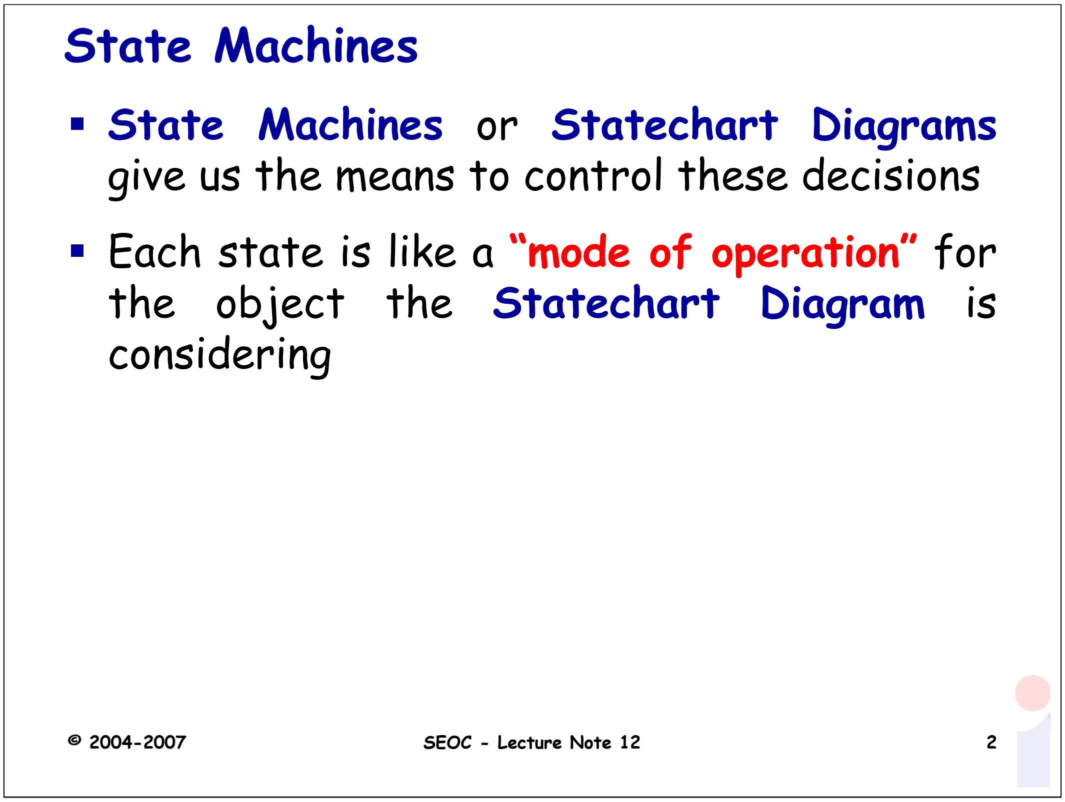
State Machines

Massimo Felici

JCMB-1402 0131 650 5899

1BP-G04 0131 650 4408

mfelici@inf.ed.ac.uk



Sequence and Communication Diagrams show how objects interact to meet some system requirements. They lack information on how the system decides what is the right thing to do. They correspond to scenarios and are decision free. There may bee many sequence or collaboration diagrams for one Use Case. The choice of how to react (that is, which scenario is appropriate) depends on a state.

State Machines or Statechart Diagrams are based on the statechart notation called HiGraph. Statechart Diagrams are finite state machines with some extra mechanism to capture the meaning of transitions.

Suggested Readings

- D. Harel. *Statecharts: A Visual Formalism for Complex Systems*. In Science of Computer Programming 8(1987):231-274.

Activity vs. State Machines

- In UML semantics **Activity Diagrams** are reducible to **State Machines** with some additional notations
- In **Activity Diagrams** the vertices represent the carrying out of an activity and the edges represent the transition on the completion of one collection of activities to the commencement of a new collection of activities
- **Activity Diagrams** capture high level activities' aspects
- In **State Machines** the vertices represent **states** of an **object** in a class and edges represent occurrences of **events**

The additional notations capture how activities are coordinated. In particular, it is possible to represent concurrency and coordination in Activity Diagrams.

Objects have behaviours and states. The state of an object depends on its current activity or condition. A Statechart Diagrams shows the possible states of the object and the transitions that cause a change in state.

State Machine Basics

- Simple,
- Complex States
  - Composite and Submachine States
  - Concurrent Substates
  - History States
  - Synch States
- Transitions
- Synchronization Bars and Decision Points
- Transition types
- Transitions to/from Composite States
- Actions

Events

- Internal or External Events trigger some activity that changes the state of the system and of some of its parts
- Events pass information, which is elaborated by Objects operations. Objects realize Events
- Design involves examining events in a State Machine and considering how those events will be supported by system objects

States

- A state is a condition of being at a certain time
- Objects (or Systems) can be viewed as moving **from state to state**
- A point in the lifecycle of a model element that satisfies some condition, where some particular action is being performed or where some event is waited
- **Start** and **End** States

Actions

- **States** can trigger **actions**
- **States** can have a second compartment that contains **actions** or activities performed while an entity is in a given state
- An **action** is an atomic execution and therefore completes **without interruption**
- Five triggers for actions:
  - **On Entry**, **Do**, **On Event**, **On Exit** and **Include**
- An activity captures complex behaviour that may run for a long duration
  - An activity may be interrupted by events, in which case it does not complete

Simple and Composite States

- **Simple states** - simplest of all states, they have no substates
- **Composite states** – have one or more regions for substates.
- **Submachine states** – semantically equivalent to composite states, submachine states have substates that are contained within a substate machine
- An **History State** - indicated by a circle with an H inside it - allows the re-entering of a composite state at the point which it was last left

• Composite States can be further broken down into substates (either within the state or in a separate diagram).

• A composite state is a state with one or more regions.

• A region is simply a container for substates.

• A composite state with two or more regions is called orthogonal.

• A composite state may have an additional compartment called the decomposition compartment, which is a detailed view of the composite state showing regions, substates and transitions.

• UML defines a submachine state as a way to encapsulate states and transitions so that they can be reused.

A composite state with two or more regions is called orthogonal. Unlike composite states, submachine states are intended to group states, so you can reuse them. Composite states are typically specific to the current state machine.

Concurrent Substates and Regions

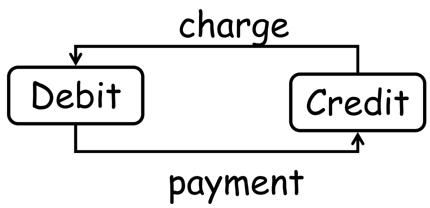
- **Concurrent Substates** are independent and can complete at different times
- Each substate is separated from the others by a dashed line

Transitions

- Viewing a system as a set of states and **transitions between states** is very useful for describing complex behaviors
- Understanding state transitions is part of system analysis and design
- A **Transition** is the movement from one state to another state
- Transitions between states occur as follows:
  1. An **element** is in a source **state**
  2. An **event** occurs
  3. An **action** is performed
  4. The **element** enters a target **state**
- **Multiple transitions** occur either when different events result in a state terminating or when there are guard conditions on the transitions
- A transition without an event and action is known as **automatic transitions**

A Transition Example

Transitions between the **Credit** and **Debit states** of an **Account class**



Transition Types

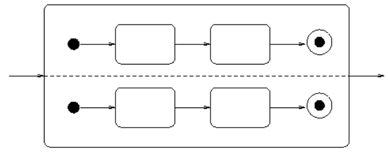
- **Compound Transition** – A representation of the change from one complete state machine configuration to another.
- **High-level Transition** – A transition from a composite state.
- **Internal Transition** – A transition between states within the same composite state. Note that transitions between regions of the same composite state are not allowed.
- **Completion Transition** – A transition from a state that has no explicit trigger.

Synchronization Bars and Decision Points

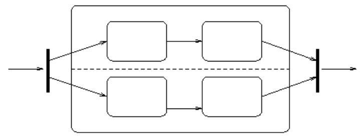
- **Synchronization Bars**
  - Allow the representation of **concurrent** states
  - Let transitions to split or combine
  - It is **important** when the overall state of a class is split into concurrent states that these states are re-combined on the same diagram
- **Decision Points**
  - Let a transition to split along a number of transitions based on a **condition**

Transitions to/from Composite States

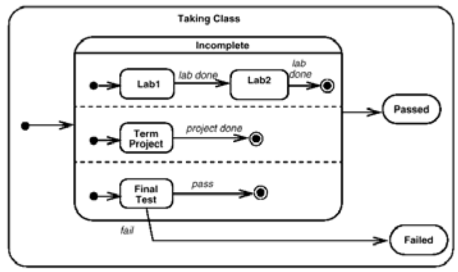
- **To composite state's boundary**
  - start the subflow at the initial state of the composite state
  - If the composite state is concurrent, then the transition is to each of the initial states
- **From composite state's boundary**
  - Immediate and effective on any of the substates
- **To the substates**
- **From substates out to other states**



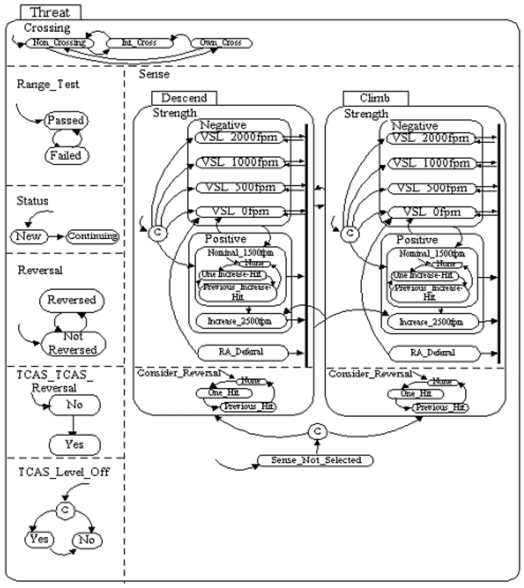
(a) State with internal concurrency



(b) Equivalent state with external synchronisation



An Example of a Very Complex State



TCAS
Traffic Alert /Collision
Avoidance System

Designing Classes with States Diagrams

- Keep the state diagram **simple**
  - State diagrams can very quickly become extremely complex and confusing
  - At all time, you should follow the aesthetic rule: ***"Less is More"***
- If the state diagram gets too complex consider **splitting** it into smaller classes
- Document states thoroughly
- Check **consistency** with the other view of the dynamics
- Think about compound state changes in a collaboration or sequence

Building Statechart Diagrams

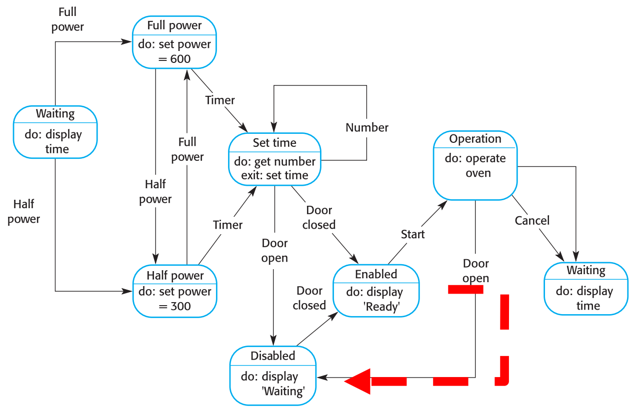
1. Identify **entities** that have **complex behaviour** - Identify a **class** participating in **behavour** whose lifecycle is to be specified
2. Model **states** - Determine the **initial** and **final states** of the entity
3. Model **transitions**
4. Model **events** - Identify the **events** that affect the entity
5. Working from the initial state, trace the impact of events and identify intermediate states
6. Identify any **entry** and **exit actions** on the states
7. Expand states using substates where necessary
8. If the entity is a class, check that the action in the state are supported by the operations and relationships of the class, and if not extend the class
9. Refine and elaborate as required

A Simple Statechart Model

A Simple Microwave Oven

1. Select the power level
2. Input the cooking time
3. Press start

- **Safety.** The oven should not operate when the door is open



Types of State Machines

- **Behavioral state machines** show the behavior of model elements such as objects. A behavioral state machine represents a specific implementation of an element.
- **Protocol state machines** show the bahavior of a protocol. They show how participants may trigger changes in a protocol's state and the corresponding changes in the system.

Some (Open) Questions

- What are the **benefits** of having states in a system?
- What are the **costs** of having states in a system?
- Every state should have an edge for every message in the class - is this the right view?
- How does this description of state relate to design by contract?
- How would you check that a Java implementation was consistent with a state diagram?
- How does this differ with the treatment of state in programming languages?
- What does this say about the different between **modeling** and **programming**?

Readings

Readings

- **UML course textbook**
  - Chapter 12 on State Machines

Suggested Readings

- D. Harel. *Statecharts: A Visual Formalism for Complex Systems*. In Science of Computer Programming 8(1987):231-274.

Summary

- Statechart Diagrams
- Activity vs. Statechart Diagrams
- Statechart Diagrams Basics
  - States and Events, Transitions, Actions, Synchronization Bars, Decision Points, Complex States (i.e., Composite States, Concurrent Substates, History States, Synch States)
- Building Statechart Diagrams