The Plan

- Thu, 19. 1.: Live Sequence Charts I
  Firstly: State-Machines Rest, Code Generation
- Tue, 24. 1.: Live Sequence Charts II
- Thu, 26. 1.: Live Sequence Charts III
- Tue, 31. 1.: Tutorial 7
- Thu, 2. 2.: Model Based/Driven SW Engineering
- Mon, 6. 2.: Inheritance
- Tue, 7. 2.: Meta-Modelling + Questions

February, 17th: The Exam.
Hierarchical State Machines: Retrospective
UML distinguishes the following kinds of states:

- **simple state**
  - entry
  - do
  - exit
  - entry
  - do
  - exit
  - entry
  - do
  - exit

- **final state**
- **composite state**

- OR

- AND

**pseudo-state**
- initial
- (shallow) history
- deep history
- fork/join
- junction, choice
- entry point
- exit point
- terminate

**submachine state**
- S : s

---

**Exercise 4.(i)**

<table>
<thead>
<tr>
<th>nr.</th>
<th>wbts</th>
<th>dsp</th>
<th>st</th>
<th>stable</th>
<th>lock</th>
<th>wis</th>
<th>st</th>
<th>stable</th>
<th>¢</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(u₁, e₁), (u₁, e₂)</td>
</tr>
<tr>
<td>1</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>1</td>
<td>e₁</td>
</tr>
<tr>
<td>2</td>
<td>s₁</td>
<td>s₂</td>
<td>0</td>
<td>0</td>
<td>s₂</td>
<td>0</td>
<td>s₁</td>
<td>0</td>
<td>1</td>
<td>e₂</td>
</tr>
<tr>
<td>3</td>
<td>s₁</td>
<td>s₂</td>
<td>0</td>
<td>0</td>
<td>s₂</td>
<td>0</td>
<td>s₁</td>
<td>0</td>
<td>1</td>
<td>e₂</td>
</tr>
<tr>
<td>4</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>1</td>
<td>e₂</td>
</tr>
<tr>
<td>5</td>
<td>s₂</td>
<td>s₁</td>
<td>0</td>
<td>0</td>
<td>s₁</td>
<td>0</td>
<td>s₂</td>
<td>0</td>
<td>1</td>
<td>e₁</td>
</tr>
<tr>
<td>6</td>
<td>s₂</td>
<td>s₁</td>
<td>0</td>
<td>0</td>
<td>s₁</td>
<td>0</td>
<td>s₂</td>
<td>0</td>
<td>1</td>
<td>e₁</td>
</tr>
<tr>
<td>7</td>
<td>s₂</td>
<td>s₁</td>
<td>0</td>
<td>0</td>
<td>s₁</td>
<td>0</td>
<td>s₂</td>
<td>0</td>
<td>1</td>
<td>e₁</td>
</tr>
<tr>
<td>8</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>1</td>
<td>e₁</td>
</tr>
<tr>
<td>9</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>1</td>
<td>e₁</td>
</tr>
<tr>
<td>10</td>
<td>s₁</td>
<td>s₂</td>
<td>0</td>
<td>0</td>
<td>s₂</td>
<td>0</td>
<td>s₁</td>
<td>0</td>
<td>1</td>
<td>e₂</td>
</tr>
<tr>
<td>11</td>
<td>s₁</td>
<td>s₂</td>
<td>0</td>
<td>0</td>
<td>s₂</td>
<td>0</td>
<td>s₁</td>
<td>0</td>
<td>1</td>
<td>e₂</td>
</tr>
<tr>
<td>12</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>1</td>
<td>e₂</td>
</tr>
<tr>
<td>13</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>1</td>
<td>e₂</td>
</tr>
<tr>
<td>14</td>
<td>s₁</td>
<td>s₂</td>
<td>s₂</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>1</td>
<td>e₂</td>
</tr>
<tr>
<td>15</td>
<td>s₁</td>
<td>s₂</td>
<td>s₂</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₂</td>
<td>1</td>
<td>e₂</td>
</tr>
</tbody>
</table>

---
Exercise 4 (ii)-(v)

(ii) Explain the difference between step and RTC-step.

(iii) Is it possible to reach #?

(iv) Considering the rule for environment interaction, how does the possible behaviour change?

(v) How does the behaviour simulated with Rhapsody compare to the results from Task (i)?

Active and Passive Objects
What about non-Active Objects?

Recall:
- We’re still working under the assumption that all classes in the class diagram (and thus all objects) are active.
- That is, each object has its own thread of control and is (if stable) at any time ready to process an event from the ether.
  → steps of active objects can interleave.

But the world doesn’t consist of only active objects.
For instance, in the Vending Machine from the exercises we could wish to have the whole system live in one thread of control.

So we have to address questions like:
- Can we send events to a non-active object?
- And if so, when are these events processed?
- etc.

Active and Passive Objects: Nomenclature

Harel and Gery (1997) propose the following (orthogonal!) notions:
- A class (and thus the instances of this class) is either active or passive as defined by the class diagram.
  → An active object has (in the operating system sense) an own thread: an own program counter, an own stack, etc.
  → A passive object doesn’t.

A class is either reactive or non-reactive.
- A reactive class has a (non-trivial) state machine.
- A non-reactive one hasn’t.

Which combinations do we (not) understand yet?

<table>
<thead>
<tr>
<th></th>
<th>active</th>
<th>passive</th>
</tr>
</thead>
<tbody>
<tr>
<td>reactive</td>
<td>✔</td>
<td>?</td>
</tr>
<tr>
<td>non-reactive</td>
<td>✔?</td>
<td>✔?</td>
</tr>
</tbody>
</table>
Passive Reactive / Rhapsody Style

- In each class, add (implicit) link `itsAct` and use it to make each object `u` know the active object `u_a` which is responsible for dispatching events to `u`.
  - If `u` is an instance of an active class, then `u_a = u`.
- Equip all signals with (implicit) association `dest` and use it to point to the destination object.
  - For each signal `F`, have a version `F_C` with an association `dest : C_{0..1}, C \in \mathcal{C}` (no inheritance yet).
In each class, add (implicit) link itsAct and use it to make each object know the active object \( u_a \) which is responsible for dispatching events to \( u \).

If \( u \) is an instance of an active class, then \( u_a = u \).

Equip all signals with (implicit) association dest and use it to point to the destination object.

For each signal \( F \), have a version \( F_C \) with an association dest: \( C_0 \wedge C \in \mathcal{C} \) (no inheritance yet).

Sending an event:
- \( n \! F \) in \( u_1 : C_1 \) becomes:
- Create an instance \( u_e \) of \( F_{C_2} \) and set \( u_e \)'s dest to \( u_d := \sigma(u_1)(n) \).
- Send to \( u_a := \sigma(\sigma(u_1)(n))(\text{itsAct}) \), i.e., \( \varepsilon' = \varepsilon \oplus (u_a, u_e) \).

Dispatching an event:
- Observation: the ether only has events for active objects.
- Say \( u_e \) is ready in the ether for \( u_a \).
- Then \( u_a \) asks \( \sigma(u_e)(\text{dest}) = u_d \) to process \( u_e \) and waits until completion of corresponding RTC.
- \( u_d \) may in particular discard event.
And What About Methods?

- In the current setting, the (local) state of objects is **only** modified by actions of transitions, which we abstract to transformers.
- In general, there are also **methods**.
- UML follows an approach to separate
  - the **interface declaration** from
  - the **implementation**.

In C++-lingo: distinguish **declaration** and **definition** of method.

- In UML, the former is called **behavioural feature** and can (roughly) be
  - a **call interface** \( f(T_1, \ldots, T_{n_1}) : T_1 \)
  - a **signal name** \( E \)

Note: The signal list can be seen as redundant (can be looked up in the state machine) of the class. But: certainly useful for documentation (or sanity check).
Semantics:

- The implementation of a behavioural feature can be provided by:
  - An operation.
  
    In our setting, we simply assume a transformer like $T_f$.
  
    It is then, e.g. clear how to admit method calls as actions on transitions: function composition of transformers (clear but tedious: non-termination).
  
    In a setting with Java as action language: operation is a method body.
  
    The class’ state-machine (“triggered operation”).

- Calling $F$ with $n_2$ parameters for a stable instance of $C$ creates an auxiliary event $F$ and dispatches it (bypassing the ether).
- Transition actions may fill in the return value.
- On completion of the RTC step, the call returns.
- For a non-stable instance, the caller blocks until stability is reached again.

Behavioural Features: Visibility and Properties

- Visibility:
  - Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods.

- Useful properties:
  - concurrency
    - concurrent – is thread safe
  
    guarded – some mechanism ensures/should ensure mutual exclusion
  
    sequential – is not thread safe, users have to ensure mutual exclusion
  
  - isQuery – doesn’t modify the state space (thus thread safe)
A Closer Look to Rhapsody Code Generation

[Diagram showing the code generation process from Rhapsody to executable output]
• Rhapsody also supports non-active objects – their instances share an event pool with an active object.

• Behavioural Features: exist.

• Semantic Variation Points are legion – but manageable, e.g. by appropriate modelling guidelines (stick to “the beaten track”).

• Interactions can be used for reflective descriptions of behaviour, i.e.
  • describe what behaviour is (un)desired, without (yet) defining how to realise it.

• One visual formalism for interactions: Live Sequence Charts
  • partially ordered locations,
  • instantaneous and asynchronous messages,
  • conditions and local invariants

Later: pre-charts.

References
References


