Contents & Goals

Last Lecture:
- (Mostly) completed discussion of modelling structure.

This Lecture:
- Educational Objectives: Capabilities for following tasks/questions.
  - What’s the purpose of a behavioural model?
  - What does this State Machine mean? What happens if I inject this event?
  - Can you please model the following behaviour.

- Content:
  - For completeness: Modelling Guidelines for Class Diagrams
  - Purposes of Behavioural Models
  - UML Core State Machines
Design Guidelines for (Class) Diagram
(parly following Ambler (2005))

General Diagramming Guidelines Ambler (2005)

(Note: “Exceptions prove the rule.”)

2.1 Readability

1.–3. Support Readability of Lines

4. Apply Consistently Sized Symbols

9. Minimize the Number of Bubbles

10. Include White-Space in Diagrams

13. Provide a Notational Legend
2.2 Simplicity

14. Show Only What You Have to Show
15. Prefer Well-Known Notation over Exotic Notation
16. Large vs. Small Diagrams
18. Content First, Appearance Second

2.3 Naming

20. Set and (23. Consistently) Follow Effective Naming Conventions

2.4 General

24. Indicate Unknowns with Question-Marks
25. Consider Applying Color to Your Diagram
26. Apply Color Sparingly
5.1 General Guidelines

- 88. Indicate Visibility Only on Design Models (in contrast to analysis models)

5.2 Class Style Guidelines

- 96. Prefer Complete Singular Nouns for Class Names
- 97. Name Operations with Strong Verbs
- 99. Do Not Model Scaffolding Code [Except for Exceptions]
  
  e.g. `get/set` methods

- 103. Never Show Classes with Just Two Compartments
- 104. Label Uncommon Class Compartments
- 105. Include an Ellipsis (...) at the End of an Incomplete List
- 107. List Operations/Attributes in Order of Decreasing Visibility
5.3 Relationships

- 112. Model Relationships Horizontally
- 115. Model a Dependency When the Relationship is Transitory
- 117. Always Indicate the Multiplicity (of have good defaults)
- 118. Avoid Multiplicity "∗"
- 119. Replace Relationship Lines with Attribute Types (to have fewer lines)

5.4 Associations

- 127. Indicate Role Names When Multiple Associations Between Two Classes Exist
- 129. Make Associations Bidirectional Only When Collaboration Occurs in Both Directions
- 131. Avoid Indicating Non-Navigability (it depends; often is meant to be)
- 133. Question Multiplicities Involving Minimums and Maximums

5.6 Aggregation and Composition

exercises
**Task: Game Development**

**Task:** develop a video game.  **Genre:** Racing.  **Rest:** open, i.e.

<table>
<thead>
<tr>
<th>Degrees of freedom:</th>
<th>Exemplary choice: 2D-Tron</th>
</tr>
</thead>
<tbody>
<tr>
<td>• simulation vs. arcade</td>
<td>arcade</td>
</tr>
<tr>
<td>• platform (SDK or not, open or proprietary, hardware capabilities...)</td>
<td>open</td>
</tr>
<tr>
<td>• graphics (3D, 2D, ...)</td>
<td>2D</td>
</tr>
<tr>
<td>• number of players, AI</td>
<td>min. 2, AI open</td>
</tr>
<tr>
<td>• controller</td>
<td>open (later determined by platform)</td>
</tr>
<tr>
<td>• game experience</td>
<td>minimal: main menu and game</td>
</tr>
</tbody>
</table>
Modelling Structure: 2D-Tron

- In many domains, there are canonical architectures – and adept readers try to see/find/match this!
- For games:

```
Main

External inputs
- Keyboard
- Joystick
- ...

Game Logic
- player scores
- interface inputs/engine

(update) notify

(Physics) Engine
- physical objects
- collision notification

Output
- Graphics (from ASCII to bitmap; native or via API)
- Sound
- ...
```

Modelling Structure: 2D-Tron

```
inputs

Joystick?
...
Keyboard?

Control

Player
- colour
- score
- direction
- speed

Gameplay

Output

OpenGL?
...

Render

Conventions:
- default \( \mu \) is 1
- default \( \xi \) is +

Tron

Engine
- area
- width
- height

world

head

notify

update

Segment
- \( x_0, y_0 \)
- \( x_1, y_1 \)
- colour

AI?

1..*
Stocktaking...

**Have:** Means to model the structure of the system.
- Class diagrams graphically, concisely describe sets of system states.
- OCL expressions logically state constraints/invariants on system states.

**Want:** Means to model behaviour of the system.
- Means to describe how system states evolve over time, that is, to describe sets of sequences
  \[ \sigma_0, \sigma_1, \ldots \in \sum^\omega \]
  of system states.
What Can Be Purposes of Behavioural Models?

Example: Pre-Image (the UML model is supposed to be the blue-print for a software system).

A description of behaviour could serve the following purposes:

- **Require Behaviour.**  **“System definitely does this”**
  “This sequence of inserting money and requesting and getting water must be possible.”
  (Otherwise the software for the vending machine is completely broken.)

- **Allow Behaviour.**  **“System does subset of this”**
  “After inserting money and choosing a drink, the drink is dispensed (if in stock).”
  (If the implementation insists on taking the money first, that’s a fair choice.)

- **Forbid Behaviour.**  **“System never does this”**
  “This sequence of getting both, a water and all money back, must not be possible.”
  (Otherwise the software is broken.)

Note: the latter two are trivially satisfied by doing nothing...

Constructive Behaviour in UML

UML provides two visual formalisms for constructive description of behaviours:

- **Activity Diagrams**
- **State-Machine Diagrams**

We (exemplary) focus on State-Machines because

- somehow “practice proven” (in different flavours),
- prevalent in embedded systems community,
- indicated useful by Dobing and Parsons (2006) survey, and
- Activity Diagram’s intuition changed (between UML 1.x and 2.x) from transition-system-like to petri-net-like...

Example state machines:
UML Model Instances

\[ S = (T, C, V, \text{attr}) \]

\[ SM = (\Sigma, D, A, \rightarrow, SM) \]

\[ \varphi \in \text{OCL} \]

Mathematics

\[ \pi = (\sigma_0, \varepsilon_0) \frac{\text{cons}, S\text{nd}_0}{u_0} (\sigma_1, \varepsilon_1) \cdots \]

\[ w = ((\sigma_i, \text{cons}_i, S\text{nd}_i))_{i \in \mathbb{N}} \]

UML State Machines: Overview
**Brief History:**
- Rooted in Moore/Mealy machines, Transition Systems, etc.
- Manifest in tool Statemate Harel et al. (1990) (simulation, code-generation); nowadays also in Matlab/Simulink, etc.
- From UML 1.x on: State Machines (not the official name, but understood: UML-Statecharts)
- Late 1990’s: tool Rhapsody with code-generation for state machines.

**Note:** there is a common core, but each dialect interprets some constructs subtly different Crane and Dingel (2007). *(Would be too easy otherwise...)*

**Roadmap: Chronologically**

**Syntax:**
(i) UML State Machine Diagrams.
(ii) Def.: Signature with signals.
(iii) Def.: **Core state machine**.
(iv) Map UML State Machine Diagrams to core state machines.

**Semantics:**
The Basic Causality Model
(v) Def.: **Ether** (aka. event pool)
(vi) Def.: **System configuration**.
(vii) Def.: **Event**.
(viii) Def.: **Transformer**.
(ix) Def.: **Transition system**, computation.
(x) Transition relation induced by core state machine.
(xi) Def.: **step, run-to-completion step**.
(xii) Later: Hierarchical state machines.
Definition. A tuple
\[ T = (T, C, V, atr, \mathcal{E}), \quad \mathcal{E} \text{ a set of signals}, \]
is called signature (with signals) if and only if
\[ (T, C \cup \mathcal{E}, V, atr) \]
is a signature (as before).

Note: Thus conceptually, a signal is a class and can have attributes of plain type, and participate in associations.
### Core State Machine

**Definition.**
A core state machine over signature $\mathcal{S} = (T, \mathcal{E}, V, atr, \mathcal{E})$ is a tuple

$$M = (S, s_0, \rightarrow)$$

where
- $S$ is a non-empty, finite set of (basic) states,
- $s_0 \in S$ is an initial state,
- and

$$\rightarrow \subseteq S \times (\mathcal{E} \cup \{\bot\}) \times \mathcal{E} \times \mathcal{E} \times S$$

is a labelled transition relation.

We assume a set $\mathcal{E}$ of boolean expressions over $\mathcal{S}$ (for instance OCL, may be something else) and a set $\mathcal{A}$ of actions.
From UML to Core State Machines: By Example

UML state machine diagram SM:

\[
\begin{array}{c}
\text{annot} \coloneqq \[ \langle \text{event} \rangle \] \[ \langle \text{event} \rangle^* \] \[ \langle \text{guard} \rangle \] \[ \langle \text{action} \rangle \]
\end{array}
\]

with
- \( \text{event} \in \mathcal{E} \),
- \( \text{guard} \in \text{Expr} \),
- \( \text{action} \in \text{Act} \).

maps to
\[
M(SM) = (\{s_1, s_2\}, \{(s_1, e_1, g_1, a_1, s_2)\})
\]

Abbreviations and Defaults in the Standard

Reconsider the syntax of transition annotations:

\[
\text{annot} \coloneqq \[ \langle \text{event} \rangle \] \[ \langle \text{event} \rangle^* \] \[ \langle \text{guard} \rangle \] \[ \langle \text{action} \rangle \]
\]

where \( \text{event} \in \mathcal{E} \), \( \text{guard} \in \text{Expr} \), \( \text{action} \in \text{Act} \).

What if things are missing?

In the standard, the syntax is even more elaborate:
- \( E(v) \) — when consuming \( E \) in object \( u \), attribute \( v \) of \( u \) is assigned the corresponding attribute of \( E \).
- \( E(v : T) \) — similar, but \( v \) is a local variable, scope is the transition
State-Machines belong to Classes

In the following, we assume that

- a UML model consists of a set $\mathcal{D}$ of class diagrams and a set $\mathcal{H}$ of state chart diagrams (each comprising one state machine $SM$).
- each state machine $SM \in \mathcal{H}$ is associated with a class $C_{SM} \in \mathcal{C}(\mathcal{H})$.
- For simplicity, we even assume a bijection, i.e., we assume that each class $C \in \mathcal{C}(\mathcal{H})$ has a state machine $SM_C$ and that its class $C_{SM_C}$ is $C$.

If not explicitly given, then this one:

$$SM_0 := (\{s_0\}, s_0, \emptyset, \text{true}, skip, s_0).$$

We will see later that this choice does no harm semantically.

**Intuition 1:** $SM_C$ describes the behaviour of the instances of class $C$.

**Intuition 2:** Each instance of class $C$ executes $SM_C$.

**Note:** we don’t consider multiple state machines per class. We will see later that this case can be viewed as a single state machine with as many AND-states.

References
References


