Event Driven Programming and Finite State Automata

Bengt J. Nilsson

Malmö University, Sweden

Source: Johan Holmgren

Event driven programming

Event driven programming

- Event-driven programming can be described as a programming paradigm, where the flow of the program is determined by events:
  - User actions (mouse clicks, key presses)
  - Sensor outputs
  - Messages from other programs
  - ...
- Useful, for example, in graphical user interfaces, where the user input triggers the program/system to perform certain action.

Event driven programming

- An event-driven application typically consists of a main loop that continuously listens for events.
- When an event occurs, a so-called callback function is triggered.
- In an embedded system, it is possible to use interrupts instead of using an infinite main loop.

MALMÖ UNIVERSITY

Finite state machines

MALMÖ UNIVERSITY

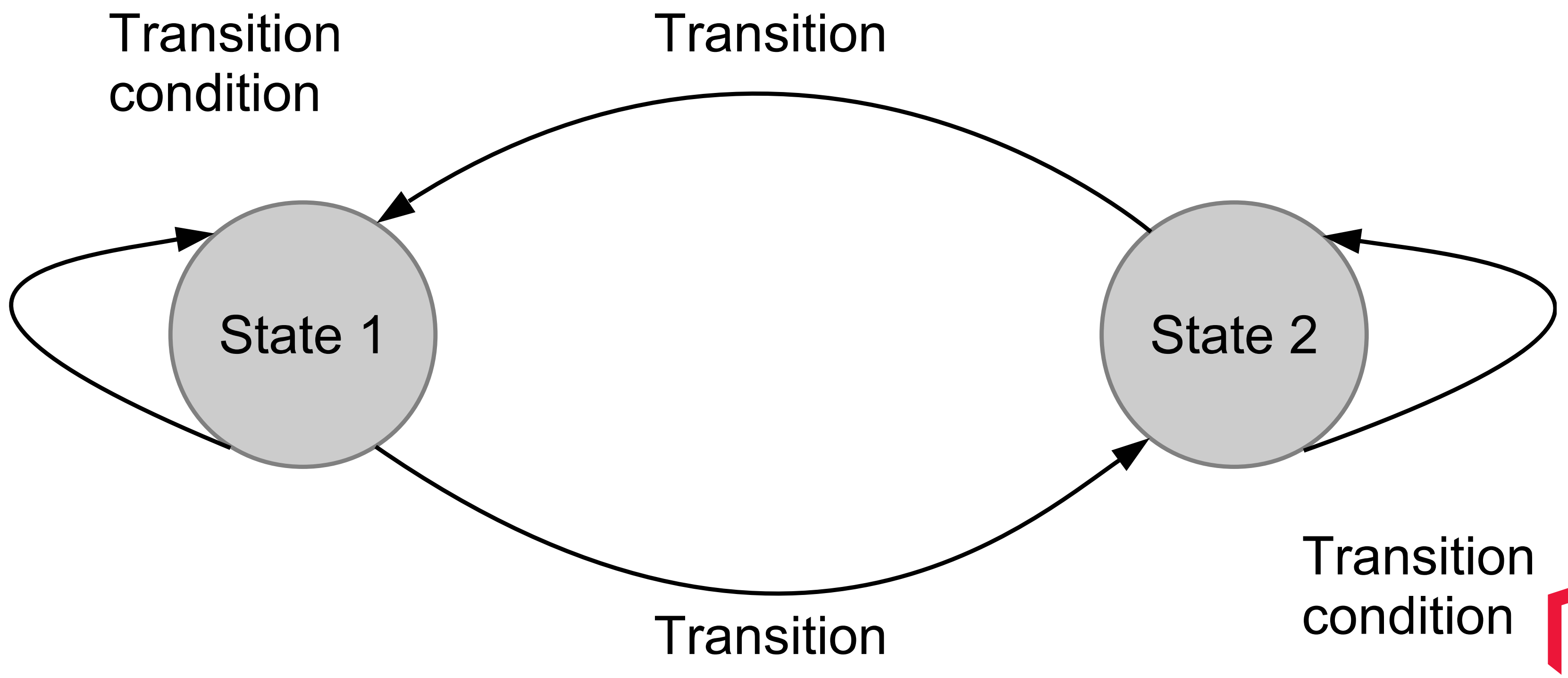
Finite-State Machine

- A finite-State Machine (FSM) is a computational model that describes the behavior of hardware and software
- An FSM contains a finite number of states, and it can be in exactly one state at a time.
- Over time, the machine moves (transits) between states according to the conditions for transitions.
- => Consists of *states*, *transitions* between states, and *conditions* for a transition.

MALMÖ UNIVERSITY

State diagrams

- A state diagram is used to describe the behavior of an FSM.
- Illustrates how the states of the FSM is connected using transitions.
- Transitions are driven by things that happens (events).



Example: Swimming pool heater

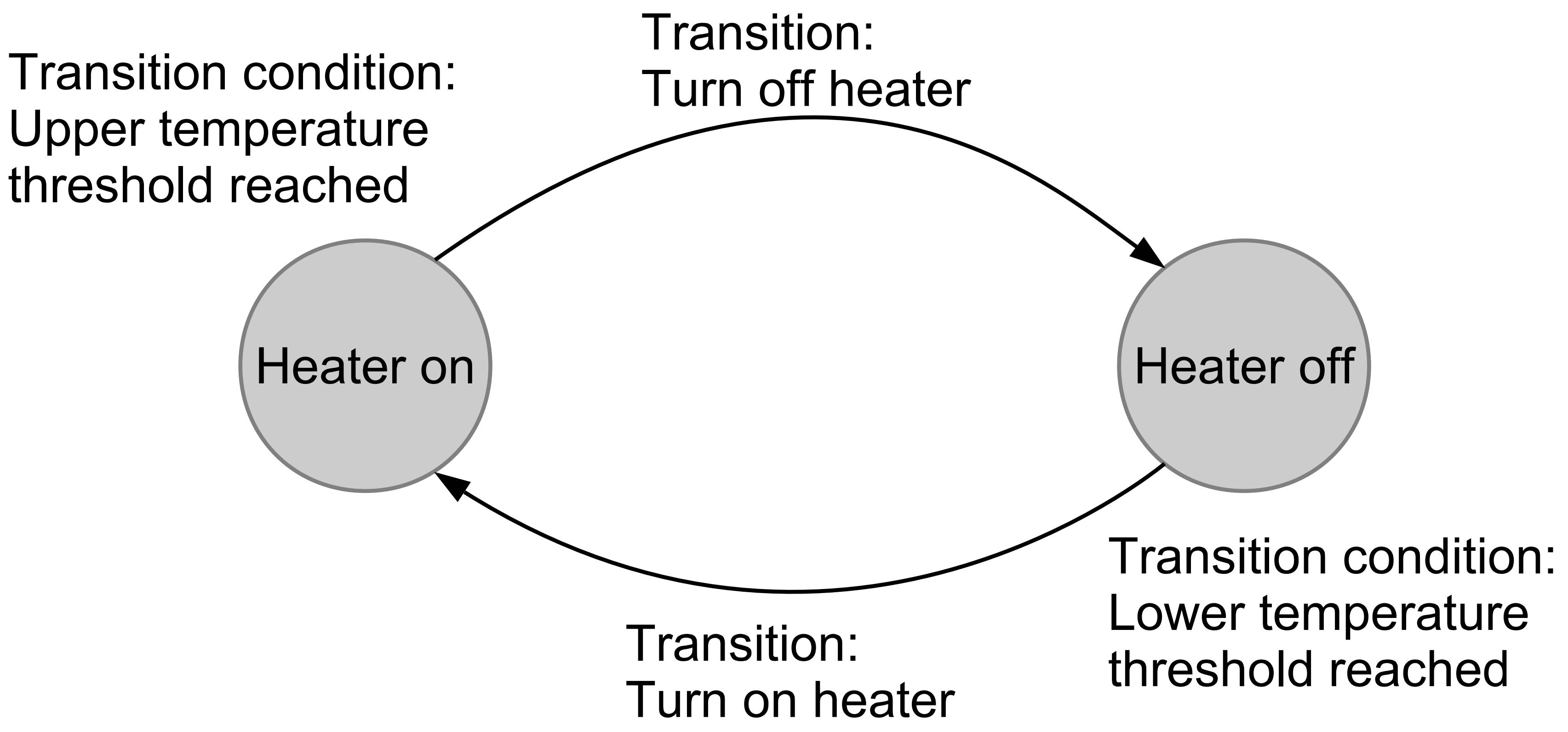
- Consider a heat pump, whose purpose is to heat up a swimming pool.
- The heat pump operates in the following way:
  - It starts heating when the water temperature falls below a lower temperature thresholds (e.g., 28 degrees)
  - It stops when the water temperature gets higher than an upper threshold temperature (e.g., 29 degrees).
- The reason for having a temperature span is to avoid having the heat pump starting and stopping all the time.

Example: Swimming pool heater

- States:
  - Heater on
  - Heater off
- Transitions
  - Turn on heater
  - Turn off heater
- Transition conditions
  - Upper temperature threshold reached (from below)
  - Lower temperature threshold reached (from above)

MALMÖ
UNIVERSITY

Example: Swimming pool heater



MALMÖ UNIVERSITY

Exercise: Swimming pool temperature regulator

- Now consider that we have a very warm summer and instead of a heat pump we need a temperature regulator that is able to both heat and cool the swimming pool.
- Your task is to create an FSM for a swimming pool temperature regulator:
  - Identify temperature thresholds, states, transitions and transition conditions
  - Draw a state diagram for your FSM.

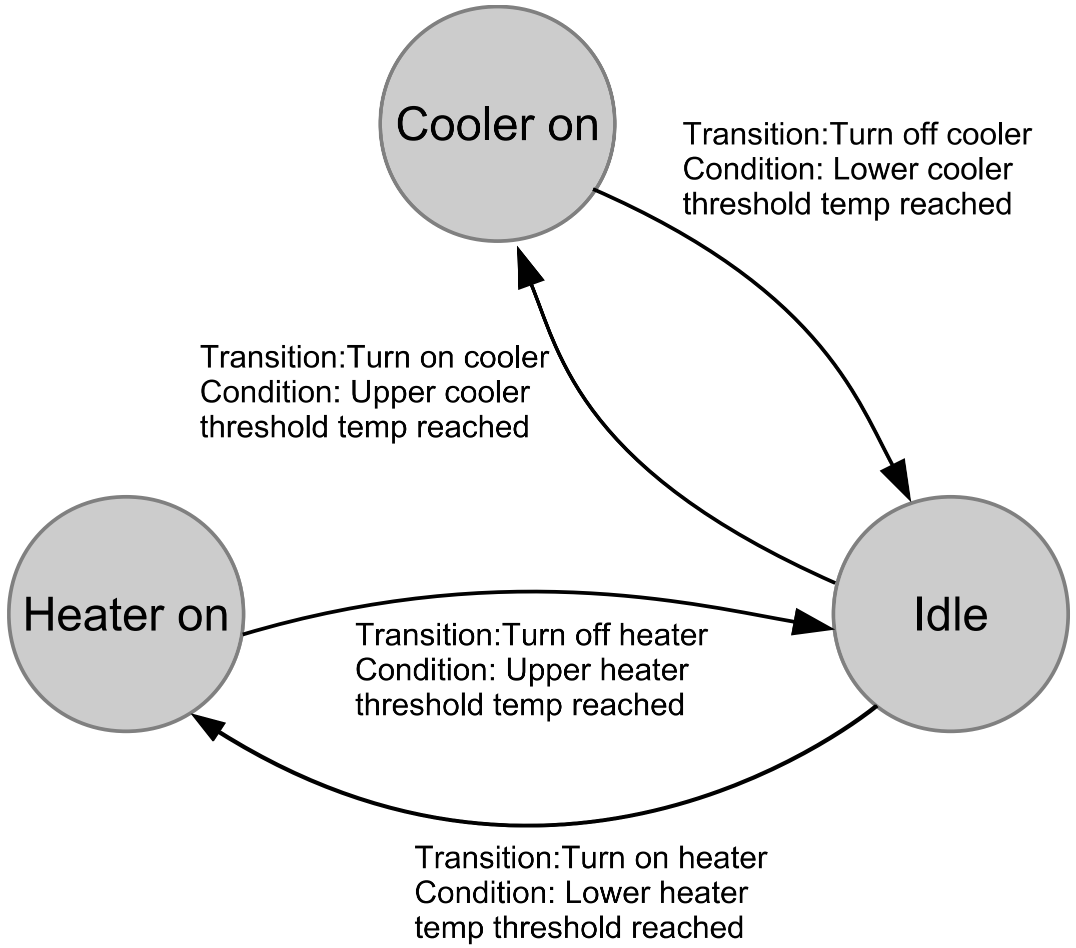
MALMÖ UNIVERSITY

Exercise: Swimming pool temperature regulator

- States:
  - Heater on
  - Cooler on
  - Idle
- Transitions
  - Turn on heater (idle -> heater on)
  - Turn off heater (heater on -> idle)
  - Turn cooler on (idle -> cooler on)
  - Turn cooler off (cooler on -> idle)
- Transition conditions
  - Lower heater temperature threshold reached (Turn on heater)
  - Upper heater temperature threshold reached (Turn off heater)
  - Upper cooler temperature threshold reached (Turn cooler on)
  - Lower cooler temperature threshold reached (Turn cooler off)

MALMÖ UNIVERSITY

Exercise: Swimming pool temperature regulator



MALMÖ UNIVERSITY

Moore and Mealy machines

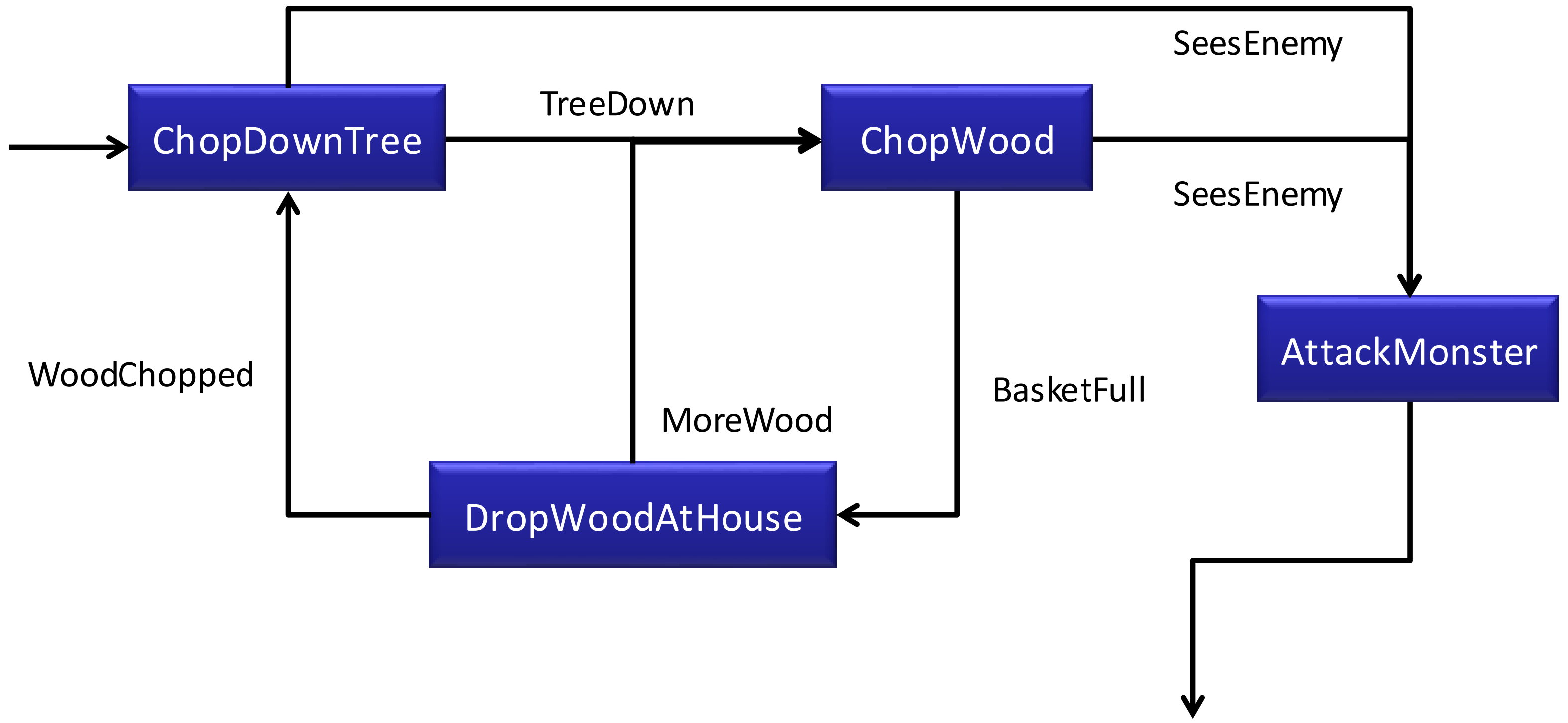
- An FSM can be a:
  - Moore machine: code is executed within states
  - Mealy machine: code is executed at transitions between states.
  - Mealy-Moore machine: combination of Moore and Mealy machines.

Checking for transitions

- A challenge when implementing an FSM is to decide how to check for state transitions.
- A possible approach is to use polling, where the machine continuously check if a condition for transition has been met.
- One disadvantage of polling is that the transition check in some cases can be computationally expensive, and it may have to be run often.
  - Not desirable in real time systems
- An alternative to polling is to use an **event-driven system** where the machine subscribes to events, and get notifications as soon as an event occur.

A problem

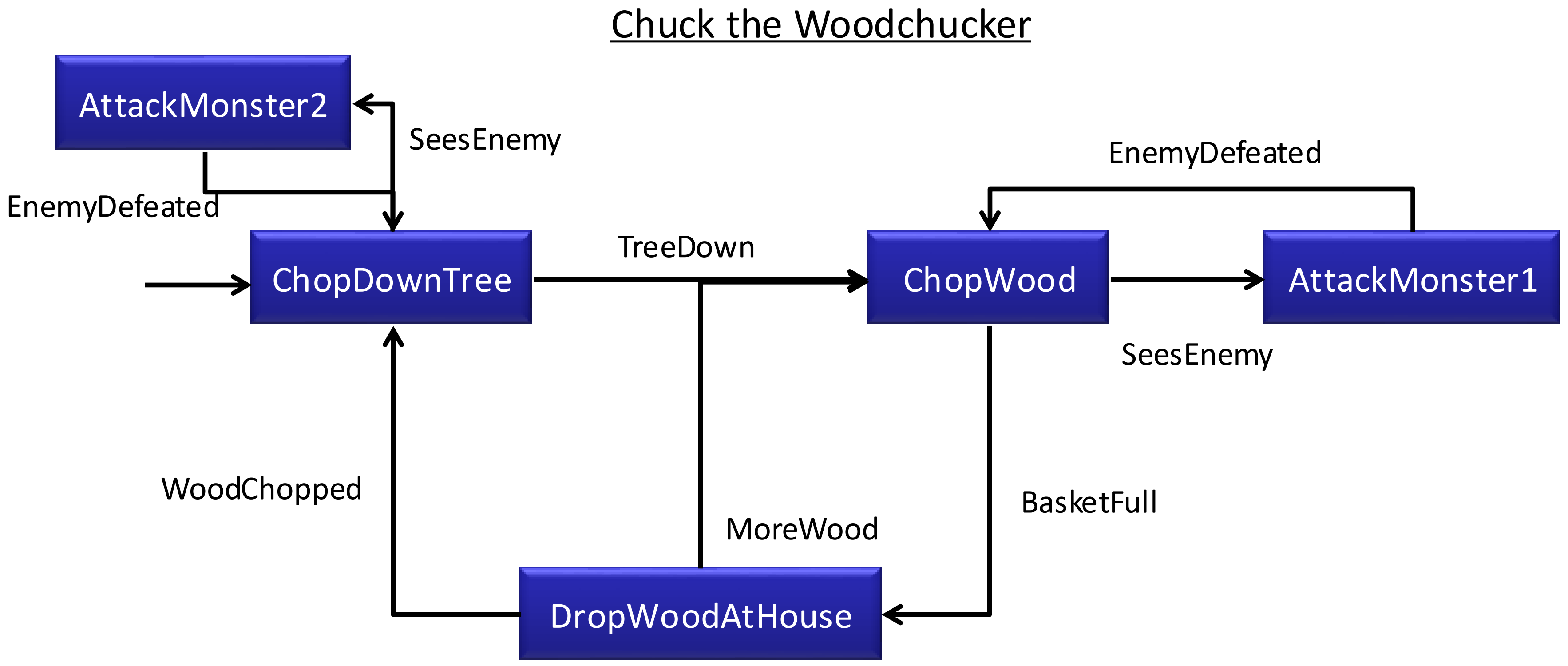
Chuck the Woodchucker



What would Chuck do now?

Source: Johan Hagelbäck

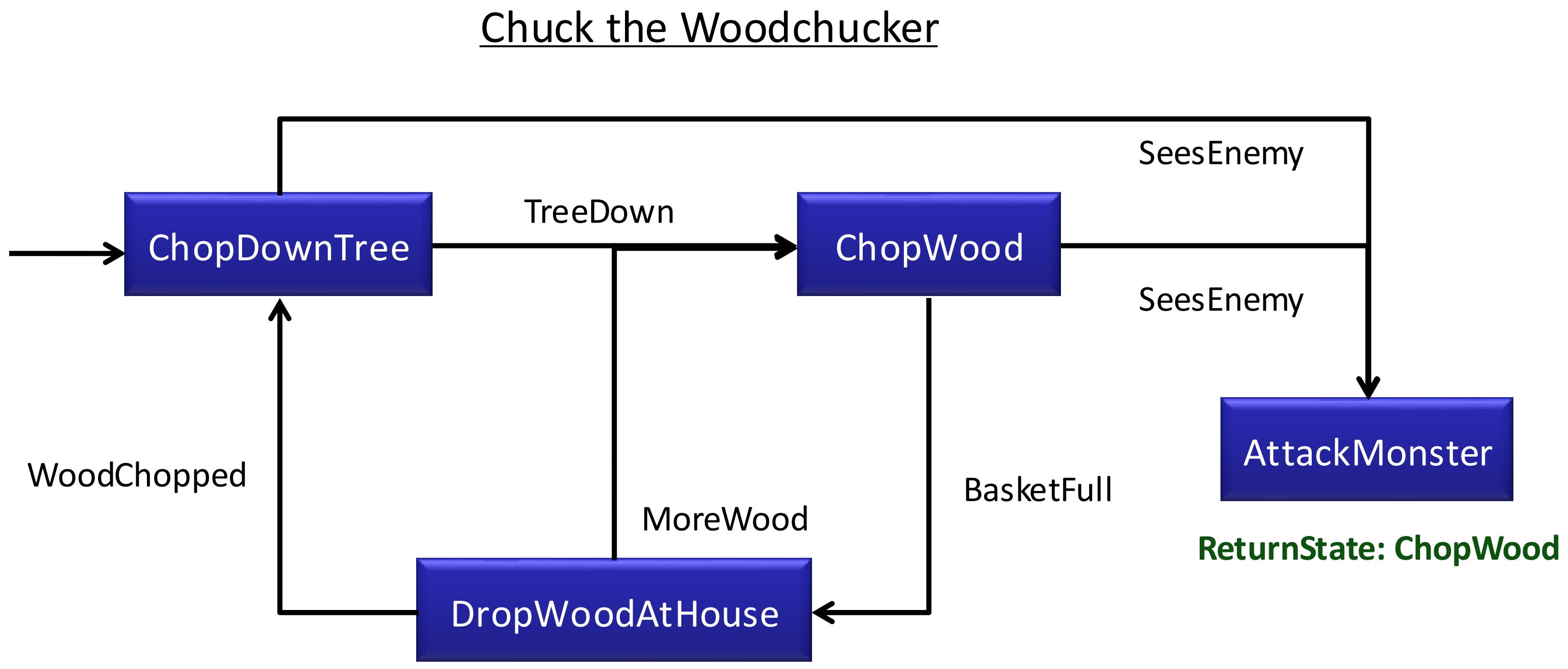
Solution 1: Multiple Attack states



Not very good since we need to duplicate states and code…

Source: Johan Hagelbäck

Solution 2: Stack-based FSM

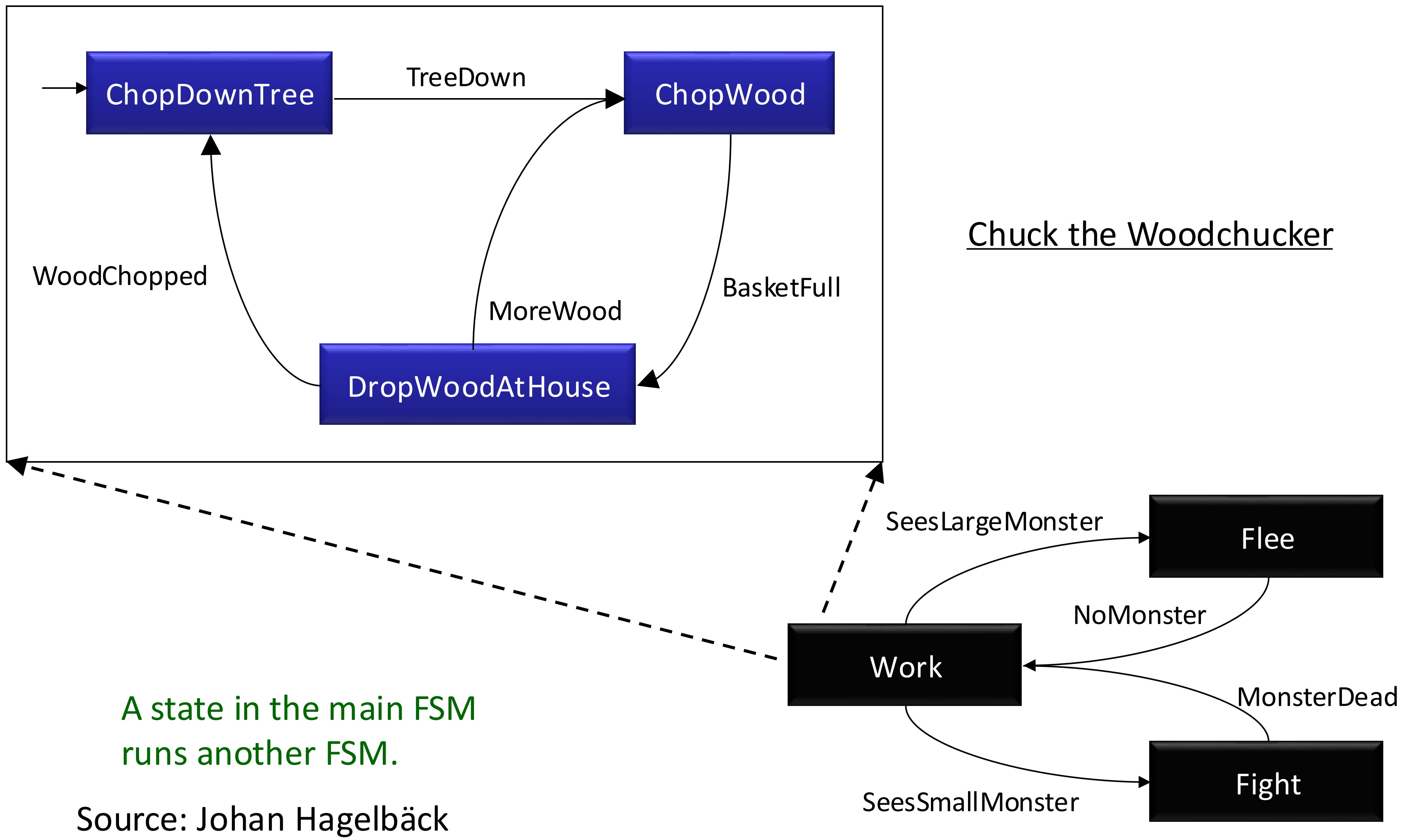


The FSM keeps a history of previous states to return to.

Source: Johan Hagelbäck

MALMÖ UNIVERSITY

Solution 3: Hierarchical FSM



Chuck the Woodchucker

A state in the main FSM runs another FSM.

Source: Johan Hagelbäck

Which solution is the best?

- If it works well with a stack-based FSM, use it.
- If the behavior is too complex for a stack-based FSM, consider using a hierarchical FSM.
  - Hierarchical FSMs are very useful for complex behaviors.
  - Please note that a hierarchical FSM also makes use of a stack.
- Sometimes even hierarchical FSMs are not scalable and flexible enough to model the behavior we want to implement.
- There are other techniques that can be used to model complex behaviours, e.g. *Behavior Trees*.
  - However, that is not included in this course.

Source: Johan Hagelbäck

Implementing an FSM

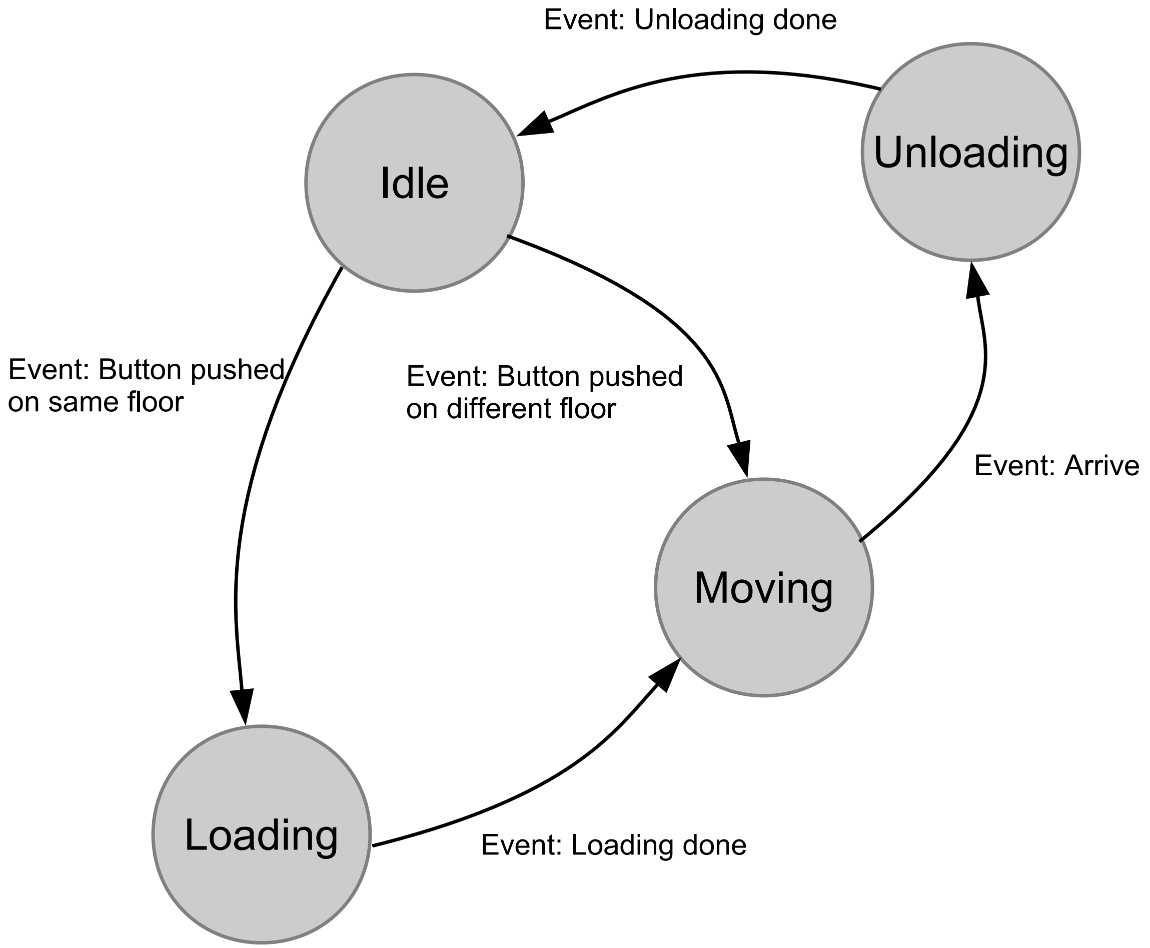
- An FSM can be implemented using different techniques (which can be combined):
  - State transition table
  - Switch case
  - Function pointers
- In the coming slides we will discuss how to implement an FSM for a two-floor elevator using function pointers (a very nice way to implement FSMs).

Two-floor elevator

- Consider an elevator traveling between two floors (for example, the elevator at Hyllie train station).
- Transitions are driven by button pushes and events created by the elevator.
- We will implement an FSM for our elevator using function pointers.
- Home exercise: Implement the same FSM using a switch case and learn what differences there are.

MALMÖ UNIVERSITY

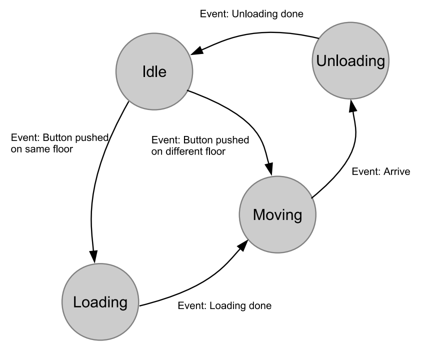
FSM: Two-floor elevator



Two-floor elevator

- States:
  - Moving
  - Idle
  - Loading
  - Unloading
- Events (drives transitions)
  - Push button on lower level
  - Push external button on upper level
  - Done loading
  - Done unloading
  - Arrive to floor

- Possible transitions
  - Idle -> Moving
  - Idle -> Loading
  - Loading -> Moving
  - Moving -> Unloading
  - Unloading -> Idle
- We also need to keep track of:
  - Current state
  - Current position of elevator



Using function pointers

- When implementing an FSM using a function pointer, each state is implemented in a separate function.
- The function pointer points to the current state function.
  - Hence keeping track of the current state of the machine.
- Each time an event (or a poll for transition) occurs, the the code for the current state is executed via the function pointer.
- If right conditions are met:
  - Code is executed in the state function.
  - Transition to new state is made by letting the function pointer point to the new state.

MALMÖ UNIVERSITY

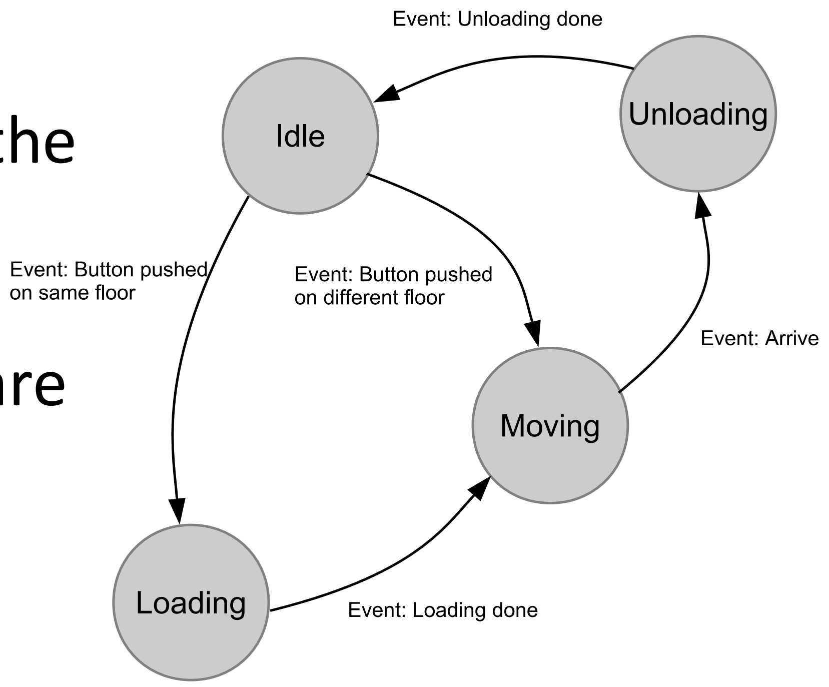
Code (external)

- I have implemented our two-floor elevator FSM in C.
- I have also implemented a terminal based simulator interface where the user can simulate the operations of our elevator.
- I will now go through the code and run the simulator.

MALMÖ UNIVERSITY

Exercise:

- For the elevator FSM presented earlier, add the possibility to push buttons (level and open door) inside the elevator.
  - How will the behavior of the elevator change as a consequence?
  - What new states/events are needed?
  - Will there be any new transitions needed?
  - Will the current transitions change?



Thank you for your attention!

Questions?

MALMÖ UNIVERSITY