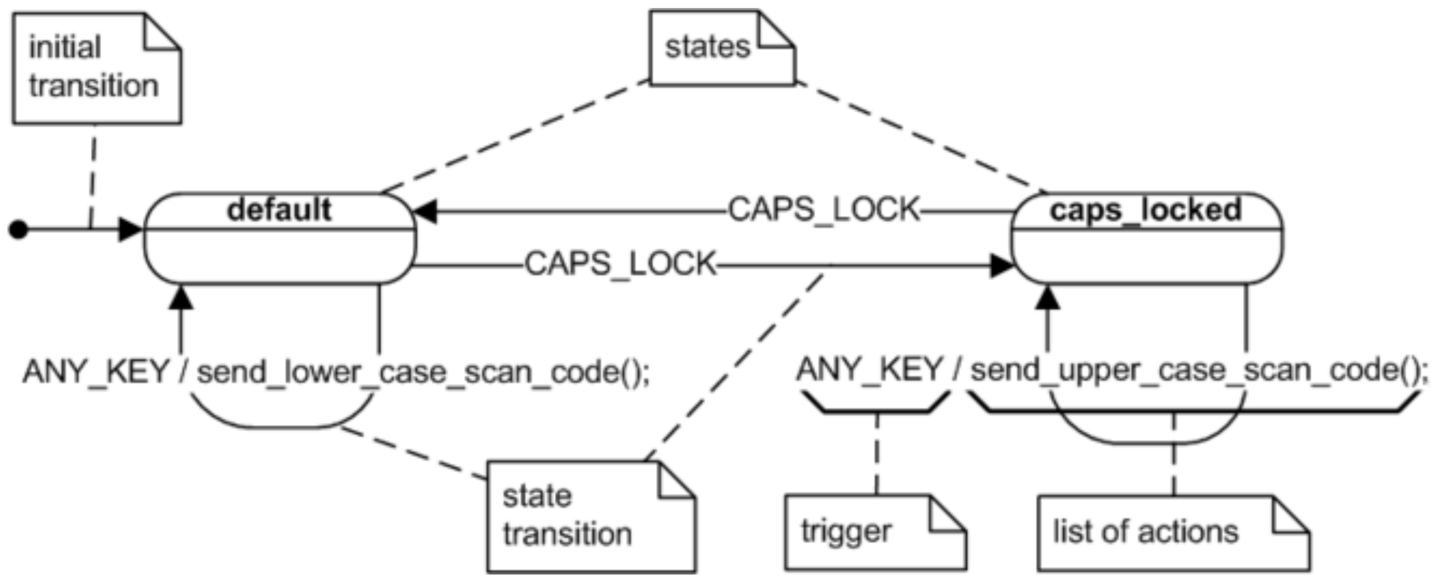
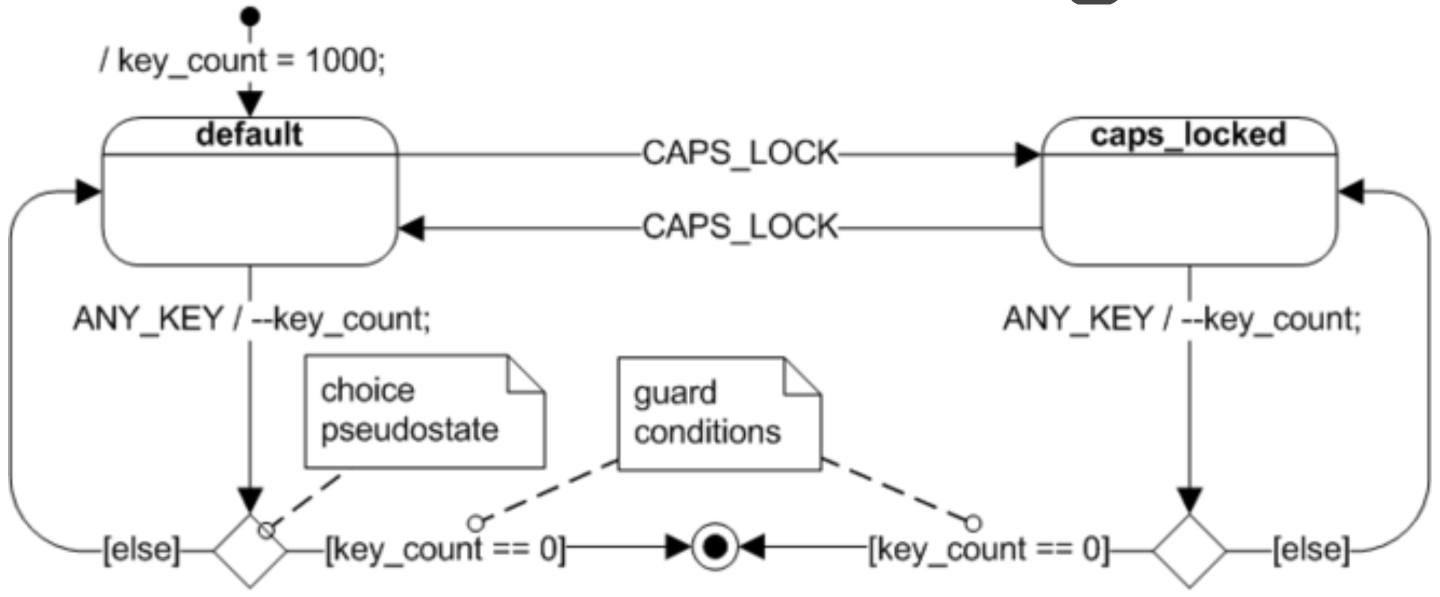
Basic UML State Diagrams



UML state diagrams are directed graphs in which nodes denote states and connectors denote state transitions. For example, Figure 1 shows a UML state diagram corresponding to the computer keyboard state machine. In UML, states are represented as rounded rectangles labeled with state names. The transitions, represented as arrows, are labeled with the triggering events followed optionally by the list of executed actions. The **initial transition** originates from the solid circle and specifies the default state when the system first begins. Every state diagram should have such a transition, which should not be labeled, since it is not triggered by an event. The initial transition can have associated actions.

Project 4 has two kinds of events, clicking the button, and the clock ticking.

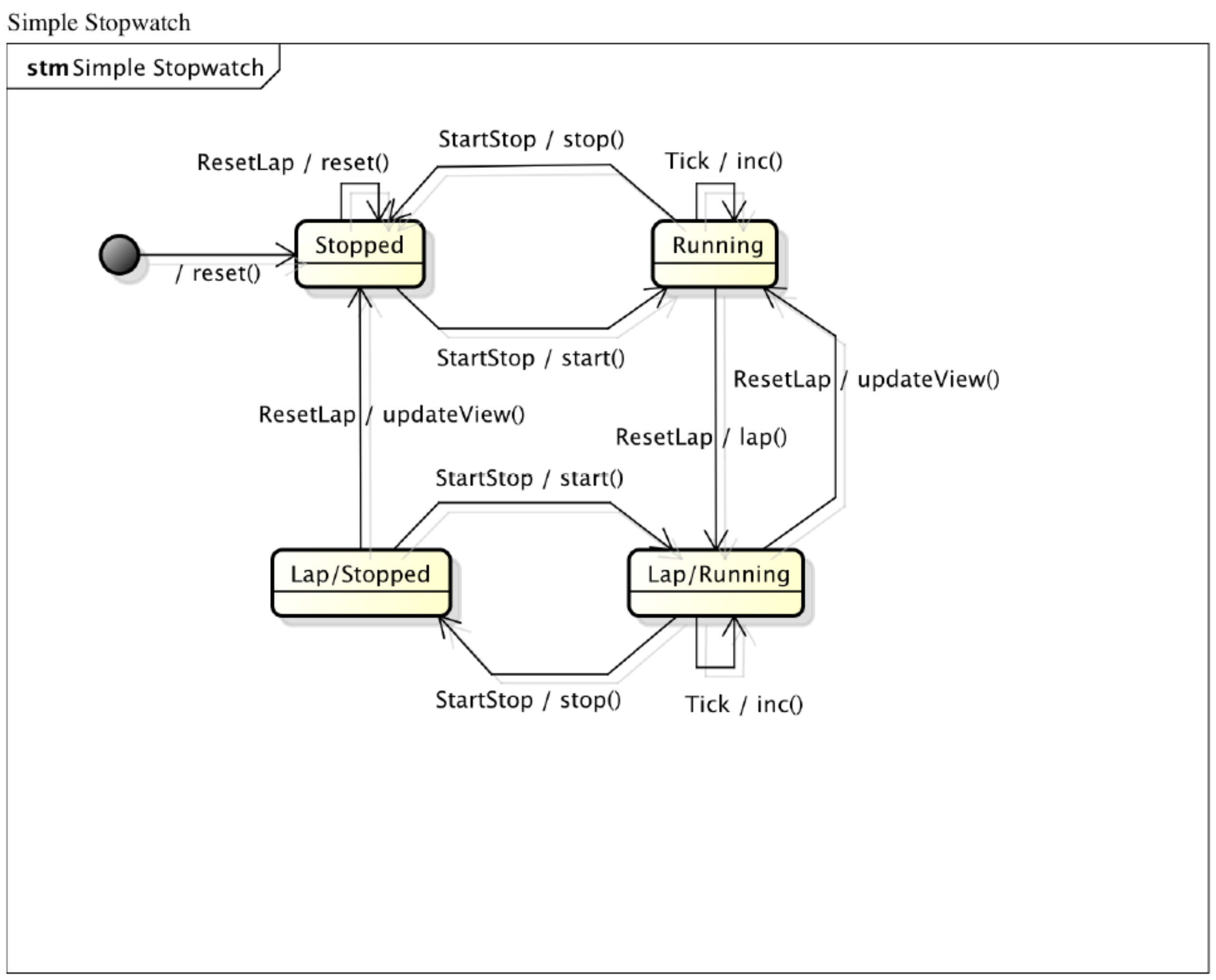
Extended UML State Diagrams



This is an example of an extended state machine, in which the complete condition of the system (called the extended state) is the combination of a qualitative aspect —the "state"—and the quantitative aspects—the extended state variables (such as the down-counter **key_count**). This keyboard “dies” after 1000 keystrokes.
The obvious advantage of extended state machines is flexibility: for example, extending the lifespan of this "cheap keyboard" from 1,000 to 10,000 keystrokes would not complicate the extended state machine at all.
This also shows the use of **guard conditions**, boolean expressions based on the value of extended state variables, like **key_count**, and/or event parameters.
Time-based guard conditions will prove useful in the Project 4 state diagram.

stopwatch-android-java

Reminder: stopwatch State Machine



Alternative Representation for the Stopwatch State Machine

state: triggering event [optional guard] / action → next state

stopwatch example (note: all transitions call update_view(), not shown):

States: **Stopped, Running, Lap/Running, Lab/Stopped**

Events: **StartStop, ResetLap, Tick** (ignored if not listed for a state)

Extended state variables: none

start: *none* / reset() → **Stopped; Stopped**: ResetLap / reset() → **Stopped**

Stopped: StartStop / start() → **Running; Running**: StartStop / stop() → **Stopped**

Running: Tick / inc() → **Running; Running**: ResetLap / lap() → **Lap/Running**

Lap/Running: ResetLap / updateView → **Running**

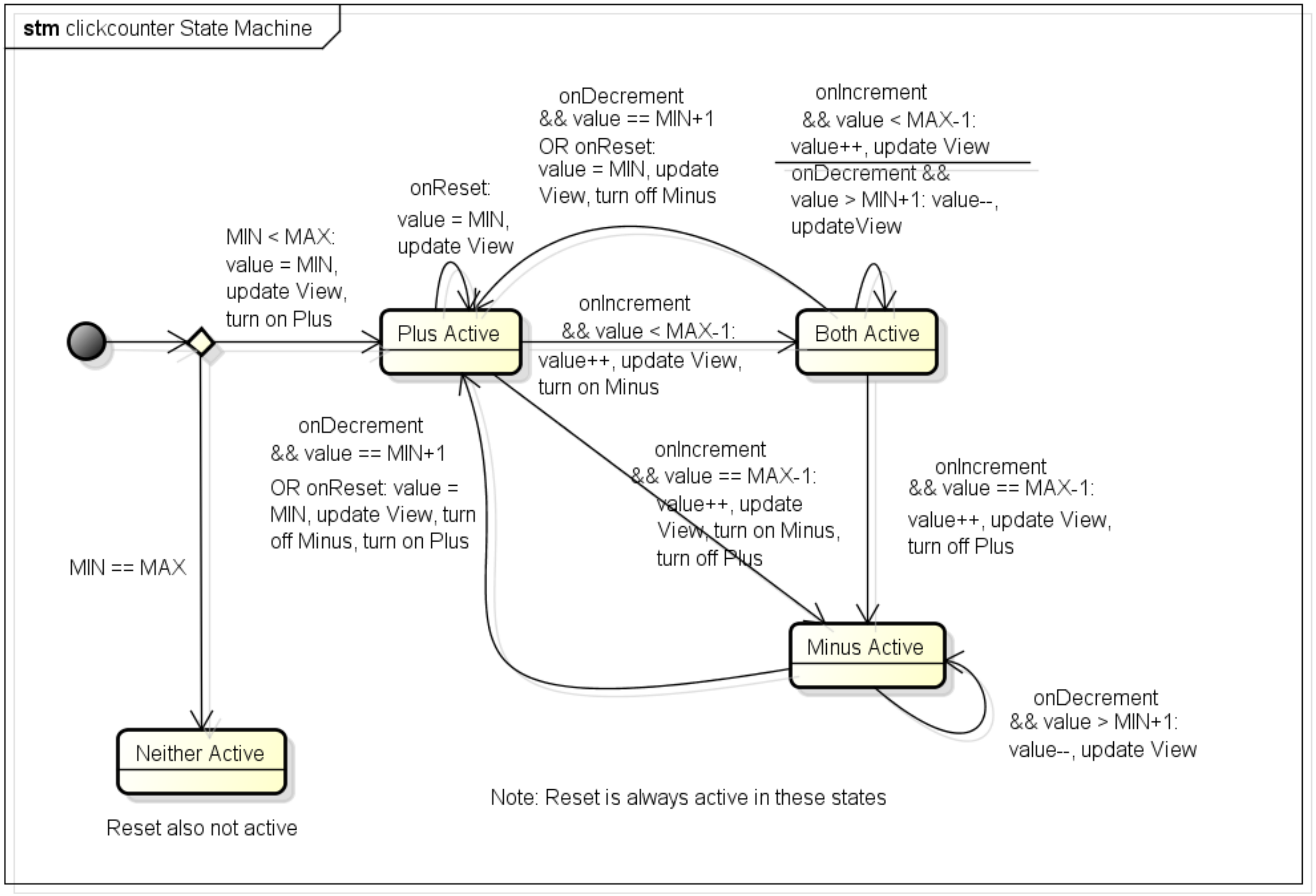
Lap/Running: Tick / inc() → **Lap/Running**

Lap/Running: StartStop / stop() → **Lap/Stopped**

Lap/Stopped: StartStop / start() → **Lap/Running**

Lap/Stopped: ResetLap / updateView() → **Stopped**

Implicit clickcounter State Machine


stm clickcounter State Machine
MIN < MAX: value = MIN, update View, turn on Plus
MIN == MAX
Neither Active
Reset also not active
Plus Active
onReset: value = MIN, update View
onDecrement && value == MIN+1 OR onReset: value = MIN, update View, turn off Minus
onIncrement && value < MAX-1: value++, update View
onDecrement && value > MIN+1: value--, updateView
Both Active
onIncrement && value < MAX-1: value++, update View, turn on Minus
onDecrement && value == MIN+1 OR onReset: value = MIN, update View, turn off Minus, turn on Plus
onIncrement && value == MAX-1: value++, update View, turn on Minus, turn off Plus
onIncrement && value == MAX-1: value++, update View, turn off Plus
Minus Active
onDecrement && value > MIN+1: value--, update View
Note: Reset is always active in these states
powered by Astah

LOYOLA UNIVERSITY CHICAGO

Alternative Representation for the Clickcounter State Machine

state: triggering event [optional guard] / action → next state

clickcounter example (note: all transitions call update_view()):

States: **start**, **inc (INCrement active)**, **dec (DECrement active)**, **both**

Events: **INC (increment), DEC (decrement), RESET**

Extended state variables: **value**; **MIN**, **MAX** (constants)

start: *none* / value = MIN, INC active, DEC inactive → **inc**

inc or both: INC [value < MAX-1] / value++, DEC active → **both**

inc: INC [value == MAX-1] / value++, DEC active, INC inactive → **dec**

dec or both : DEC [value > MIN+1] / value-- → **both**

dec or both: DEC [value == MIN+1] / value = MIN, INC active, DEC inactive → **inc**

all states: RESET / value = MIN, INC active, DEC inactive → **inc**

Project 4 – Functional Requirements – Slide 1 of 2

- The timer has the following controls:
  - One two-digit display of the form 88.
  - One multi-function button. **(Clicking the button causes a click event.)**
- The timer behaves as follows (part 1 of 2):
  - The timer always displays the remaining time in seconds.
  - Initially, the timer is stopped and the (remaining) time is zero.
  - If the button is pressed when the timer is stopped, the time is incremented by one up to a preset maximum of **99**.
    (The button acts as an **increment** button.)
  - If the time is greater than zero and three seconds elapse from the most recent time the button was pressed, then the timer beeps once and starts running. **(When the remaining time is greater than 0 the clock model (see** stopwatch**) is used to send tick events to the state machine.)**

LOYOLA UNIVERSITY CHICAGO

<u>Project 4</u> – Functional Requirements – Slide 2 of 2

- The timer behaves as follows (part 2 of 2):
  - While running, the timer subtracts one from the time for every second that elapses. **(Caused by a clock model <u>tick</u> event.)**
  - If the timer is running and the button is pressed, the timer stops and the time is reset to zero. (The button acts as a **cancel** button.)
  - If the timer is running and the time reaches zero by itself (without the button being pressed), then the timer stops and the <u>alarm</u> starts beeping continually and indefinitely.
  - If the alarm is sounding and the button is pressed, the alarm stops sounding; the timer is now stopped and the (remaining) time is zero. (The button acts as a **stop** button.)
  - The timer <u>handles rotation</u> by continuing in its current state.
- **Your Groups will now create an extended state machine diagram for Project 4's Simple Timer app**