Contents & Goals

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

This Lecture:
• Educational Objectives:
  • 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 constraints 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 \((\sigma, \varepsilon)\) \((\text{cons}, \text{Snd})\) \(-\rightarrow\) \((\sigma', \varepsilon')\) 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.

Notions of Steps: The Run-to-Completion Step

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:
\[
\begin{align*}
  s_1 &\rightarrow E [x > 0] / \sigma_0 : C x = 2 \\
  s_2 &\rightarrow E [x > 0] / \epsilon_0 : E \text{ for } u
\end{align*}
\]

Proposal: Let \((\sigma_0, \epsilon_0)\) \((\text{cons}_0, \text{Snd}_0)\) \(-\rightarrow\) \(u_0\).

. . . \((\text{cons}_{n-1}, \text{Snd}_{n-1})\) \(-\rightarrow\) \(u_{n-1}\) \((\sigma_n, \epsilon_n))\), \(n > 0\),

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)), i.e. \(\text{cons} = \{ u_E \}\), \(u_E \in \text{dom}(\sigma_0) \cap \text{D}(E)\)

• if \(u\) becomes stable or disappears, then in the last step, i.e. \(\forall i > 0 \cdot (\sigma_i(u)(\text{stable}) = 1 \lor u / \in \text{dom}(\sigma_i)) = \Rightarrow i = n\)

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

Then we call the sequence \((\sigma_0(u)) =) \sigma_{k_1}(u), \sigma_{k_2}(u), \ldots, \sigma_{k_N}(u), \sigma_n(u)\) 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 $\sigma_0$ from (local) configuration $\sigma_0(u)$ if and only if there is an infinite, consecutive sequence $(\sigma_0, \epsilon_0)(\text{cons}_0, S_{\text{snd}}_0)\rightarrow u_0(\sigma_1, \epsilon_1)(\text{cons}_1, S_{\text{snd}}_1)\rightarrow \ldots$ 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.

**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”.
  - *(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**

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

- $S$: system configurations $(\sigma, \epsilon)$
- $\rightarrow$: labelled transition relation $(\sigma, \epsilon)\rightarrow u(\sigma', \epsilon')$.

**Wanted**: initial states $S_0$.

**Proposal**: Require a (finite) set of object diagrams $OD$ as part of a UML model $(C_D, S_M, OD)$. And set $S_0 = \{ (\sigma, \epsilon) | \sigma \in G^{-1}(OD), OD \in OD, \epsilon = \text{empty} \}$.

**Other Approach**: (used by Rhapsody tool) multiplicity of classes (plus initialisation code).

We can read that as an abbreviation for an object diagram.

**Semantics of UML Model (So Far)**

The semantics of the UML model $M = (C_D, S_M, OD)$ where:

- some classes in $C_D$ 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 $C_D$, 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 $M = (C, D, S, M, O, D)$ be a UML model.

We call $M$ consistent iff, for each OCL constraint $expr \in \text{Inv}(C, D)$, $\sigma | = expr$ for each "reasonable point" $(\sigma, \epsilon)$ of computations of $M$.

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

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

Pragmatics:

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

If the developer makes a mistake, then $M'$ is inconsistent.

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

---

Transformer: Create

abstract syntax concrete syntax

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 $expr: T \rightarrow D, v \in \text{atr}(D)$, $\text{atr}(C) = \{\langle v_1: T_1, expr_0 \rangle | 1 \leq i \leq n\}$

semantics . . . observables . . . (error) conditions $I/llbracket expr/rrbracket(\sigma, \beta)$ 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 $\text{dom}(\sigma)$.

• 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.
Valid proposal for simple analysis: monotone frame semantics, no destruction at all. But provide garbage collection — and models shall (in general) be correct without assumptions. This is in line with "expect the worst", because there are target platforms which don't destroy $x$, $u$ or remove $\sigma$ alone? (garbage collection)

$\sigma, \beta \vdash [I] = [I]$, $u$ may have been the last one linking to object $C \in$ observables

$\sigma \epsilon ''$ and $\sigma, \beta \vdash [I] = [I]$, $u$ may still refer to it via association

Assume object $\sigma, \beta \vdash [I] = [I]$, $u$ may still refer to it via association

Create an object of class $\sigma, \beta \vdash [I] = [I], u$ create

Destroy the object denoted by expression $\sigma, \beta \vdash [I] = [I], u$ destroy

Transformer: Create
UML distinguishes the following kinds of states:

- **Simple States**
  - **Entry**
  - **Do**
  - **Exit**

- **Composite States**
  - **AND**
  - **OR**

- **Pseudo-States**
  - **Initial**
  - **Terminate**

- **References**
  - **(Shallow) History**
  - **Deep History**
  - **Fork/Join**, **Junction**, **Choice**
  - **Entry Point**
  - **Exit Point**

References: