

Statechart Diagrams

Massimo Felici

Room 1402, JCMB, KB

0131 650 5899

mfelici@inf.ed.ac.uk

Statechart Diagrams

- **Sequence and Collaboration Diagrams**
 - show how objects **interact** to meet some system requirements
 - lack information on how the system decides what is the right thing to do
 - correspond to scenarios and are decision free. There may be 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.
- **Statechart Diagrams** give us the means to control these decisions.
- Each state is like a “**mode of operation**” for the object the **Statechart Diagram** is considering
- Based on the statechart notation introduced by Harel (also called HiGraph)
 - Statechart Diagrams are finite state machines with some extra mechanism to capture the meaning of transitions.

Activity vs. Statechart Diagrams

- In UML semantics **Activity Diagrams** are reducible to **Statechart Diagrams** with some additional notations
 - The additional notations capture how activities are coordinated. In particular, it is possible to represent concurrency and coordination in **Activity Diagrams**
- 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** take a "high level" of activities
- In **Statechart Diagrams** the vertices represent **states** of an **object** in a class and edges represent occurrences of **events**

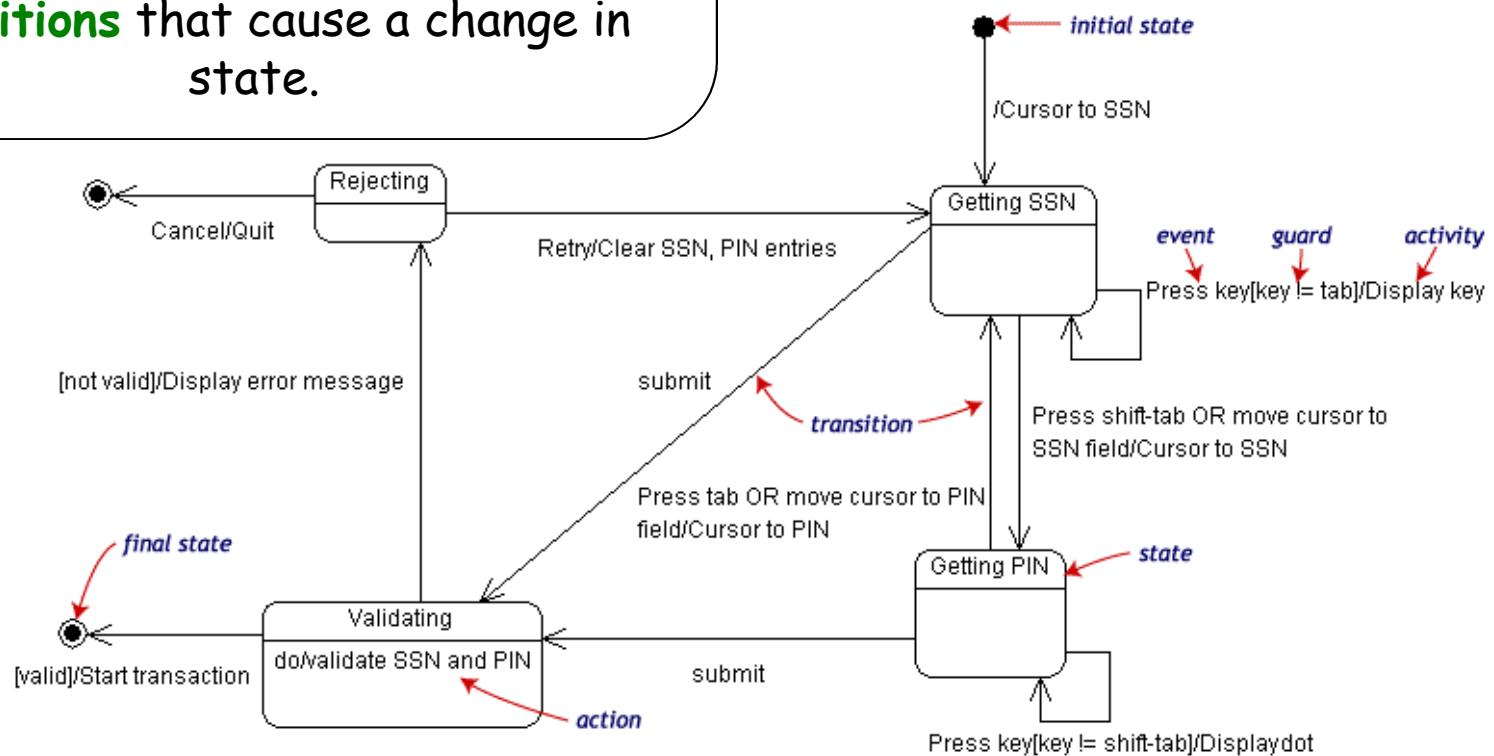


Statechart Diagrams at a Glance

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.



Statechart Diagrams Basics

- States and Events
- Transitions
- Actions
- Synchronization Bars
- Decision Points
- Complex States
 - Composite States
 - Concurrent Substates
 - History States
 - Synch States
- Transitions to and from Composite States



States and Events

▪ States

- Objects (or Systems) can be viewed as moving **from state to state**
- 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 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
- **Simple** or **Composite** States
- **Start** and **End** States

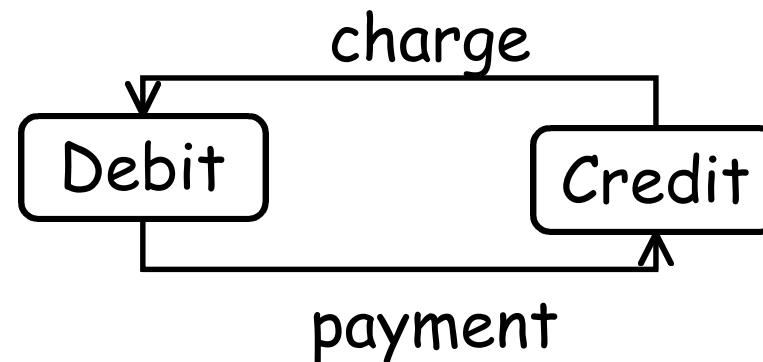
▪ Events

- 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 **Statechart Diagram** and considering how those events will be supported by system objects

Transitions

- 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**

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



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



Synchronization Bars & 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**

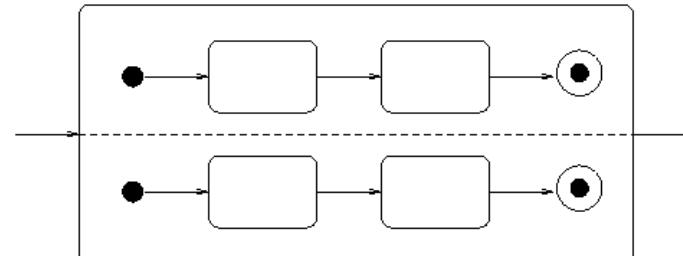


Complex States

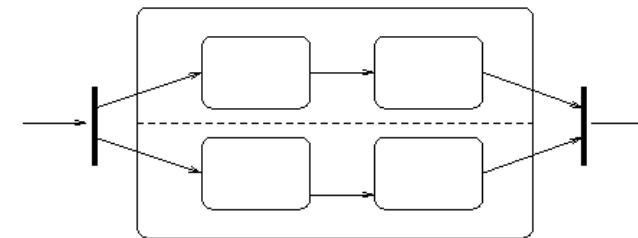
- **Composite States**: can be further broken down into **substates** (either within the state or in a separate diagram)
- **Concurrent Substates** (each substate is separated from the others by a dashed line) are independent and can complete at different times
- **History States**  allow the re-entering of a composite state at the point which it was last left
- **Synch States**  is used in a concurrent state to indicate that the associated concurrent transitions must meet up (or sync) before entering the next step

Transitions to and from Composite States

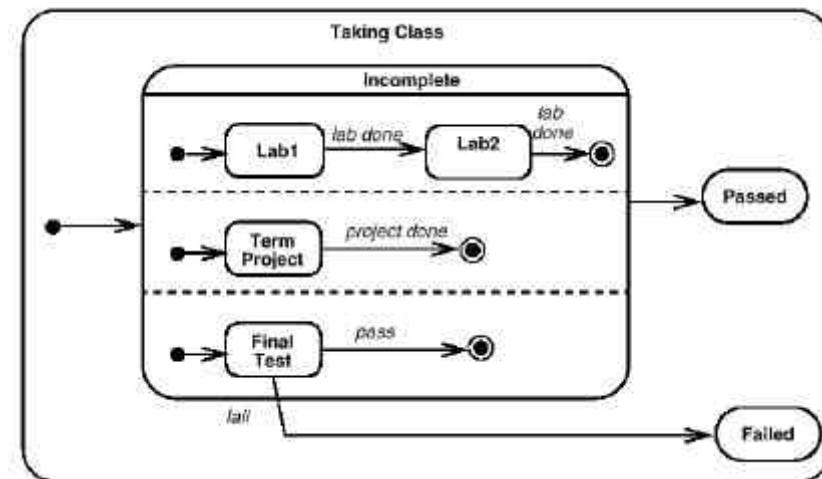
- To composite state's boundary
 - start the subflow at the initial state of the composite state
 - If the composite state is concurrent, then the transition is to each of the initial states
- From composite state's boundary
 - Immediate and effective on any of the substates
- To the substates
- From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



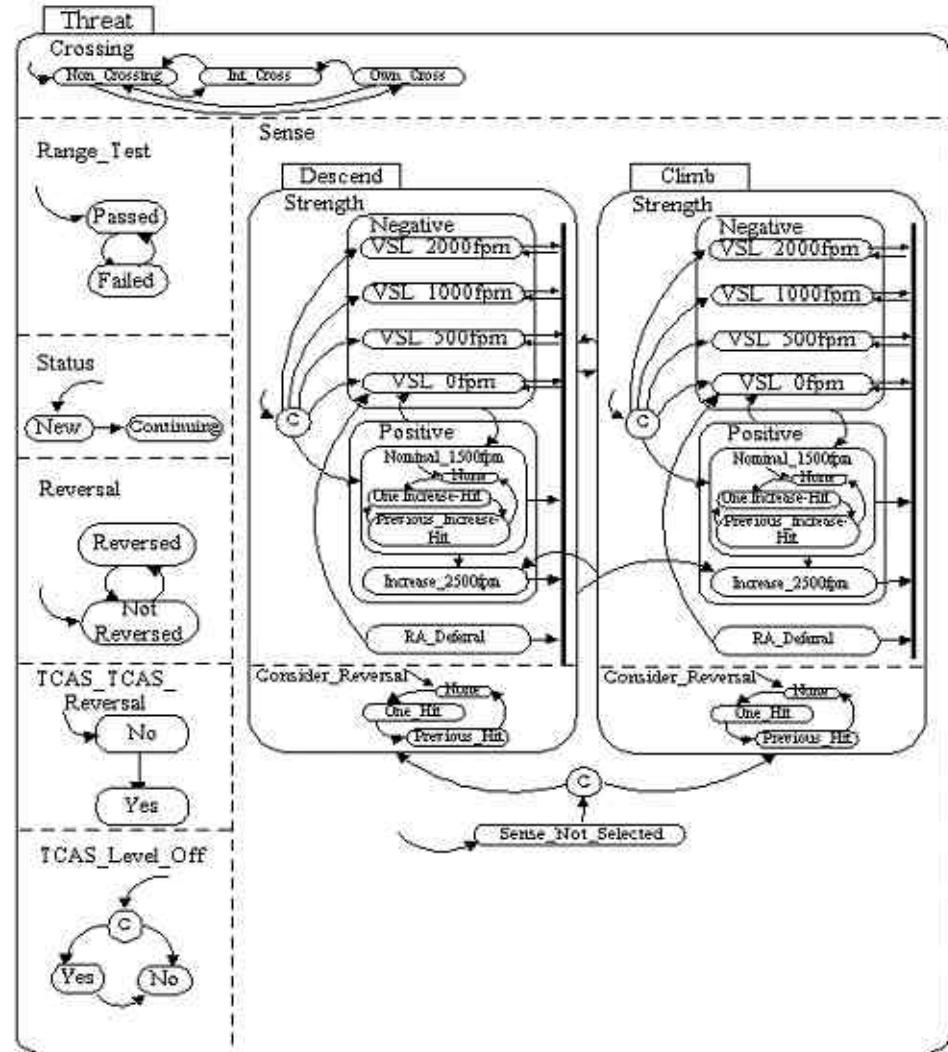
(a) State with internal concurrency



(b) Equivalent state with external synchronisation



An Example of Very Complex State



TCAS
Traffic Alert /Collision
Avoidance System

Advice on 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**
2. Determine the **initial** and **final states** of the entity
3. Identify the **events** that affect the entity
4. Working from the initial state, trace the impact of events and identify intermediate states
5. Identify any **entry** and **exit actions** on the states
6. Expand states using substates where necessary
7. 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



A Simple Process for Statechart

1. Identify a **class** participating in **behaviour** whose lifecycle is to be specified
2. Model **states**
3. Model **events**
4. Model **transitions**
5. Refine and elaborate as required



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 difference between **modelling** and **programming**?



Reading/Activity

- David 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

