Contents & Goals

Last Lecture:
- Transitions by Rule (i) to (v).

This Lecture:
- **Educational Objectives**: Capabilities for following tasks/questions.
  - What is a step / run-to-completion step?
  - What is divergence in the context of UML models?
  - How to define what happens at “system / model startup”?
  - What are roles of OCL contraints in behavioural models?
  - Is this UML model consistent with that OCL constraint?
  - What do the actions create / destroy do? What are the options and our choices (why)?

- **Content**:
  - Step / RTC-Step revisited, Divergence
  - Initial states
  - Missing pieces: create / destroy transformer
  - A closer look onto code generation
  - Maybe: hierarchical state machines
Step and Run-to-Completion
Notions of Steps: The Step

**Note:** we call one evolution

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

a **step**.

Thus in our setting, a step directly corresponds to

one object (namely \(u\)) taking a single transition between regular states.

(We will extend the concept of “single transition” for hierarchical state machines.)

**That is:** We’re going for an interleaving semantics without true parallelism.
What is a **run-to-completion** step...?

- **Intuition**: a **maximal** sequence of steps of one object, where the first step is a **dispatch** step, all later steps are **continue** steps, and the last step establishes stability (or object disappears).

**Note**: while 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:**

\[
\begin{align*}
\mathcal{SYC} : & \quad E[x > 0] / \quad [x > 0] / x := x - 1 \\
\sigma : & \quad \begin{array}{c}
u : C \\
x = 2 \\
\end{array} \quad \xrightarrow{u} (\epsilon(w), \sigma) \\
\varepsilon : & \quad E \text{ for } u \quad \xrightarrow{E} (\ii) \\
\end{align*}
\]
Proposal: Let

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

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

- \((cons_0, Snd_0)\) indicates dispatching to \(u := u_0\) (by Rule (ii)) \(\sigma (i)\)
  i.e. \(cons = \{u_E\}, \ u_E \in \text{dom}(\sigma_0) \cap D(\mathcal{E})\),

- if \(u\) becomes stable or disappears, then in the last step, i.e.
  \[\forall i > 0 \bullet (\sigma_i(u)(\text{stable}) = 1 \lor u \notin \text{dom}(\sigma_i)) \implies i = n\]

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

\[(\sigma_0(u) =) \quad \sigma_{k_1}(u), \sigma_{k_2}(u), \ldots, \sigma_{k_N}(u), \sigma_n(u)\]

a (!) run-to-completion step of \(u\) (from (local) configuration \(\sigma_0(u)\) to \(\sigma_n(u)\)).
We say, object \( u \) **can diverge** on reception \( \text{cons}_0 \) from (local) configuration \( \sigma_0(u) \) if and only if there is an **infinite**, consecutive sequence

\[
\left( \sigma_0, \varepsilon_0 \right) \xrightarrow{(\text{cons}_0, \text{Snd}_0)} \left( \sigma_1, \varepsilon_1 \right) \xrightarrow{(\text{cons}_1, \text{Snd}_1)} \ldots
\]

where \( u_i = u \) for infinitely many \( i \in \mathbb{N}_0 \) and \( \sigma_i(u)(\text{stable}) = 0, \ i > 0 \), i.e. \( u \) does not become stable again.
Our definition of RTC-step 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.**

- **(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**.

*(Proof left as exercise...)*
Putting It All Together
Recall: a labelled transition system is $(S, A, \rightarrow, S_0)$. We have

- $S$: system configurations $(\sigma, \varepsilon)$
- $\rightarrow$: labelled transition relation $(\sigma, \varepsilon) \xrightarrow{(cons, Snd)}_{u} (\sigma', \varepsilon')$.

Wanted: initial states $S_0$.

Proposal:

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

$$(C D, S M, \mathcal{OD}).$$

And set

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

Other Approach: (used by Rhapsody tool) multiplicity of classes (plus initialisation code). We can read that as an abbreviation for an object diagram.
The semantics of the UML model

\[ M = (CD, SM, OD) \]

where

- some classes in \( CD \) 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,
- \( OD \) is a set of object diagrams over \( CD \),

is the transition system \((S, A, \rightarrow, S_0)\) constructed on the previous slide(s).

The computations of \( M \) are the computations of \((S, A, \rightarrow, S_0)\).
Let $\mathcal{M} = (\mathcal{CD}, \mathcal{SM}, \mathcal{OD})$ be a UML model.

We call $\mathcal{M}$ consistent iff, for each OCL constraint $\text{expr} \in \text{Inv(\mathcal{CD})} \cup \text{Inv(\mathcal{SM})}$, $\sigma \models \text{expr}$ for each “reasonable point” $(\sigma, \varepsilon)$ of computations of $\mathcal{M}$.

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

**Note:** we could define $\text{Inv(\mathcal{SM})}$ similar to $\text{Inv(\mathcal{CD})}$.

**Diagram:**

```
M1: x > 0
M2: s1

\text{control C: inv: st = s1 imply x > 0}
```
OCL Constraints and Behaviour

• Let $\mathcal{M} = (\mathcal{C} \mathcal{D}, \mathcal{S} \mathcal{M}, \mathcal{O} \mathcal{D})$ be a UML model.
• We call $\mathcal{M}$ consistent iff, for each OCL constraint $expr \in \text{Inv}(\mathcal{C} \mathcal{D})$,

$$\sigma \models expr$$

for each “reasonable point” $(\sigma, \varepsilon)$ of computations of $\mathcal{M}$.

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

Note: we could define $\text{Inv}(\mathcal{S} \mathcal{M})$ similar to $\text{Inv}(\mathcal{C} \mathcal{D})$.

Pragmatics:

• In UML-as-blueprint mode, if $\mathcal{S} \mathcal{M}$ doesn’t exist yet, then $\mathcal{M} = (\mathcal{C} \mathcal{D}, \emptyset, \mathcal{O} \mathcal{D})$ is typically asking the developer to provide $\mathcal{S} \mathcal{M}$ such that $\mathcal{M}' = (\mathcal{C} \mathcal{D}, \mathcal{S} \mathcal{M}, \mathcal{O} \mathcal{D})$ is consistent.

If the developer makes a mistake, then $\mathcal{M}'$ is inconsistent.

Not common: if $\mathcal{S} \mathcal{M}$ is given, then constraints are also considered when choosing transitions in the RTC-algorithm. In other words: even in presence of mistakes, the $\mathcal{S} \mathcal{M}$ never move to inconsistent configurations.
1. Model is "broken"

2. Model behaviour considers constraints, (*) would not be taken if \( x \leq 0 \) after creation.
Last Missing Piece: Create and Destroy Transformer
abstract syntax
create(C, expr, v)

concrete syntax
\texttt{exp}.v := \texttt{new }C^	exttt{}'

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

well-typedness
\[
\begin{align*}
\text{expr} : T_D, & \quad v \in \text{atr}(D), \\
& \quad v \in \mathcal{C}_{0,1}.
\end{align*}
\]
\[
\text{atr}(C) = \{ \langle \texttt{v}_i : T_1, \text{expr}_i^0 \rangle \mid 1 \leq i \leq n \}
\]

semantics
\[
\ldots
\]

observables
\[
\ldots
\]

(error) conditions
\[
I[\text{expr}](\sigma, \beta) \text{ not defined.}
\]

\text{instead}
\[
\texttt{x} := (\texttt{new }C').y + (\texttt{new }D).z;
\]

\text{write}
\[
\begin{align*}
\texttt{temp}_1 & := \texttt{new }C' \\
\texttt{temp}_2 & := \texttt{new }D' \\
\texttt{x} & := \texttt{temp}_1.y + \texttt{temp}_2.z;
\end{align*}
\]
## Transformer: Create

### Abstract Syntax

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create(C, expr, v)</code></td>
<td><code>create(C, expr, v)</code></td>
</tr>
</tbody>
</table>

### Intuitive Semantics

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

### Well-Typedness

$$expr : T_D, \ v \in atr(D),$$

$$atr(C) = \{⟨v_1 : T_1, expr_1^0⟩ | 1 \leq i \leq n\}$$

### 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 since then expressions would need to modify the system state.

- Also for simplicity: no parameters to construction ($\sim$ parameters of constructor). Adding them is straightforward (but somewhat tedious).
How To Choose New Identities?

- **Re-use**: choose any identity that is not alive now, i.e. not in dom(σ).
  - 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 dom(σ) and any predecessor in current run.
  - Depends on history.
  - Dangling references remain dangling – could mask “dirty” effects of platform.
**Transformer: Create**

<table>
<thead>
<tr>
<th>Abstract Syntax</th>
<th>Concrete Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create(C, expr, v)</code></td>
<td><code>create(C, expr, v)</code></td>
</tr>
</tbody>
</table>

**Intuitive Semantics**

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

**Well-Typedness**

$$expr : T_D, \; v \in atr(D),$$

$$atr(C) = \{ \langle v_1 : T_1, expr_0^i \rangle \mid 1 \leq i \leq n \}$$

**Semantics**

$$t_{create}(C, expr, v)[u_x]$$

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

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

$$u_0 = I[expr](\sigma, u_x); \; d_i = I[expr_0^i](\sigma, \emptyset) \text{ if } expr_0^i \neq \emptyset \text{ and arbitrary value from } \mathcal{D}(T_i) \text{ otherwise.}$$

**Observables**

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

**Error Conditions**

$$I[expr](\sigma, u_x) \text{ not defined.}$$
Create Transformer Example

\[ \text{sM}_{D} : \]

\[
\begin{align*}
/n := \text{new } C
\end{align*}
\]

create\((C, \text{expr}, v)\)

\[ t_{\text{create}(C, \text{expr}, v)[u_x]}(\sigma, \varepsilon) = \cdots \]

\[ \mathbb{D}(\text{int}) = \mathbb{Z} \]
abstract syntax  
\textbf{destroy}(expr)

concrete syntax

\begin{tabular}{|l|}
\hline
\textbf{intuitive semantics}  \\
\textit{Destroy the object denoted by expression $expr$.}  \\
\hline
\textbf{well-typedness}  \\
$expr : T_C, C \in \mathcal{C}$  \\
\hline
\textbf{semantics}  \\
$\ldots$  \\
\hline
\textbf{observables}  \\
$\text{Obs}_{\text{destroy}}[u_x] = \{(u_x, \bot, (+, \emptyset), u)\}$  \\
\hline
\textbf{(error) conditions}  \\
$I[expr](\sigma, \beta)$ not defined.  \\
\hline
\end{tabular}
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? (garbage collection)

- 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

<table>
<thead>
<tr>
<th>Abstract Syntax</th>
<th>Concrete Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>destroy(expr)</code></td>
<td></td>
</tr>
</tbody>
</table>

### Intuitive Semantics

Destroy the object denoted by expression `expr`.

### Well-Typedness

\[ expr : T_C, \ C \in \mathcal{C} \]

### Semantics

\[
\begin{align*}
 t_{\text{destroy}(expr)}[ux](\sigma, \varepsilon) &= \{(\sigma', \varepsilon')\}, \\
 \text{where } \sigma' &= \sigma |_{\text{dom(\sigma)\{u\}}} \text{ with } u = I[expr](\sigma, ux).
\end{align*}
\]

### Observables

\[
\begin{align*}
 \text{Obs}_{\text{destroy}(expr)}[ux] &= \{(+, u)\}
\end{align*}
\]

### (Error) Conditions

\[
I[expr](\sigma, ux) \text{ not defined.}
\]
Hierarchical State-Machines
UML distinguishes the following kinds of states:

<table>
<thead>
<tr>
<th>State Type</th>
<th>Example</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple state</td>
<td><img src="image" alt="Simple State" /></td>
<td>pseudo-state</td>
</tr>
<tr>
<td>final state</td>
<td><img src="image" alt="Final State" /></td>
<td>initial</td>
</tr>
<tr>
<td>composite state</td>
<td><img src="image" alt="Composite State" /></td>
<td>(shallow) history</td>
</tr>
<tr>
<td>OR</td>
<td><img src="image" alt="OR State" /></td>
<td>deep history</td>
</tr>
<tr>
<td>AND</td>
<td><img src="image" alt="AND State" /></td>
<td>fork/join</td>
</tr>
<tr>
<td></td>
<td></td>
<td>junction, choice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>entry point</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exit point</td>
</tr>
<tr>
<td></td>
<td></td>
<td>terminate</td>
</tr>
<tr>
<td>submachine state</td>
<td><img src="image" alt="Submachine State" /></td>
<td>terminate</td>
</tr>
</tbody>
</table>

**pseudo-state**

- initial
- (shallow) history
- deep history
- fork/join
- junction, choice
- entry point
- exit point
- terminate

**submachine state**

- ![Submachine State](image)
References
References
