Contents & Goals

Last Lecture:
- 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:
  - Purposes of Behavioural Models
  - Constructive vs. Reflective
  - UML Core State Machines (first half)
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**

\[ \sigma_0, \sigma_1, \ldots \in \Sigma^\omega \]

of system states.
What Can Be Purposes of Behavioural Models?

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

Example: Pre-Image

(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.
  
  “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.
  
  “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.
  
  “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:

```
\[
E[n \neq \emptyset]/x := x + 1; n!F
\]

\[
F/x := 0
\]

\[
/n := \emptyset
\]
Course Map

UML

Model

Mathematics

Instances

UML State Machines: Overview

\[ G = (N, E, f) \]

\[ \pi = (\sigma_0, e_0) \rightarrow (\sigma_1, e_1) \rightarrow \cdots \]

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

\[ \mathcal{F} = (\mathcal{S}, \mathcal{E}, V, \text{attr}) \]

\[ \mathcal{M} = (\Sigma_{\mathcal{F}}, A_{\varphi}, \rightarrow_{\text{SM}}) \]

\[ \mathcal{B} = (Q_{\text{SD}}, q_0, A_{\varphi}, \rightarrow_{\text{SD}}, F_{\text{SD}}) \]

\[ \pi = (\sigma_0, e_0) \rightarrow (\sigma_1, e_1) \rightarrow \cdots \]

\[ w_e = ((\sigma_i, \text{cons}_i, \text{Snd}_i))_{i \in \mathbb{N}} \]

\[ \pi = (\sigma_0, e_0) \rightarrow (\sigma_1, e_1) \rightarrow \cdots \]

\[ M = (\Sigma_{\mathcal{F}}, A_{\varphi}, \rightarrow_{\text{SM}}) \]

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

\[ \mathcal{F} = (\mathcal{S}, \mathcal{E}, V, \text{attr}) \]

\[ \mathcal{M} = (\Sigma_{\mathcal{F}}, A_{\varphi}, \rightarrow_{\text{SM}}) \]

\[ \mathcal{B} = (Q_{\text{SD}}, q_0, A_{\varphi}, \rightarrow_{\text{SD}}, F_{\text{SD}}) \]

\[ \pi = (\sigma_0, e_0) \rightarrow (\sigma_1, e_1) \rightarrow \cdots \]

\[ w_e = ((\sigma_i, \text{cons}_i, \text{Snd}_i))_{i \in \mathbb{N}} \]

\[ \pi = (\sigma_0, e_0) \rightarrow (\sigma_1, e_1) \rightarrow \cdots \]

\[ M = (\Sigma_{\mathcal{F}}, A_{\varphi}, \rightarrow_{\text{SM}}) \]

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

\[ \mathcal{F} = (\mathcal{S}, \mathcal{E}, V, \text{attr}) \]

\[ \mathcal{M} = (\Sigma_{\mathcal{F}}, A_{\varphi}, \rightarrow_{\text{SM}}) \]

\[ \mathcal{B} = (Q_{\text{SD}}, q_0, A_{\varphi}, \rightarrow_{\text{SD}}, F_{\text{SD}}) \]

\[ \pi = (\sigma_0, e_0) \rightarrow (\sigma_1, e_1) \rightarrow \cdots \]

\[ w_e = ((\sigma_i, \text{cons}_i, \text{Snd}_i))_{i \in \mathbb{N}} \]
**UML State Machines**

\[
E[n \neq 0]/x := x + 1; n ! F
\]

\[
F/x := 0
\]

Brief History:
- Rooted in Moore/Mealy machines, Transition Systems
- [Harel, 1987]: Statecharts as a concise notation, introduces in particular hierarchical states.
- 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

(i) What do we (have to) cover? UML State Machine Diagrams Syntax.
(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.
UML State Machines: Syntax

UML State-Machines: What do we have to cover?


Ein Zustand löst von sich aus bestimmte Ereignisse aus:
- **entry** beim Betreten;
- **do** während des Aufenthaltes;
- **completion** beim Erreichen des Endzustandes einer Unter-Zustandsmaschine;
- **exit** beim Verlassen.

Diese und andere Ereignisse können als Auslöser für Aktivitäten herangezogen werden.

Ein komplexer Zustand mit einer Region.

Ein Eintrittspunkt definiert, dass ein komplexer Zustand an einer anderen Stelle betreten wird, als durch den Anfangszustand definiert ist.

Ein Austrittspunkt erlaubt es, von einem inneren inneren Zustand, den der Übergang zu verlassen.
**UML State-Machines: What do we have to cover?**

**Proven approach:**

Start out simple, consider the essence, namely

- basic/leaf states
- transitions,

then extend to cover the complicated rest.

**Signature With Signals**

**Definition.** A tuple

\[ \mathcal{S} = (\mathcal{F}, \mathcal{C}, V, atr, \epsilon) \]

is called signature (with signals) if and only if

\[ (\mathcal{F}, \mathcal{C}, V, atr) \]

is a signature (as before).

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

**Definition.**
A core state machine over signature \( \mathcal{S} = (T, C, V, \text{atr}) \) is a tuple \( \mathcal{M} = (S, s_0, \rightarrow) \) where
- \( S \) is a non-empty, finite set of **basic states**,
- \( s_0 \in S \) is an **initial state**,
- and
  \[ \rightarrow \subseteq S \times (E \cup \{\bot\}) \times \text{Expr}_\mathcal{S} \times \text{Act}_\mathcal{S} \times S \]

is a labelled transition relation.

We assume a set \( \text{Expr}_\mathcal{S} \) of boolean expressions over \( \mathcal{S} \) (for instance OCL, may be something else) and a set \( \text{Act}_\mathcal{S} \) of **actions**.

---

**From UML to Core State Machines: By Example**

UML state machine diagram \( SM \):

maps to

\[
\mathcal{M}(SM) = (\{s_1, s_2\}, s_1, (s_1, \text{event}, \text{guard}, \text{action}, s_2))
\]
Annotations and Defaults in the Standard

Reconsider the syntax of transition annotations:

\[
\text{annot ::= } [\langle \text{event}\rangle[\cdot \langle \text{event}\rangle]^*][\cdot[\langle \text{guard}\rangle\cdot]][\cdot[\langle \text{action}\rangle]]
\]

and let’s play a bit with the defaults:

\[
\begin{align*}
\text{true} & \quad \rightarrow \quad \text{true, skip} \\
\text{false} & \quad \rightarrow \quad \text{false, skip} \\
E & \quad \rightarrow \quad E, \text{true, skip} \\
E & \quad \rightarrow \quad E, \text{true, act} \\
E & \quad \rightarrow \quad E, \text{true, act} \\
E & \quad \rightarrow \quad E, \text{true, skip}
\end{align*}
\]

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 \(E\).
- \(E(v : \tau)\) — similar, but \(v\) is a local variable, scope is the transition

State-Machines belong to Classes, Are Executed by Objects

- In the following, we assume that a UML models consists of a set \(\mathcal{CD}\) of class diagrams and a set \(\mathcal{MM}\) of state chart diagrams (each comprising one state machine \(SM\)).
- Furthermore, we assume that each state machine \(SM \in \mathcal{MM}\) is associated with a class \(C_{SM} \in \mathcal{C}(\mathcal{I})\).
- For simplicity, we even assume a bijection, i.e. we assume that each class \(C \in \mathcal{C}(\mathcal{I})\) has a state machine \(SM_C\) and that its class \(C_{SM_C}\) is \(C\).

If not explicitly given, then this one:

\[
SM_0 := ([s_0], s_0, true, skip, s_0).
\]

We’ll see later that, semantically, this choice does no harm.

- Intuition 1: \(SM_C\) describes the behaviour of the instances of class \(C\).
- Intuition 2: Each instance of class \(C\) executes \(SM_C\) but with a local \(\text{program\_context}\).

Note: we don’t consider multiple state machines per class. Because later (when we have AND-states) we’ll see that this case can be viewed as a single state machine with as many AND-states.
References
References


