Software Design, Modelling and Analysis in UML

Lecture 10: Constructive Behaviour, State Machines Overview

2012-11-28

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

Albert-Ludwigs-Universität Freiburg, Germany

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**

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

not real-time,
just counting steps here

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.)

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!

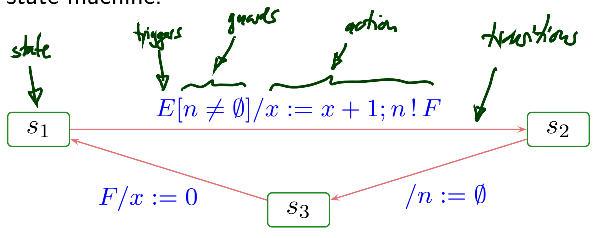
Constructive UML

UML provides two visual formalisms for constructive description of behaviours:

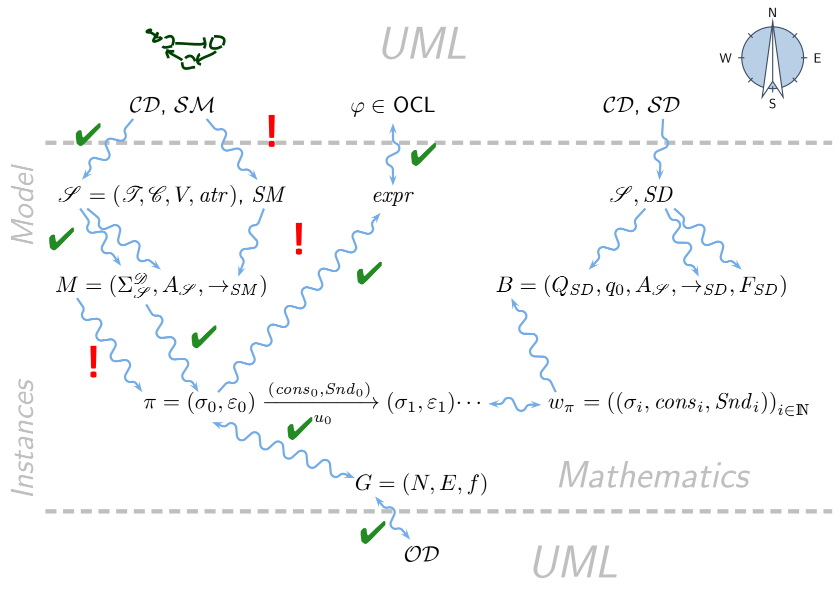
- **Activity Diagrams**
- **State-Machine Diagrams**

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:



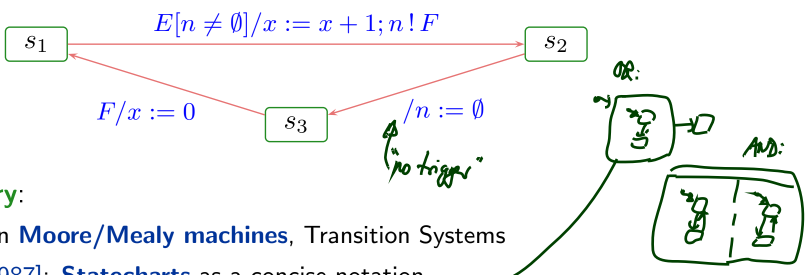
Course Map



UML
$\mathcal{CD}, \mathcal{SM}$
$\varphi \in$ OCL
$\mathcal{CD}, \mathcal{SD}$
Model
$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr),\ SM$
$expr$
$\mathscr{S}, SD$
$M = (\Sigma_{\mathscr{S}}^{\mathscr{D}}, A_{\mathscr{S}}, \to_{SM})$
$B = (Q_{SD}, q_0, A_{\mathscr{S}}, \to_{SD}, F_{SD})$
Instances
$\pi = (\sigma_0, \varepsilon_0) \xrightarrow[u_0]{(cons_0, Snd_0)} (\sigma_1, \varepsilon_1) \cdots$
$w_\pi = ((\sigma_i, cons_i, Snd_i))_{i \in \mathbb{N}}$
$G = (N, E, f)$
Mathematics
$\mathcal{OD}$
UML
N
W
E
S

UML State Machines: Overview

UML State Machines



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** (in State Chart Diagrams) (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**.

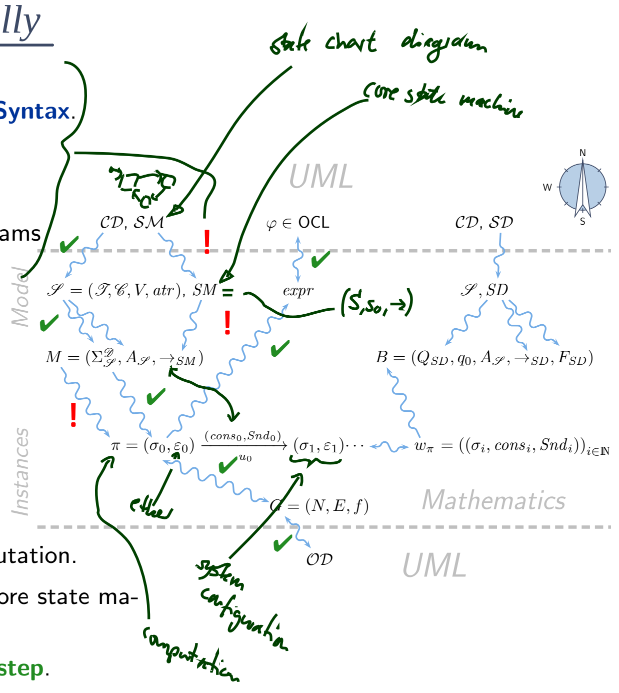
(viii) Def.: **Transformer**.

(ix) Def.: **Transition system**, computation.

(x) Transition relation induced by core state machine.

(xi) Def.: **step**, **run-to-completion step**.

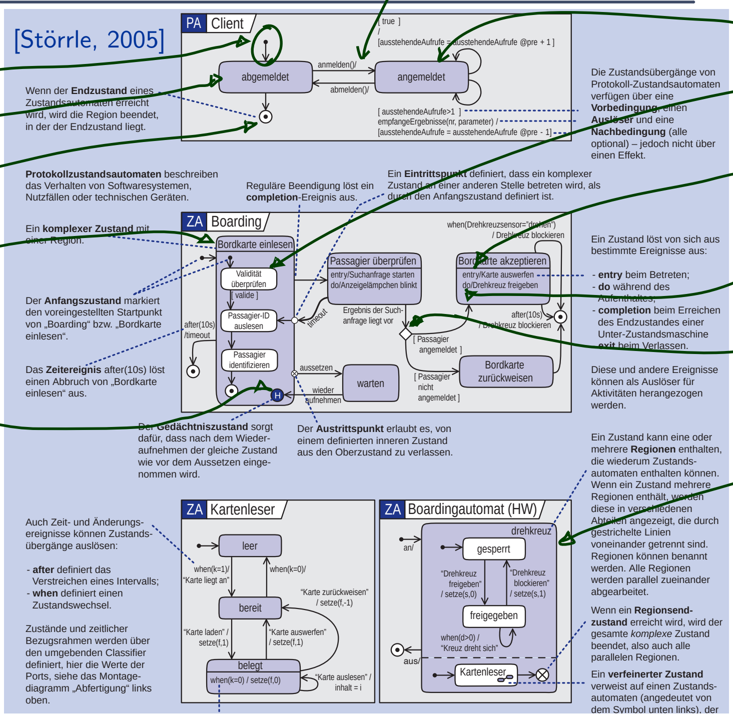
(xii) Later: Hierarchical state machines.



state chart diagram
core state machine
UML
$\mathcal{CD}, \mathcal{SM}$
$\varphi \in$ OCL
$\mathcal{CD}, \mathcal{SD}$
N
W
S
Model
$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr),\ SM$ =
$expr$
$(S, s_0, \rightarrow)$
$\mathscr{S}, SD$
$M = (\Sigma_{\mathscr{S}}^{\mathscr{D}}, A_{\mathscr{S}}, \rightarrow_{SM})$
$B = (Q_{SD}, q_0, A_{\mathscr{S}}, \rightarrow_{SD}, F_{SD})$
Instances
$\pi = (\sigma_0, \varepsilon_0) \xrightarrow[u_0]{(cons_0, Snd_0)} (\sigma_1, \varepsilon_1) \cdots$
$w_\pi = ((\sigma_i, cons_i, Snd_i))_{i \in \mathbb{N}}$
ether
$G = (N, E, f)$
Mathematics
$\mathcal{OD}$
UML
system configuration
computation

UML State Machines: Syntax

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



Wenn der **Endzustand** eines Zustandsautomaten erreicht wird, wird die Region beendet, in der der Endzustand liegt.

Die Zustandsübergänge von Protokoll-Zustandsautomaten verfügen über eine **Vorbedingung**, einen **Auslöser** und eine **Nachbedingung** (alle optional) – jedoch nicht über einen Effekt.

Protokollzustandsautomaten beschreiben das Verhalten von Softwaresystemen, Nutzfällen oder technischen Geräten.

Reguläre Beendigung löst ein **completion**-Ereignis aus.

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

Ein **komplexer Zustand** mit einer Region.

Der **Anfangszustand** markiert den voreingestellten Startpunkt von „Boarding" bzw. „Bordkarte einlesen".

Das **Zeitereignis** after(10s) löst einen Abbruch von „Bordkarte einlesen" aus.

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.

Der **Gedächtniszustand** sorgt dafür, dass nach dem Wiederaufnehmen der gleiche Zustand wie vor dem Aussetzen eingenommen wird.

Der **Austrittspunkt** erlaubt es, von einem definierten inneren Zustand aus den Oberzustand zu verlassen.

Ein Zustand kann eine oder mehrere **Regionen** enthalten, die wiederum Zustandsautomaten enthalten können. Wenn ein Zustand mehrere Regionen enthält, werden diese in verschiedenen Abteilen angezeigt, die durch gestrichelte Linien voneinander getrennt sind. Regionen können benannt werden. Alle Regionen werden parallel zueinander abgearbeitet.

Auch Zeit- und Änderungsereignisse können Zustandsübergänge auslösen:

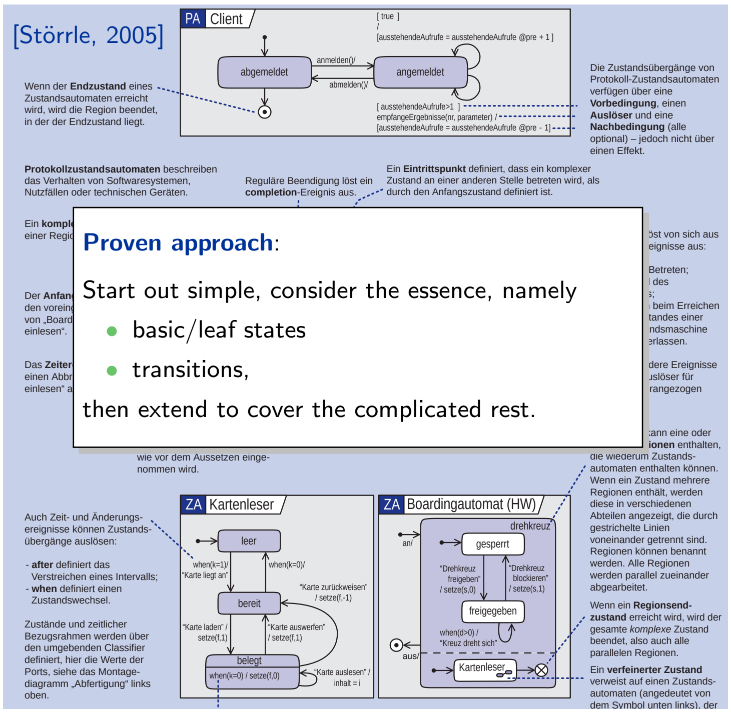
- **after** definiert das Verstreichen eines Intervalls;
- **when** definiert einen Zustandswechsel.

Zustände und zeitlicher Bezugsrahmen werden über den umgebenden Classifier definiert, hier die Werte der Ports, siehe das Montagediagramm „Abfertigung" links oben.

Wenn ein **Regionsendzustand** erreicht wird, wird der gesamte *komplexe* Zustand beendet, also auch alle parallelen Regionen.

Ein **verfeinerter Zustand** verweist auf einen Zustandsautomaten (angedeutet von dem Symbol unten links), der

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


[Störrle, 2005]
PA Client
abgemeldet
anmelden()/
abmelden()/
angemeldet
[true] / [ausstehendeAufrufe = ausstehendeAufrufe @pre + 1]
[ausstehendeAufrufe>1] empfangeErgebnisse(nr, parameter) / [ausstehendeAufrufe = ausstehendeAufrufe @pre - 1]
ZA Kartenleser
leer
when(k=1)/ "Karte liegt an"
when(k=0)/
bereit
"Karte zurückweisen" / setze(f,-1)
"Karte laden" / setze(f,1)
"Karte auswerfen" / setze(f,1)
belegt
when(k=0) / setze(f,0)
"Karte auslesen" / inhalt = i
ZA Boardingautomat (HW)
drehkreuz
an/
gesperrt
"Drehkreuz freigeben" / setze(s,0)
"Drehkreuz blockieren" / setze(s,1)
freigegeben
when(d>0) / "Kreuz dreht sich"
aus/
Kartenleser

Wenn der **Endzustand** eines Zustandsautomaten erreicht wird, wird die Region beendet, in der der Endzustand liegt.

Die Zustandsübergänge von Protokoll-Zustandsautomaten verfügen über eine **Vorbedingung**, einen **Auslöser** und eine **Nachbedingung** (alle optional) – jedoch nicht über einen Effekt.

Protokollzustandsautomaten beschreiben das Verhalten von Softwaresystemen, Nutzfällen oder technischen Geräten.

Reguläre Beendigung löst ein **completion**-Ereignis aus.

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

Ein **komple** einer Regio

Der **Anfan** den voreing von „Board einlesen".

Das **Zeiter** einen Abbr einlesen" a

öst von sich aus eignisse aus:

Betreten; des s; beim Erreichen tandes einer ndsmaschine erlassen.

dere Ereignisse uslöser für rangezogen

kann eine oder **ionen** enthalten, die wiederum Zustandsautomaten enthalten können. Wenn ein Zustand mehrere Regionen enthält, werden diese in verschiedenen Abteilen angezeigt, die durch gestrichelte Linien voneinander getrennt sind. Regionen können benannt werden. Alle Regionen werden parallel zueinander abgearbeitet.

wie vor dem Aussetzen eingenommen wird.

Auch Zeit- und Änderungsereignisse können Zustandsübergänge auslösen:

- **after** definiert das Verstreichen eines Intervalls;
- **when** definiert einen Zustandswechsel.

Zustände und zeitlicher Bezugsrahmen werden über den umgebenden Classifier definiert, hier die Werte der Ports, siehe das Montagediagramm „Abfertigung" links oben.

Wenn ein **Regionsendzustand** erreicht wird, wird der gesamte *komplexe* Zustand beendet, also auch alle parallelen Regionen.

Ein **verfeinerter Zustand** verweist auf einen Zustandsautomaten (angedeutet von dem Symbol unten links), der

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

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr\cancel{)}, \mathscr{E}) \quad \overset{\subseteq \mathscr{C}}{\mathscr{E}} \text{ a set of signals,}$$

is called signature (with signals) if and only if

$$(\mathscr{T}, \mathscr{C}\cancel{\cup \mathscr{E}}, V, atr)$$

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 $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$ is a tuple

$$\mathcal{S}M = (S, s_0, \to)$$

where

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

set of signals   source state   destination state

disjoint union, $\sim \& \mathscr{E}$

$$\to \; \subseteq S \times \underbrace{(\mathscr{E} \,\dot{\cup}\, \{\_\})}_{\text{trigger}} \times \underbrace{Expr_{\mathscr{S}}}_{\text{guard}} \times \underbrace{Act_{\mathscr{S}}}_{\text{action}} \times S$$

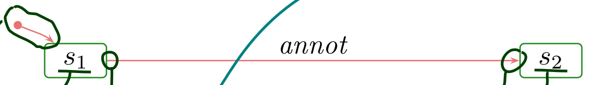
is a labelled transition relation.

We assume a set $Expr_{\mathscr{S}}$ of boolean expressions over $\mathscr{S}$ (for instance OCL, may be something else) and a set $Act_{\mathscr{S}}$ of **actions**.

From UML to Core State Machines: By Example

UML state machine diagram $\mathcal{SM}$:

not considered in lecture



guard   action

$annot ::= \big[\langle event\rangle[\text{ ‘,’ } \langle event\rangle]^*\big] \; [\text{ ‘[’ } \langle guard\rangle \text{ ‘]’ }] \; [\text{ ‘/’ } [\langle action\rangle]]$

trigger

with

- $event \in \mathscr{E}$,
- $guard \in Expr_{\mathscr{S}}$ (default: *true*, assumed to be in $Expr_{\mathscr{S}}$)
- $action \in Act_{\mathscr{S}}$ (default: `skip`, assumed to be in $Act_{\mathscr{S}}$)

maps to

$$\mathcal{S}M(\mathcal{SM}) = (\underbrace{\{s_1, s_2\}}_{S}, \underbrace{s_1}_{s_0}, \underbrace{(s_1, event, guard, action, s_2)}_{\to})$$

Annotations and Defaults in the Standard

Reconsider the syntax of transition annotations:

$$annot ::= \big[\langle event\rangle[\text{ ‘.’ } \langle event\rangle]^*\big] \; [\text{ ‘[’ } \langle guard\rangle \text{ ‘]’ }] \;\; [\text{ ‘/’}\big[\langle action\rangle\big]\big]$$

and let's play a bit with the defaults:

the empty annotation $\rightarrow$

$$\begin{aligned}
& & \rightsquigarrow \; & \_\,,\ true\,,\ skip \\
/ & & \rightsquigarrow \; & \_\,,\ true\,,\ skip \\
E\ / & & \rightsquigarrow \; & E,\ true\,,\ skip \\
/\ act & & \rightsquigarrow \; & \_\,,\ true\,,\ act \\
E\ /\ act & & \rightsquigarrow \; & E,\ true\,,\ act \\
E\,[gd]\ /\ act & & \rightsquigarrow \; & E,\ gd\,,\ act
\end{aligned}$$

$act \in Act_{\mathscr{S}}$

$gd \in Expr_{\mathscr{S}}$

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

$Msg(x)/\ldots$

$\rightsquigarrow Msg\,/\,x := params \to x;\ \ldots$

State-Machines belong to Classes, Are Executed by Objects

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

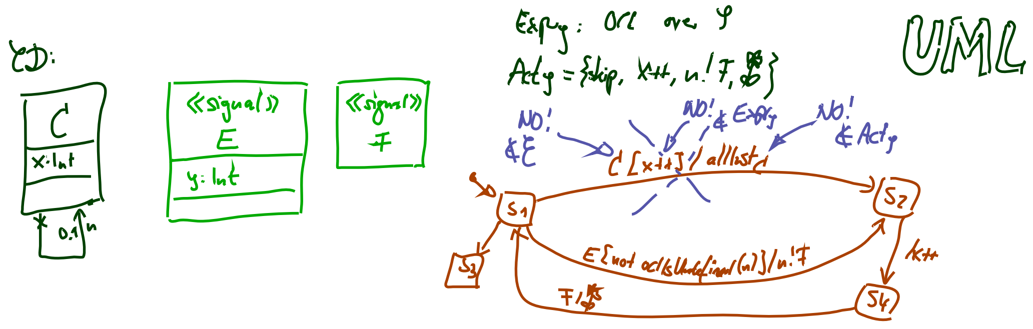
  If not explicitly given, then this one:

  maybe even better: $\emptyset$

  $$\mathcal{SM}_0 := (\{s_0\}, s_0, \{(s_0, \_, \mathit{true}, \texttt{skip}, s_0)\}).$$

  We'll see later that, semantically, this choice does no harm.
- **Intuition 1**: $\mathcal{SM}_C$ describes the behaviour of **the instances** of class C.
  Intuition 2: Each instance of class C executes $\mathcal{SM}_C$ but with a local "program counter".

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.



$\mathscr{S} = (\{Int\}, \{C, E, F\},$
$\{x : Int, n : C_{0,1}, y : Int\},$
$\{C \mapsto \{x, n\}, E \mapsto \{y\}, F \mapsto \emptyset\},$
$\underbrace{\{E, F\}}_{\mathscr{E}})$

$SM = (\{s_1, s_2, s_3, s_4\}, s_1,$
$\{(s_1, -, true, skip, s_3),$
$(s_1, E, not\ oclIsUndefined(n), n ! F, s_2),$
$(s_2, -, true, x{+}{+}, s_4),$
$(s_4, F, true, [illegible], s_1)\})$

MATH

References

References

[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.

[OMG, 2007b] OMG (2007b). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.