04-630
Data Structures and Algorithms for Engineers

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 2

Formalisms for representing algorithms

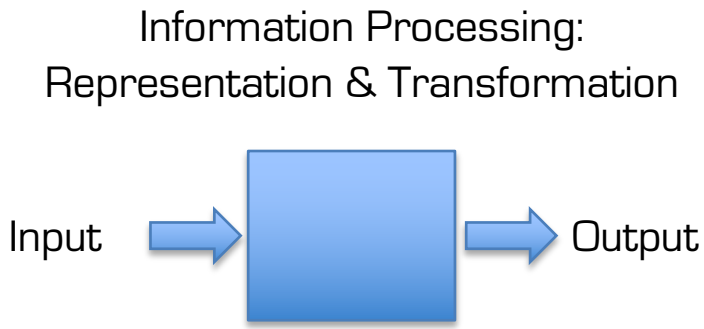
- Definition of an algorithm
- Modelling software
- Relational modelling
- State modelling
- Practical Representations
  - Pseudo code
  - Flow charts
  - Finite state machines
  - UML

(This lecture is adapted from 17-630 Computer Science for Practicing Engineers)

Definition of an Algorithm

Informal definition

An algorithm is a systematic procedure for transforming information from one (input) state to another (output) state

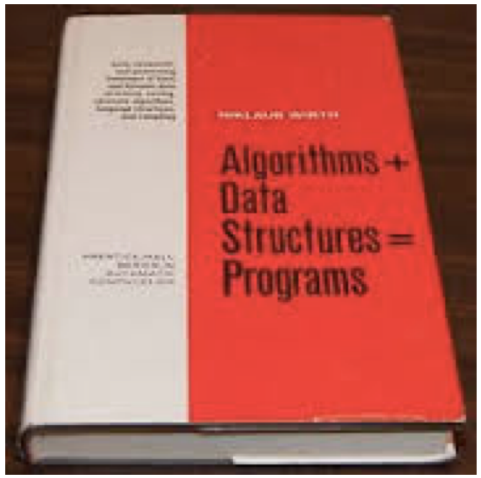


We will present a formal definition in the section on computability theory

Definition of an Algorithm

Information much be represented in some way

Typically there is a strong link between an algorithm and the information representation, i.e. the data structure



Modelling Software

Goal: introduce basis ways to represent algorithms to support practical analysis

- Complexity
- Correctness

Modelling Software

Practical analysis techniques

- Formal rigorous analysis can involve complex mathematics:
  - quickly reaches a point of diminishing returns
  - difficult to communicate
  - takes too long and costs too much to be of practical value

- It is often more useful and practical to
  - Use simplified mathematics to analyze trends, characteristics, and general properties
  - focus on the essence of the algorithm (memory/execution)

Modelling Software

- Software is an intangible product
  - We can't see, touch, smell, or otherwise directly measure software
  - We can only conceptualize its structure and only see evidence of its execution

- We create models that facilitate both communication and analysis. These models can be:
  - Graphical
  - Textual
  - Execution (prototype/experiments)
  - Mathematical
  - Combinations of these

Modelling Software

- To support analysis, we need ways to abstract and represent algorithms

- There are two extremes
  1. Formal mathematics

     Often too abstract and too removed from implementations and applications

  2. Computer language

     Often too many unnecessary details that complicate analysis

- Our initial task: how can we effectively represent, communicate, and analyze algorithms?

Relational Modelling

- Assume we have a requirement

  "We need to be able to take text files, search for certain phrases that may or may not be present, and produce a formatted document with the phrases removed and placed in another file in a specified format for printing and/or further analysis."

- Such requirements can be thought of in terms of a relational model

  Relational models describe a requirement in terms of preconditions, rules, and post conditions

Relational Modelling

Relational Models are typically functional

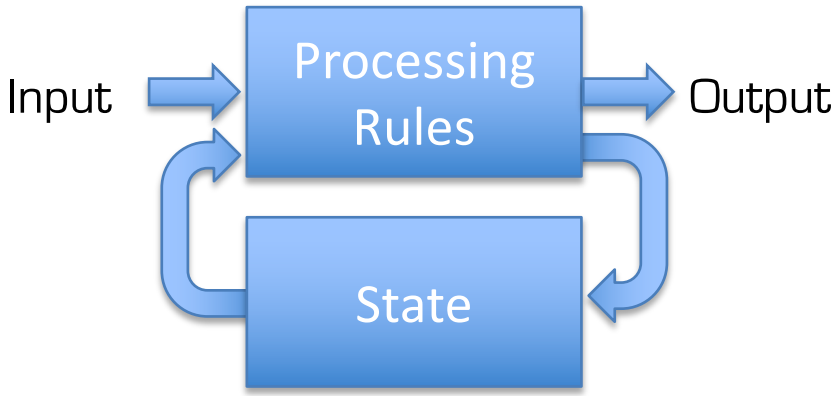
- Rigorously specify
  - Input (domain)
  - Processing rules
  - Output (range)

- Processing rules are algorithms
- State is not retained in relational models

State Modelling

- In most systems, state is retained and transformed throughout execution

- In these cases simple functional models are inadequate to model the system



State Modelling

- In state models, we often need to model precisely
  - Preconditions (inputs)
  - Post conditions (outputs)
  - Processing rules (algorithms)
  - States
  - State transitions

Practical Representations

Required characteristics

- Simple, clear, and intuitive (as far as possible)
- As rigorous as practical – but keeping the math as simple as possible
- Language neutral
- Factor out the hardware and operating systems
- Focus on algorithmic essence
- Properly scoped (not too big, not too trivial or obvious)

Practical Representations

- Some candidate representations
  - Pseudo Code
  - Flow Charts
  - State Diagrams
  - Formalisms
  - Modeling Methodologies (e.g. UML)

- Many engineers use these, but some use them
  - At the wrong time
  - To model the wrong kinds of things (poor scoping)
  - Incorrectly
  - Mix “what is needed” with “how we will build it”

Pseudo Code

- Pseudo code is an informal abstraction of an algorithm that:
  - uses the structural conventions of a programming language
  - is simplified for human reading rather than machine compilation
  - omits details that are not essential for algorithmic analysis
  - shows the temporal relation of instruction execution (sequencing)
- Despite many attempts, no standard for pseudo code syntax currently exists

Pseudo Code

- Declaration

  `type variable;`

  `integer A; string name;`

- Assignment

  `variable = value;`

  `a = 45; x = y`

- Basic mathematical operators

  `result = variable_value operator variable_value`

  `y = a+b; z = 5.0/e; j = k*l; r = 2*(22/7)*(r^2)`

- Basic functions and subroutines

  `read(), write(), print(),…`

  Assumed functions should be clearly defined prior to use

  More on functions and subroutines later

Pseudo Code

Control Structures

- Direct sequence

  `do X, then do Y`

- Conditional branching

  if Q then do X, else do Y

- Bounded iteration

  do Z exactly X times

- Conditional or unbounded iteration

  do Z until Q becomes true

  while Q is true do Z

Pseudo Code

Example: Algorithm to find the greatest common denominator (GCD)

- How the **`read()`** function work is not important for our analysis
- We focus on the essence of the algorithm, not on checking input, formatting output, error handling, and so forth
- Now that the algorithm has been distilled to its essence we can analyze: *how do we know we solved the problem? how quickly does it compute the answer?*

```
a = read()
b = read()
if a = 0
    return b
while b ≠ 0
    if a > b
        a := a - b
    else
        b := b - a
return a
```

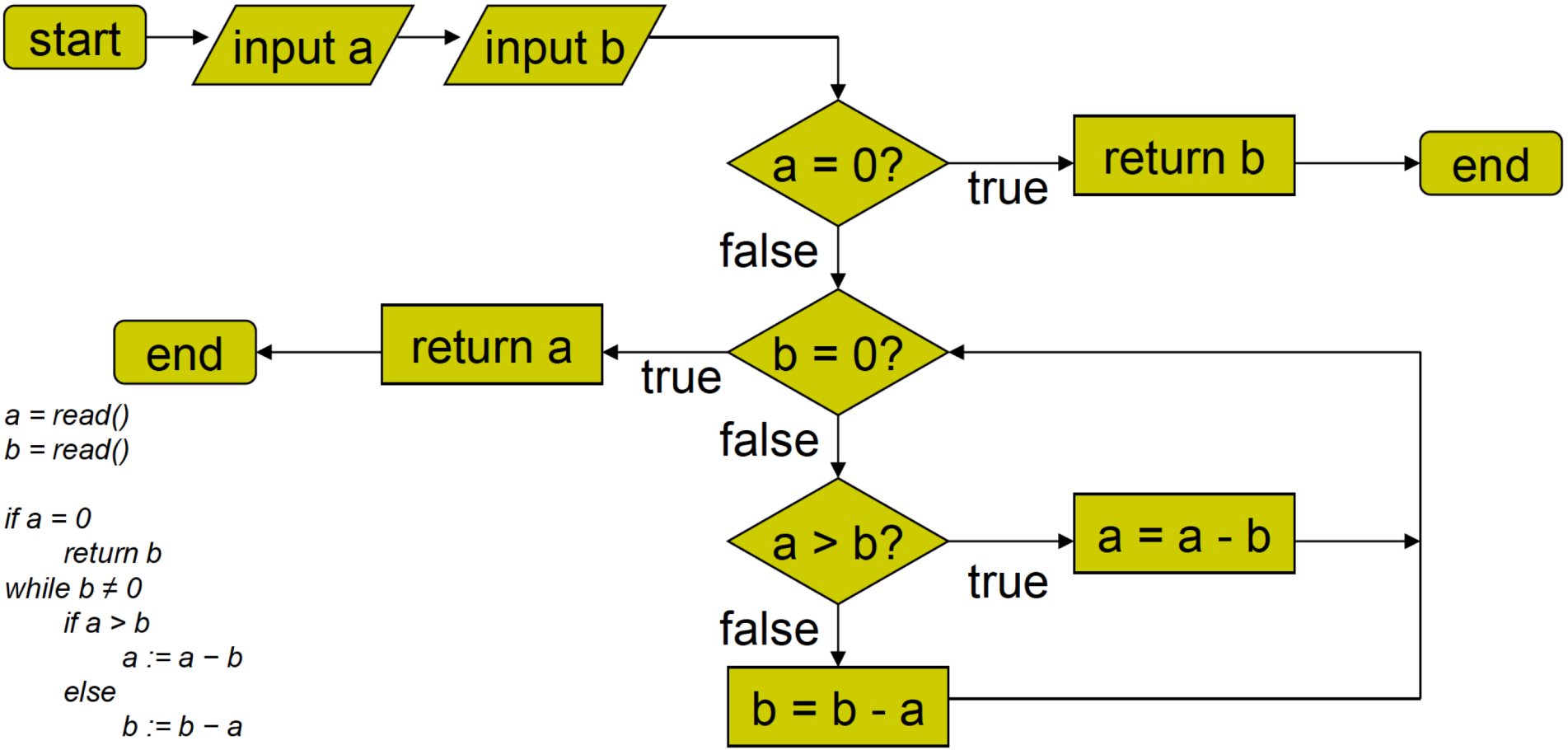
Pseudo Code

- Pseudo code is attractive because
  - It looks like the computer-interpretable code
  - It is complete in terms of describing computer algorithms
- In practice, pseudo code is sometimes extended and violates notions of minimalism
  - Pseudo code should only support what is necessary to describe the algorithm – and no more!
  - Sometimes, pseudo code is used to describe entire applications, and becomes too cumbersome to support analysis of algorithms

Flowcharts

- Graphical representation of the behavior of an algorithm
  - Represents the steps of an algorithm by geometric shapes
  - Temporal relationships are shown by connections
- Developed in the early 20th century for use in industrial engineering
  - Used in many domains for the last 100 years
  - John von Neumann developed the flow chart while working at IBM as a means to describe how programs operated
  - Flowcharts are still used to describe computer algorithms- UML activity diagrams are an extension of the flowchart
- There are many flowchart notation standards

Flowcharts




```
a = read()
b = read()

if a = 0
      return b
while b ≠ 0
      if a > b
            a := a − b
      else
            b := b − a
return a
```

Flowcharts

Strengths and weaknesses

- The set of defined constructs is both minimal and complete
- The resulting algorithms can be hard to understand and analyze
- Graphical methods do not scale well – very difficult to represent large and/or complex algorithms
- Hard to distribute, share, and reuse

Finite State Machines (FSM)

- Behavioral models composed of a finite number of states, transitions between those states, and actions
- FSMs are represented by state diagrams
- State diagrams have been used for 50+ years in software, hardware, and system design and there are a variety of notations and approaches
  - Traditional Mealy-Moore state machines
  - Harel state machines
  - UML state machines

Finite State Machines (FSM)

A traditional (e g. Mealy-Moore) type of FSM is a quintuple $(\Sigma, S, s_0, \delta, F)$

Σ is the input alphabet where Σ is finite $\wedge$ $\Sigma \neq \emptyset$

S is a set of states where S is finite $\wedge$ $S \neq \emptyset$

s_0 is an initial state where, $s_0 \in S$

$\delta(q, x)$ is the state transition function where $q \in S \wedge x \in \Sigma$

(If the FSM is nondeterministic, then δ could be a set of states)

F is the set of final states where $F \subseteq of\ S \cup \{\emptyset\}$

Finite State Machines (FSM)

$\delta(q, x)$ may be a partial function:

$\delta(q, x)$ does not have to be defined for every combination of q and x

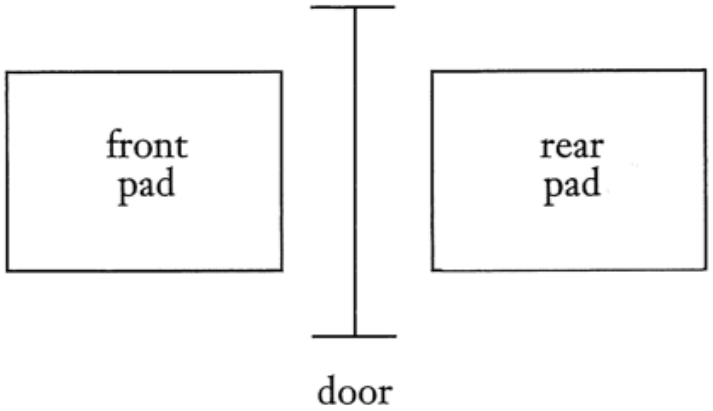
If it is not defined then the FSM can enter an error state or reject the input

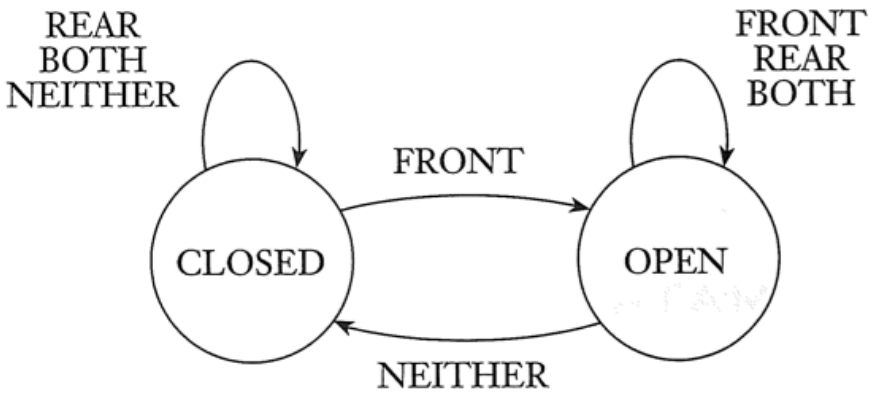
Finite State Machines (FSM)

The following (limiting) assumptions are made regarding traditional (deterministic) Mealy-Moore FSAs

- an FSA can only be in one state at a time and must be in exactly one state at all times
- States of one FSA are independent from the states of all other FSAs
- Transitions between states are not interruptible
- Actions are atomic and run to completion
- Actions may be executed on entry into a state, on exit from a state, or during the transition from one state to another

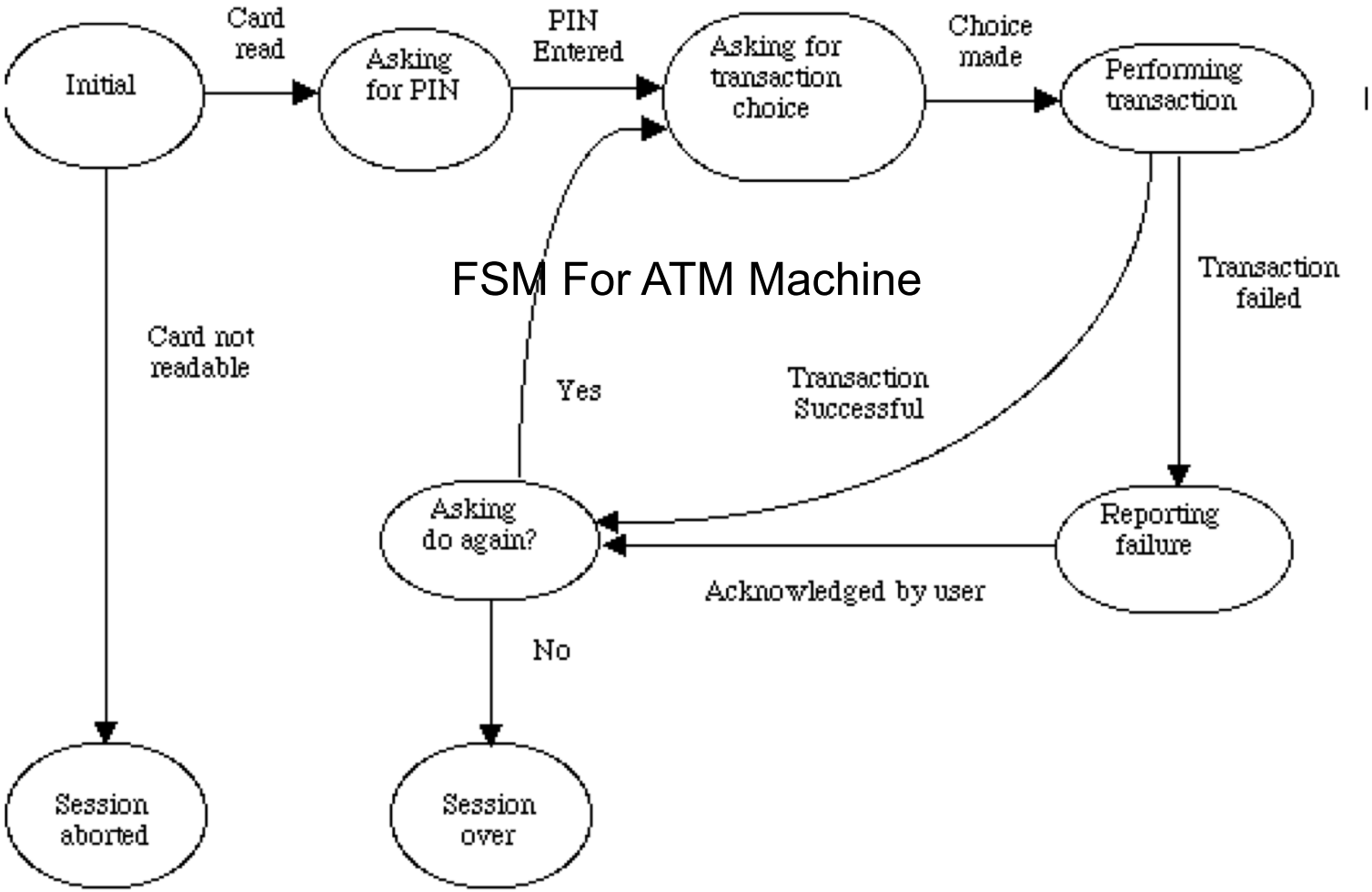
Finite State Machines (FSM)





Finite State Machines (FSM)

FSM For ATM Machine



FSM For ATM Machine

Finite State Machines (FSM)

Transitions indicate state change from one state to another state that are described by

- a condition that needs to be fulfilled to enable a transition
- an action which is an activity that is to be performed at some point in the transition
  - Entry action: which is performed when entering the state
  - Exit action: which is performed when exiting the state
  - Input action: which is performed depending on present state and input conditions
  - Transition action: which is performed when performing a certain transition

Finite State Machines (FSM)

- Popular form of FSM are the Harel State Diagrams
- A variant which was adopted for
  Unified Modeling Language (UML) State Machines
- There are two types of UML State Machines
  - Behavioral State Machines (BSM)
    Model the behaviour of objects
  - Protocol State Machines (PSM)
    Model protocols of interfaces and ports
- Most use users of UML don't differentiate

Finite State Machines (FSM)

FSAs are limited and its difficult to model concurrency, complex object states, threads, multi-tasking

UML state machines extend the traditional automata theory in several ways that include

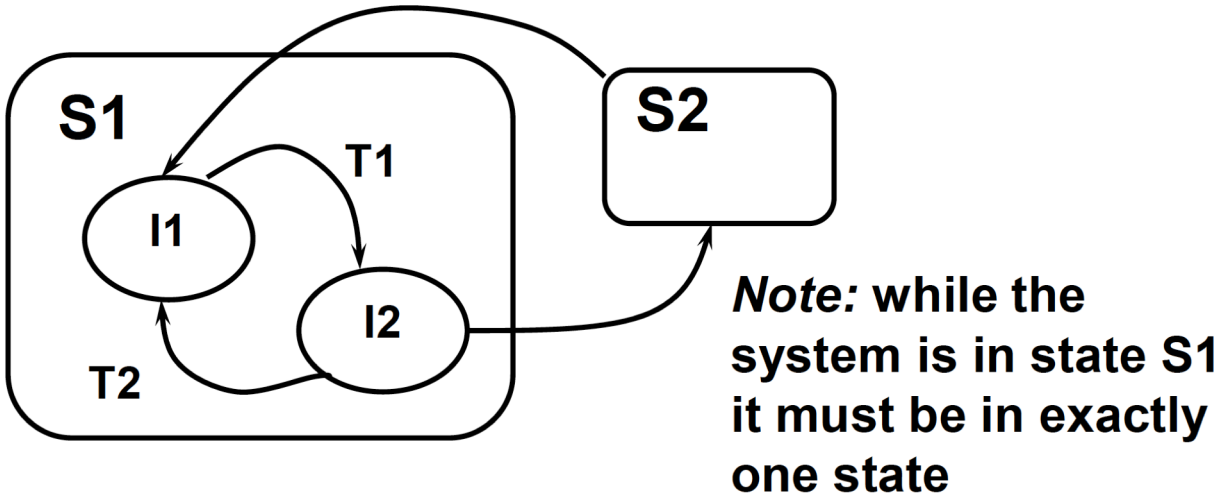
- nested state
- guards
- actions
- activities
- orthogonal components
- concurrent state models

Finite State Machines (FSM)

UML State Machines – Nested States

Outer state is call the superstate

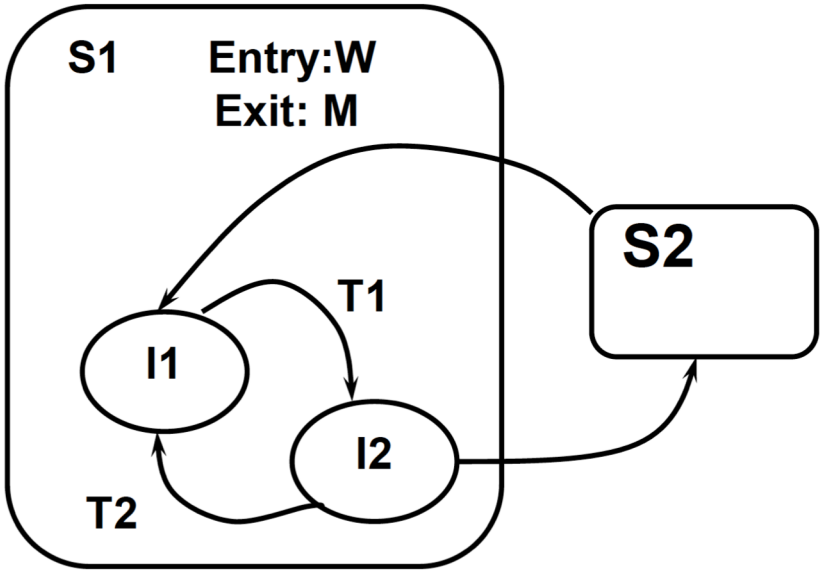
Inner states are called substates



Finite State Machines (FSM)

UML State Machines – Actions

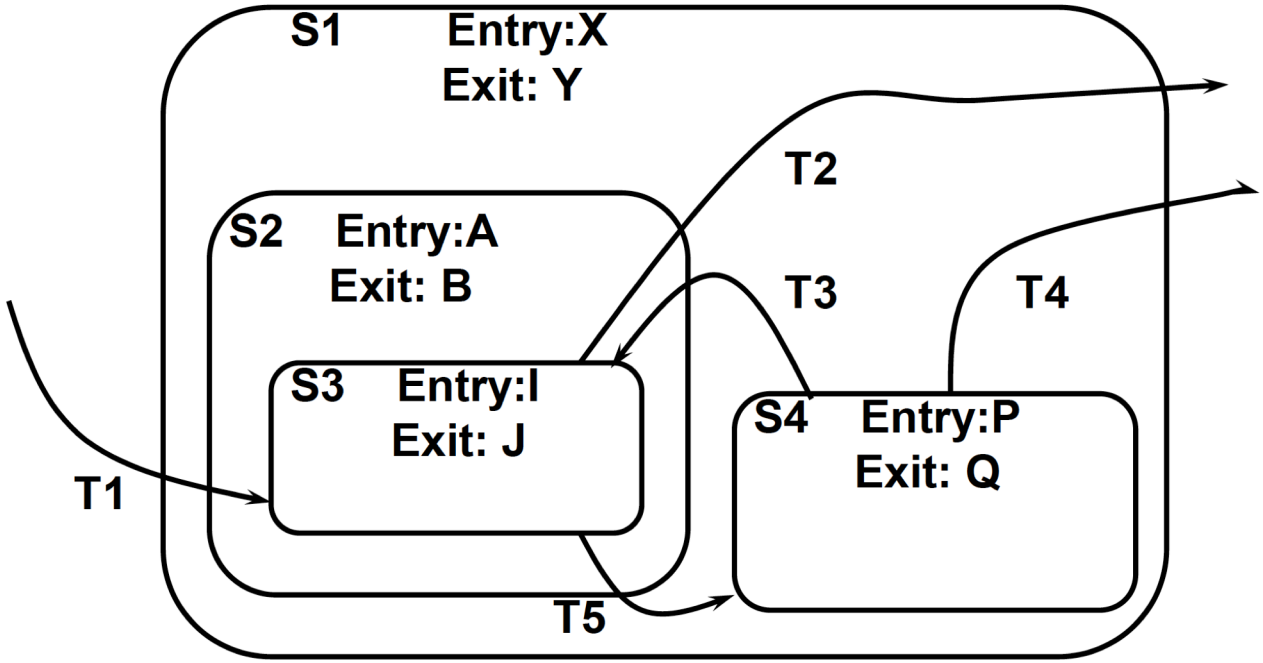
You can specify state entry and exit actions



Finite State Machines (FSM)

UML State Machines – Actions

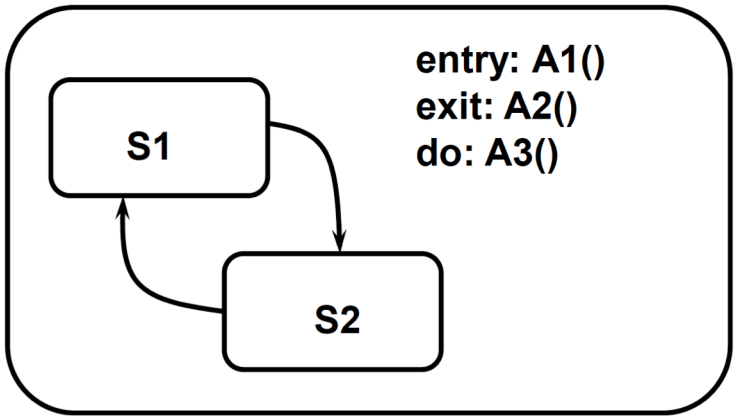
You can nest entry and exit actions



Finite State Machines (FSM)

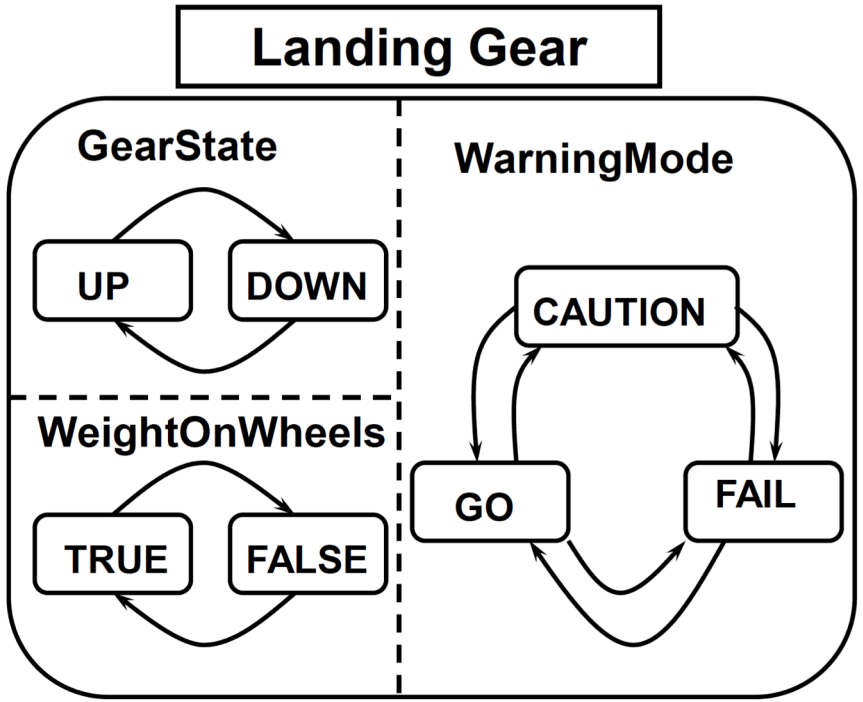
UML State Machines – Activities

- Like actions except they are performed as long as the state is active
- Activities are indicated with a do: statement



Finite State Machines (FSM)

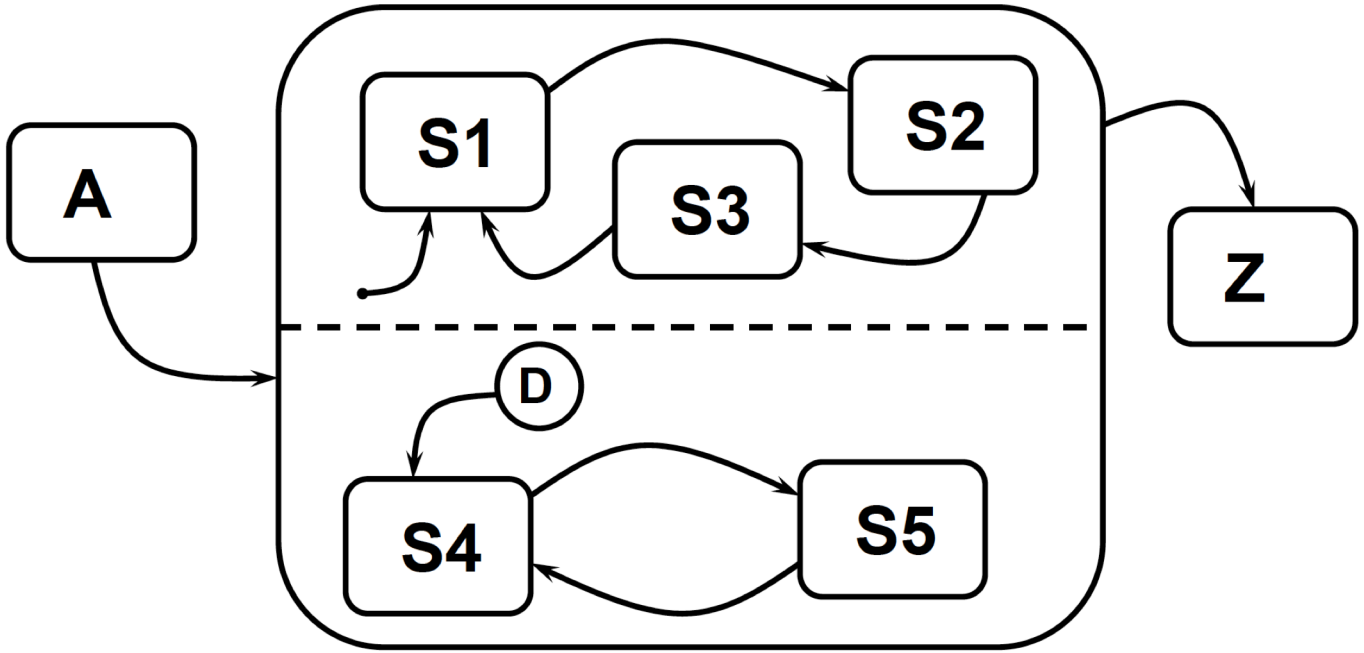
UML State Machines – Orthogonal Components



Finite State Machines (FSM)

UML State Machines – Concurrent State Models

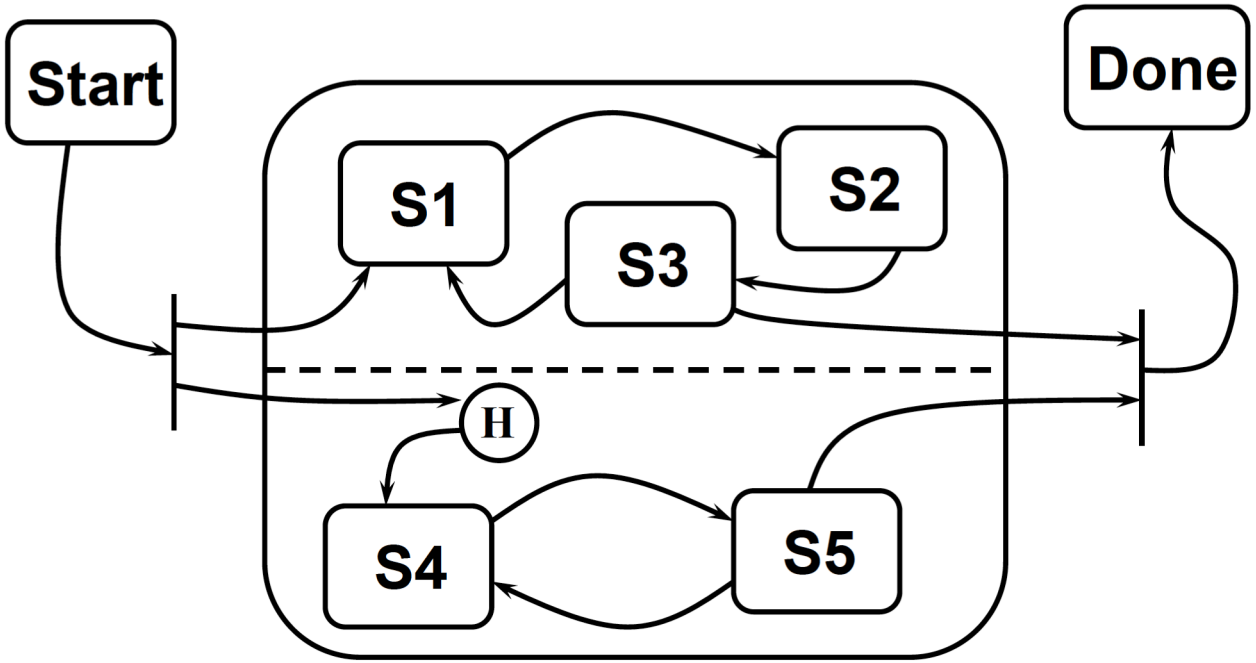
Concurrent threading can be modelled



Finite State Machines (FSM)

UML State Machines – Concurrent State Models

Forking / Joining can be modelled



Finite State Machines (FSM)

- In traditional FSA, transitions carry little or no information
- UML state machine transitions carry a lot of information:
  - Event Name - Name of triggering event
  - Parameters - data passed with event
  - Guard - condition that must be true for the transition to occur
  - Action List - list of actions executed
  - Event List - list of events executed

Finite State Machines (FSM)

- The key problem with FSM technologies is that they simply do not scale up well
  - state explosion is a common problem
  - care must be taken to restrict the scope of what is being modeled

- FSMs often abstract away the very algorithms we want to model
  - Care must be taken to maintain a proper and consistent level of abstraction
  - Can violate notions of completeness

Finite State Machines (FSM)

UML state machines are really more like a notation than traditional FSMs

- Violates minimalism
- Any benefit gain in applying mathematical rigor may be lost