

Software Design, Modelling and Analysis in UML

Lecture 11: Core State Machines I

2014-12-04

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

- Associations (up to some rest)

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
 - What does this State Machine mean? What happens if I inject this event?
 - Can you please model the following behaviour.
 - What is: Signal, Event, Ether, Transformer, Step, RTC.
- **Content:**
 - Associations cont'd, back to main track
 - Core State Machines
 - UML State Machine syntax

Associations: The Rest

Recapitulation: Consider the following association:

$$\langle r : \langle role_1 : C_1, \mu_1, P_1, \xi_1, \nu_1, o_1 \rangle, \dots, \langle role_n : C_n, \mu_n, P_n, \xi_n, \nu_n, o_n \rangle \rangle$$

- **Association name** r and **role names/types** $role_i/C_i$ induce extended system states λ .
- **Multiplicity** μ is considered in OCL syntax.
- **Visibility** ξ /**Navigability** ν : well-typedness.

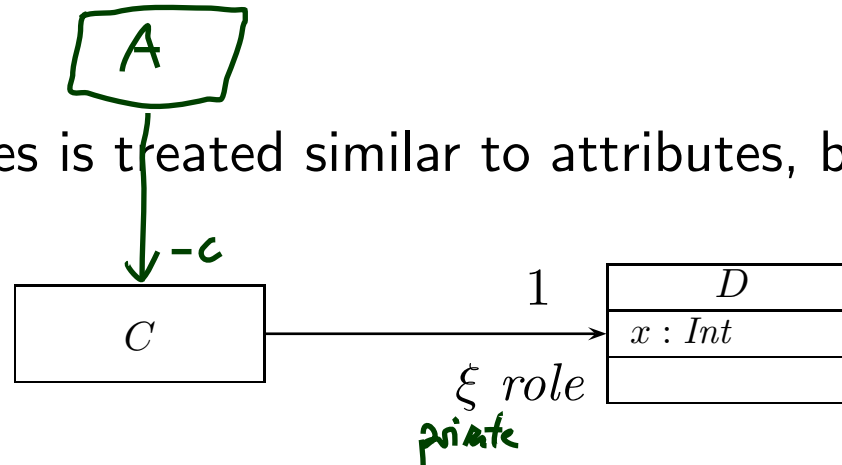
Now the rest:

- **Multiplicity** μ : we propose to view them as constraints.
- **Properties** P_i : even more typing.
- **Ownership** o : getting closer to pointers/references.
- **Diamonds**: exercise.

Visibility

Visibility of role-names is treated similar to attributes, by **typing rules**.

Question: given



$self.A.C$ OK
 $self.A.C.role$ NOT OK
 $self.C.C$ OK

is the following OCL expression well-typed or not (wrt. visibility):

context C inv : $self.role.x > 0$

Basically the same rule as before (similar for other multiplicities):

$role(w) : \tau_C \rightarrow \tau_D \quad \mu = 0..1 \text{ or } \mu = 1,$

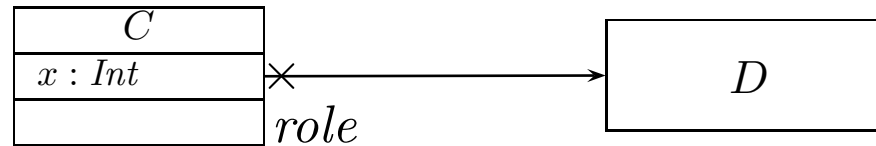
$role(expr_1(w)) : \tau_C \rightarrow \tau_D \quad \mu = 0..1 \text{ or } \mu = 1, expr_1(w) : \tau_C,$
 $w : \tau_{C_1}, \text{ and } C_1 = C \text{ or } \xi = +$

$\langle r : \dots \langle role : D, \mu, -, \xi, -, - \rangle, \dots \langle role' : C, -, -, -, -, - \rangle, \dots \rangle \in V$

Navigability

Navigability is similar to visibility: expressions over non-navigable association ends ($\nu = \times$) are **basically** type-correct, but **forbidden**.

Question: given



is the following OCL expression well-typed or not (wrt. navigability)?

context D inv : $self.role.x > 0$

The standard says: navigation is...

- '—': ...possible
- '>': ...efficient
- '×': ...not possible

So: In general, UML associations are different from pointers/references!

But: Pointers/references can faithfully be modelled by UML associations.

The Rest of the Rest

Recapitulation: Consider the following association:

$$\langle r : \langle role_1 : C_1, \mu_1, P_1, \xi_1, \nu_1, o_1 \rangle, \dots, \langle role_n : C_n, \mu_n, P_n, \xi_n, \nu_n, o_n \rangle \rangle$$

- **Association name** r and **role names/types** $role_i/C_i$ induce extended system states λ .
- **Multiplicity** μ is considered in OCL syntax.
- **Visibility** ξ /**Navigability** ν : well-typedness.

Now the rest:

- **Multiplicity** μ : we propose to view them as constraints.
- **Properties** P_i : even more typing.
- **Ownership** o : getting closer to pointers/references.
- **Diamonds**: exercise.

Multiplicities as Constraints

Recall: The multiplicity of an association end is a term of the form:

$$\mu ::= * \mid N \mid N..M \mid N..* \mid \mu, \mu \quad (N, M \in \mathbb{N})$$

Proposal: View multiplicities (except 0..1, 1) as additional invariants/constraints.

Recall: we can normalize each multiplicity μ to the form

$$\mu = \underbrace{N_1..N_2}, \dots, \underbrace{N_{2k-1}..N_{2k}}$$

where $N_i \leq N_{i+1}$ for $1 \leq i \leq 2k$, $N_1, \dots, N_{2k-1} \in \mathbb{N}$, $N_{2k} \in \mathbb{N} \cup \{*\}$.

Multiplicities as Constraints

$$\mu = N_1..N_2, \dots, N_{2k-1}..N_{2k}$$

where $N_i \leq N_{i+1}$ for $1 \leq i \leq 2k$, $N_1, \dots, N_{2k-1} \in \mathbb{N}$, $N_{2k} \in \mathbb{N} \cup \{*\}$.

Define $\mu_{\text{OCL}}^C(\text{role}) := \text{context } C \text{ inv} :$

$$(N_1 \leq \text{role} \rightarrow \text{size}() \leq N_2) \text{ or } \dots \text{ or } (N_{2k-1} \leq \text{role} \rightarrow \text{size}() \leq \underbrace{N_{2k}}_{\text{omit if } N_{2k} = *})$$

for each $\mu \neq 0..1$, $\mu \neq 1$,

$$\langle r : \dots, \langle \text{role} : D, \mu, -, -, -, - \rangle, \dots, \langle \text{role}' : C, -, -, -, - \rangle, \dots \rangle \in V \text{ or}$$

$$\langle r : \dots, \langle \text{role}' : C, -, -, -, - \rangle, \dots, \langle \text{role} : D, \mu, -, -, -, - \rangle, \dots \rangle \in V, \text{role} \neq \text{role}'.$$

For $\mu=0$: context C inv; ocl is Undefined (role)

And **define**

$$\mu_{\text{OCL}}^C(\text{role}) := \text{context } C \text{ inv} : \text{not}(\text{ocllsUndefined}(\text{role}))$$

for each $\mu = 1$.

Note: in n -ary associations with $n > 2$, there is redundancy.

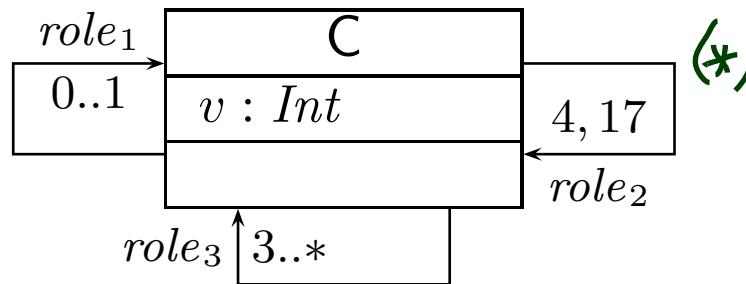
Multiplicities as Constraints Example

$\mu_{\text{OCL}}^C(\text{role}) = \text{context } C \text{ inv :}$

$(N_1 \leq \text{role} \rightarrow \text{size}() \leq N_2) \text{ or } \dots \text{ or } (N_{2k-1} \leq \text{role} \rightarrow \text{size}() \leq N_{2k})$

note:
0..1
is equivalent to
0..0, 1..1

CD :



$\text{Inv}(CD) =$

- $\{ \text{context } C \text{ inv : } 4 \leq \text{role}_2 \rightarrow \text{size}() \leq 4 \text{ or } 17 \leq \text{role}_2 \rightarrow \text{size}() \leq 17, (*) \}$
 $= \{ \dots \}$
- $\cup \{ \text{context } C \text{ inv : } 3 \leq \text{role}_3 \rightarrow \text{size}() \}$

Why Multiplicities as Constraints?

More precise, can't we just use **types**? (cf. Slide 26)

- $\mu = 0..1, \mu = 1$:
many programming language have direct correspondences (the first corresponds to type pointer, the second to type reference) — therefore treated specially.
- $\mu = *$:
could be represented by a set data-structure type without fixed bounds — no problem with our approach, we have $\mu_{\text{OCL}} = \text{true}$ anyway.
- $\mu = 0..3$: \exists
use array of size ~~3~~ — if model behaviour (or the implementation) adds 5th identity, we'll get a runtime error, and thereby see that the constraint is violated.
Principally acceptable, but: checks for array bounds everywhere...?
- $\mu = 5..7$:
could be represented by an array of size 7 — but: few programming languages/data structure libraries allow lower bounds for arrays (other than 0).
If we have 5 identities and the model behaviour removes one, this should be a violation of the constraints imposed by the **model**.
The implementation which does this removal is **wrong**. How do we see this...?

Multiplicities Never as Types...?

Well, if the **target platform** is known and fixed, **and** the target platform has, for instance,

- reference types,
- range-checked arrays with positions $0, \dots, N$,
- set types,

then we could simply **restrict** the syntax of multiplicities to

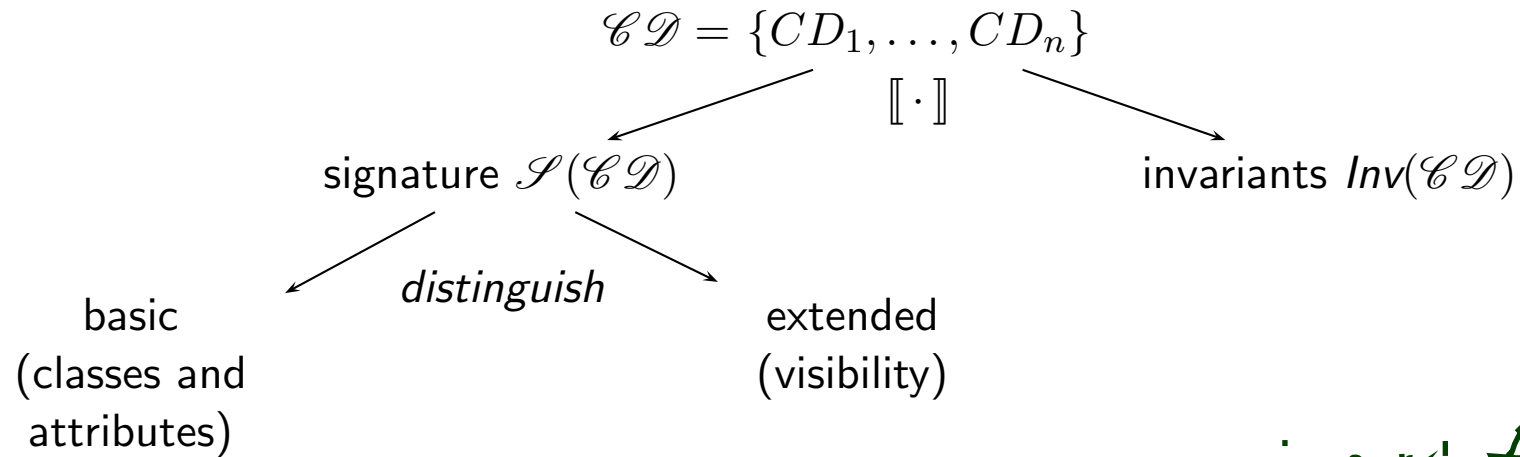
$$\mu ::= 1 \mid 0..N \mid *$$

and don't think about constraints
(but use the obvious 1-to-1 mapping to types)...

In general, **unfortunately**, we don't know.

Multiplicities as Constraints of Class Diagram

Recall/Later:



From now on: $Inv(\mathcal{CD}) = \{\text{constraints occurring in notes}\} \cup \{\mu_{OCL}^C(\text{role}) \mid$

in a minute

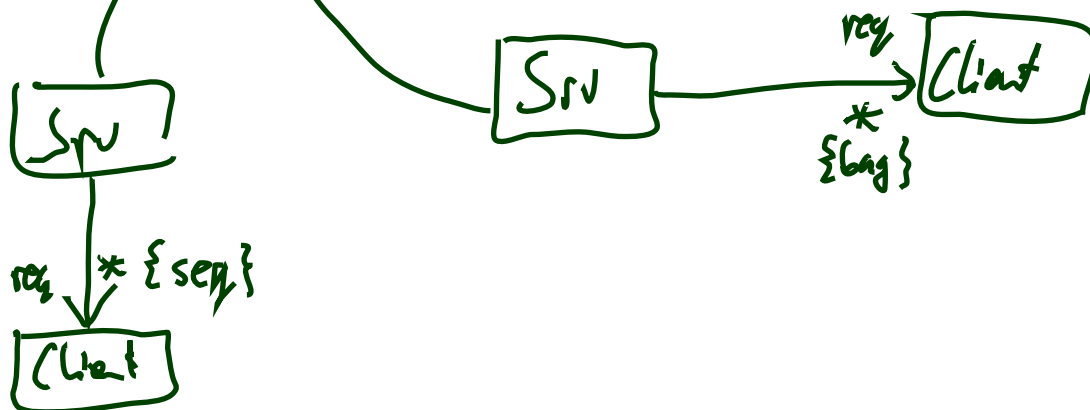
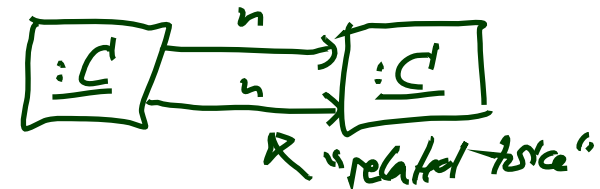
$\langle r : \dots, \langle \text{role} : D, \mu, -, -, -, - \rangle, \dots, \langle \text{role}' : C, -, -, -, -, - \rangle, \dots \rangle \in V$ or
 $\langle r : \dots, \langle \text{role}' : C, -, -, -, -, - \rangle, \dots, \langle \text{role} : D, \mu, -, -, -, - \rangle, \dots \rangle \in V,$
 $\text{role} \neq \text{role}', \mu \notin \{0..1\}$.

Properties

We don't want to cover association **properties** in detail, only some observations (assume binary associations):

Property	Intuition	Semantical Effect
unique	one object has at most one r -link to a single other object	current setting
bag	one object may have multiple r -links to a single other object	have $\lambda(r)$ yield multi-sets
ordered, sequence	an r -link is a sequence of object identities (possibly including duplicates)	have $\lambda(r)$ yield sequences

does not allow



Properties

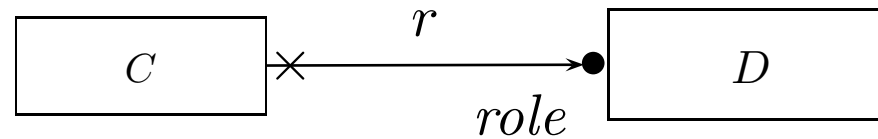
We don't want to cover association **properties** in detail, only some observations (assume binary associations):

Property	Intuition	Semantical Effect
unique	one object has at most one r -link to a single other object	current setting
bag	one object may have multiple r -links to a single other object	have $\lambda(r)$ yield multi-sets
ordered, sequence	an r -link is a sequence of object identities (possibly including duplicates)	have $\lambda(r)$ yield sequences

Property	OCL Typing of expression $role(expr)$
unique	$\tau_D \rightarrow Set(\tau_C)$
bag	$\tau_D \rightarrow Bag(\tau_C)$
ordered, sequence	$\tau_D \rightarrow Seq(\tau_C)$

For **subsets**, **redefines**, **union**, etc. see [OMG, 2007a, 127].

Ownership



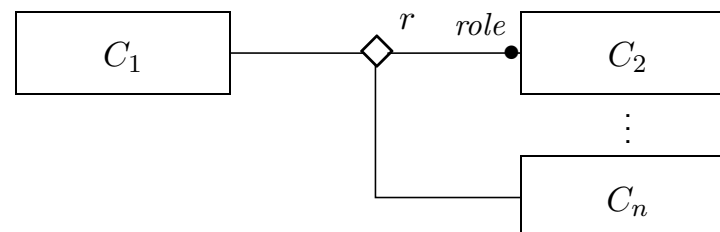
Intuitively it says:

Association r is **not a “thing on its own”** (i.e. provided by λ), but association end ‘ $role$ ’ is **owned** by C (!). (That is, it’s stored inside C object and provided by σ).

So: if multiplicity of $role$ is 0..1 or 1, then the picture above is very close to concepts of pointers/references.

Actually, ownership is seldom seen in UML diagrams. Again: if target platform is clear, one may well live without (cf. [OMG, 2007b, 42] for more details).

Not clear to me:



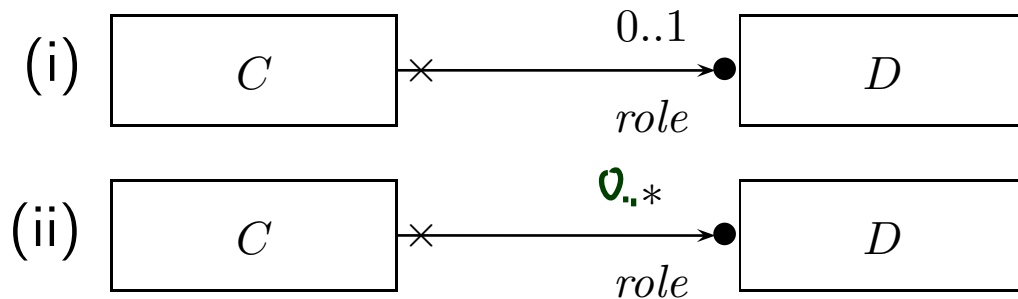
Back to the Main Track

Back to the main track:

Recall: on some earlier slides we said, the extension of the signature is **only** to study associations in “full beauty” ..
For the remainder of the course, we should look for something simpler...

Proposal:

- **from now on**, we only use associations of the form



(And we may omit the non-navigability and ownership symbols.)

- Form (i) introduces $role : C_{0,1}$, and form (ii) introduces $role : C_*$ in V .
- In both cases, $role \in atr(C)$.
- We drop λ and go back to our nice σ with $\sigma(u)(role) \subseteq \mathcal{D}(D)$.

OCL Constraints in (Class) Diagrams

Where Shall We Put OCL Constraints?

Two options:

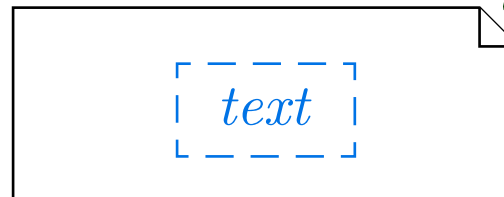
(0) *additional documents*

(i) Notes.

(ii) Particular dedicated places.

(i) **Notes:**

A UML **note** is a picture of the form



*Eseleohr
(dog's ear)*

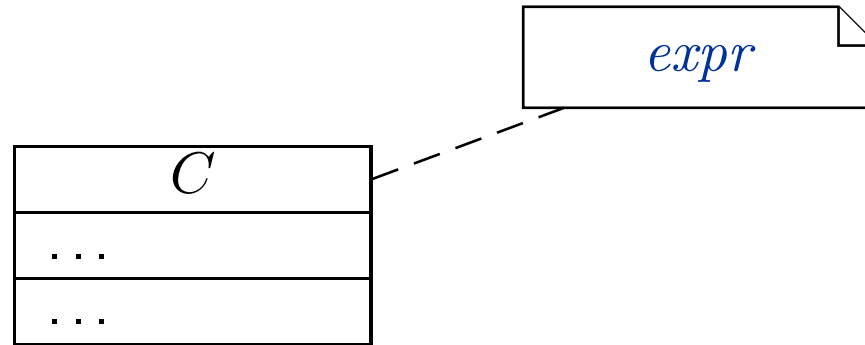


text can principally be **everything**, in particular **comments** and **constraints**.

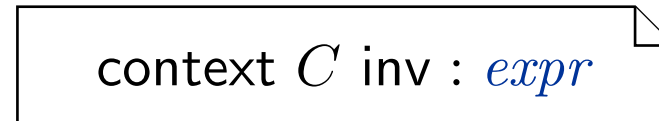
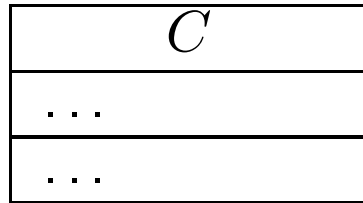
Sometimes, content is **explicitly classified** for clarity:



OCL in Notes: Conventions

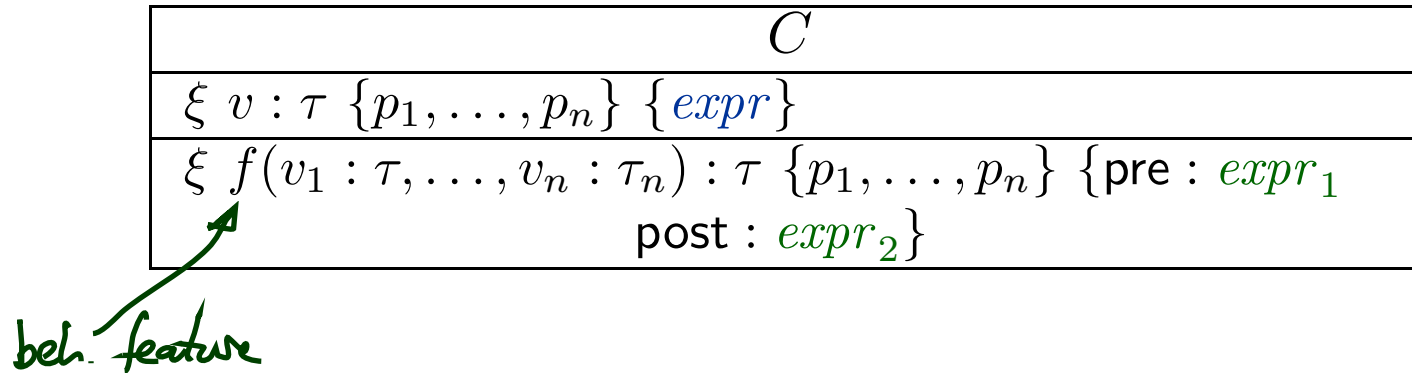


stands for

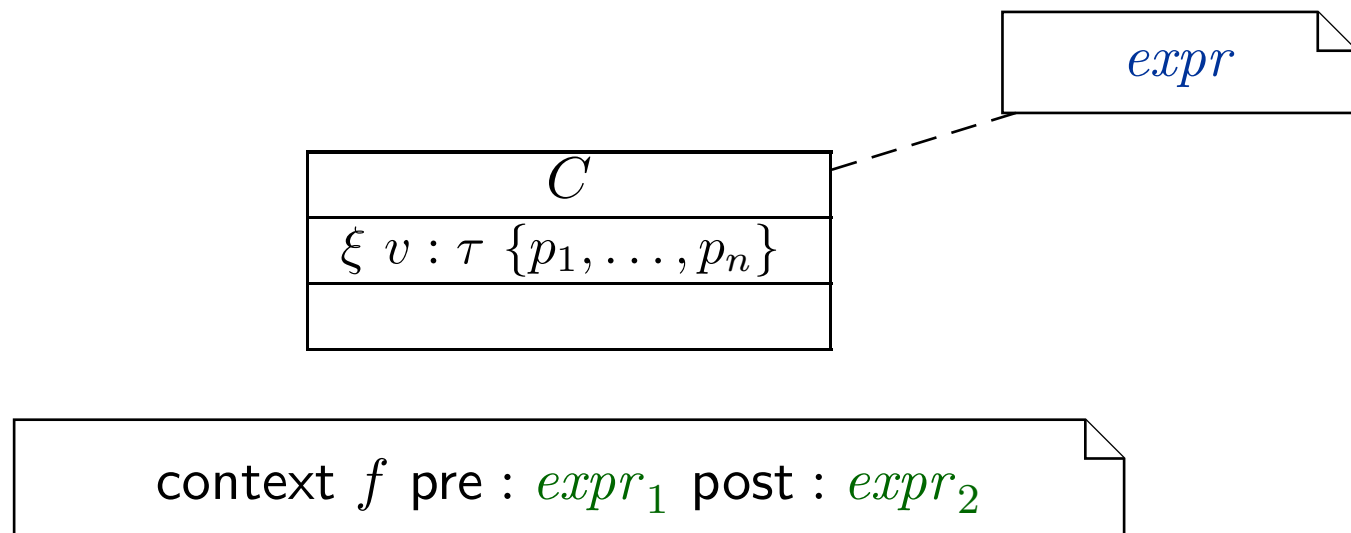


Where Shall We Put OCL Constraints?

- (ii) **Particular dedicated places** in class diagrams: (behav. feature: later)



For simplicity, we view the above as an abbreviation for



Invariants of a Class Diagram

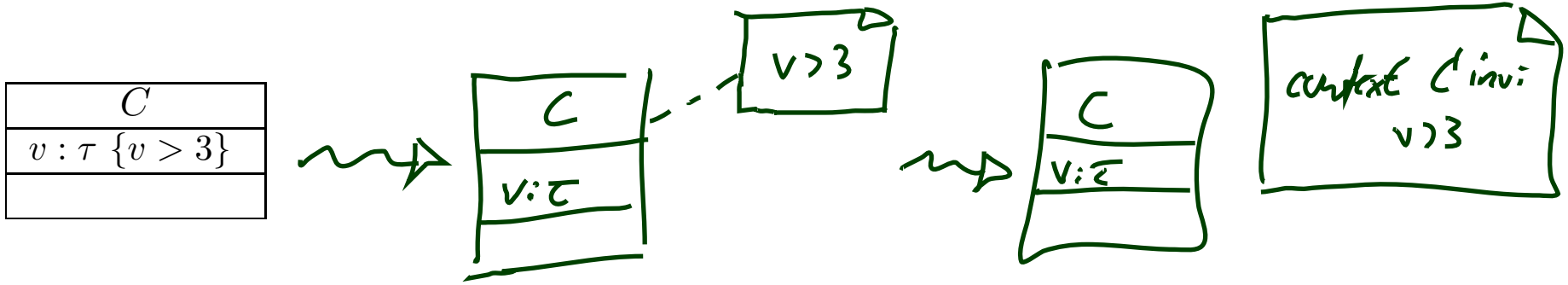
- Let \mathcal{CD} be a class diagram.
- As we (now) are able to recognise OCL constraints when we see them, we can define

$$Inv(\mathcal{CD})$$

as the set $\{\varphi_1, \dots, \varphi_n\}$ of OCL constraints **occurring** in notes in \mathcal{CD} — after **unfolding** all abbreviations (cf. next slides).

- As usual: $Inv(\mathcal{CD}) := \bigcup_{\mathcal{CD} \in \mathcal{CD}} Inv(\mathcal{CD})$.
- **Principally clear**: $Inv(\cdot)$ for any kind of diagram.

Invariant in Class Diagram Example



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

- $Inv(\mathcal{CD}) = Inv(CD) = \dots$

Semantics of a Class Diagram

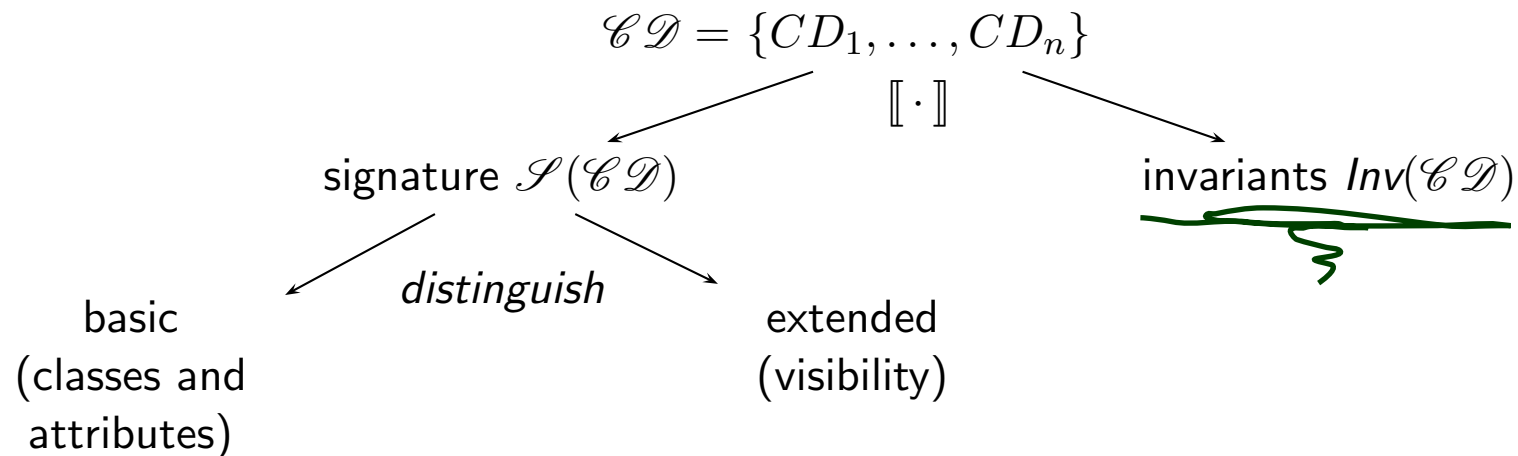
Definition. Let \mathcal{CD} be a set of class diagrams.

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 $\Sigma_{\mathcal{D}}^{\mathcal{S}}$, of which **some** may satisfy $\text{Inv}(\mathcal{CD})$.

In pictures:



Pragmatics

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

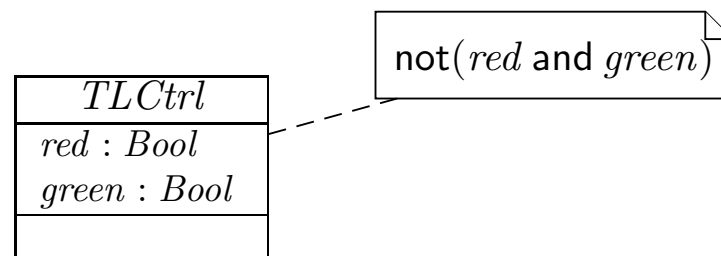
A set of class diagrams $\mathcal{C}\mathcal{D}$ with invariants $Inv(\mathcal{C}\mathcal{D})$ describes the **structure** of system states.

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.

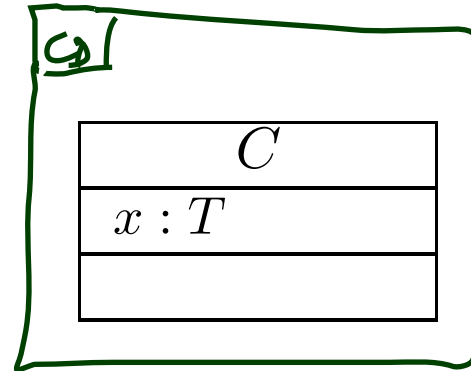
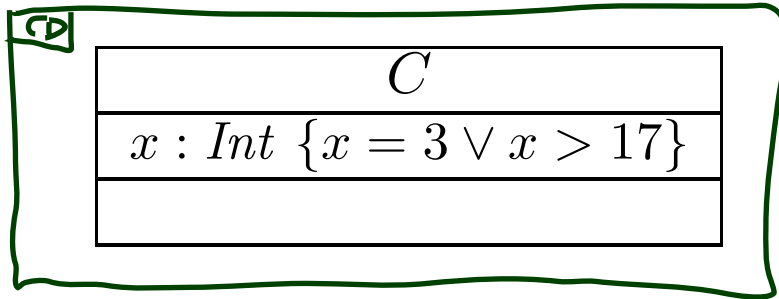
(The exact meaning of “use” will become clear when we study behaviour — intuitively: the system states that are reachable from the initial system state(s) by calling methods or firing transitions in state-machines.)

Example: highly abstract model of traffic lights controller.



Constraints vs. Types

Find the 10 differences:



$$\mathcal{D}(T) = \{3\} \\ \cup \{n \in \mathbb{N} \mid n > 17\}$$

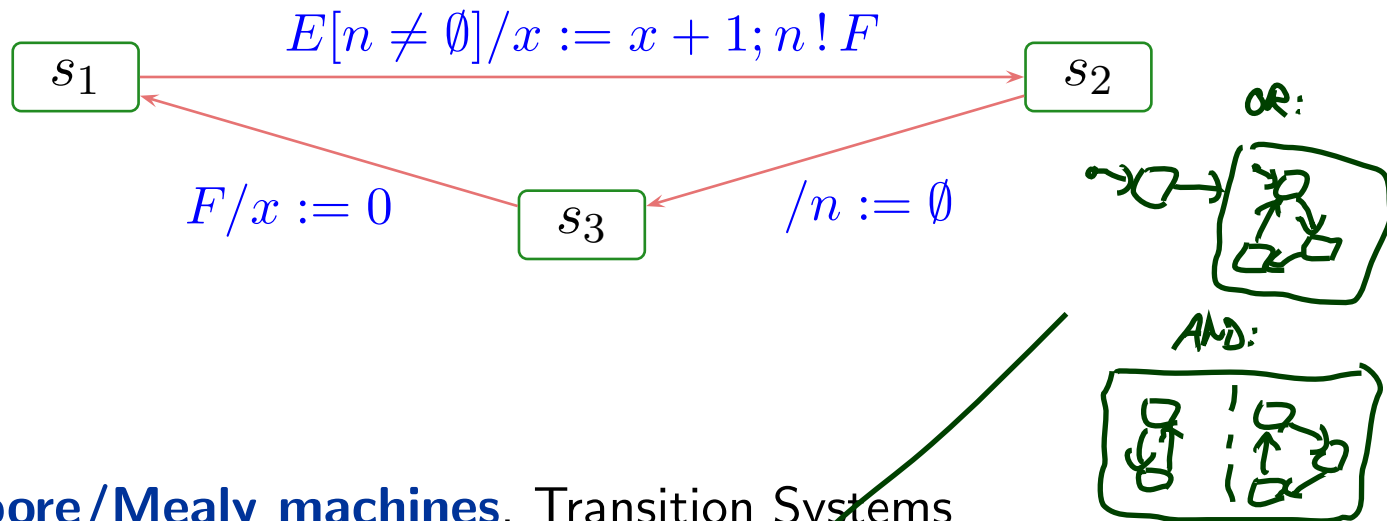
- $x = 4$ is well-typed in the left context, a system state satisfying $x = 4$ violates the constraints of the diagram.
- $x = 4$ is not even well-typed in the right context, there cannot be a system state with $\sigma(u)(x) = 4$ because $\sigma(u)(x)$ is supposed to be in $\mathcal{D}(T)$ (by definition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

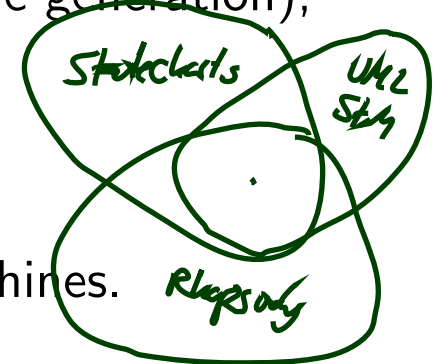
UML State Machines

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**.

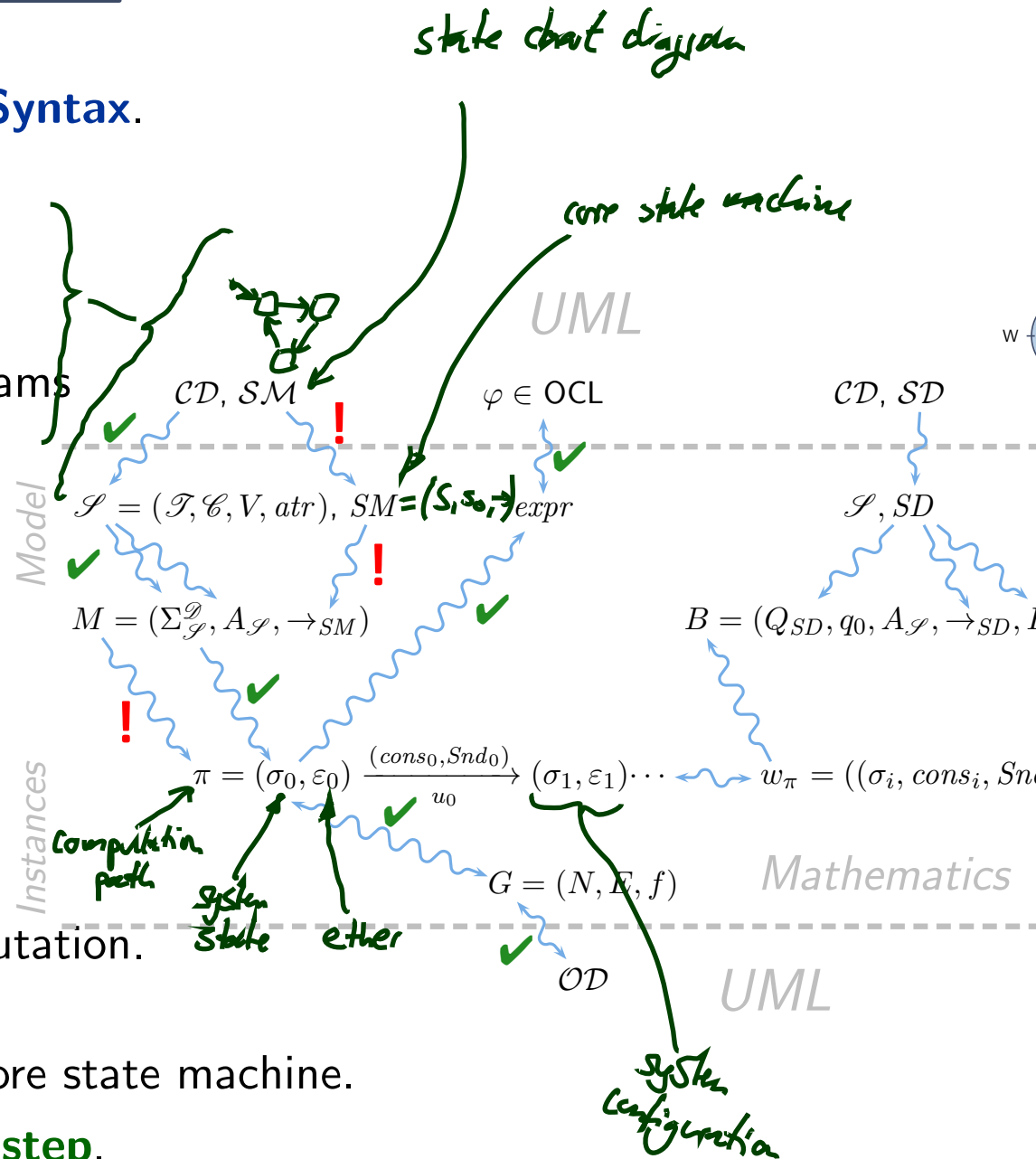
(viii) Def.: **Transformer**.

(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.

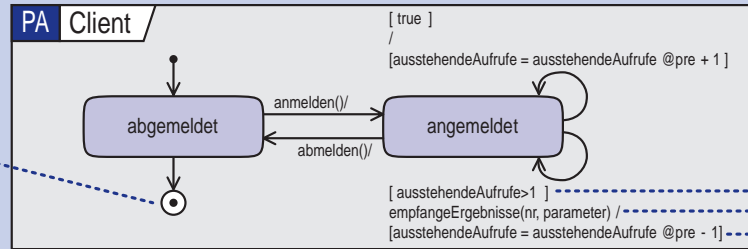


UML State Machines: Syntax

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

[Störrle, 2005]

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.

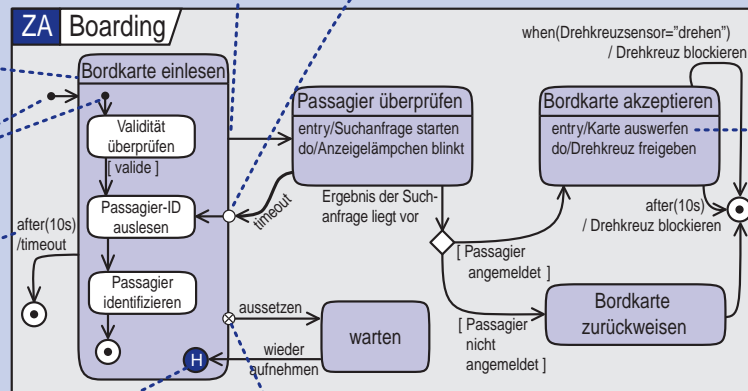
Reguläre Beendigung löst ein **completion**-Ereignis aus.

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.

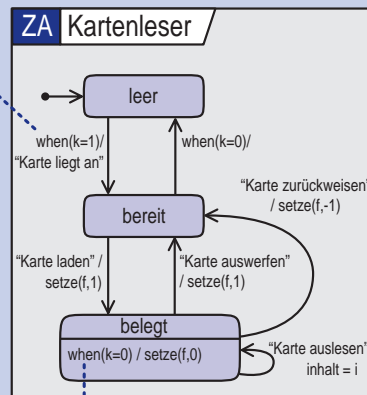
Wenn ein **Regionsendzustand** erreicht wird, wird der gesamte **komplexe** Zustand beendet, also auch alle parallelen Regionen.

Ein **verfeinerter Zustand** verweist auf einen Zustandsautomaten (angedeutet von dem Symbol unten links), der das Verhalten des Zustandes definiert.

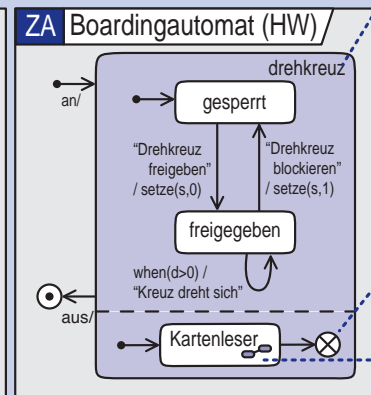
Auch Zeit- und Änderungsereignisse können Zustandsübergänge auslösen:

- **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 Montage-diagramm „Abfertigung“ links oben.



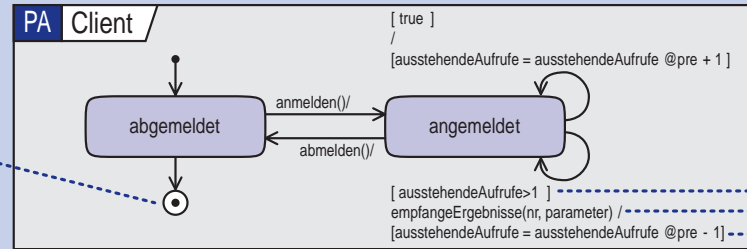
Ereignisse können innerhalb eines Zustands Aktionen auslösen.



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

[Störrle, 2005]

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** einer Region

Proven approach:

Start out simple, consider the essence, namely

- basic/leaf states
- transitions,

then extend to cover the complicated rest.

Der **Anfang** den voreingestellten von „Board einlesen“.

Das **Zeiter** einen Abbruch einlesen“ a

öst von sich aus Ereignisse aus:

Betretend; I des S; i beim Erreichen tandes einer ndsmaschine erlassen.

derer Ereignisse uslöser für rangezogen

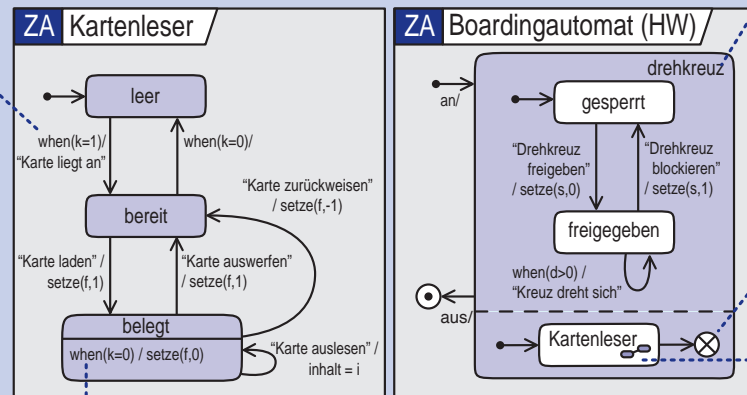
kann eine oder ionen enthalten, Zustands-

nommen 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 Montage-diagramm „Abfertigung“ links oben.



Ereignisse können innerhalb eines Zustands Aktionen auslösen.

automaten 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.

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 das Verhalten des Zustandes definiert.

Signature With Signals

Definition. A tuple

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

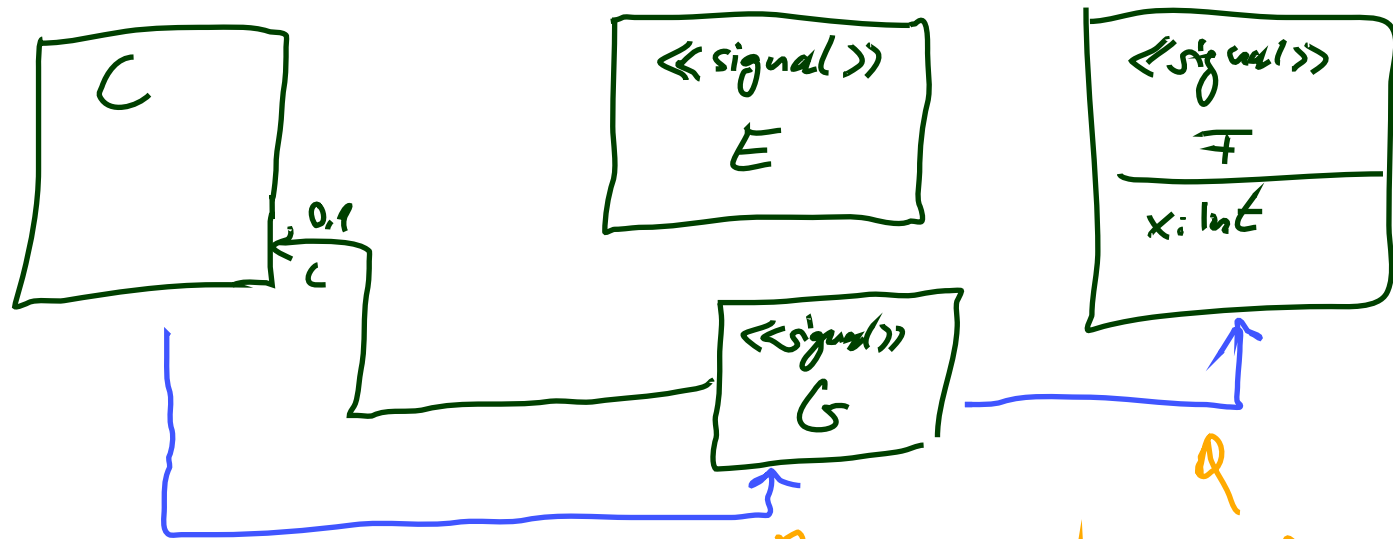
is called **signature (with signals)** if and only if

$$(\mathcal{T}, \mathcal{C}, V, atr)$$

is a signature (as before).

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

Signature With Signals: Example



$$\mathcal{Y} = \left(\mathcal{J}, \{C, E, F, G\}, \{x: \text{int}, c: C_{0,1}\}, \right.$$

$$\left. \{C \mapsto \emptyset, E \mapsto \emptyset, F \mapsto \{x: \text{int}\}, G \mapsto \{c: C_{0,1}\}\}, \right.$$

$$\left. \{E, F, G\} \right)$$

allowed by def., ruled out later

Core State Machine

Definition.

A **core state machine** over signature $\mathcal{S} = (\mathcal{I}, \mathcal{C}, V, atr, \mathcal{E})$ 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

source state → $\rightarrow \subseteq S \times (\mathcal{E} \cup \{-\}) \times \underbrace{Expr_{\mathcal{S}}}_{\text{guard}} \times \underbrace{Act_{\mathcal{S}}}_{\text{action}} \times S$ *set of signals* *destination state*

"∪" disjoint union
- & E

is a labelled transition relation.

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

From UML to Core State Machines: By Example

UML state machine diagram \mathcal{SM} :


$$\text{annot} ::= [\langle \text{event} \rangle [\text{'.'} \langle \text{event} \rangle]^* [\text{'['} \langle \text{guard} \rangle \text{' '}] [\text{'/'} \langle \text{action} \rangle]]$$

with

- $\text{event} \in \mathcal{E}$,
- $\text{guard} \in \text{Expr}_{\mathcal{J}}$ (default: *true*, assumed to be in $\text{Expr}_{\mathcal{J}}$)
- $\text{action} \in \text{Act}_{\mathcal{J}}$ (default: *skip*, assumed to be in $\text{Act}_{\mathcal{J}}$)

maps to

$$M(\mathcal{SM}) = \left(\underbrace{\{s_1, s_2\}}_S, \underbrace{s_1}_{s_0}, \underbrace{(s_1, \text{event}, \text{guard}, \text{action}, s_2)}_{\rightarrow} \right)$$

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.

[Harel, 1987] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

[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.

[Störrle, 2005] Störrle, H. (2005). *UML 2 für Studenten*. Pearson Studium.