2.2 State Machines
Model-driven Development with State Machines

- Statecharts / State Machines
  - UML 1.3: State Machines (2001)
  - UML 2.1: State Machines (2007)
  - Other definitions (e.g. Rhapsody, also by Harel) with slightly different syntax and semantics

- Why State Machines (in this lecture)?
  - Can describe behavior of a (part of a) system
  - Description is model-based
  - Modeling of states and concurrency (Alternatives: Petri nets, SDL, Z, Esterel)
Agenda

- Repetition: Automata and State Machines
- Advanced concepts for State Machines
- Conflicting transitions
- Comparison of different semantics
  - Stateflow
  - UML
  - Sequential Statecharts
Recapitulation: Automata

- Formal definition
  - 5-tuple \((S, \Sigma, \delta, q_0, F)\)
    - \(S\): set of states
    - \(\Sigma\): input alphabet
    - \(\delta\): transition function
    - \(q_0\): start state
    - \(F\): set of accept states

- Automaton models states of a system
- System component has finite set of internal states (called “configuration”)
- Example: vending machine from lecture “Modellierung”
- Moore-Automaton: output/actions in states
- Mealey-Automaton: output/actions at transitions
Recapitulation: State Machines

- Similar to automata
- Combination of Moore- and Mealy-Automata
- Describe the life-cycle of an object
- Notion of “Signals” ⇒ Communication between State Machines

UML Elements
- StateMachine
- SimpleState
- FinalState
- CompositeState
- Regions, Submachine,
- Pseudostates:
  - Junction, Choice, Fork, Join, ShallowHistory, Initial
- ConnectionPointReference
- ChangeEvent, TimeEvent, Signal
Transitions: Events

- Events trigger transitions, i.e. they cause a state change

- An event is anything that can *happen* in a system
  - Signal trigger: Asynchronous triggering of a transition
  - Call trigger: Invocation of an operation
  - Time trigger: Deadline by which a transition is triggered
  - Change trigger: Transition is triggered when an attribute changes

- State Machines can communicate via signals, i.e. a signal event by one State Machine can trigger a transition in another State Machine
Transitions: Parameters and conditions

- Signal Trigger Events can have parameters
- Transition may only fire if condition is fulfilled
- All elements are optional (⇒ ε transitions for automata)

### Diagram

1. **Signal Trigger Event**: MoneyReceived(newAmount)
2. **Parameter**: [currentAmount + newAmount = price]
3. **Condition**: / currentAmount += newAmount
4. **Action**: Show „OK“
5. **Show price**
Transitions: Compound Transitions

- Model alternatives in a well arranged way

Normal transitions

- CoffeeButtonPressed() [cupInPlace]
- EspressoButtonPressed() [cupInPlace]
- ChocolateButtonPressed() [cupInPlace]

Make Coffee
Make Espresso
Make Hot Chocolate

Compound transitions with dynamic choice

- CoffeeButtonPressed()
- EspressoButtonPressed()
- ChocolateButtonPressed()

Make Coffee
Make Espresso
Make Hot Chocolate
Transitions: Compound Transition

- Junction: Chain Transitions, No Semantics
- Choice: Dynamic Conditional Branch

Show price / x = 1

Show „OK“

[cupInPlace] / x = 3

[x=1] [x=2] [x=3]

Make Coffee Make Espresso Make Hot Chocolate

Show price / x = 1

Show „OK“

[cupInPlace] / x = 3

[x=1] [x=2] [x=3]

Make Coffee Make Espresso Make Hot Chocolate
States: Actions and activities in states

- Actions (in states)
  - Entry and exit actions
  - Can change attributes (and thereby redefine the state!)
  - May not be interrupted (always run to completion)

- Activities
  - Marked with “Do”
  - Run while State Machine is in corresponding state
  - May be interrupted, hence cannot change the state of an object

**Make Coffee**

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry / BoilWater()</td>
</tr>
<tr>
<td>Do / ShowWait()</td>
</tr>
<tr>
<td>Exit / RefillWater()</td>
</tr>
</tbody>
</table>
Hierarchical States: Composite States

- Sometimes events must be responded to equally, regardless of the current state
Hierarchical States: Simple composite state

- Reduce visual complexity by using a hierarchy of states
- Only one simple state is active at a given time (hence Composite)

![State Machine Diagram]

- System running
- Composite state
- Basic states

- Machine opened
- MaintenancePause
- Show price
- Show "OK"
- Make Coffee
- Make Espresso
- Make Hot Chocolate
Hierarchical States: Simple composite state

- Reduce visual complexity by using a hierarchy of states
- Only one simple state is active at a given time (hence Composite)

Composite state icon

System running → Maintenance Pause
machine opened

Basic state
Hierarchical States: Orthogonal Composite State

- Model concurrency with “regions” (= parallel states)
- One basic state in each region is active

State Machines
Transitions: Fork and Join

- **fork** vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices.
- **join** vertices serve to merge several transitions emanating from source vertices in different orthogonal regions.

![Diagram of State Machines](image-url)
Hierarchical States: Shallow History

- Composite state with “memory”
- If a composite state is re-entered, the last active basic state directly below the composite state is activated
- Composite state can still be entered “normally” (bypassing history)
Hierarchical States: Deep History

• If an Composite state is re-entered, the last active basic state at every nested level is activated
Entry and Exit Points

- An entry point pseudostate
  - Entry point of a state machine or composite state
  - At most a single transition to a vertex within the same region.
- An exit point pseudostate
  - Exit point of a state machine or composite state
  - Implies the exit of this composite state or submachine state
  - Triggering of the transition that has this exit point as source
Submachine State

- Reuse of state machines
- Embed a state machine into another
- Inner state machine can terminate normally, abort or be interrupted
- ExitPoints, EntryPoints

State Machines
Conflicting transitions in Orthogonal Composite State

- Different semantics for execution of orthogonal states
  - Microstep Semantics (e.g. in Stateflow, UML 2)
  - Superstep Semantics (e.g. in Statemate)
  - Sequential Semantics (proposed by A. Zündorf)
Both transitions are taken synchronously and consume e1
No order for evaluation of actions defined!
Result not uniquely determined (race condition)
Both transitions are taken synchronously and consume e1

No order for evaluation of events defined! Queue, Stack, Priorized

Result not uniquely determined (race condition)
Each parallel state is executed on his own as far as possible.

Result depends on how local variables are merged.
Non-determinism introduced by guards

\[ e_1 / e_2 \]
\[ x := y + 1; \]

\[ e_3 \]
\[ y := 2 \times x; \]

\[ e_4 \]
\[ e_5 [x \leq 0] \]
\[ e_5 [x \geq 0] \]
Sequential Semantics

- Explicit use of priorities to remove non-determinism
- In this example: higher value = higher priority
- Define if high-level transitions have higher or lower priority
Problems with Semantics

- Non-determinism
  - If there are several possibilities for the execution of a step, the result depends on the implementation.

- Lifetime of events
  - According to Harel\(^1\) Events live for one step of the system, i.e. each State Machine has a chance to use an event that has been produced in the previous step. After that the events are discarded.

- Publishing of events in Superstep semantics
  - In Superstep semantics it depends on the implementation if a State Machine published all self-produced events after one superstep or if only the events of the last microstep are published.

---

1 Harel, Naamad: The STATEMATE Semantics of Statecharts, ACM Transactions of Software Engineering, 1996
Overview: Semantics

- Microstep semantics:
  - Changes become immediately visible (i.e. in the same step)
  - Can lead to non-determinism regarding assignments and state changes
  - Concrete semantics depend on implementation

- Superstep semantics:
  - Run-to-completion principle
  - Determinism regarding state changes, but still problems with assignments
  - Concrete semantics depend on implementation

- Sequential semantics:
  - Use explicit priorities to remove non-determinism
  - Useful for code generation
“Event occurrences are detected, dispatched, and then processed by the state machine, one at a time. The order of dequeuing is not defined, leaving open the possibility of modeling different priority-based schemes.”

“In the presence of orthogonal regions it is possible to fire multiple transitions as a result of the same event occurrence — as many as one transition in each region in the current state configuration. In case where one or more transitions are enabled, the state machine selects a subset and fires them. [...] The order in which selected transitions fire is not defined.”

Summary

- State Machines are better suited to describe complex behavior than Deterministic Finite Automaton (DFA)
- State Machine are not more powerful in a computational sense (= can be mapped to DFA)
- Advanced notation elements
  - Transitions: events, conditions, actions, compound transitions
  - States: basic states, Composite states, Orthogonal Composite states, history
- Problems
  - Conflicting transitions
  - Execution semantics