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
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

**Definition.** Let $\Sigma_{\mathcal{P}}$ the set of system configurations over some $\mathcal{P}_0$, $\mathcal{D}_0$ and $\mathcal{E}$ and ether. We call a relation $t \subseteq \mathcal{D}(\mathcal{P}) \times (\Sigma_{\mathcal{P}} \times \mathcal{E}) \times (\Sigma_{\mathcal{P}} \times \mathcal{E})$ a (system configuration) transformer.

- In the following, we assume that each application of a transformer $t$ to some system configuration $(\sigma, \varepsilon)$ for object $u_x$ is associated with a set of observations:

  $$\text{Obs}_t[u_x](\sigma, \varepsilon) \in 2^{\mathcal{P}(\mathcal{P}) \times \mathcal{D}(\mathcal{E}) \cup \{\ast, +\} \times \mathcal{D}(\mathcal{P})}.$$

- An observation $(u_{\text{src}}, (E, \vec{d}), u_{\text{dst}}) \in \text{Obs}_t[u_x](\sigma, \varepsilon)$ represents the information that, as a “side effect” of $u_x$ executing $t$, an event $(E, \vec{d})$ has been sent from object $u_{\text{src}}$ to object $u_{\text{dst}}$.

  **Special cases:** creation/destroyation.

**Why Transformers?**

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

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

- **Clear:** $(\text{event})$ is from $\mathcal{E}$ of the corresponding signature.

- **But:** What are $(\text{guard})$ and $(\text{action})$?

  - UML can be viewed as being parameterized in expression language (providing $(\text{guard})$) and action language (providing $(\text{action})$).

  **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,

and that we’re given
- a semantics for boolean expressions in form of a partial function
  \[
  I[\cdot \cdot] : (\cdot \cdot) : Expr \rightarrow ((\Sigma_{\cdot \cdot} \times (\{\cdot\} \rightarrow \mathcal{P}(\cdot))) \rightarrow \mathbb{B})
  \]
  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
  \[
  t_{act} \subseteq \mathcal{P}(\cdot) \times (\Sigma_{\cdot \cdot} \times Eth) \times (\Sigma_{\cdot \cdot} \times Eth).
  \]

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 “⊥”.
- 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 $\varepsilon$ — interesting, because state machines are built around sending/consuming events
- create/destroy: modify domain of $\sigma$ — 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

\[
\text{Act}_\text{op} := \{ \text{step} \}
\]

\[
\begin{align*}
\forall \sigma, \varepsilon \in \text{OpExp}, \sigma', \varepsilon' \in \text{OpExp} \land \sigma, \varepsilon \in \text{OpExp} \land \sigma', \varepsilon' \in \text{OpExp} \\
\text{update}(\sigma', \varepsilon, \sigma_1, \varepsilon_1, \sigma_2, \varepsilon_2) \in \text{OpExp} \\
\text{send}(\sigma, \varepsilon, \sigma_1, \varepsilon_1, \sigma_2, \varepsilon_2) \in \text{OpExp} \\
\text{create}(\sigma, \varepsilon, \varepsilon_1) \in \text{OpExp} \\
\text{destroy}(\sigma) \in \text{OpExp}
\end{align*}
\]
### Transformer: Skip

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>skip</code></td>
<td><code>skip</code></td>
</tr>
</tbody>
</table>

**intuitive semantics**: *do nothing*

**well-typedness**: `./.`

**semantics**

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

**observables**

\[
 \text{Obs}_{skip}[u_x](\sigma, \varepsilon) = \emptyset
\]

**error conditions**

### Transformer: Update

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>update(expr_1, v, expr_2)</code></td>
<td><code>expr_1.v := expr_2</code></td>
</tr>
</tbody>
</table>

**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 \text{atr}(C)`;
- `expr_2 : \tau`;
- `expr_1`, `expr_2` obey visibility and navigability

**semantics**

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

where

\[
 \sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[expr_2](\sigma, \beta)]]
\]

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

**observables**

\[
 \text{Obs}_{\text{update}}(expr_1, v, expr_2)[u_x] = \emptyset
\]

**error conditions**

*Not defined if \(I[expr_1](\sigma, \beta)\) or \(I[expr_2](\sigma, \beta)\) not defined.*
**Update Transformer Example**

\[ SMC: \]

\[ s_1 \quad /x := x + 1 \quad s_2 \]

\[
\text{update}(\text{expr}_1, v, \text{expr}_2)
\]

\[
t_{\text{update}}(\text{expr}_1, v, \text{expr}_2)[u_2](\sigma, \varepsilon) = (\sigma[u \mapsto \sigma(u)[v \mapsto I[\text{expr}_2](\sigma, \beta)]], \varepsilon),
\]

\[
u = I[\text{expr}_1](\sigma, \beta)
\]

\[ \sigma: \]

\[
\begin{array}{l}
  u_1: C \\
  x = 4 \\
  y = 0
\end{array}
\]

\[
\begin{array}{l}
  : \sigma' \\
  x = 5 \\
  y = 0
\end{array}
\]

\[ \varepsilon: \]

\[
\begin{array}{l}
  I \times y \times D(\tau, i, y, u_3) = 5
\end{array}
\]

**Transformer: Send**

**abstract syntax**

\[
\text{send}(E(\text{expr}_1, ..., \text{expr}_n), \text{expr}_{\text{dst}})
\]

**concrete syntax**

\[
\text{expr}_{\text{dst}}(E(\text{expr}_1, ..., \text{expr}_n))
\]

**intuitive semantics**

Object \( u_2: C \) sends event \( E \) to object \( \text{expr}_{\text{dst}} \), i.e. create a fresh signal instance, fill in its attributes, and place it in the ether.

**well-typedness**

\[
\text{expr}_{\text{dst}}: \tau_D, C, D \in \mathcal{C}; E \in \mathcal{E}; \text{atr}(E) = \{v_1: \tau_1, ..., v_n: \tau_n\};
\]

\[
\text{expr}_i: \tau_i, 1 \leq i \leq n;
\]

all expressions obey visibility and navigability in \( C \)

**semantics**

\[
t_{\text{send}}(E(\text{expr}_1, ..., \text{expr}_n), \text{expr}_{\text{dst}})[u_2](\sigma, \varepsilon) \Rightarrow (\sigma', \varepsilon')
\]

where \( \sigma' = \sigma \cup \{u \mapsto \{v_i \mapsto d_i, \quad 1 \leq i \leq n\}\}; \quad \varepsilon' = \varepsilon \oplus (u_{\text{dst}}, u);
\]

if \( u_{\text{dst}} = I[\text{expr}_{\text{dst}}]([\sigma, \beta] \in \text{dom}(\sigma)); \quad d_i = I[\text{expr}_i]([\sigma, \beta] \text{ for } 1 \leq i \leq n);
\]

\[ u \in \mathcal{D}(E) \text{ a fresh identity, i.e. } u \notin \text{dom}(\sigma), \]

and where \( (\sigma', \varepsilon') = (\sigma, \varepsilon) \text{ if } u_{\text{dst}} \notin \text{dom}(\sigma); \beta = \{u_{\text{dst}} \mapsto u_{\text{dst}}\}\]

**observables**

\[
\text{Obs}_{send}[u_2] = \{(u_2, (E, d_1, ..., d_n), u_{\text{dst}})\}
\]

**error conditions**

\[ I[\text{expr}]([\sigma, \beta]) \text{ not defined for any } \text{expr} \in \{\text{expr}_{\text{dst}}, \text{expr}_1, ..., \text{expr}_n\}\]
Send Transformer Example

\[ SMC: \]

\[ s_1 \to \ldots \to !F(x + 1); \ldots \to s_2 \]

\[ \text{send}(E(expr_1, \ldots, expr_n), expr_{dst}) \]

\[ t_{\text{send}}(expr_{src}, E(expr_1, \ldots, expr_n), expr_{dst})(\sigma, \epsilon) = \ldots \]

\[ \sigma: \]

\[ \begin{array}{c}
  u_1 : C \\
  x = 5
\end{array} \]

\[ \varepsilon: \]

\[ \begin{array}{c}
  u_2 : C \\
  x = 13
\end{array} \]

Transformer: Create

abstract syntax

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

concrete syntax

\[ expr.v := \text{new}(C) \]

intuitive semantics

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

well-typedness

\[ expr : \tau_D, v \in \text{atr}(D), \text{atr}(C) = \{(v_i : \tau_i, expr_i^0) | 1 \leq i \leq n\} \]

semantics

\[ \ldots \]

observables

\[ \ldots \]

(error) conditions

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

- 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 such as order of evaluation (and thus creation).
- Also for simplicity: no parameters to construction (∼ parameters of constructor). Adding them is straightforward (but somewhat tedious).
Create Transformer Example

\[S\mathcal{M}_C:\]

\[... ; n := \text{new } C ; ...\]

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

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

\[\sigma:\]

\[
\begin{array}{c}
  d : D \\
  n = \emptyset
\end{array}
\]

\[\varepsilon:\]

\[\sigma':\]

\[\varepsilon':\]

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.
### 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>expr = new(C)</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**

\[
equiv{expr}{\tau_D, v \in \text{atr}(D), \text{atr}(C) = \{\langle v_1 : \tau_1, expr_1 \rangle | 1 \leq i \leq n\}}
\]

**semantics**

\[
\text{if } \equiv{\sigma, \varepsilon}{t} \text{ then } \equiv{\sigma, \varepsilon'}{t}
\]

\[
u_0 = I[expr](\sigma, \beta); d_i = I[expr_i](\sigma, \beta) \text{ if } \equiv{expr_i}{''} \text{ and arbitrary value from } \mathcal{D}(\tau_i) \text{ otherwise}; \beta = \{\text{this } \mapsto u_x\}.
\]

**observables**

\[
\text{Obs}_{create}[u_x] = \{ (u_x, (\ast, \emptyset), u) \}
\]

**error conditions**

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

---

### Transformer: Destroy

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>destroy(expr)</code></td>
<td><code>...</code></td>
</tr>
</tbody>
</table>

**intuitive semantics**

Destroy the object denoted by expression `expr`.

**well-typedness**

\[
equiv{expr}{\tau_C, C \in \mathcal{E}}
\]

**semantics**

... 

**observables**

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

**error conditions**

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

\[ SM_C: \]

\[ s_1 \quad / \ldots; \text{delete} \quad n; \ldots \quad s_2 \]

\[ \text{destroy} (expr) \]

\[ t_{\text{destroy} (expr)} [u_2] (\sigma, \varepsilon) = \ldots \]

\[ \sigma': \]

\[ \varepsilon': \]

---

**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.

---

\[ \sigma': \]

\[ \varepsilon': \]
### Transformer: Destroy

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>destroy(expr)</td>
<td></td>
</tr>
</tbody>
</table>

**Intuitive semantics**

Destroy the object denoted by expression `expr`.

**Well-typedness**

`expr : τ_C, C ∈ C'`

**Semantics**

\[ t[u_x](σ, ε) = (σ', ε) \]

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

**Observables**

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

**Error conditions**

\( I[expr](σ, β) \) not defined.

---

### Sequential Composition of Transformers

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

\[
(t_2 \circ t_1)[u_x](σ, ε) = t_2[u_x](t_1[u_x](σ, ε))
\]

with observation

\[
\text{Obs}_{(t_2 \circ t_1)}[u_x](σ, ε) = \text{Obs}_{t_1}[u_x](σ, ε) \cup \text{Obs}_{t_2}[u_x](t_1(σ, ε)).
\]

- **Clear**: not defined if one the two intermediate “micro steps” is not defined.
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.
Transition Relation, Computation

Definition. Let $A$ be a set of actions and $S$ a (not necessarily finite) set 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 \( \mathcal{S}_0 = (\mathcal{P}, \mathcal{C}_0, V_0, atr_0, \Delta_0) \) be a signature with signals (all classes active), \( \mathcal{R}_0 \) a structure of \( \mathcal{S}_0 \), and \( (E\text{th}, \text{ready}, \oplus, \ominus, \lbrack brack) \) an ether over \( \mathcal{S}_0 \) and \( \mathcal{R}_0 \). Assume there is one core state machine \( M_C \) per class \( C \in \mathcal{C} \).

We say, the state machines induce the following labelled transition relation on states \( S := \Sigma \cup \{\#\} \) with actions \( A := 2^{\mathcal{P}(\mathcal{C})} \times \text{Evs}(\mathcal{E}, \mathcal{P}) \times 2^{\mathcal{P}(\mathcal{C})} \times \text{Evs}(\mathcal{E}, \mathcal{P}) \times \mathcal{P}(\mathcal{C}) \):

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

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 \( u \),

\[
s \xrightarrow{\text{cons,} \#} \#
\]

if and only if

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

### (i) Discarding An Event

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

if

- an \( E \)-event (instance of signal \( E \)) is ready in \( \varepsilon \) for \( u \) object of a class \( C \), i.e.
  \[\exists u \in \text{dom}(\sigma) \cap \mathcal{P}(C) \exists u_E \in \mathcal{P}(\mathcal{E}) : u_E \in \text{ready}(\varepsilon, u)\]
- \( u \) is stable and in state machine state \( s \), i.e. \( \sigma(u)(\text{stable}) = 1 \) and \( \sigma(u)(\text{st}) = s \),
- 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)

\[
\forall (s, F, \text{expr}, \text{act}, s') \in \rightarrow(\mathcal{S}M_C) : F \neq E \lor I[\text{expr}]\sigma = 0
\]

and

- the system configuration doesn’t change, i.e. \( \sigma' = \sigma \)
- the event \( u_E \) is removed from the ether, i.e.
  \[\varepsilon' = \varepsilon \oplus u_E,\]
- consumption of \( u_E \) is observed, i.e.
  \[\text{cons} = \{(u, (E, \sigma(u_E)))\}, \text{Snd} = \emptyset.\]
Example: Discard

\[ x > 0 \text{ if } x := x - 1; n! J \]

\[ G[x > 0]/x := y \]

\[ H/z := y/x \]

\[ c : C \]

\[ x = 1, z = 0, y = 2 \]

\[ st = s_1 \]

\[ stable = 1 \]

\[ \varepsilon : \{ \text{for } c, G \text{ for } c \} \]

\[ (\sigma, \varepsilon) \]

\[ (\sigma', \varepsilon') \]

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

\[ \exists u \in \text{dom}(\sigma) \cap \mathcal{D}(C) \]

\[ \exists u_E \in \mathcal{D}(\mathcal{E}) : u_E \in \text{ready}(\varepsilon, u) \]

\[ \forall (s, F, \text{expr}, \text{act}, s') \in \rightarrow (SM_C) : \]

\[ F \neq E \lor I[\text{expr}] (\sigma) = 0 \]

\[ \sigma'(\text{stable}) = 1, \sigma(u)(\text{stable}) = 1, \sigma(u)(st) = s, \]

\[ \sigma(u)(\text{stable}) = 1 \text{ and } \sigma(u)(st) = s, \]

\[ \text{a transition is enabled, i.e.} \]

\[ \exists (s, F, \text{expr}, \text{act}, s') \in \rightarrow (SM_C): F = E \land I[\text{expr}] (\sigma) = 1 \]

\[ \text{where } \sigma = \sigma[u.\text{params}_E \mapsto u_e]. \]

\[ \left( \sigma'_n, \varepsilon'_t \right) = t_{\text{act}} (\sigma, \varepsilon \ominus u_E), \]

\[ \sigma'_t = (\sigma''[u.st \mapsto s', u.\text{stable} \mapsto b, u.\text{params}_E \mapsto \emptyset]) |_{\mathcal{D}(\varepsilon)} \setminus \{u_E\} \]

\[ \text{where } b \text{ depends:} \]

\[ \text{If } u \text{ becomes stable in } s', \text{ then } b = 1. \text{ It does become stable if and only if there is no transition without trigger enabled for } u \text{ in } (\sigma', \varepsilon'). \]

\[ \text{Otherwise } b = 0. \]

\[ \text{Consumption of } u_E \text{ and the side effects of the action are observed, i.e.} \]

\[ \text{cons} = \{(u, (E, \sigma(u_E)))\}, \text{Snd} = \text{Obs}_{\text{act}} (\tilde{\sigma}, \varepsilon \ominus u_E). \]
Example: Dispatch

\[ x > 0 \vdash x := x - 1; \text{n}! \text{J} \]

SMC:

\[ s_1 \xrightarrow{G \geq 0} x := y \]

\[ H \geq y / x \]

\[ s_2 \]

\[ n \]

\[ 0.1 \]

\[ x, z : \text{Int} \]

\[ y : \text{Int} \langle \text{env} \rangle \]

\[ n_0, n_1 \]

\[ \sigma : C \]

\[
\begin{array}{c}
\text{x = 1, z = 0, y = 2} \\
\text{st} = s_1 \\
\text{stable} = 1
\end{array}
\]

\[ \varepsilon : G \text{ for } c \]

\[ (G, \varepsilon) \]

\[ (G, \varepsilon) \]

\[ \varepsilon' \]

\[ \sigma' \]

\[ \sigma'' \]

\[ \text{cons} = \emptyset, \text{Snd} = \text{Obs}_{\text{act}}(\sigma, \varepsilon) \]

(iii) Commence Run-to-Completion

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

if

- there is an unstable object of a class \( \mathcal{C} \), i.e.

\[ \exists u \in \text{dom}(\sigma) \cap \mathcal{C} \subseteq \mathcal{D}: \sigma(u)(\text{stable}) = 0 \]

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

\[ \exists (s, \_ \text{ expr, act, s'}) \in \to (\text{SMC}) : F = E \land \bigwedge \text{expr}(\delta) = 1 \]

\[ \delta = \sigma[u, \text{params}_E \mapsto u_E]. \]

\[ \sigma(u)(\text{stable}) = 1, \sigma(u)(\text{st}) = s, \]

\[ (\sigma'', \varepsilon') = \text{t}_{\text{act}}(\delta, \varepsilon \oplus u_E) \]

\[ \sigma' = \sigma''[u, \text{st} \mapsto s', u, \text{stable} \mapsto b, u, \text{params}_E \mapsto u_E](\varepsilon) \]

\[ \text{cons} = (\langle n, (E, \sigma(u_E)) \rangle), \text{Snd} = \text{Obs}_{\text{act}}(\delta, \varepsilon \oplus u_E) \]

and

- \( (\sigma', \varepsilon') \) results from applying \( \text{t}_{\text{act}} \) to \( (\sigma, \varepsilon) \), i.e.

\[ (\sigma', \varepsilon') \in \to \text{t}_{\text{act}}(\sigma, \varepsilon), \quad \sigma' = \sigma''[u, \text{st} \mapsto s', u, \text{stable} \mapsto b] \]

where \( b \) depends as before.

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

\[ \text{cons} = \emptyset, \text{Snd} = \text{Obs}_{\text{act}}(\sigma, \varepsilon). \]
Example: Commence

\[
\begin{align*}
| x > 0 | x & := x - 1; n! J \\
G & := y \\
H & := y/x \\
G, J & := x - 1;
\end{align*}
\]

\[
\begin{array}{c}
\text{Example: Commence} \\
\text{SMC:} \\
\text{SMC:} \\
\text{SMC:}
\end{array}
\]

\[
\begin{array}{c}
s_1 \\
G \\
H/z := y/x \\
s_2
\end{array}
\]

\[
\begin{array}{c}
\sigma: \\
\varepsilon: \\
\end{array}
\]

\[
\begin{array}{c}
x = 2, z = 0, y = 2 \\
\text{st} = s_2 \\
\text{stable} = 0 \\
\end{array}
\]

\[
\begin{array}{c}
\sigma' \\
\varepsilon'
\end{array}
\]

\[
\begin{array}{c}
\exists u \in \text{dom}(\sigma) \cap \mathcal{D}(C): \sigma(u)(\text{stable}) = 0 \\
\exists (s, \text{expr}, \text{act}, s') \in \to(SMC): I[\text{expr}](\sigma) = 1 \\
\sigma(u)(\text{stable}) = 1, \sigma(u)(\text{st}) = s, \\
\sigma'(u''(\text{act}) = t_{\text{act}}(\sigma, \varepsilon), \sigma' = \sigma''[u, \text{st} \mapsto s', u, \text{stable} \mapsto b] \\
\text{cons} = \emptyset, \text{Snd} = \text{Obs}_{env}(\sigma, \varepsilon)
\end{array}
\]

\[
\begin{array}{c}
\text{(iv) Environment Interaction} \\
\text{Assume that a set } \mathcal{E}_{env} \subseteq \mathcal{E} \text{ is designated as environment events and a set of attributes } v_{env} \subseteq V \text{ is designated as input attributes.} \\
\text{Then} \\
(\sigma, \varepsilon) \xrightarrow{(\text{cons}, \text{Snd})}_{env} (\sigma', \varepsilon') \\
\text{if} \\
\exists \sigma' = \sigma \cup \{u_E \mapsto \{v_i \mapsto d_i \mid 1 \le i \le n\}, \varepsilon' = \varepsilon \oplus u_E \\
\text{where } u_E \notin \text{dom}(\sigma) \text{ and } \text{atr}(E) = \{v_1, \ldots, v_n\}. \\
\text{or} \\
\forall v \in V \forall u \in \text{dom}(\sigma): \sigma'(u)(v) \neq \sigma(u)(v) \implies v \in V_{env}. \\
\text{and no objects appear or disappear, i.e. } \text{dom}(\sigma') = \text{dom}(\sigma). \\
\varepsilon' = \varepsilon.
\end{array}
\]
**Example: Environment**

\[ x > 0 \mid x := x - 1; n! J \]

\[ S\!M\!C: \]

- **\( s_1 \)**
- **\( G[x > 0] / x := y \)**
- **\( H/z := y / x \)**
- **\( H \)**
- **\( n \)**
- **\( x, z : \text{Int} \)**
- **\( y : \text{Int} \ (\{\text{env}\}) \)**

**\( \sigma \):**

- **\( c : C \)**
  - \( x = 0, z = 0, y = 2 \)
  - \( st = s_2 \)
  - \( \text{stable} = 1 \)

**\( \varepsilon \):**

- \( \varepsilon' = \varepsilon \cup u \in \sigma \) where \( u \notin \text{dom}(\sigma) \) and \( \text{atr}(E) = \{v_1, \ldots, v_n\} \)

\[ (v) \text{ Error Conditions} \]

- if, in (ii) or (iii),
  - \( I[\text{expr}] \) is not defined for \( \sigma \), or
  - \( t_{\text{act}} \) is not defined for \( (\sigma, \varepsilon) \),

and

- consumption is observed according to (ii) or (iii), but \( \text{Snd} = \emptyset \).

**Examples:**

- \( s_1 \rightarrow E[x > 0] / \text{act} \rightarrow s_2 \)
- \( s_1 \rightarrow E[\text{true}] / \text{act} \rightarrow s_3 \)
- \( s_1 \rightarrow E[\text{expr}] / x := x / 0 \rightarrow s_2 \)
**Example: Error Condition**

\[
S\mathcal{MC}: \quad \begin{array}{c}
\leq 1 \xrightarrow{s_1} G[x > 0]/x := y \\
H/z := y/x \xrightarrow{s_2}
\end{array}
\]

\[
\sigma: \quad x = 0, z = 0, y = 27 \\
st = s_2 \\
stable = 1
\]

\[
\varepsilon: \quad H \text{ for } c
\]

- \(I[\text{expr}]\) not defined for \(\sigma\), or
- Consumption according to (ii) or (iii)
- \(t_{\text{act}}\) is not defined for \((\sigma, \varepsilon)\)
- \(\text{Snd} = \emptyset\)

**Notions of Steps: The Step**

**Note:** we call one evolution \(\sigma, \varepsilon \xrightarrow{(\text{cons, Snd})} \sigma', \varepsilon'\) 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.
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:**

```
s_1 \rightarrow_s E[x > 0] /
\sigma: \quad : C
x = 2
\varepsilon: \quad E \text{ for } u
\rightarrow_s /x := x - 1
```

---

What about this *Example*:

```
\begin{array}{c}
\sigma: \quad : C \\
x = 2 \\
\varepsilon: \quad E \text{ for } u \\
\end{array}
```

---

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

**Proposal:** Let

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

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

- object \(u\) is alive in \(\sigma_0\),
- \(u_0 = u\) and \((\text{cons}_0, \text{Snd}_0)\) indicates dispatching to \(u\), i.e. \(\text{cons} = \{(u, \vec{v} \mapsto \vec{d})\}\),
- there are no receptions by \(u\) in between, i.e. \(\text{cons}_i \cap \{u\} \times \text{Evs}(\mathcal{E}, \mathcal{D}) = \emptyset, i > 1,\)
- \(u_{n-1} = u\) and \(u\) is stable only in \(\sigma_0\) and \(\sigma_n\), i.e. \(\sigma_0(u)(\text{stable}) = \sigma_n(u)(\text{stable}) = 1\) and \(\sigma_i(u)(\text{stable}) = 0\) for \(0 < i < n,\)

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

\[(\sigma_0(u) = \sigma_{k_1}(u), \sigma_{k_2}(u), \ldots, \sigma_{k_N}(u) = \sigma_{n-1}(u))\]

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

---

**Divergence**

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

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

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 \((S, \rightarrow, S_0)\). We have

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

Wanted: initial states \(S_0\).

Proposal:

Require a (finite) set of object diagrams \(OD\) as part of a UML model \((\mathcal{C}D, \mathcal{M}, \mathcal{O}D)\).

And set

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

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

\[ \mathcal{M} = (\mathcal{C}D, \mathcal{M}, \mathcal{O}D) \]

where

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

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

The computations of \(\mathcal{M}\) are the computations of \((S, \rightarrow, S_0)\).
\textbf{OCL Constraints and Behaviour}

- Let $\mathcal{M} = (\mathcal{C}, \mathcal{M}, \mathcal{O})$ be a UML model.
- We call $\mathcal{M}$ \textbf{consistent} iff, for each OCL constraint $\text{expr} \in \text{Inv}(\mathcal{C})$, $\sigma \models \text{expr}$ for each “reasonable point” $(\sigma, \varepsilon)$ of computations of $\mathcal{M}$.

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

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

\textbf{Pragmatics:}

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

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

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

\textbf{References}
References

