

Software Design, Modelling and Analysis in UML

Lecture 10: Constructive Behaviour, State Machines Overview

2013-12-02

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

– 10 – 2013-12-02 – main –

Contents & Goals

Last Lecture:

- (Mostly) completed discussion of modelling **structure**.

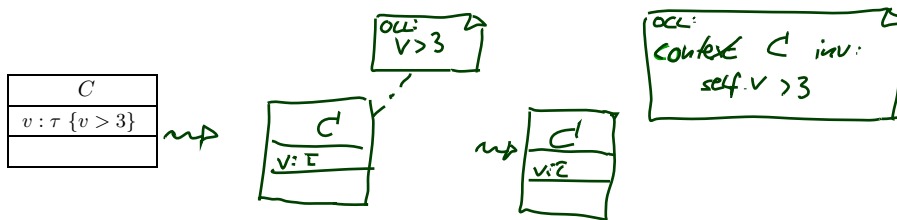
This Lecture:

- **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)

– 10 – 2013-12-02 – Prelim –

OCL Constraints in (Class) Diagrams

Invariant in Class Diagram Example

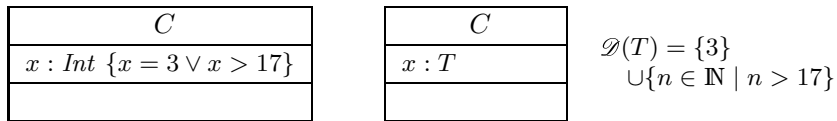


If \mathcal{CD} consists of only CD with the single class C , then

- $\text{Inv}(\mathcal{CD}) = \text{Inv}(CD) = \{ \text{context } C \text{ inv: } v > 3 \}$

Constraints vs. Types

Find the 10 differences:



- $x = 4$ 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 $\mathcal{D}(T)$ (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.

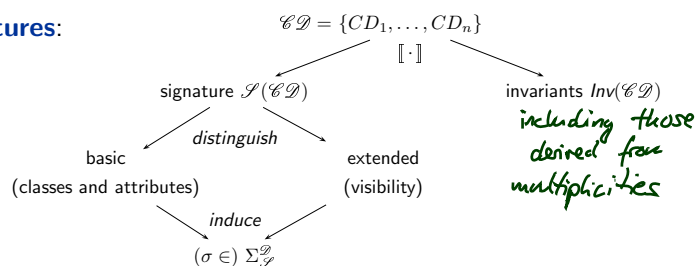
We say, the **semantics** of \mathcal{CD} is the signature it induces and the set of OCL constraints occurring in \mathcal{CD} , denoted

$$\llbracket \mathcal{CD} \rrbracket := \langle \mathcal{S}(\mathcal{CD}), \text{Inv}(\mathcal{CD}) \rangle.$$

Given a structure \mathcal{D} of \mathcal{S} (and thus of \mathcal{CD}), the class diagrams **describe** the system states $\Sigma_{\mathcal{D}}$. Of those, **some** satisfy $\text{Inv}(\mathcal{CD})$ and some don't.

We call a system state $\sigma \in \Sigma_{\mathcal{D}}$ **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.

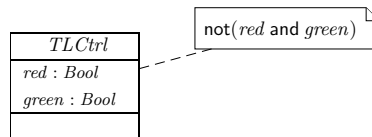
A set of class diagrams $\mathcal{C}\mathcal{D}$ with invariants $Inv(\mathcal{C}\mathcal{D})$ describes the **structure** of system states.

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



7/96

Addendum: Semantics of OCL Boolean Operations

8/96

- semantics of operator is monotone
(\perp propagates through, once a sub-expression evaluates to \perp , the whole expression does)
- $I(+)(x,y) = \begin{cases} x+y, & \text{if } x \neq \perp \text{ and } y \neq \perp \\ \perp, & \text{otherwise} \end{cases}$
de...
- if $I(\text{or})(p,q) = \begin{cases} \text{true}, & \text{if } p=\text{true} \text{ or } q=\text{true} \text{ and } p \neq \perp \text{ and } q \neq \perp \\ \perp, & \text{otherwise} \end{cases}$
then
not $\text{oclUndefined}(\text{self}.n)$ imply $\text{self}.n.x > 0$
 $\Leftrightarrow \text{oclUndefined}(\text{self}.n) \text{ or } \text{self}.n.x > 0$
would not do what we want

Correct Semantics of OCL Boolean Operations

Table A.2 - Semantics of boolean operations

b_1	b_2	$b_1 \text{ and } b_2$	$b_1 \text{ or } b_2$	$b_1 \text{ xor } b_2$	$b_1 \text{ implies } b_2$	not b_1
false	false	false	false	false	true	true
false	true	false	true	true	true	true
true	false	false	true	true	false	false
true	true	true	true	false	true	false
false	\perp	false	\perp	\perp	true	true
true	\perp	\perp	true	\perp	\perp	false

188

Object Constraint Language, v2.0

Table A.2 - Semantics of boolean operations

\perp	false	false	\perp	\perp	\perp	\perp
\perp	true	\perp	true	\perp	true	\perp
\perp	\perp	\perp	\perp	\perp	\perp	\perp

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 \mathcal{D} and asked to conduct task T .

- Can you do T with \mathcal{D} ?
(semantics sufficiently clear? all necessary information available? ...)
- Does doing T with \mathcal{D} cost you more nerves/time/money/... than it should?”
(syntactical well-formedness? readability? intention of deviations from standard syntax clear? reasonable selection of information? layout? ...)

In other words:

- the things **most relevant** for T , do they **stand out** in \mathcal{D} ? YES
- the things **less relevant** for T , do they **disturb** in \mathcal{D} ? NO

*answer should be
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”

13/96

– 10 – 2013-12-02 – Selements –

General Diagramming Guidelines [Amber, 2005]

(Note: “Exceptions prove the rule.”)



2.1 Readability



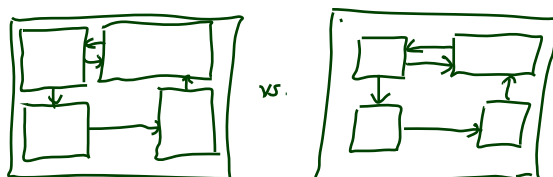
- 1.–3. Support Readability of Lines

- 4. Apply Consistently Sized Symbols



- 9. Minimize the Number of Bubbles/Things

- 10. Include White-Space in Diagrams



14/96

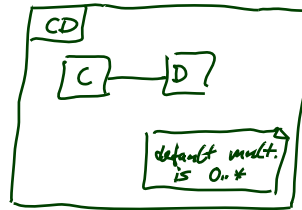
– 10 – 2013-12-02 – Selements –

General Diagramming Guidelines [Amber, 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



General Diagramming Guidelines [Amber, 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

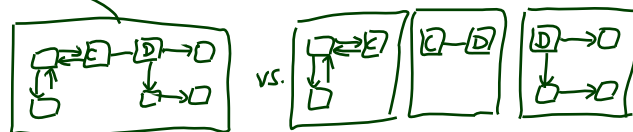
depends on audience



->*	0/8
-x*	5/3
->*	10/0
->*	9/1
-Ford*	10/0

"I did not know in Sept. 13"

"I did know"



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]**

eg get/set methods

Class Diagram Guidelines [Ambler, 2005]

5.2 Class Style Guidelines

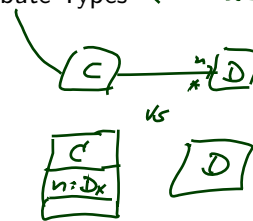
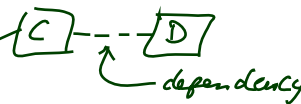
- 103. Never Show Classes with Just Two Compartments
- 104. Label Uncommon Class Compartments
- 105. Include an Ellipsis (...) at the End of an Incomplete List
- 107. List Operations/Attributes in Order of Decreasing Visibility



Class Diagram Guidelines [Ambler, 2005]



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..* or 1..*)
- 119. Replace Relationship Lines with Attribute Types (to have fewer lines)



Class Diagram Guidelines [Ambler, 2005]

- **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

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

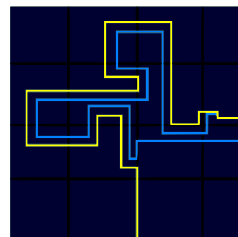
Baz Luhrmann/Mary Schmich

Example: Modelling Games

Task: Game Development

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

Degrees of freedom:	Exemplary choice: 2D-Tron
<ul style="list-style-type: none">• simulation vs. arcade	arcade
<ul style="list-style-type: none">• platform (SDK or not, open or proprietary, hardware capabilities...)	open
<ul style="list-style-type: none">• graphics (3D, 2D, ...)	2D
<ul style="list-style-type: none">• number of players, AI	min. 2, AI open
<ul style="list-style-type: none">• controller	open (later determined by platform)
<ul style="list-style-type: none">• game experience	minimal: main menu and game

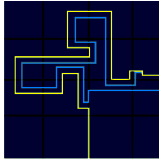


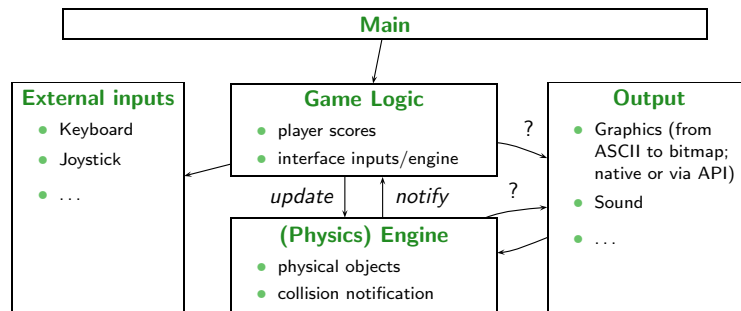
Modelling Structure: 2D-Tron

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

2D-Tron

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

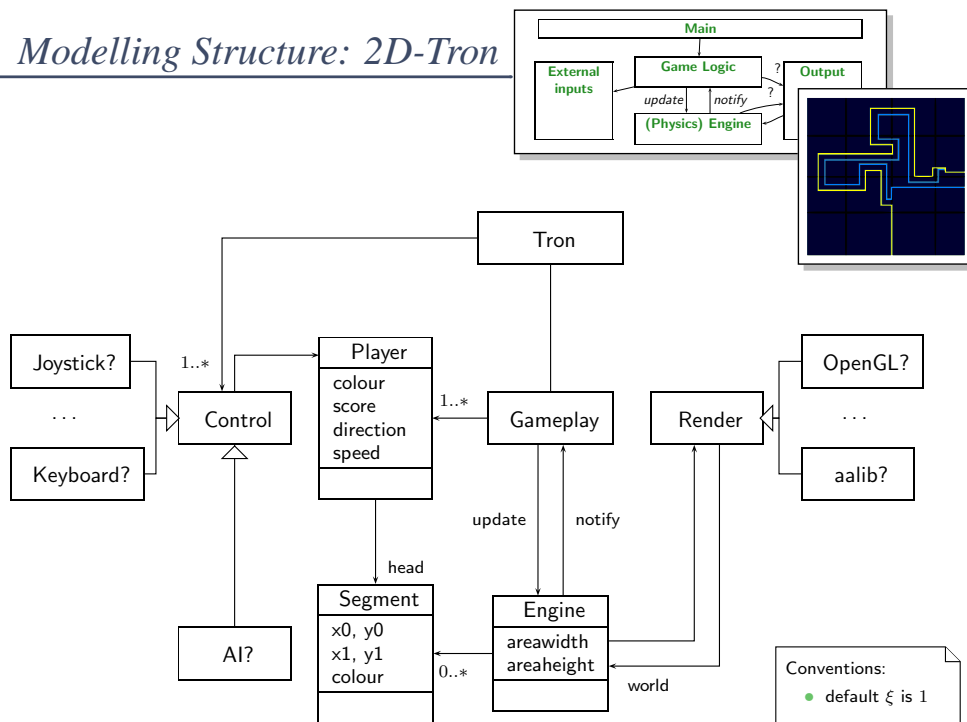




- 10 - 2013.12.02 - Stron -

23/96

Modelling Structure: 2D-Tron



- 10 - 2013.12.02 - Stron -

24/96

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**

$$\sigma_0, \sigma_1, \dots \in \Sigma^\omega$$

of system states.

not real-time,
just counting
steps here

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..

28/96

– 10 – 2013-12-02 – Sbehav –

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!

29/96

– 10 – 2013-12-02 – Sbehav –

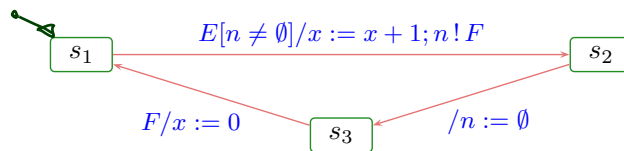
Constructive UML

UML provides two visual formalisms for constructive description of behaviours:

- Activity Diagrams
- State-Machine Diagrams

We (exemplary) focus on State-Machines because

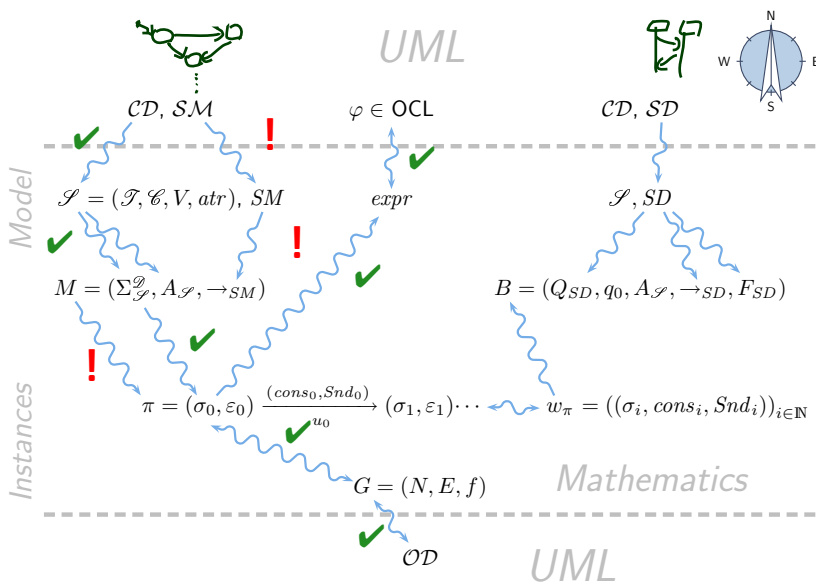
- somehow “practice proven” (in different flavours),
- prevalent in embedded systems community,
- indicated useful by [Dobing and Parsons, 2006] survey, and
- Activity Diagram’s intuition changed (between UML 1.x and 2.x) from transition-system-like to petri-net-like...
- Example state machine:



- 10 - 2013-12-02 - Sbrhahav -

30/96

Course Map



- 10 - 2013-12-02 - Sbrhahav -

31/96

References

References

- [Ambler, 2005] Ambler, S. W. (2005). *The Elements of UML 2.0 Style*. Cambridge University Press.
- [Crane and Dingel, 2007] Crane, M. L. and Dingel, J. (2007). UML vs. classical vs. rhapsody statecharts: not all models are created equal. *Software and Systems Modeling*, 6(4):415–435.
- [Dobing and Parsons, 2006] Dobing, B. and Parsons, J. (2006). How UML is used. *Communications of the ACM*, 49(5):109–114.
- [Harel, 1987] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.
- [Harel, 1997] Harel, D. (1997). Some thoughts on statecharts, 13 years later. In Grumberg, O., editor, *CAV*, volume 1254 of *LNCS*, pages 226–231. Springer-Verlag.
- [Harel and Gery, 1997] Harel, D. and Gery, E. (1997). Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42.
- [Harel et al., 1990] Harel, D., Lachover, H., et al. (1990). Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414.
- [OMG, 2007a] OMG (2007a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-04.