

**Contents & Goals**

**Last Lecture:**
- Legal state configurations
- Legal transitions
- Rules (i) to (v) for hierarchical state machines

**This Lecture:**

- **Educational Objectives:** Capabilities for following tasks/questions.
  - How do entry / exit actions work? What about do-actions?
  - What is the effect of shallow / deep history pseudo-states?
  - What about junction, choice, terminate, etc.?
  - What is the idea of deferred events?
  - How are passive reactive objects treated in Rhapsody’s UML semantics?
  - What about methods?

- **Content:**
  - Entry / exit / do actions, internal transitions
  - Remaining pseudo-states; deferred events
  - Passive reactive objects
  - Behavioural features
Entry and Exit Actions
Entry/Do/Exit Actions

- In general, with each state \( s \in S \) there is associated
  - an **entry**, a **do**, and an **exit** action (default: **skip**)
  - a possibly empty set of trigger/action pairs called **internal transitions**, (default: empty).

**Note:** ‘entry’, ‘do’, ‘exit’ are reserved names; \( E_1, \ldots, E_n \in \mathcal{E} \).

- **Recall:** each action is supposed to have a transformer; assume \( t_{act_1}^{entry}, t_{act_1}^{exit}, \ldots \)
- Taking the transition above then amounts to applying

\[
t_{act_2}^{entry} \circ t_{act} \circ t_{act_1}^{exit}
\]

instead of just

\[
t_{act}
\]

\( \leadsto \) adjust Rules (ii), (iii), and (v) accordingly.
Internal Transitions

- Taking an **internal transition**, e.g. on $E_1$, only executes $t_{act_{E_1}}$.
- **Intuition**: The state is neither left nor entered, so: no exit, no entry action.
- **Note**: internal transitions also start a run-to-completion step.

$\leadsto$ adjust Rules (i), (ii), and (v) accordingly.

**Note**: the standard seems not to clarify whether internal transitions have priority over regular transitions with the same trigger at the same state.

Some code generators assume that internal transitions have priority!
That is: Entry / Internal / Exit don’t add expressive power to Core State Machines. If internal actions should have priority, $s_1$ can be embedded into an OR-state.

Abbreviation view may avoid confusion in context of hierarchical states.
**Intuition**: after entering a state, start its do-action.

- If the do-action terminates,
  - then the state is considered **completed** (like **final state**),
- otherwise,
  - if the state is left before termination, the do-action is stopped.

Recall the overall UML State Machine philosophy:

> “An object is either idle or doing a run-to-completion step.”

Now, what is it exactly while the do action is executing...?
The Concept of History, and Other Pseudo-States
What happens on...

- $R_s$?
  - $s_0$, $s_2$

- $R_d$?
  - $s_0$, $s_2$

- $A, B, C, S, R_s$?
  - $s_0$, $s_1$, $s_2$, $s_3$, $s_{4p}$, $s_3$

- $A, B, C, S, R_d$?
  - $s_0$, $s_1$, $s_2$, $s_3$, $s_{4p}$, $s_3$

- $A, B, C, D, E, S, R_s$?
  - $s_0$, $s_1$, $s_2$, $s_3$, $s_{4p}$, $s_3$

- $A, B, C, D, E, S, R_d$?
  - $s_0$, $s_1$, $s_2$, $s_3$, $s_{4p}$, $s_3$
Junction and Choice

- Junction ("static conditional branch"): 

- Choice: ("dynamic conditional branch")
• Junction ("static conditional branch"):  
  - **good**: abbreviation  
  - unfolds to so many similar transitions with different guards, the unfolded transitions are then checked for enabledness  
  - at best, start with trigger, branch into conditions, then apply actions  

• Choice: ("dynamic conditional branch")  
  - **evil**: may get stuck  
  - enters the transition **without knowing** whether there’s an enabled path  
  - at best, use “else” and convince yourself that it cannot get stuck  
  - maybe even better: **avoid**
Hierarchical states can be "folded" for readability.
(but: this can also hinder readability.)

Can even be taken from a different state-machine for re-use.

$S : s$
Hierarchical states can be “folded” for readability. (but: this can also hinder readability.)

Can even be taken from a different state-machine for re-use.

**Entry/exit points**
- Provide connection points for finer integration into the current level, than just via initial state.
- Semantically a bit tricky:
  - First the exit action of the exiting state,
  - then the actions of the transition,
  - then the entry actions of the entered state,
  - then action of the transition from the entry point to an internal state,
  - and then that internal state’s entry action.

**Terminate Pseudo-State**
- When a terminate pseudo-state is reached, the object taking the transition is immediately killed.
Are We Done?
UML distinguishes the following **kinds of states**: 

<table>
<thead>
<tr>
<th><strong>simple state</strong></th>
<th><strong>example</strong></th>
<th><strong>pseudo-state</strong></th>
<th><strong>example</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>final state</strong></td>
<td><img src="image1" alt="example" /></td>
<td>initial</td>
<td><img src="image2" alt="example" /></td>
</tr>
<tr>
<td><strong>composite state</strong></td>
<td><strong>OR</strong></td>
<td>(shallow) history</td>
<td><img src="image3" alt="example" /></td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td></td>
<td>deep history</td>
<td><img src="image4" alt="example" /></td>
</tr>
</tbody>
</table>

**fork/join**

**junction, choice**

**entry point**

**exit point**

**terminate**

**submachine state**
Deferred Events in State-Machines
Deferred Events: Idea

UML state machines comprises the feature of **deferred events**.

The idea is as follows:

- Consider the following state machine:

  ![State Machine Diagram](image)

  - Assume we’re stable in $s_1$, and $F$ is ready in the ether.
  - In the framework of our course, $F$ is discarded.
  - But we may find it a pity to discard the poor event and we may want to remember it for later processing, e.g. in $s_2$, in other words: defer it.

General **options** to satisfy such needs:

- Provide a pattern how to “program” this (use self-loops and helper attributes).
- Turn it into an original language concept. (← OMG’s choice)
Deferred Events: Syntax and Semantics

- **Syntactically**, each state has (in addition to the name) a set of deferred events. **Default**: the empty set.

- The **semantics** is a bit intricate, something like:
  - if Rule (i) (discard) would apply,
  - **but** $E$ is in the deferred set of the current state configuration,
  - then stuff $E$ into some “deferred events space” of the object, (e.g. into the ether ($\epsilon$) or into the local state of the object ($\sigma$))
  - and turn attention to the next event.

- **Not so obvious**:
  - Is there a priority between deferred and regular events?
  - Is the order of deferred events preserved?

- ... 

Fecher and Schönborn (2007), e.g., claim to provide semantics for the complete Hierarchical State Machine language, including deferred events.
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 crossing controller 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.
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 and Reactive / Rhapsody Style: Example

\[ C_1 \xrightarrow{n} C_2 \xrightarrow{m} D \]

\[ S M_{C_1}: \quad s_1 \xrightarrow{E/n!F, m!G} s_2 \]

\[ S M_{C_2}: \quad s_1 \xrightarrow{F/} s_2 \xrightarrow{} s_3 \]

\[ S M_D: \quad s_1 \xrightarrow{G/} s_2 \xrightarrow{} s_3 \]

\[ \sigma: \quad u_1 : C_1 \xrightarrow{n} u_2 : C_2 \xrightarrow{m} u_3 : D \]

\[ \varepsilon: (u_1, E_1) \]

Wanted: If all active \( E, F, G \),

\[ (u_1, \varepsilon) \xrightarrow{E, F, G} u_1 \]

\[ u_2 \xrightarrow{F,} u_2 \]

\[ u_3 \xrightarrow{G} u_3 \]

\[ \ldots \]

\[ (\sigma, \varepsilon) \xrightarrow{E, F, G} u_1 \]

\[ u_2 \xrightarrow{F,} u_2 \]

\[ u_3 \xrightarrow{G} u_3 \]

\[ \ldots \]
In each class, add (implicit) link \textit{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 \( u_a = u \).
In each class, add (implicit) link \( \text{itsAct} \) 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 \( \text{dest} \) and use it to point to the destination object.

For each signal \( F \), have a version \( F_C \) with an association \( \text{dest} : C_{0,1} \), \( C \in \mathcal{C} \) (no inheritance yet).
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).

\[
\text{Sending an event:}
\]

- $n!F$ in $u_1 : C_1$ becomes:
- Create an instance $u_e$ of $F_C$ 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., $\varepsilon' = \varepsilon \oplus (u_a, u_e)$.

\[
\text{Dispatching an event:}
\]

- 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.
Discussion
Semantic Variation Points

**Pessimistic view:** They are legion...

- **For instance,**
  - allow **absence of initial pseudo-states**
    - object may then “be” in enclosing state without being in any substate;
    - or assume one of the children states non-deterministically
  - (implicitly) **enforce determinism**, e.g.
    - by considering the order in which things have been added to the CASE tool’s repository,
    - or some graphical order (left to right, top to bottom)
  - allow **true concurrency**
  - etc. etc.

**Exercise:** Search the standard for “semantical variation point”.

**Optimistic view:** tools exist with complete and consistent code generation.

- **Crane and Dingel (2007),** e.g., provide an in-depth comparison of Statemate, UML, and Rhapsody state machines — the bottom line is:
  - **the intersection is not empty**
    - (i.e. there are pictures that mean the same thing to all three communities)
  - **none is the subset of another**
    - (i.e. for each pair of communities exist pictures meaning different things)
And What About Methods?
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.
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)

- For simplicity, we leave the notion of steps untouched, we construct our semantics around state machines. Yet we could explain pre/post in OCL (if we wanted to).
References
References


