Software Design, Modelling and Analysis in UML

Lecture 10: Constructive Behaviour, State Machines Overview

Contents & Goals

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

This Lecture:
- Educational Objectives: Capabilities for following tasks/questions.
  - Discuss the style of this class diagram.
  - What’s the difference between reflective and constructive descriptions of behaviour?
  - 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
  - Constructive vs. Reflective
  - UML Core State Machines (first half)
Invariant in Class Diagram Example

If $\mathcal{C} \mathcal{D}$ consists of only $\mathcal{C} \mathcal{D}$ with the single class $C$, then

- $\text{Inv}(\mathcal{C} \mathcal{D}) = \text{Inv}(\mathcal{C} \mathcal{D}) = \{ \text{context } C \text{ inv: } v > 3 \}$
Constraints vs. Types

Find the 10 differences:

<table>
<thead>
<tr>
<th>$C$</th>
<th>$P(T) = {3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : \text{Int} { x = 3 \lor x &gt; 17 }$</td>
<td>$\bigcup { n \in \mathbb{N} \mid n &gt; 17 }$</td>
</tr>
</tbody>
</table>

- $x = 4$ is well-typed in the left context, a system state satisfying $x = 4$ violates the constraints of the diagram.
- $x = 4$ is not even well-typed in the right context, there cannot be a system state with $\sigma(u)(x) = 4$ because $\sigma(u)(x)$ is supposed to be in $P(T)$ (by definition of system state).

Rule-of-thumb:
- If something “feels like” a type (one criterion: has a natural correspondence in the application domain), then make it a type.
- If something is a requirement or restriction of an otherwise useful type, then make it a constraint.

Semantics of a Class Diagram

Definition. Let $\mathcal{CD}$ be a set of class diagrams. We say, the semantics of $\mathcal{CD}$ is the signature it induces and the set of OCL constraints occurring in $\mathcal{CD}$, denoted

$$[\mathcal{CD}] := (\mathcal{S}(\mathcal{CD}), \text{Inv}(\mathcal{CD})).$$

Given a structure $\mathcal{S}$ of $\mathcal{CD}$ (and thus of $\mathcal{CD}$), the class diagrams describe the system states $\Sigma_{\mathcal{CD}}$. Of those, some satisfy $\text{Inv}(\mathcal{CD})$ and some don’t. We call a system state $\sigma \in \Sigma_{\mathcal{CD}}$ consistent if and only if $\sigma \models \text{Inv}(\mathcal{CD})$.

In pictures:
**Pragmatics**

**Recall:** A UML model is an image or pre-image of a software system.

A set of class diagrams $\mathcal{C}$ with invariants $\text{Inv}(\mathcal{C})$ describes the structure of system states.

Together with the invariants it can be used to state:

- **Pre-image:** Dear programmer, please provide an implementation which uses only system states that satisfy $\text{Inv}(\mathcal{C})$.

- **Post-image:** Dear user/maintainer, in the existing system, only system states which satisfy $\text{Inv}(\mathcal{C})$ are used.

(The exact meaning of “use” will become clear when we study behaviour — intuitively, the system states that are reachable from the initial system state(s) by calling methods or firing transitions in state-machines.)

**Example:** highly abstract model of traffic lights controller.

![Traffic Lights Controller Diagram](image)

**Addendum: Semantics of OCL Boolean Operations**
• semantics of operator is monotone
   (T propagates though, once a sub-expression evaluates to T, the whole expression does)

\[ I(t)(x,y) = \begin{cases} x \land y, & \text{if } x \neq T \text{ and } y \neq T \\ \bot, & \text{otherwise} \end{cases} \]

• \( I(\lor)(p,q) = \begin{cases} \text{true, if } p = \text{true} \text{ or } q = \text{true} \text{ and } p \neq T \text{ and } q \neq T \\ \bot, & \text{otherwise} \end{cases} \)

Then
- not(defined(self.x)) \impliedby \text{self.x} > 0
- defined(self.x) \lor \text{self.x} > 0

would not do what we want

---

Correct Semantics of OCL Boolean Operations

Table A.2 - Semantics of boolean operations

<table>
<thead>
<tr>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( b_1 ) and ( b_2 )</th>
<th>( b_1 ) or ( b_2 )</th>
<th>( b_1 ) xor ( b_2 )</th>
<th>( b_1 ) implies ( b_2 )</th>
<th>not ( b_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>( \bot )</td>
<td>false</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>false</td>
</tr>
</tbody>
</table>

---

Table A.2 - Semantics of boolean operations

<table>
<thead>
<tr>
<th>( \bot )</th>
<th>false</th>
<th>false</th>
<th>( \bot )</th>
<th>( \bot )</th>
<th>( \bot )</th>
<th>( \bot )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bot )</td>
<td>true</td>
<td>( \bot )</td>
<td>true</td>
<td>( \bot )</td>
<td>true</td>
<td>( \bot )</td>
</tr>
<tr>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
</tbody>
</table>
Design Guidelines for (Class) Diagram

(partly following [Ambler, 2005])

Be careful whose advice you buy, but,
be patient with those who supply it.

Baz Luhrmann/Mary Schmich

Main and General Modelling Guideline (admittedly: trivial and obvious)

Be good to your audience.

“Imagine you’re given your diagram $D$ and asked to conduct task $T$.

- Can you do $T$ with $D$?
  (semantics sufficiently clear? all necessary information available? …)

- Does doing $T$ with $D$ cost you more nerves/time/money/… than it should?”

In other words:
- the things most relevant for $T$, do they stand out in $D$? __YES__
- the things least relevant for $T$, do they disturb in $D$? __NO__

For a good diagram
Main and General Quality Criterion (again: trivial and obvious)

- **Q:** When is a (class) diagram a good diagram?
- **A:** If it serves its purpose/makes its point.

**Examples** for purposes and points and rules-of-thumb:

- **Analysis/Design**
  - realizable, no contradictions
  - abstract, focused, admitting degrees of freedom for (more detailed) design
  - platform independent – as far as possible but not (artificially) farer

- **Implementation/A**
  - close to target platform
    \( C_{0,1} \) is easy for Java, \( C_* \) comes at a cost — other way round for RDB

- **Implementation/B**
  - complete, executable

- **Documentation**
  - Right level of abstraction: “if you’ve only one diagram to spend, illustrate the concepts, the architecture, the difficult part”
  - The more detailed the documentation, the higher the probability for regression
    “outdated/wrong documentation is worse than none”

---

**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
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
**General Diagramming Guidelines [Ambler, 2005]**

- **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

---

**Class Diagram Guidelines [Ambler, 2005]**

- **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
5.2 Class Style Guidelines

- 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 (or have good defaults)
- 118. Avoid Multiplicity “*” (explicitly use 0..1 or 1..*)
- 119. Replace Relationship Lines with Attribute Types (to have four lines)
Class Diagram Guidelines [Ambler, 2005]

- **5.4 Associations**
  - 127. Indicate Role Names When Multiple Associations Between Two Classes Exist
  - 129. Make AssociationsBidirectional Only When Collaboration Occurs in Both Directions
  - **131. Avoid Indicating Non-Navigability** (it depends, of the direction)
  - 133. Question Multiplicities Involving Minimums (and Maximums)

- **5.6 Aggregation and Composition**
  - → exercises

[...] But trust me on the sunscreen.

Baz Luhrmann/Mary Schmich
**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>
In many domains, there are canonical architectures — and adept readers try to see/find/match this!

For games:

Modelling Structure: 2D-Tron

- arcade
- platform open
- 2D
- min. 2, AI open
- controller open
- only game, no menus

2D-Tron

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
- ...

Notify update?

Main

External inputs

Game Logic

(update) notify

(Physics) Engine

Output

notify update?

Player
- colour
- score
- direction
- speed

Gameplay

update notify

Renderer

update notify

OpenGL?

aalib?

AI?

Segment
- x0, y0
- x1, y1
- colour

AI?

Tron

Control

Joystick?

Keyboard?

Player

Gameplay

Segment

Engine

Conventions:
- default $\xi$ is 1
**Modelling Behaviour**

---

**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* \( \sigma_0, \sigma_1, \cdots \in \Sigma^\omega \)
  of system states.
What Can Be Purposes of Behavioural Models?

(We will discuss this in more detail in Lecture 22.)

Example: Pre-Image 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 vs. Reflective Descriptions

[Harel, 1997] proposes to distinguish constructive and reflective descriptions:

- “A language is **constructive** if it contributes to the dynamic semantics of the model. That is, its constructs contain information needed in executing the model or in translating it into executable code.”
  A constructive description tells **how** things are computed (which can then be desired or undesired).

- “Other languages are **reflective** or **assertive**, and can be used by the system modeler to capture parts of the thinking that go into building the model – behavior included –, to derive and present views of the model, statically or during execution, or to set constraints on behavior in preparation for verification.”
  A reflective description tells **what** shall or shall not be computed.

Note: No sharp boundaries!
Constructive 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 machine:

```
E[n ≠ ∅]/x := x + 1; n! F
F/x := 0
/n := ∅
```

Course Map
References


