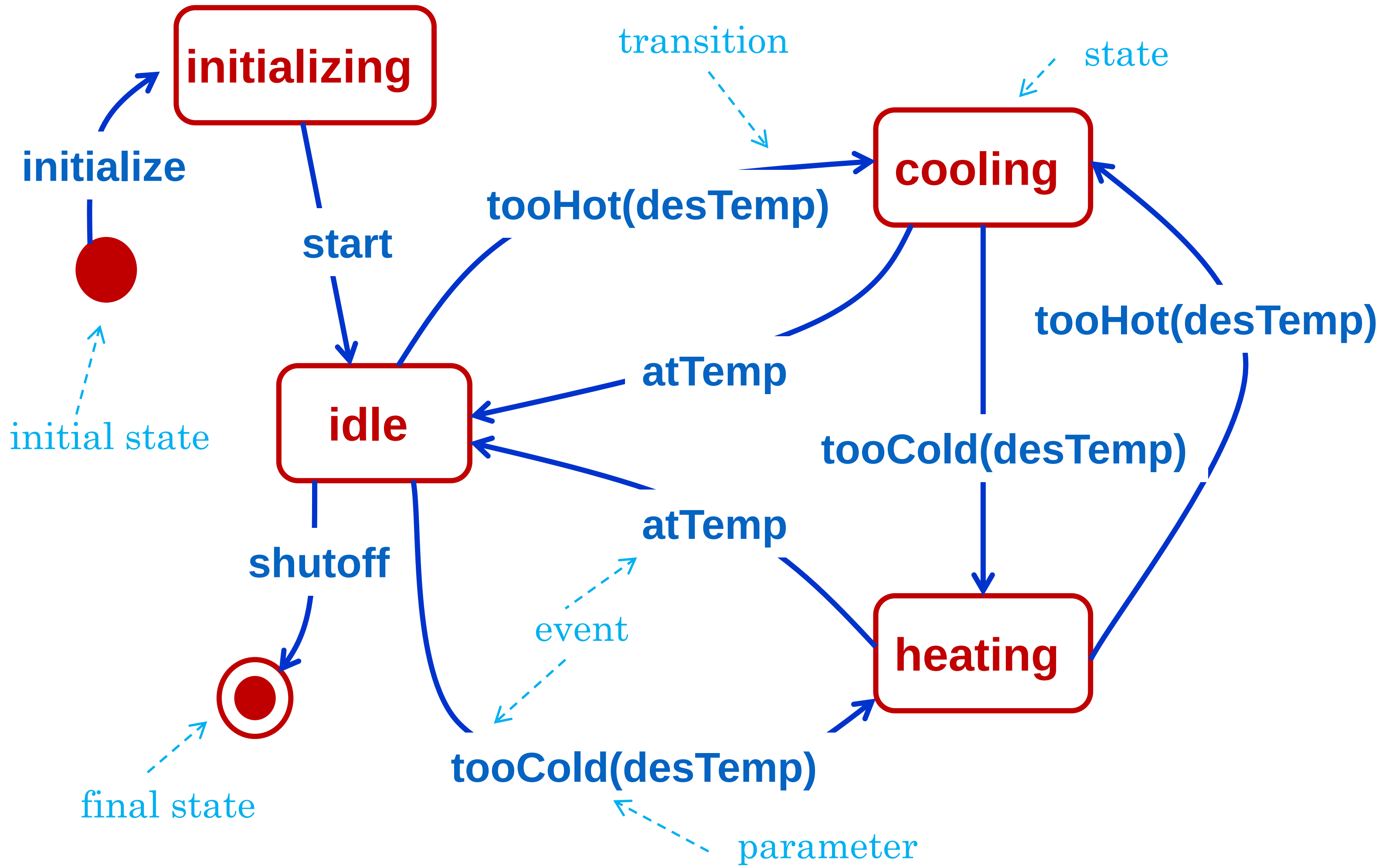
A state machine has of a collection of states and transitions

- **State** – a condition or situation in the lifetime of an object during which some condition is satisfied, some activity occurs, or waits
  - **name** – a textual string that is distinguishable from all other states of the object
  - **entry/exit actions** – actions executed upon entering/exiting a (simple, substate, or composite) state
  - **internal transitions** – transitions that do not cause state change
- State machine can have two special states – **initial & final**
- **Transition** – a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs, and conditions are satisfied

Simple and Advanced States



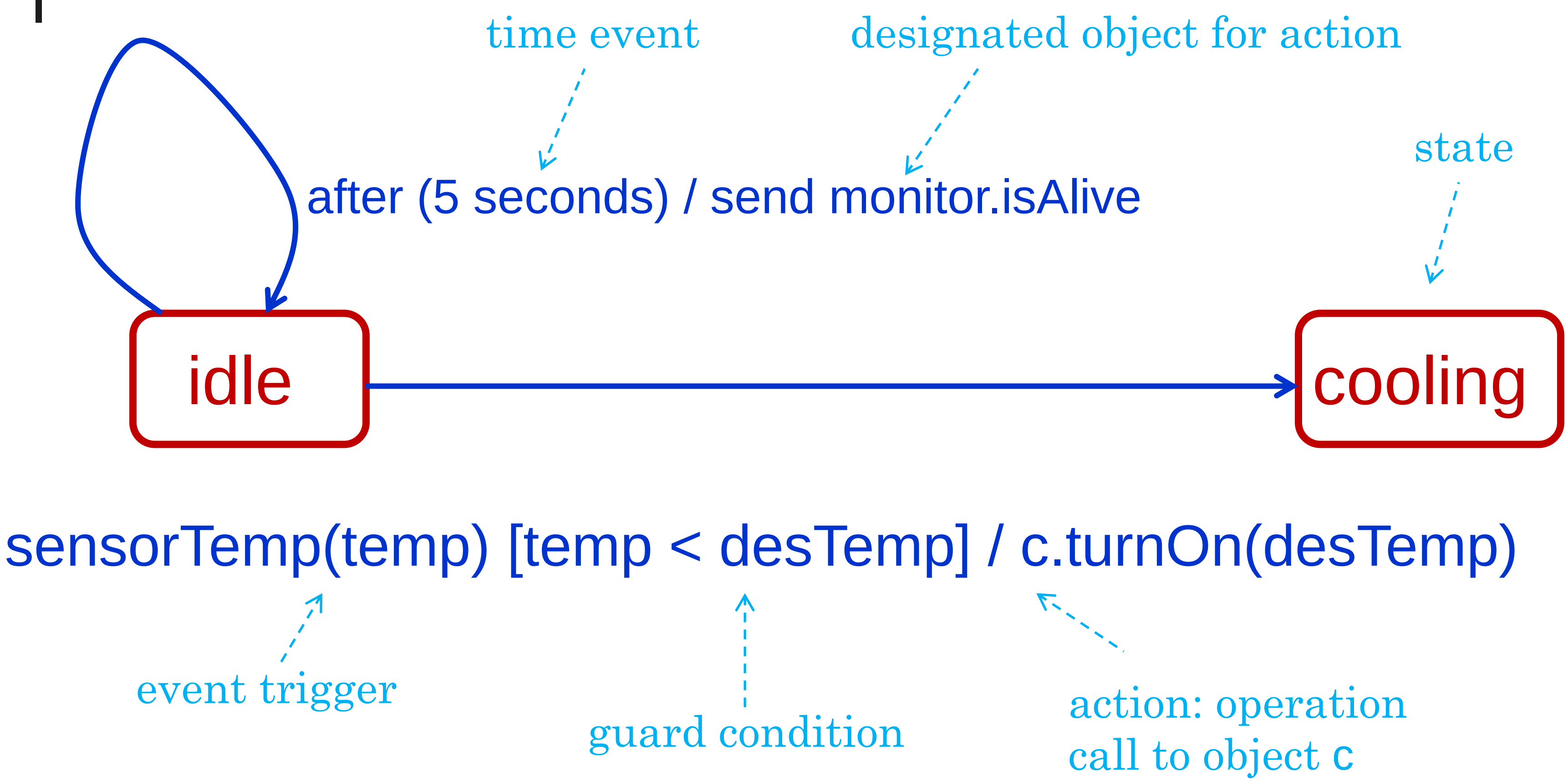
Thermostat State Machine Example



State Machines Artifacts (cont.)

- Transition – a relationship between two states specifying that an object in a state carries out some actions and enters into another state when an event occurs, and some specified condition is satisfied
  - A transition is said to **fire** when a change of state occurs – e.g., Cooler might transition from state "idle" to "active" when event tooHot(desTemp) occurs
- Prior to transition firing, the object is in **source state** and in the **target state** after the transition firing

Transitions



$\texttt{event-name}_{opt}(\texttt{parameter-list})_{opt}$ $[\texttt{guard-condition}]_{opt}$ $/$ $\texttt{action-list}_{opt}$

State Machines Artifacts (cont.)

- Transition has source and target states, trigger event and guard condition, and actions
- A transition which has the same source and target states is called self-transition
- A transition may have multiple sources – represents a join from multiple concurrent states)
- A transition may have multiple targets – represents a fork to multiple concurrent states
- An event that does not trigger a transition is lost or ignored
- Only one transition may fire (within one thread of control) in response to one event occurrence

Transition Artifacts (cont.)

Transition has

- **Source state** – state affected by the transition
- **Target state** – the state that becomes active after the completion of the transition (i.e., transition firing)
- **Event trigger** – the event whose reception triggers a state transition (transition becomes eligible to fire)
- **Guard condition** – a Boolean expression which is evaluated upon the reception of a trigger event; true evaluation results in event to be triggered and false evaluation results in the event to be lost unless another transition is triggered
  - Guard conditions for multiple transitions from a given source state need to be mutually exclusive – i.e., state transition is deterministic
- **Action** – an executable atomic computation; it can act on the object itself (direct) or act on other objects (indirect)

Event Trigger

- Receipt of an event trigger in the source state makes the transition eligible to fire provided its guard condition is satisfied if one is given
- **A transition can occur without a trigger** – called triggerless (or completion) transition is a transition that occurs upon the completion of some activity in the source state
- A signal or call may have parameters whose values are available to the transition
- **An event trigger can be polymorphic** – e.g., a transition whose trigger event is hardwareFault (parent signal) can also be triggered by visionFault (child signal)

Action

- **Action** – an executable atomic operation that results in a change in state, returns a value, or both
  - operation calls to the object that owns the state machine or any other visible object
  - creation and destruction of another object – note that stereotyped call events "created" and "destroyed" are distinct from the creation and destruction actions
  - sending a signal to an object (the keyword **send** is used as prefix to the signal name – visual cue)
  - atomic action – action must complete; *no event can interrupt the execution*

- **Activity** – a non-atomic execution within a state machine
  - an activity is associated with a given state
  - an activity is a sequence of actions – an event may interrupt it

Advanced State Machines

- Advanced features – advanced states and substates – support modeling complex behavior while reducing the number of states and transitions

- Use of idioms (e.g., dispatching the same action whenever a given state is entered) simplify specification of state machines

- UML advanced states and transitions enable multilayer state machines in a systematic fashion

  - **Entry action**
  - **Exit action**
  - **Internal transitions**
  - **Activities**
  - **Deferred events**

Other Artifacts of State Machines

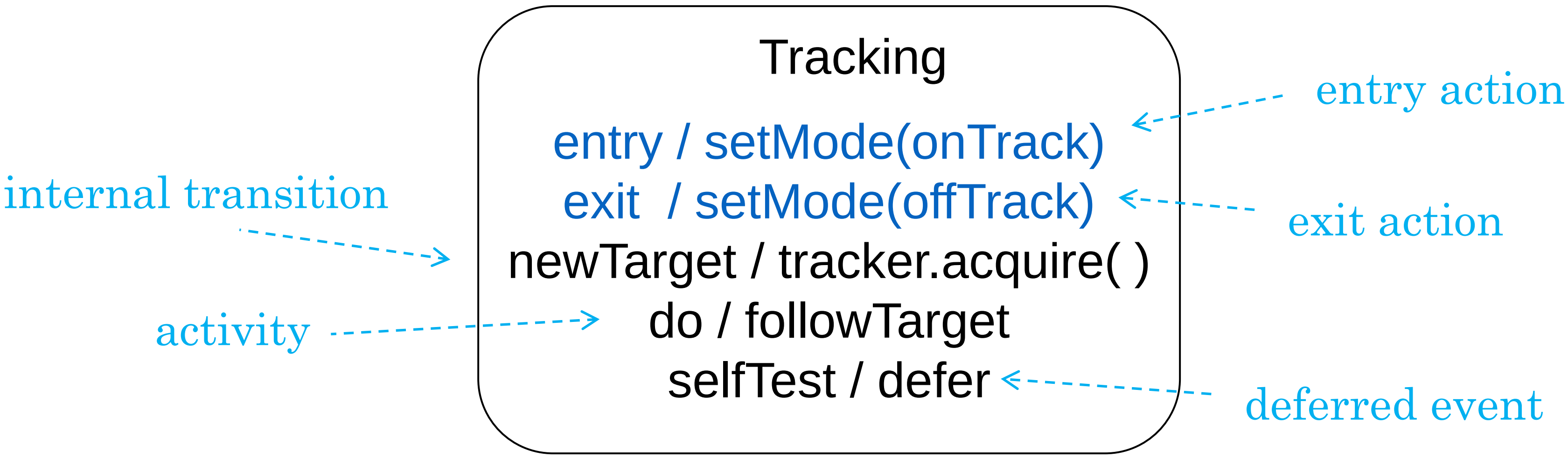
State machines can have sub-state machines

State has

- **name** – a textual string that is distinguishable from all other states of the object
- **entry/exit actions** – actions executed upon entering/exiting a (simple, substate, or composite) state
- **substates** – nested structure of state which may involve disjoint or concurrent substates
- **internal transitions** – transitions that do not cause state change
- **deferred events** – a list of events that are postponed/queued for handling in another state

Advanced State Example

Vehicle Guidance System



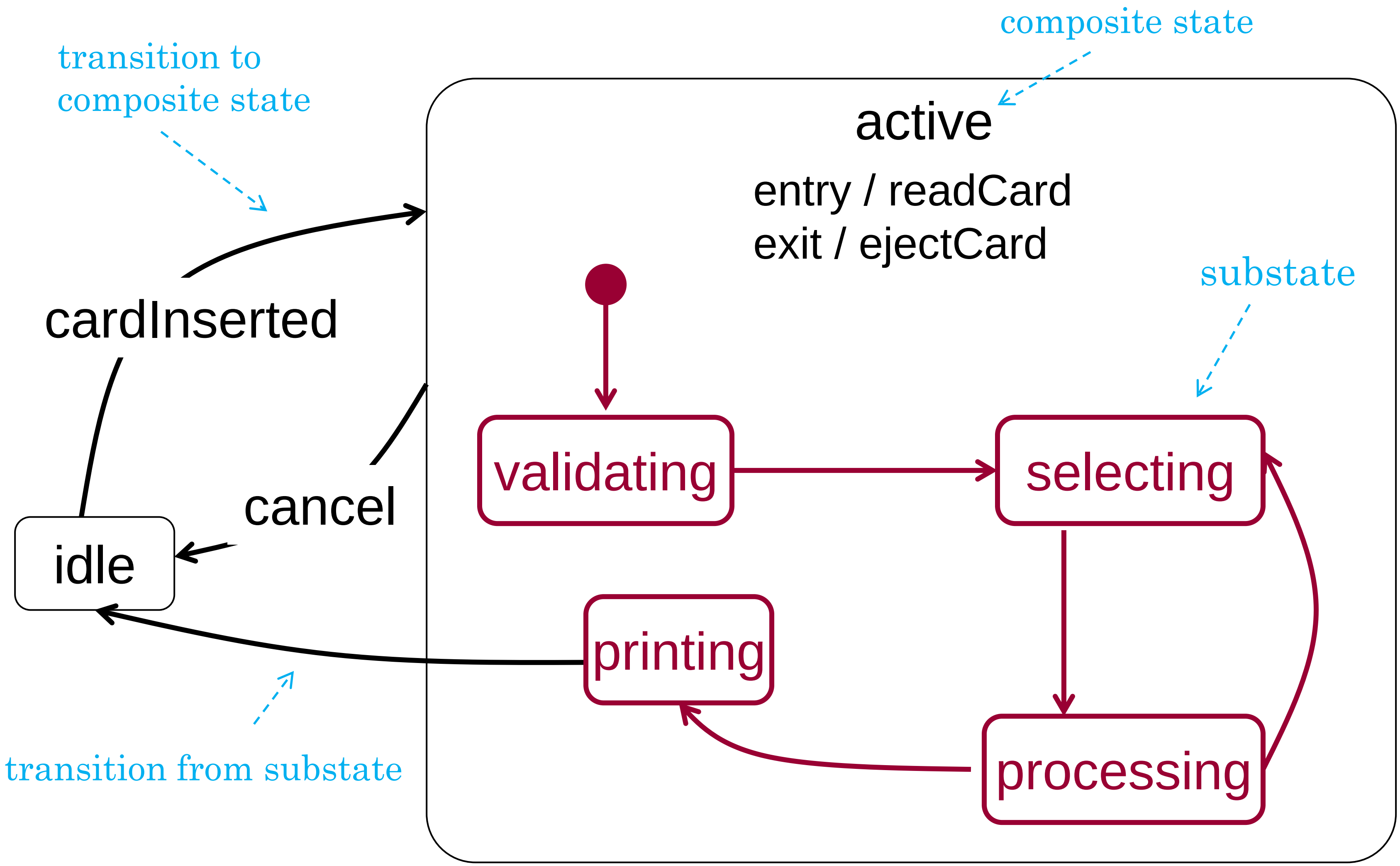
entry and exit actions: each action is dispatched for every state entry/exit regardless of which transition can lead into/out-of a state

internal transitions: these transitions do not cause a state transition as in a self-transition. Specifically, internal transition do not dispatch entry and exit actions

Composite State and Substates

- A composite state has substates or nested states; substates may be nested to any level
- A **composite state** has either
  - sequential states – a disjoint partitioning of a composite state
  - concurrent states – orthogonal partitioning of a composite state
- **Sequential substates**
  - Allows specifying one transition to be used for any number of the sequential substates – one transition is specified for the composite state instead of one transition for each of the sequential substates
- **Concurrent substates**
  - Allows specifying two or more state machines to execute concurrently in the context of an enclosing object

Composite State/Sequential Substates Example



Modeling State Machines

- Determine the context for the state machine of a class
  - collect the neighboring classes – includes parent classes and any class that is reachable by dependency or association
  - neighboring classes can be targets for actions or candidates for including in guard conditions
- Determine **initial** and **final** states; include **preconditions** and **postconditions** associated with initial and final states
- Determine the events object may be subjected to
  - some events may already be available in the object's interface
  - if events are not given in the object's interface, choose/specify neighboring objects which dispatch events to be consumed by the object

Modeling State Machines (cont.)

- Determine start and final states followed by determining the main (top-level) states the object can be in at a given time
  - Any of the main states may have substates
- Connect these states with appropriate transitions (e.g., a transition having a triggering event, a guard condition, and an action)
- Identify entry and exit actions and apply advanced states and transition idioms as applicable
- Expand any of the top-level state as necessary to its substates and all necessary transitions among them
- Verify that all events appearing in state machine match the object's events (signal, call, time, and change events)

Modeling State Machines (cont.)

- Verify that all events expected by the object's interface are accounted for in the state machine – some events may be intentionally ignored

- Verify that all actions in the state machine can be achieved given object's relationships, methods, and operations

- Trace through all possible sequences of state changes/transitions (either manually or by using tools) – verify expected sequences of events and their responses
  - account for any ***unreachable*** (or dead) state
  - eliminate ***superfluous* states** (reduce complexity!!!)

Attributes of a Well-Structured State Machine

- has a **well-defined context** – has access to all the objects visible to the enclosing object
- is **minimal and efficient** – it has minimum number of states and transitions
- states and transitions are **named appropriately** given the vocabulary of the system
- **substates are used as necessary** – for most systems one to two levels will handle complex behavior
- **concurrent substates are used sparingly**

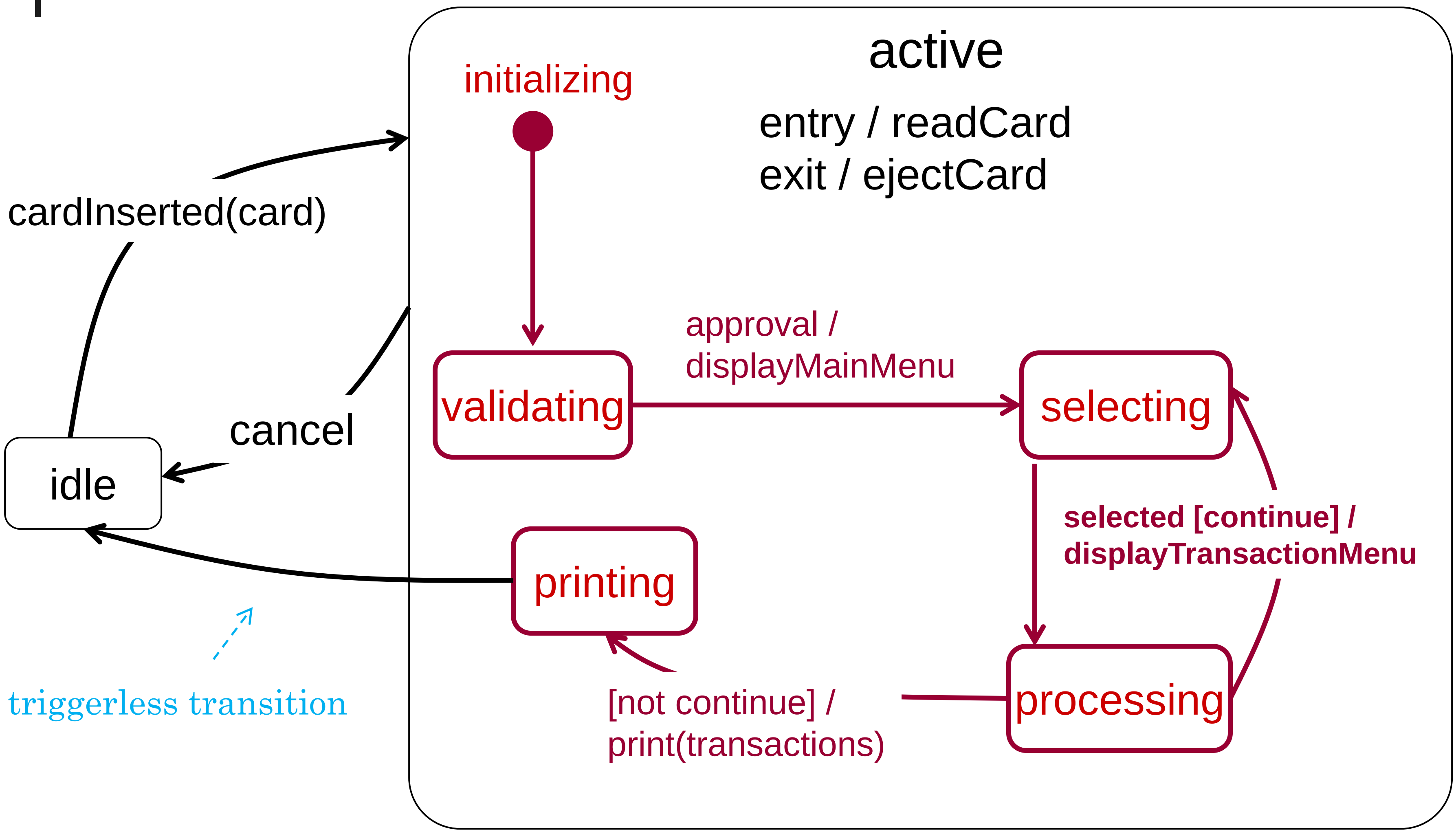
Statechart Diagram

- Statechart diagrams represent **dynamic aspects** (event-ordered behavior) of a system

- Statechart diagrams are especially important for **reactive objects / software**
  - a reactive object is one that its behavior is characterized primarily by its response to events dispatched from outside of its own context

- Statechart diagram shows **flow of control from state to state** – it specifies the <u>sequences of state changes</u> an object can experience during its <u>lifetime</u>

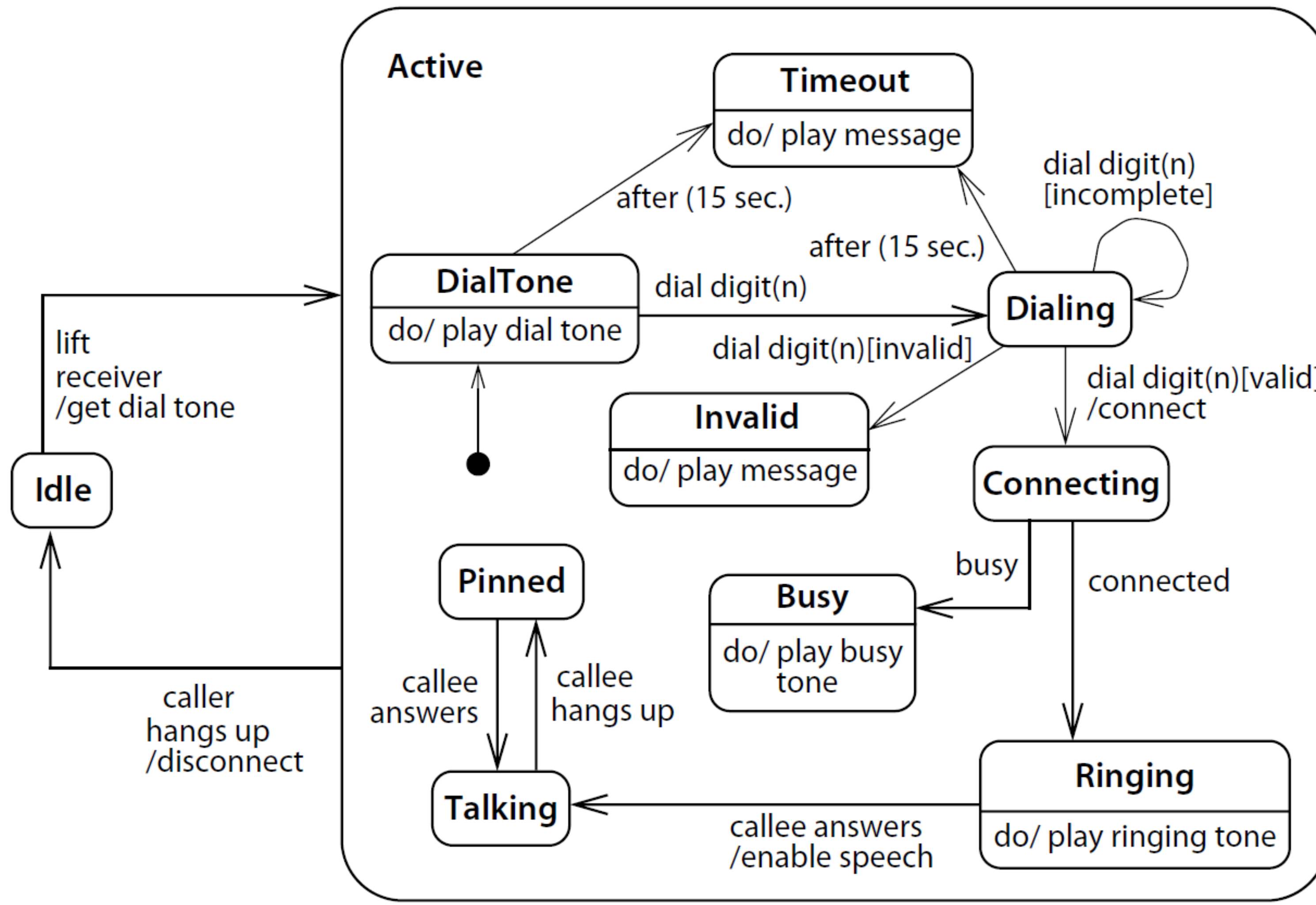
Statechart Diagram Example



References

- *Object-Oriented Analysis and Design with Applications, 3rd Edition, G. Booch, et. al. Addison Wesley, 2007*
- *OMG Unified Modeling Language Specification, http://www.omg.org/spec/, 2019*

Example State Machine



Source: The Unified Modeling Language Reference Manual, Page 439