Software Design, Modelling and Analysis in UML

Lecture 12: Core State Machines III

2011-12-21

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

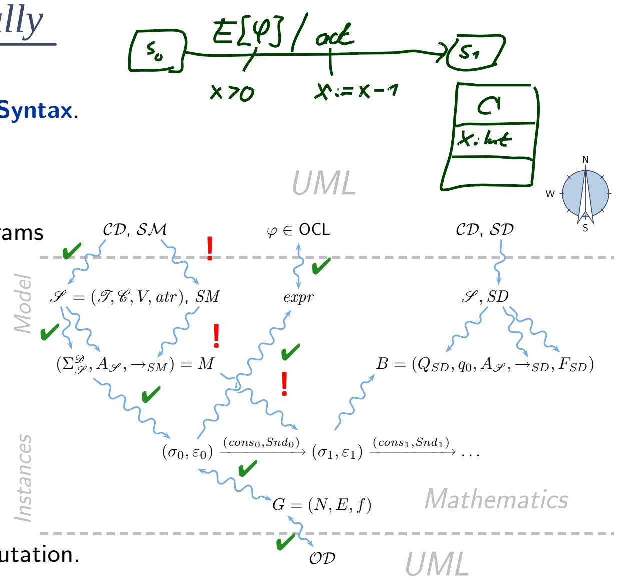
- The basic causality model
- Ether, System Configuration, Event, Transformer

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:**
  - Examples for transformer
  - Run-to-completion Step
  - Putting It All Together

System Configuration, Ether, Transformer

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.

Transformer

non-determinism

the object "executing" the action

Definition.
Let $\Sigma_{\mathscr{S}}^{\mathscr{D}}$ the set of system configurations over some $\mathscr{S}_0$, $\mathscr{D}_0$ and Eth and ether. We call a relation

$$t \subseteq \mathscr{D}(\mathscr{C}) \times (\Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth) \times (\Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth)$$

a (system configuration) **transformer**.

- In the following, we assume that each application of a transformer t to some system configuration (σ, ε) for object u_x is associated with a set of **observations**

$$Obs_t[u_x](\sigma, \varepsilon) \in 2^{\mathscr{D}(\mathscr{C}) \times Evs(\mathscr{E} \,\dot{\cup}\, \{*,+\}, \mathscr{D}) \times \mathscr{D}(\mathscr{C})}.$$

- An observation $(u_{src}, (E, \vec{d}), u_{dst}) \in Obs_t[u_x](\sigma, \varepsilon)$ represents the information that, as a "side effect" of u_x executing t, an event (!) $(E, \vec{d})$ has been sent from object u_{src} to object u_{dst}.
Special cases: creation/destruction.

Why Transformers?

- **Recall** the (simplified) syntax of transition annotations:

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

- **Clear**: $\langle event \rangle$ is from $\mathscr{E}$ of the corresponding signature.
- **But:** What are $\langle guard \rangle$ and $\langle action \rangle$?
  - UML can be viewed as being **parameterized** in **expression language** (providing $\langle guard \rangle$) and **action language** (providing $\langle action \rangle$).
  - **Examples**:
    - **Expression Language**:
      - OCL
      - Java, C++, ... expressions
      - ...
    - **Action Language**:
      - UML Action Semantics, "Executable UML"
      - Java, C++, ... statements (plus some event send action)
      - ...

Transformers as Abstract Actions!

In the following, we assume that we're **given**

- an **expression language** $Expr$ for guards, and
- an **action language** Act for actions,

partial function
I may not defined
for some expr ∈ Expr

and that we're **given**

- a **semantics** for boolean expressions in form of a partial function

$$I[\![\cdot]\!](\cdot,\cdot) : Expr \to ((\Sigma_{\mathscr{S}}^{\mathscr{D}} \times (\{\underset{\text{self}}{\cancel{this}}\} \to \mathscr{D}(\mathscr{C}))) \nrightarrow \mathbb{B})$$

  which evaluates expressions in a given system configuration,

  Assuming I to be partial is a way to treat "undefined" during runtime. If I is not defined (for instance because of dangling-reference navigation or division-by-zero), we want to go to a designated "error" system configuration.

- a **transformer** for each action: For each $act \in Act$, we assume to have

$$t_{act} \subseteq \mathscr{D}(\mathscr{C}) \times (\Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth) \times (\Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth).$$

Expression/Action Language Examples

We can make the assumptions from the previous slide because **instances exist**:

- for OCL, we have the OCL semantics from Lecture 03. Simply remove the pre-images which map to "$\perp$".
- for Java, the operational semantics of the SWT lecture uniquely defines transformers for sequences of Java statements.

We distinguish the following kinds of transformers:

- **skip**: do nothing — recall: this is the default action
- **send**: modifies ε — interesting, because state machines are built around sending/consuming events
- **create**/**destroy**: modify domain of σ — not specific to state machines, but let's discuss them here as we're at it
- **update**: modify own or other objects' local state — boring

In the following we discuss

$$
\begin{aligned}
Act_{\mathscr{S}} := \{skip\}\\
&\cup \{update(expr_1, v, expr_2) \mid expr_1, expr_2 \in OCLExpr, \; v \in V\}\\
&\cup \{send(expr_1, E, expr_2) \mid expr_1, expr_2 \in OCLExpr, E \in \mathscr{E}(\mathscr{S})\}\\
&\cup \{create(C, expr, v) \mid expr \in OCLExpr, C \in \mathscr{C}, v \in V\}\\
&\cup \{destroy(expr) \mid expr \in OCLExpr\}
\end{aligned}
$$

Transformer Examples: Presentation

abstract syntax	**concrete syntax**
op	
intuitive semantics	
...	
well-typedness	
...	
semantics	
$((\sigma, \varepsilon), (\sigma', \varepsilon')) \in t_{\mathsf{op}}[u_x]$ iff ... or $t_{\mathsf{op}}[u_x](\sigma, \varepsilon) = \{(\sigma', \varepsilon')\}$ where ...	
observables	
$Obs_{\mathsf{op}}[u_x](\sigma, \varepsilon) = \{\dots\}$, not a relation, depends on choice	
(error) conditions	
Not defined if ...	

Transformer: Skip

abstract syntax | **concrete syntax**

`skip` | skip

intuitive semantics

do nothing

well-typedness

./.

semantics

$$t[u_x](\sigma, \varepsilon) = \{(\sigma, \varepsilon)\}$$

observables

$$Obs_{\texttt{skip}}[u_x](\sigma, \varepsilon) = \emptyset$$

(error) conditions

Transformer: Update

abstract syntax | **concrete syntax**

$\texttt{update}(expr_1, v, expr_2)$ | $expr_1.v := expr_2$

intuitive semantics

Update attribute v in the object denoted by $expr_1$ to the value denoted by $expr_2$.

well-typedness

$expr_1 : \tau_C$ and $v : \tau \in atr(C)$; $\quad expr_2 : \tau$;

$expr_1, expr_2$ obey visibility and navigability

semantics

$$t_{\texttt{update}(expr_1, v, expr_2)}[u_x](\sigma, \varepsilon) = \{(\sigma', \varepsilon)\}$$

doesn't change

where $\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[\![expr_2]\!](\sigma, \beta)]]$ with

$u = I[\![expr_1]\!](\sigma, \beta)$, $\beta = \{\cancel{\text{this}}\ \text{self} \mapsto u_x\}$.

observables

$$Obs_{\texttt{update}(expr_1, v, expr_2)}[u_x] = \emptyset$$

(error) conditions

Not defined if $I[\![expr_1]\!](\sigma, \beta)$ or $I[\![expr_2]\!](\sigma, \beta)$ not defined.

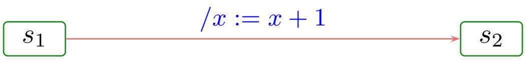
id of the object whose attribute is updated

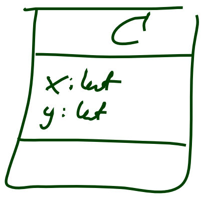
the attribute that's updated

new value for v, as given by "old" system state σ

Update Transformer Example

$\mathcal{SM}_C$:

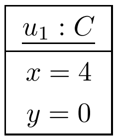




$$\texttt{update}(expr_1, v, expr_2)$$

$$t_{\texttt{update}(expr_1,v,expr_2)}[u_x](\sigma,\varepsilon) = (\sigma[u \mapsto \sigma(u)[v \mapsto I[\![expr_2]\!](\sigma,\beta)]], \varepsilon),$$
$$u = I[\![expr_1]\!](\sigma,\beta)$$

σ:

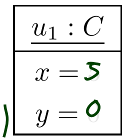


$$t_{upd}[u_1](\sigma,\varepsilon) = \overbrace{}^{u_x = u_1} \quad \text{implicit self.}$$

$$(\sigma[u \mapsto \sigma(u)[x \mapsto \underbrace{I[\![x+1]\!](\sigma,\beta)}_{5}]], \varepsilon)$$

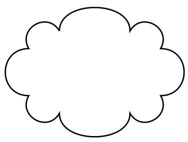
only x changes

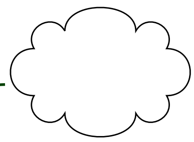
$I[\![\text{self}.x+1]\!](\sigma, \{\text{self} \mapsto u_1\}) = 5$



$:\sigma'$

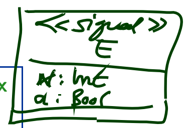
ε:





$:\varepsilon'$

Transformer: Send



abstract syntax

$\texttt{send}(E(expr_1, ..., expr_n), expr_{dst})$

concrete syntax

$expr_{dst}\ \text{!}\ E(\ldots)$

intuitive semantics

Object $u_x : C$ sends event E to object $expr_{dst}$, i.e. create a fresh signal instance, fill in its attributes, and place it in the ether.

well-typedness

$expr_{dst} : \tau_D,\ C, D \in \mathscr{C};\ E \in \mathscr{E};\ atr(E) = \{v_1 : \tau_1, \ldots, v_n : \tau_n\};$ $\backslash E(\mathscr{S})\ (\mathscr{S});$

$expr_i : \tau_i,\ 1 \le i \le n;$

all expressions obey visibility and navigability in C

semantics

$$t_{\texttt{send}(E(expr_1,...,expr_n),expr_{dst})}[u_x](\sigma,\varepsilon) \ni (\sigma',\varepsilon')$$

where $\sigma' = \sigma \,\dot\cup\, \{u \mapsto \{v_i \mapsto d_i \mid 1 \le i \le n\}\}$; $\quad \varepsilon' = \varepsilon \oplus (u_{dst}, u)$;
if $u_{dst} = I[\![expr_{dst}]\!](\sigma,\beta) \in \mathrm{dom}(\sigma)$; $\quad d_i = I[\![expr_i]\!](\sigma,\beta)$ for $1 \le i \le n$;
$u \in \mathscr{D}(E)$ a fresh identity, i.e. $u \notin \mathrm{dom}(\sigma)$,
and where $(\sigma',\varepsilon') = (\sigma,\varepsilon)$ if $u_{dst} \notin \mathrm{dom}(\sigma)$; $\beta = \{\cancel{\text{this}}\ \text{self} \mapsto u_x\}$

u is a fresh identity

// our choice – we could turn it into an error condition as well

observables

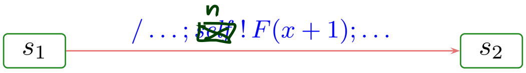
$$Obs_{\texttt{send}}[u_x] = \{(u_x, (E, d_1, \ldots, d_n), u_{dst})\}$$

(error) conditions

$I[\![expr]\!](\sigma,\beta)$ not defined for any
$expr \in \{expr_{dst}, expr_1, \ldots, expr_n\}$

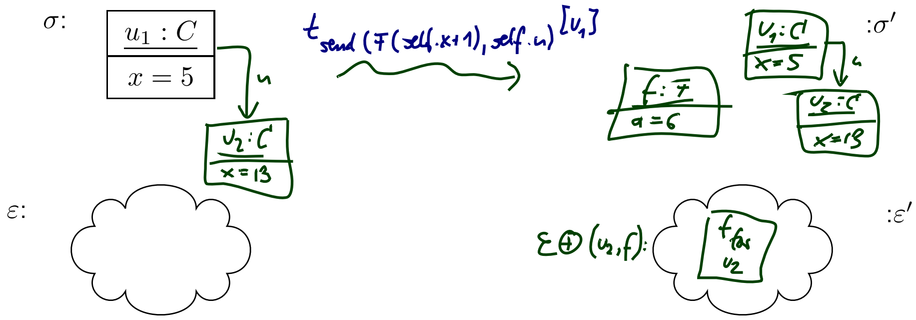
Send Transformer Example

$\mathcal{SM}_C$:



$$\texttt{send}(E(expr_1, ..., expr_n), expr_{dst})$$

$$t_{\texttt{send}(expr_{src}, E(expr_1,...,expr_n), expr_{dst})}[u_x](\sigma, \varepsilon) = ...$$



Transformer: Create

abstract syntax
$\texttt{create}(C, expr, v)$

concrete syntax
expr.v := new(C)

intuitive semantics
Create an object of class C and assign it to attribute v of the object denoted by expression $expr$.

well-typedness
$expr : \tau_D,\ v \in atr(D),\ atr(C) = \{\langle v_1 : \tau_1, expr_i^0 \rangle \mid 1 \le i \le n\}$

semantics
...

observables
...

(error) conditions
$I[\![expr]\!](\sigma, \beta)$ not defined.

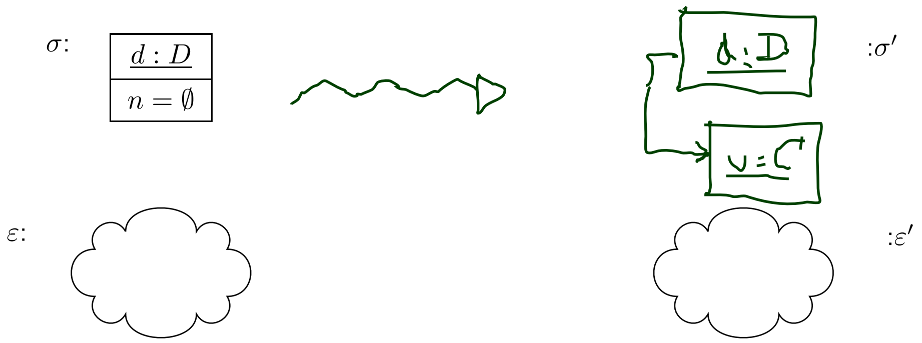
- We use an "and assign"-action for simplicity — it doesn't add or remove expressive power, but moving creation to the expression language raises all kinds of other problems such as order of evaluation (and thus creation).
- Also for simplicity: no parameters to construction ($\sim$ parameters of constructor). Adding them is straightforward (but somewhat tedious).

Create Transformer Example

$\mathcal{SM}_C$: $s_1 \xrightarrow{/ \ldots; n := new\ C; \ldots} s_2$

$\texttt{create}(C, expr, v)$

$t_{\texttt{create}(C,expr,v)}(\sigma, \varepsilon) = \ldots$



How To Choose New Identities?

- **Re-use**: choose any identity that is not alive **now**, i.e. not in $\mathrm{dom}(\sigma)$.
  - Doesn't depend on history.
  - May "undangle" dangling references – may happen on some platforms.

- **Fresh**: choose any identity that has not been alive **ever**, i.e. not in $\mathrm{dom}(\sigma)$ and any predecessor in current run.
  - Depends on history.
  - Dangling references remain dangling – could mask "dirty" effects of platform.

Transformer: Create

expr.v = new(C)

C

x : Int = 13

abstract syntax **concrete syntax**

$\texttt{create}(C, expr, v)$

intuitive semantics

Create an object of class C and assign it to attribute v of the object denoted by expression $expr$.

well-typedness

$expr : \tau_D,\ v \in atr(D),\ atr(C) = \{\langle v_1 : \tau_1, expr_i^0 \rangle \mid 1 \leq i \leq n\}$

semantics

$$((\sigma, \varepsilon), (\sigma', \varepsilon')) \in t$$

initial values as given by Class Diagram

iff $\sigma' = \sigma[u_0 \mapsto \sigma(u_0)[v \mapsto u]] \cup \{u \mapsto \{v_i \mapsto d_i \mid 1 \leq i \leq n\}\}$,

$\varepsilon' = [u](\varepsilon)$; $u \in \mathscr{D}(C)$ fresh, i.e. $u \notin \mathrm{dom}(\sigma)$;

$u_0 = I[\![expr]\!](\sigma, \beta)$; $d_i = I[\![expr_i^0]\!](\sigma, \beta)$ if $expr_i^0 \neq$ '' and arbitrary value from $\mathscr{D}(\tau_i)$ otherwise; $\beta = \{\mathsf{this} \mapsto u_x\}$.

clear either for u

observables

$$Obs_{\texttt{create}}[u_x] = \{(u_x, (*, \emptyset), u)\}$$

(error) conditions

$I[\![expr]\!](\sigma)$ not defined.

Transformer: Destroy

abstract syntax **concrete syntax**

$\texttt{destroy}(expr)$

intuitive semantics

Destroy the object denoted by expression $expr$.

well-typedness

$expr : \tau_C,\ C \in \mathscr{C}$

semantics

$\ldots$

observables

$$Obs_{\texttt{destroy}}[u_x] = \{(u_x, (+, \emptyset), u)\}$$

(error) conditions

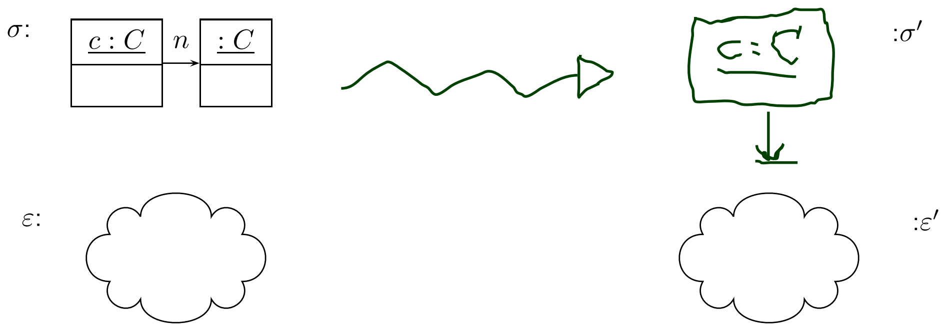
$I[\![expr]\!](\sigma, \beta)$ not defined.

Destroy Transformer Example

$\mathcal{SM}_C$: $s_1 \xrightarrow{/ \ldots;\, delete\ n; \ldots} s_2$

$\texttt{destroy}(expr)$

$t_{\texttt{destroy}(expr)}[u_x](\sigma, \varepsilon) = \ldots$



What to Do With the Remaining Objects?

Assume object u_0 is destroyed...

- object u_1 may still refer to it via association r:
  - allow dangling references?
  - or remove u_0 from $\sigma(u_1)(r)$?
- object u_0 may have been the last one linking to object u_2:
  - leave u_2 alone?
  - or remove u_2 also?
- Plus: (temporal extensions of) OCL may have dangling references.

Our choice: Dangling references and no garbage collection!
This is in line with "expect the worst", because there are target platforms which don't provide garbage collection — and models shall (in general) be correct without assumptions on target platform.

But: the more "dirty" effects we see in the model, the more expensive it often is to analyse. Valid proposal for simple analysis: monotone frame semantics, no destruction at all.

Transformer: Destroy

abstract syntax	**concrete syntax**
`destroy`$(expr)$	
intuitive semantics	
Destroy the object denoted by expression $expr$.	
well-typedness	
$expr : \tau_C,\ C \in \mathscr{C}$	
semantics	
$t[u_x](\sigma, \varepsilon) = (\sigma', \varepsilon)$ where $\sigma' = \sigma\vert_{\{\mathrm{dom}(\sigma) \setminus \{u\}\}}$ with $u = I[\![expr]\!](\sigma, \beta)$.	
observables	
$Obs_{\texttt{destroy}}[u_x] = \{(u_x, (+, \emptyset), u)\}$	
(error) conditions	
$I[\![expr]\!](\sigma, \beta)$ not defined.	

Sequential Composition of Transformers

- **Sequential composition** $t_1 \circ t_2$ of transformers t_1 and t_2 is canonically defined as

$$(t_2 \circ t_1)[u_x](\sigma, \varepsilon) = t_2[u_x](t_1[u_x](\sigma, \varepsilon))$$

with observation

$$Obs_{(t_2 \circ t_1)}[u_x](\sigma, \varepsilon) = Obs_{t_1}[u_x](\sigma, \varepsilon) \cup Obs_{t_2}[u_x](t_1(\sigma, \varepsilon)).$$

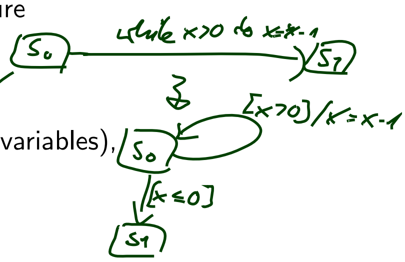
- **Clear**: not defined if one the two intermediate "micro steps" is not defined.

x := x+1 ; y := 27 ; u.F

Transformers And Denotational Semantics

Observation: our transformers are in principle the **denotational semantics** of the actions/action sequences. The trivial case, to be precise.

Note: with the previous examples, we can capture



- empty statements, skips,
- assignments,
- conditionals (by normalisation and auxiliary variables),
- create/destroy,

but not **possibly diverging loops**.

Our (Simple) Approach: if the action language is, e.g. Java, then (syntactically) forbid loops and calls of recursive functions.

Other Approach: use full blown denotational semantics.

No show-stopper, because loops in the action annotation can be converted into transition cycles in the state machine.

Run-to-completion Step

Transition Relation, Computation

Definition. Let A be a set of **actions** and S a (not necessarily finite) set of of **states**.

We call

$$\rightarrow \subseteq S \times A \times S$$

a (labelled) **transition relation**.

Let $S_0 \subseteq S$ be a set of **initial states**. A sequence

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots$$

with $s_i \in S$, $a_i \in A$ is called **computation** of the **labelled transition system** $(S, \rightarrow, S_0)$ if and only if

- **initiation**: $s_0 \in S_0$
- **consecution**: $(s_i, a_i, s_{i+1}) \in \rightarrow$ for $i \in \mathbb{N}_0$.

Note: for simplicity, we only consider infinite runs.

Active vs. Passive Classes/Objects

- **Note**: From now on, assume that all classes are **active** for simplicity.

  We'll later briefly discuss the Rhapsody framework which proposes a way how to integrate non-active objects.

- **Note**: The following RTC "algorithm" follows [Harel and Gery, 1997] (i.e. the one realised by the Rhapsody code generation) where the standard is ambiguous or leaves choices.

From Core State Machines to LTS

Definition. Let $\mathscr{S}_0 = (\mathscr{T}_0, \mathscr{C}_0, V_0, atr_0, \cancel{\mathscr{E}_0})$ be a signature with signals (all classes **active**), $\mathscr{D}_0$ a structure of $\mathscr{S}_0$, and $(Eth, ready, \oplus, \ominus, [\,\cdot\,])$ an ether over $\mathscr{S}_0$ and $\mathscr{D}_0$. Assume there is one core state machine M_C per class $C \in \mathscr{C}$.

We say, the state machines induce the following labelled transition relation on states $S := \Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth \;\dot{\cup}\; \{\#\}$ with actions $A := 2^{\mathscr{D}(\mathscr{C}) \times Evs(\mathscr{E},\mathscr{D})} \times 2^{\mathscr{D}(\mathscr{C}) \times Evs(\mathscr{E},\mathscr{D})} \times \mathscr{D}(\mathscr{C})$:

"crash"

$$(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon')$$

if and only if

(i) an event with destination u is discarded,

(ii) an event is dispatched to u, i.e. stable object processes an event, or

(iii) run-to-completion processing by u commences,
i.e. object u is not stable and continues to process an event,

(iv) the environment interacts with object u,

$$s \xrightarrow{(cons, \emptyset)} \#$$

if and only if

(v) $s = \#$ and $cons = \emptyset$, or an error condition occurs during consumption of $cons$.

(i) Discarding An Event

$$(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon')$$

if

- an E-event (instance of signal E) is ready in ε for ~~an~~ object of a class $\mathscr{C}$, i.e.

$$\cancel{\exists}\, u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C)\; \exists\, u_E \in \mathscr{D}(\mathscr{E}) : u_E \in ready(\varepsilon, u)$$

- u is stable and in state machine state s, i.e. $\sigma(u)(stable) = 1$ and $\sigma(u)(st) = s$,
- but there is no corresponding transition enabled (all transitions incident with current state of u either have other triggers or the guard is not satisfied)

$$\forall\, (s, F, expr, act, s') \in \rightarrow (\mathcal{SM}_C) : F \neq E \vee I[\![expr]\!](\sigma) = 0$$

and

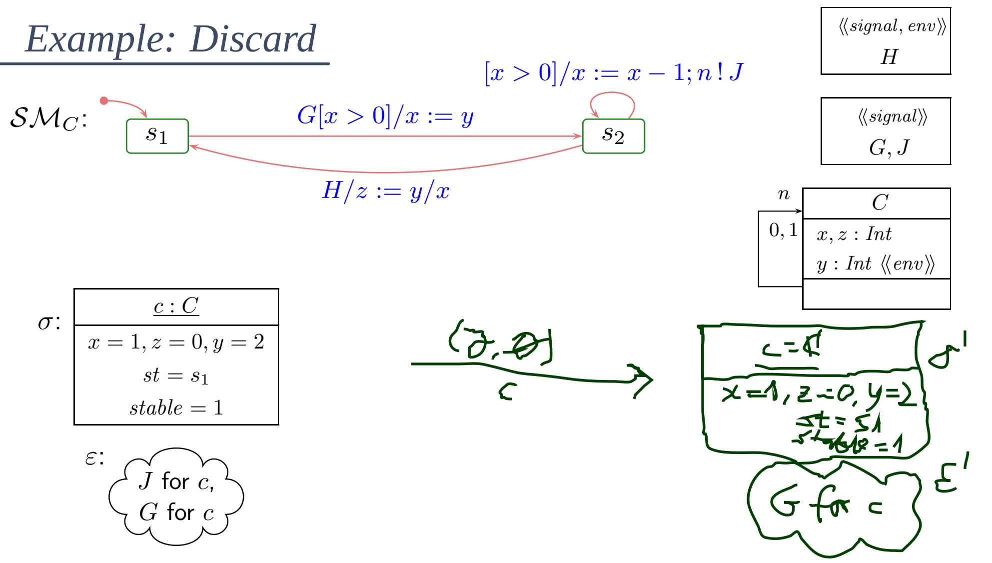
- the system configuration doesn't change, i.e. $\sigma' = \sigma$
- the event u_E is removed from the ether, i.e.

$$\varepsilon' = \varepsilon \ominus u_E,$$

- consumption of u_E is observed, i.e.

$$cons = \{(u, (E, \sigma(u_E)))\}, Snd = \emptyset.$$

Example: Discard



- $\exists\, u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C)$
  $\exists\, u_E \in \mathscr{D}(\mathscr{E}) : u_E \in ready(\varepsilon, u)$
- $\forall\, (s, F, expr, act, s') \in \rightarrow (\mathcal{SM}_C)$:
  $F \neq E \vee I[\![expr]\!](\sigma) = 0$
- $\sigma(u)(stable) = 1,\ \sigma(u)(st) = s,$
- $\sigma' = \sigma,\ \varepsilon' = \varepsilon \ominus u_E$
- $cons = \{(u, (E, \sigma(u_E)))\},\ Snd = \emptyset$

(ii) Dispatch

$$(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon') \text{ if}$$

- $\exists\, u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C)\ \exists\, u_E \in \mathscr{D}(\mathscr{E}) : u_E \in ready(\varepsilon, u)$
- u is stable and in state machine state s, i.e. $\sigma(u)(stable) = 1$ and $\sigma(u)(st) = s$,
- a transition is enabled, i.e.

$$\exists\, (s, F, expr, act, s') \in \rightarrow (\mathcal{SM}_C) : F = E \wedge I[\![expr]\!](\tilde{\sigma}) = 1$$

  where $\tilde{\sigma} = \sigma[u.params_E \mapsto u_e]$.

and

- (σ', ε') results from applying t_{act} to (σ, ε) and removing u_E from the ether, i.e.

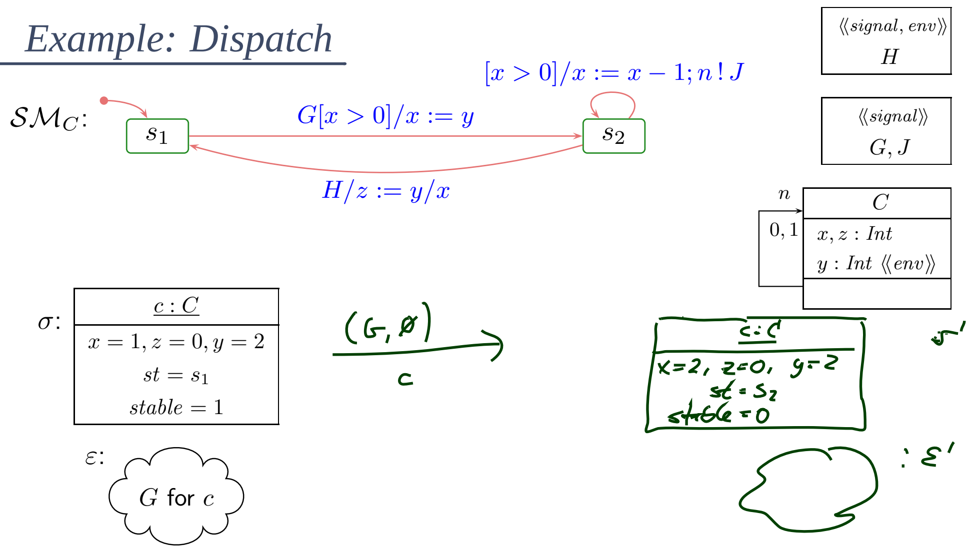
$$(\sigma'', \varepsilon') = t_{act}(\tilde{\sigma}, \varepsilon \ominus u_E),$$

$$\sigma' = (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset])|_{\mathscr{D}(\mathscr{C}) \setminus \{u_E\}}$$

  where b **depends**:
  - If u becomes stable in s', then $b = 1$. It **does** become stable if and only if there is no transition **without trigger** enabled for u in (σ', ε').
  - Otherwise $b = 0$.
- Consumption of u_E and the side effects of the action are observed, i.e.

$$cons = \{(u, (E, \sigma(u_E)))\}, Snd = Obs_{t_{act}}(\tilde{\sigma}, \varepsilon \ominus u_E).$$

Example: Dispatch



- $\exists\, u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C)$
  $\exists\, u_E \in \mathscr{D}(\mathscr{E}) : u_E \in ready(\varepsilon, u)$
- $\exists\, (s, F, expr, act, s') \in \!\!\to (\mathcal{SM}_C)$:
  $F = E \wedge I[\![expr]\!](\tilde{\sigma}) = 1$
- $\tilde{\sigma} = \sigma[u.params_E \mapsto u_e]$.
- $\sigma(u)(stable) = 1$, $\sigma(u)(st) = s$,
- $(\sigma'', \underline{\varepsilon'}) = t_{act}(\tilde{\sigma}, \underline{\varepsilon \ominus u_E})$
- $\sigma' = (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset])|_{\mathscr{D}(\mathscr{C}) \setminus \{u_E\}}$
- $cons = \{(u, (E, \sigma(u_E)))\}$, $Snd = Obs_{t_{act}}(\tilde{\sigma}, \varepsilon \ominus u_E)$

(iii) Commence Run-to-Completion

$$(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon')$$

if

- there is an unstable object u of a class $\mathscr{C}$, i.e.

$$\exists\, u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C) : \sigma(u)(stable) = 0$$

- there is a transition without ~~guard~~ trigger enabled from the current state $s = \sigma(u)(st)$, i.e.

$$\exists\, (s, \_, expr, act, s') \in \!\!\to (\mathcal{SM}_C) : I[\![expr]\!](\sigma) = 1$$

and

- (σ', ε') results from applying t_{act} to (σ, ε), i.e.

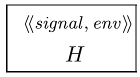
$$(\sigma'', \varepsilon') \in t_{act}[u](\sigma, \varepsilon), \quad \sigma' = \sigma''[u.st \mapsto s', u.stable \mapsto b]$$

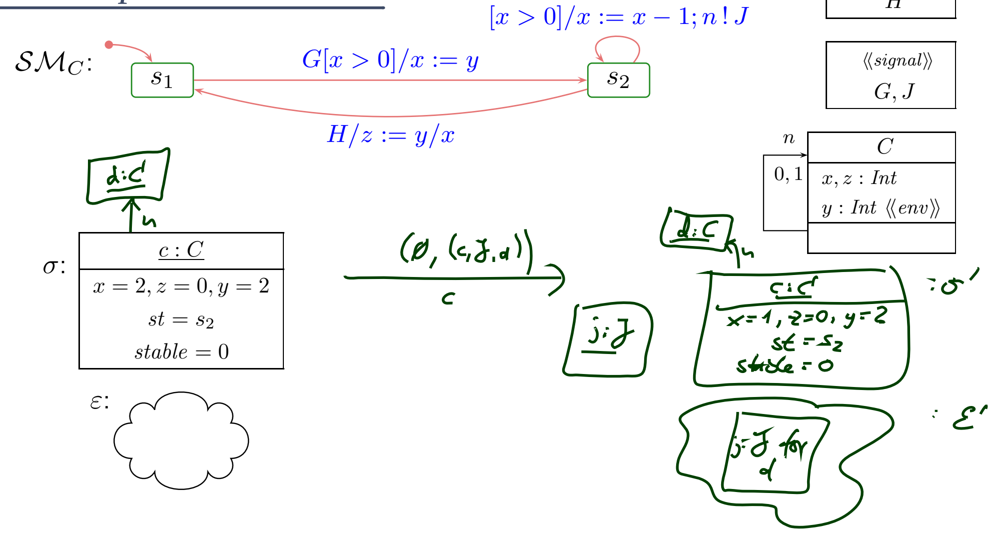
where b **depends** as before.

- Only the side effects of the action are observed, i.e.

$$cons = \emptyset, Snd = Obs_{t_{act}}(\sigma, \varepsilon).$$

Example: Commence





- $\exists\, u \in \text{dom}(\sigma) \cap \mathscr{D}(C) : \sigma(u)(stable) = 0$
- $\exists\, (s, \_, expr, act, s') \in\to (\mathcal{SM}_C) : I[\![expr]\!](\sigma) = 1$
- $\sigma(u)(stable) = 1,\ \sigma(u)(st) = s,$
- $(\sigma'', \varepsilon') = t_{act}(\sigma, \varepsilon),$
  $\sigma' = \sigma''[u.st \mapsto s', u.stable \mapsto b]$
- $cons = \emptyset,\ Snd = Obs_{t_{act}}(\sigma, \varepsilon)$

(iv) Environment Interaction

Assume that a set $\mathscr{E}_{env} \subseteq \mathscr{E}$ is designated as **environment events** and a set of attributes $v_{env} \subseteq V$ is designated as **input attributes**.

Then

$$(\sigma, \varepsilon) \xrightarrow[env]{(cons, Snd)} (\sigma', \varepsilon')$$

if

- an environment event $E \in \mathscr{E}_{env}$ is spontaneously sent to an alive object $u \in \mathscr{D}(\sigma)$, i.e.

  $$\sigma' = \sigma \,\dot\cup\, \{u_E \mapsto \{v_i \mapsto d_i \mid 1 \leq i \leq n\}, \quad \varepsilon' = \varepsilon \oplus u_E$$

  where $u_E \notin \text{dom}(\sigma)$ and $atr(E) = \{v_1, \ldots, v_n\}$.
- Sending of the event is observed, i.e. $cons = \emptyset,\ Snd = \{(env, E(\vec{d}))\}$.

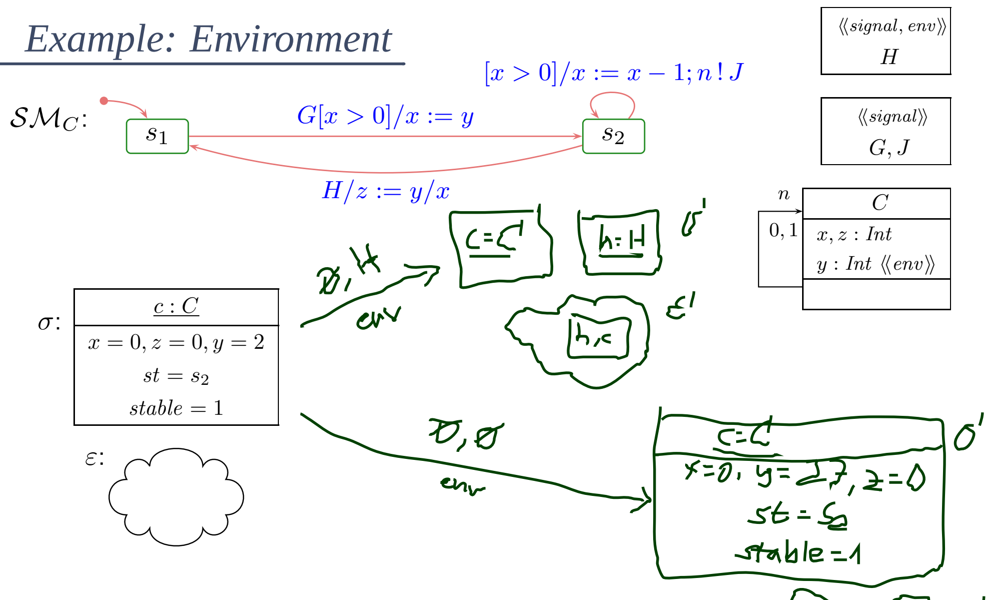
or

- Values of input attributes change freely in alive objects, i.e.

  $$\forall\, v \in V\ \forall\, u \in \text{dom}(\sigma) : \sigma'(u)(v) \neq \sigma(u)(v) \implies v \in V_{env}.$$

  and no objects appear or disappear, i.e. $\text{dom}(\sigma') = \text{dom}(\sigma)$.
- $\varepsilon' = \varepsilon$.

Example: Environment



- $\sigma' = \sigma \,\dot\cup\, \{u_E \mapsto \{v_i \mapsto d_i \mid 1 \leq i \leq n\}$
- $\varepsilon' = \varepsilon \oplus u_E$ where $u_E \notin \mathrm{dom}(\sigma)$ and $atr(E) = \{v_1, \dots, v_n\}$.
- $u \in \mathrm{dom}(\sigma)$
- $cons = \emptyset$, $Snd = \{(env, E(\vec{d}))\}$.

(v) Error Conditions

$$s \xrightarrow[u]{(cons, Snd)} \#$$

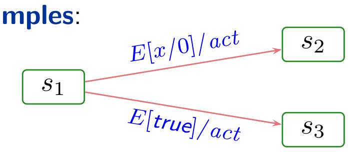
if, in (ii) or (iii),

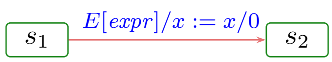
- $I[\![expr]\!]$ is not defined for σ, or
- t_{act} is not defined for (σ, ε),

and

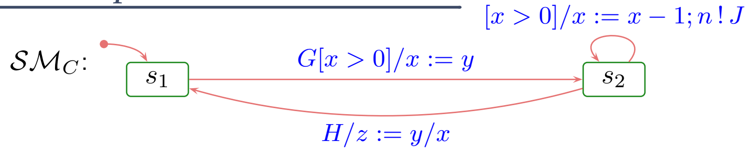
- consumption **is observed** according to (ii) or (iii), but $Snd = \emptyset$.

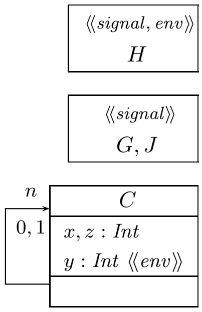
Examples:

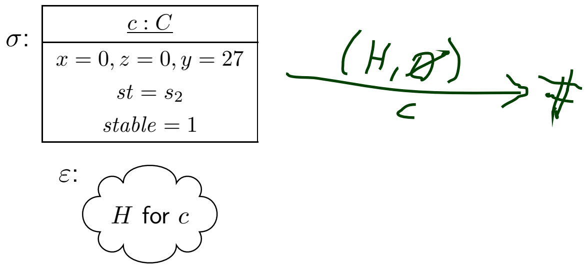
- 

- 

Example: Error Condition







- $I[\![expr]\!]$ not defined for σ, or
- t_{act} is not defined for (σ, ε)
- consumption according to (ii) or (iii)
- $Snd = \emptyset$

Notions of Steps: The Step

Note: we call one evolution $(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon')$ a **step**.

Thus in our setting, **a step directly corresponds** to

one object (namely u) takes **a single transition** between regular states.

(We have to extend the concept of "single transition" for hierarchical state machines.)

That is: We're going for an interleaving semantics without true parallelism.

Remark: With only methods (later), the notion of step is not so clear.
For example, consider

- c_1 calls `f()` at c_2, which calls `g()` at c_1 which in turn calls `h()` for c_2.
- Is the completion of `h()` a step?
- Or the completion of `f()`?
- Or doesn't it play a role?

It does play a role, because **constraints**/**invariants** are typically (= by convention) assumed to be evaluated at step boundaries, and sometimes the convention is meant to admit (temporary) violation in between steps.

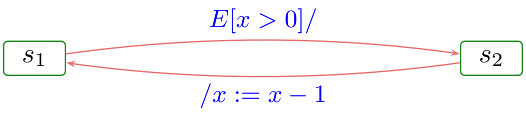
Notions of Steps: The Run-to-Completion Step

What is a **run-to-completion** step...?

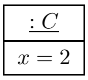
- **Intuition**: a maximal sequence of steps, where the first step is a **dispatch** step and all later steps are **commence** steps.
- **Note**: one step corresponds to one transition in the state machine.

  A run-to-completion step is in general not syntactically definable — one transition may be taken multiple times during an RTC-step.

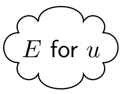
Example:



σ:

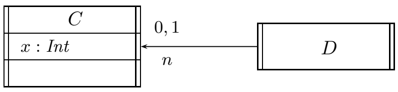


ε:

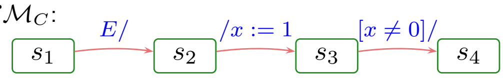


Notions of Steps: The Run-to-Completion Step

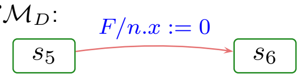
What about this **Example**:



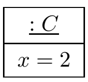
$\mathcal{SM}_C$:



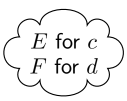
$\mathcal{SM}_D$:



σ:



ε:



Notions of Steps: The Run-to-Completion Step Cont'd

Proposal: Let

$$(\sigma_0, \varepsilon_0) \xrightarrow[u_0]{(cons_0, Snd_0)} \dots \xrightarrow[u_{n-1}]{(cons_{n-1}, Snd_{n-1})} (\sigma_n, \varepsilon_n), \quad n > 0,$$

be a finite (!), non-empty, maximal, consecutive sequence such that

- object u is alive in σ_0,
- $u_0 = u$ and $(cons_0, Snd_0)$ indicates dispatching to u, i.e. $cons = \{(u, \vec{v} \mapsto \vec{d})\}$,
- there are no receptions by u in between, i.e.

$$cons_i \cap \{u\} \times Evs(\mathscr{E}, \mathscr{D}) = \emptyset, i > 1,$$

- $u_{n-1} = u$ and u is stable only in σ_0 and σ_n, i.e.

$$\sigma_0(u)(stable) = \sigma_n(u)(stable) = 1 \text{ and } \sigma_i(u)(stable) = 0 \text{ for } 0 < i < n,$$

Let $0 = k_1 < k_2 < \dots < k_N = n$ be the maximal sequence of indices such that $u_{k_i} = u$ for $1 \le i \le N$. Then we call the sequence

$$(\sigma_0(u) =) \quad \sigma_{k_1}(u), \sigma_{k_2}(u) \dots, \sigma_{k_N}(u) \quad (= \sigma_{n-1}(u))$$

a (!) **run-to-completion computation** of u (from (local) configuration $\sigma_0(u)$).

Divergence

We say, object u **can diverge** on reception $cons$ from (local) configuration $\sigma_0(u)$ if and only if there is an infinite, consecutive sequence

$$(\sigma_0, \varepsilon_0) \xrightarrow{(cons_0, Snd_0)} (\sigma_1, \varepsilon_1) \xrightarrow{(cons_1, Snd_1)} \dots$$

such that u doesn't become stable again.

- **Note**: disappearance of object not considered in the definitions. By the current definitions, it's neither divergence nor an RTC-step.

Run-to-Completion Step: Discussion.

What people may **dislike** on our definition of RTC-step is that it takes a **global** and **non-compositional** view. That is:

- In the projection onto a single object we still **see** the effect of interaction with other objects.
- Adding classes (or even objects) may change the divergence behaviour of existing ones.
- Compositional would be: the behaviour of a set of objects is determined by the behaviour of each object "in isolation".
  Our semantics and notion of RTC-step doesn't have this (often desired) property.

Can we give (syntactical) criteria such that any global run-to-completion step is an interleaving of local ones?

Maybe: **Strict interfaces**. (*Proof left as exercise...*)

- **(A)**: Refer to private features only via "self".
  (Recall that other objects of the same class can modify private attributes.)
- **(B)**: Let objects only communicate by events, i.e.
  don't let them modify each other's local state via links **at all**.

Putting It All Together

The Missing Piece: Initial States

Recall: a labelled transition system is $(S, \to, S_0)$. We **have**

- S: system configurations (σ, ε)
- $\to$: labelled transition relation $(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon')$.

Wanted: initial states S_0.

Proposal:
Require a (finite) set of **object diagrams** $\mathcal{OD}$ as part of a UML model

$$(\mathscr{CD}, \mathscr{SM}, \mathscr{OD}).$$

And set

$$S_0 = \{(\sigma, \varepsilon) \mid \sigma \in G^{-1}(\mathcal{OD}), \mathcal{OD} \in \mathscr{OD}, \varepsilon \text{ empty}\}.$$

Other Approach: (used by Rhapsody tool) multiplicity of classes.
We can read that as an abbreviation for an object diagram.

Semantics of UML Model — So Far

The **semantics** of the **UML model**

$$\mathcal{M} = (\mathscr{CD}, \mathscr{SM}, \mathscr{OD})$$

where

- some classes in $\mathscr{CD}$ are stereotyped as ‘signal’ (standard), some signals and attributes are stereotyped as ‘external’ (non-standard),
- there is a 1-to-1 relation between classes and state machines,
- $\mathscr{OD}$ is a set of object diagrams over $\mathscr{CD}$,

is the **transition system** $(S, \to, S_0)$ constructed on the previous slide.

The **computations of** $\mathcal{M}$ are the computations of $(S, \to, S_0)$.

OCL Constraints and Behaviour

- Let $\mathcal{M} = (\mathscr{CD}, \mathscr{SM}, \mathscr{OD})$ be a UML model.
- We call $\mathcal{M}$ **consistent** iff, for each OCL constraint $expr \in \mathit{Inv}(\mathscr{CD})$,

  $\sigma \models expr$ for each "reasonable point" (σ, ε) of computations of $\mathcal{M}$.

  (Cf. exercises and tutorial for discussion of "reasonable point".)

Note: we could define $\mathit{Inv}(\mathscr{SM})$ similar to $\mathit{Inv}(\mathscr{CD})$.

Pragmatics:

- In **UML-as-blueprint mode**, if $\mathscr{SM}$ doesn't exist yet, then $\mathcal{M} = (\mathscr{CD}, \emptyset, \mathscr{OD})$ is typically asking the developer to provide $\mathscr{SM}$ such that $\mathcal{M}' = (\mathscr{CD}, \mathscr{SM}, \mathscr{OD})$ is consistent.

  If the developer makes a mistake, then $\mathcal{M}'$ is inconsistent.
- **Not common**: if $\mathscr{SM}$ is given, then constraints are also considered when choosing transitions in the RTC-algorithm. In other words: even in presence of mistakes, the $\mathscr{SM}$ never move to inconsistent configurations.

References

References

[Harel and Gery, 1997] Harel, D. and Gery, E. (1997). Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42.

[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.