Statecharts (hierarchical FSMs)
Hierarchical FSMs (Statecharts)

Introduced by D Harel (1987)
Hierarchical FSMs (Statecharts)

The FSM model presented until now can be extended by allowing
- Guards (conditions) and
- Actions on transitions

The original Statechart proposal added
- Hierarchy
- Concurrency
- Communication
- Transition joins
- History and deep history states

Completed with
- Entry,
- Exit,
- Do (while)

Actions on states
Conditions and actions

- The following example shows the general label format for a transition entering a state:

  \[
  \text{S1} \quad \text{event [condition] \{condition\_action\} / transition\_action} \quad \text{S2}
  \]

- When an event occurs, the condition for the transition with a matching event ([condition]) is evaluated.
- If the condition condition evaluates to true, the condition action condition\_action (\{condition\_action\}) is executed.
- If the destination state is determined to be a valid destination, the transition is taken. State S1 is exited.
  - The difference between condition and transition actions is more clear when junctions are involved
- The transition action transition\_action is executed when the transition is taken.
- State S2 is entered.
Hierarchical FSMs (Statecharts)

**Motivation for hierarchy:** a complex system cannot be described naively in a flat, unstratified multitude of states, because of the very large number of such states. A state-based approach must be modular, hierarchical and well structured, without having to represent explicitly the combination of all states.

Need to handle statements as
- “in all airborne states, when yellow handle is pulled, the seat will be ejected” (*cluster states into superstates*)
- “gearbox change of state is independent from braking system” (*independence or orthogonality*)
- “when selection button is pressed, enter selection mode” (*general transitions*)
- Display mode consists of time-display, date-display and stopwatch-display (*refinement of states*)
Hierarchical FSMs (Statecharts)

OR-clustering of states

Allows abstraction
refines ... (saves arrows)
Hierarchical FSMs (Statecharts)

Transitions that apply to all the states of a superstate

$\epsilon$ and $\phi$ apply to all states of $K$ (A B C D E F )

Same as ... (saves arrows)
Hierarchical FSMs (Statecharts)

AND-clustering of states

(UML notation)
The subsystem is explicitly defined as a number of FSM evolving in parallel

Same as …
There is a considerable reduction in the number of states and transitions. 
α is taken synchronously!
Statecharts: AND (parallel)states

- Stateflow notation for AND states
Hierarchical FSMs (Statecharts)

Transition and action inheritance
Inner states inherit the transitions and actions of parent states

if subS1 is active

events ev1 and ev2 are processed according to the definitions of the parent state
All during actions in the hierarchy are executed
Inner and outer junctions

Difference between inner and outer transitions
• In inner transitions exit and entry statements are not executed
• In outer transitions, the state is exited and re-entered
Default transition, history and deep history state

A default transition is taken when a chart (subchart) is entered first.

Extends the concept of initial state providing an initial transition

History state (H): the last active state in the current subchart

Deep history state (H\*): the last active state in the current subchart and its subcharts (recursively)

Several options are possible for subcharted states
State actions

States can have different action types, which include entry, during, exit, bind, and, on event_name actions. In stateflow, the actions for states are assigned using the following general format:

```
name/
entry: entry actions
during: during actions
exit: exit actions
bind: data_name, event_name
on event_name: on event_name actions
```

Power_on/
entry: ent_action();
during: dur_action();
exit: exit_action();
on Switch_off: on_action();

![Stateflow diagram](image)
State actions

Stateflow syntax

actions are preceded by the prefix or the short prefix, followed by a required colon (:), followed by one or more actions. Separate multiple actions with a carriage return, semicolon (;), or a comma (,). If you enter the name and slash followed directly by actions, the actions are interpreted as entry action(s).
State actions

Entry Actions (entry: or en:)
Entry actions are executed for a state when the state is entered (becomes active).

Exit Actions (exit: or ex:)
Exit actions for a state are executed when the state is active and a transition out of the state is taken.

During Actions (during: or du:)
During actions are executed for a state when it is active and an event occurs and no valid transition to another state is available.

On Event_Name Actions (on event event_name:)
There can be multiple on event_name lines for different events. On event_name actions are executed when the state is active and the event event_name is received by the state. This is also accompanied by the execution of any during actions for the state.
Transition junctions

- Transition junctions are additional shortcuts that allow to group together conditions/actions that apply to multiple transitions.
Connective junctions

- To/from AND states
Connective junctions

Other types:
From Mathworks’ definition:
The connective junction enables representation of different possible transition paths for a single transition. Connective junctions are used to help represent the following:

- Variations of an if-then-else decision construct, by specifying conditions on some or all of the outgoing transitions from the connective junction
- A self-loop transition back to the source state if none of the outgoing transitions is valid
Connective junctions

Other types:

- ... 
  - Variations of a for loop construct, by having a self-loop transition from the connective junction back to itself 
  - Transitions from a common source to multiple destinations 
  - Transitions from multiple sources to a common destination 
  - Transitions from a source to a destination based on common events
Connective junctions

- Connective junctions can be used to define a program dataflow

```python
if [c1]{
a1
  if [c2]{
    a2
  } else if [c3]{
    a3
  }
}
```
Transitions with junctions may force backtracking. Initially, state A is active and conditions c1, c2, and c3 are true.

- Condition c1 is true, therefore action a1 is executed.
- Condition c3 is true, therefore action a3 is executed.
- Condition c4 is not true, therefore control flow is backtracked to state A.
- The Stateflow diagram root checks to see if there is another valid transition from state A. There is a valid transition segment marked with the condition c2 from state A to a connective junction.
- Condition c2 is true, therefore action a2 is executed.
- Condition c3 is true, therefore action a3 is executed.
- Condition c4 is not true, therefore control flow is backtracked to state A.

Both a1 and a2 are executed, a3 is executed twice.
Backtracking on junctions

To resolve this problem, consider the following. The previous example is amended with two terminating junctions that allow flow to terminate if either c3 or c4 is not true. This leaves state A active without taking any unnecessary actions.
Statecharts

Harel’s stopwatch
• a summary of the notation ....