The order of dequeuing is run-to-completion step \( n \in \{ c_{01}, c_{02}, \ldots \} \), \( n \in (\cdots, 1, 0) \)

- **Def.:** \( E_n \) : effects of actions
- **Def.:** \( \text{cons} = (((S_{\text{CD}} \cdot \text{ModelInstances}), A_{\text{DS}}), M_{\text{CD}}) \)
- **Def.:** \( N, E, f = (G_{\text{UML}}, T, S_{\text{SM}}) \)
- **Def.:** \( n = (n_{\text{CD}}, A_{\text{DS}}, M_{\text{CD}}) \)
- **Def.:** \( u_{\text{1st}} = 1 \)
- **Def.:** \( u_{\text{2nd}} = 2 \)
- **Def.:** \( D_{\text{1st}} \cdot u_{\text{1st}} = 28 \)
- **Def.:** \( D_{\text{1st}} \cdot u_{\text{2nd}} = 28 \)
- **Def.:** \( E_{\text{1st}} \cdot st = 28 \)
- **Def.:** \( E_{\text{2nd}} \cdot st = 28 \)

The semantics of event occurrence process is based on the run-to-completion step. The run-to-completion step in a configuration with all stable state \( \text{cons} = (\text{ModelInstances}, A_{\text{DS}}, M_{\text{CD}}) \) leaving open the possibility of modeling different priority-based schemes.
• The order of dequeuing is not defined, leaving open the possibility of modeling different priority-based schemes.

Ether and OMG (2011b) – 11 – 2016-12-08 – Sether – 7/34

The standard distinguishes (among others)
• SignalEvent (OMG, 2011b, 450) and Reception (OMG, 2011b, 447).

On SignalEvents, it says
A signal event represents the receipt of an asynchronous signal instance. A signal event may, for example, cause a state machine to trigger a transition. (OMG, 2011b, 449)

Semantic Variation Points
The means by which requests are transported to their target depend on the type of requesting action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is instantaneous and completely reliable while in others it may involve transmission delays of variable duration, loss of requests, reordering, or duplication. (See also the discussion on page 421.)

Our ether (→ in a minute) is a general representation of many possible choices. Often seen minimal requirement: order of sending by one object is preserved.

Example: FIFO Queue
A (single, global, shared, reliable) FIFO queue is an ether:

• Eth = D(C) × D(E)* the set of finite sequences of pairs (u, e) ∈ D(C) × D(E)

• ready: Eth × D(C) → 2D(E) ((u1, e).ε, u2) ↦ { (u1, e) } if u1 = u2 ∅ otherwise

• ⊕: Eth × D(C) × D(E) → Eth (ε, u, e) ↦ ε.

• ⊖: Eth × D(E) → Eth (ε., u, e1), e2) ↦ { ε., (u, e1) } if e2 = e1 ε., (u, e1) otherwise

• [·]: Eth × D(C) → Eth remove all (u, e) from ε.
We call an object \( \sigma \) stable in system configuration \( S \) if and only if for each \( \varepsilon \) over \( S \), and each \( \mathcal{E} \in \mathcal{E}(\varepsilon) \), we have

\[
\mathcal{E}(\varepsilon) = \mathcal{E}(\varepsilon) \cap \sigma \text{ dom}(\varepsilon).
\]

System Configuration Step-by-Step

System Configuration: Example
In the following, we assume that we're going back and forth between Java, C++, and other statements (plus some event send action).

The set of the guard is from $I$ action source because of dangling-reference navigation or division-by-zero. We want to go to a transformer that evaluates expressions in a given system configuration, $\sigma, \epsilon$.

An observation $\mathcal{V}$, $\mathcal{C}$ for object $\mathcal{D}$.

Examples

A Simple Action Language

Examples

Java, C++, . . . statements (plus some event send action)

Java, C++, . . . expressions

Java, C++, . . .<

Java, C++, . . .>
\[
\text{Obs: } \sigma \in \mathcal{E} \setminus \mathcal{D}_1 \times \mathcal{E} \setminus \mathcal{D}_2 \text{ or } \sigma = (\sigma, \epsilon) \text{ not defined.}\]

\[
\text{Clear } x = (\sigma, u) \text{ or } \text{update } x = (\sigma, \epsilon) \text{ do nothing.}\]

\[
\text{Sequential composition: } t_1 \circ t_2 = (\sigma, u) \text{ not defined.}\]

\[
\text{Transformer: Skip}\]

\[
\text{Transformer: Send}\]

\[
\text{Extended Computation of Transformers}\]

\[
\text{Well-typedness of Transformers}\]
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 (later),

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.

Tell Them What You've Told Them...