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
Notions of Steps: The Step

Note: we call one evolution

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

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:**

\[ \begin{align*} 
\sigma_0 & \xrightarrow{\epsilon} \sigma_1 \\
\sigma_1 & \xrightarrow{\epsilon} \sigma_2 \\
\sigma_2 & \xrightarrow{\epsilon} \sigma_3 \\
\sigma_3 & \xrightarrow{\epsilon} \sigma_4 \\
\sigma_4 & \xrightarrow{\epsilon} \sigma_5 \\
\sigma_5 & \xrightarrow{\epsilon} \sigma_6 \\
\sigma_6 & \xrightarrow{\epsilon} \sigma_7 \\
\end{align*} \]

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

**Proposal**: Let

\[(\sigma_0, \varepsilon_0) \xrightarrow{(\text{cons}_0, \text{Snd}_0)} u_0 \xrightarrow{\epsilon_1} \ldots \xrightarrow{(\text{cons}_{n-1}, \text{Snd}_{n-1})} u_{n-1} \xrightarrow{(\sigma_n, \varepsilon_n)} u_n, \quad n > 0,\]

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

- \((\text{cons}_0, \text{Snd}_0)\) indicates dispatching to \(u := u_0\) (by Rule (ii)) \(\sigma(u) \in \text{dom}(\sigma) \cap \mathcal{G}(E)\),

- if \(u\) becomes stable or disappears, then in the last step, i.e.

\[ \forall i > 0 \bullet (\sigma_i(u)(\text{stable}) = 1 \lor u \not\in \text{dom}(\sigma_i)) \implies i = n \]

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

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

a (!) run-to-completion step of \(u\) (from (local) configuration \(\sigma_0(u)\) to \(\sigma_n(u)\)).
Divergence

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

$$(\sigma_0, \varepsilon_0) \xrightarrow{\text{cons}_0, \text{Snd}_0} (\sigma_1, \varepsilon_1) \xrightarrow{\text{cons}_1, \text{Snd}_1} \cdots$$

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.

Run-to-Completion Step: Discussion.

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. (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

Initial States

Recall: a labelled transition system is \((S, A, \rightarrow, S_0)\). We have

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

Wanted: initial states \(S_0\).

Proposal:
Require a (finite) set of object diagrams \(\mathcal{OD}\) as part of a UML model
\((\mathcal{CD}, \mathcal{SM}, \mathcal{OD})\).

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

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

\[ M = (C_{\mathcal{P}}, \mathcal{M}, \mathcal{O}) \]

where

- some classes in \( C_{\mathcal{P}} \) 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,
- \( \mathcal{O} \) is a set of object diagrams over \( C_{\mathcal{P}} \),

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

The computations of \( M \) are the computations of \((S, A, \rightarrow, S_0)\).

OCL Constraints and Behaviour

- Let \( M = (C_{\mathcal{P}}, \mathcal{M}, \mathcal{O}) \) be a UML model.
- We call \( M \) consistent iff, for each OCL constraint \( expr \in \text{Inv}(C_{\mathcal{P}}) \cup \text{Inv}(\mathcal{M}) \)
  \[ \sigma \models expr \]
  for each "reasonable point" \((\sigma, \varepsilon)\) of computations of \( M \).

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

Note: we could define \( \text{Inv}(\mathcal{M}) \) similar to \( \text{Inv}(C_{\mathcal{P}}) \).
OCL Constraints and Behaviour

- Let $M = (CD, SM, OD)$ be a UML model.
- We call $M$ consistent iff, for each OCL constraint $expr \in Inv(CD)$,
  \[
  \sigma \models expr \text{ for each "reasonable point" } (\sigma, \varepsilon) \text{ of computations of } M.
  \]
  (Cf. exercises and tutorial for discussion of "reasonable point".)

**Note:** we could define $Inv(SM)$ similar to $Inv(CD)$.

**Pragmatics:**
- In **UML-as-blueprint mode**, if $SM$ doesn’t exist yet, then $M = (CD, \emptyset, OD)$ is typically asking the developer to provide $SM$ such that $M' = (CD, SM, OD)$ is consistent. If the developer makes a mistake, then $M'$ is inconsistent.
- **Not common**: if $SM$ is given, then constraints are also considered when choosing transitions in the RTC-algorithm. In other words: even in presence of mistakes, the $SM$ never move to inconsistent configurations.
Abstract Syntax
create(C, expr, v)

Concrete Syntax
expr := 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 : T_D, v ∈ atr(D), v : C_{expr}, atr(C) = \{(v_i : T_i, expr_{expr}) | 1 \leq i \leq n\}

Semantics

Observables

Error Conditions
I[expr](σ, β) not defined.

Instead
x := (true c).y + (true d).z;

P₁
lcp₁ := new c₁;
lcp₂ := new d₁;
x := lcp₁.y + lcp₂.z;
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></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**

\[
\text{expr} : T_D, \ v \in \text{atr}(D), \ \text{atr}(C) = \{ (v_1 : T_1, \text{expr}_0^1) \mid 1 \leq i \leq n \}
\]

**semantics**

\[
\ldots
\]

**observables**

\[
\ldots
\]

**error conditions**

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

- 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 (~ 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 \( \text{dom}(\sigma) \).
  - 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 \( \text{dom}(\sigma) \) and any predecessor in current run.
  - Depends on history.
  - Dangling references remain dangling – could mask “dirty” effects of platform.
Create Transformer Example

\[SM_D:\]
\[
\begin{array}{c}
\texttt{s}_1 \\
\text{/n := new } C \\
\texttt{s}_2
\end{array}
\]

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

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

\[
\sigma: \\
\begin{array}{c}
d: D \\
n = \emptyset \\
\text{let } y = \emptyset \text{ in } z \\
\end{array}
\]

\[
\epsilon: \\
\begin{array}{c}
\text{let } \epsilon' \text{ in } z \\
\end{array}
\]

\[
\sigma': \\
\begin{array}{c}
d: D \\
n = \emptyset \\
\text{let } y = \emptyset \text{ in } z \\
\end{array}
\]
abstract syntax

destroy(expr)

concrete syntax

intuitive semantics

Destroy the object denoted by expression expr.

well-typedness

expr : T_C, C ∈ ℭ

semantics

observables

\[ \text{Obs}_{\text{destroy}}[u_x] = \{(u_x, \bot, (+, \emptyset), u)\} \]

(error) conditions

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

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 ∈ ℂ`

**Semantics**

\[
t_{destroy}(expr)[u_x](σ, ε) = \{(σ', ε)\}, \quad ε \in \llbracket ε \rrbracket(ε)
\]

where \(σ' = σ|_{\text{dom}(σ) \setminus \{u\}}\) with \(u = I[expr](σ, u_x)\).

**Observables**

\[
\text{Obs}_{destroy(expr)}[u_x] = \{(+, u)\}
\]

**Error conditions**

\(I[expr](σ, u_x)\) not defined.

---

**Hierarchical State-Machines**
UML distinguishes the following kinds of states:

- **simple state**
  - entry/act
  - exit/act

- **final state**

- **composite state**
  - OR
  - AND

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

- **submachine state**

References
References
