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
System Configuration, Ether, Transformer
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 $\Sigma$ the set of system configurations over some $\mathcal{I}_0$, $\mathcal{D}_0$ and $\mathcal{E}$ and ether. We call a relation

$$t \subseteq \mathcal{D}(\mathcal{C}) \times (\Sigma \times \mathcal{E}) \times (\Sigma \times \mathcal{E})$$

a (system configuration) transformer.

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

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

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

Special cases: creation/destruction.
Why Transformers?

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

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

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

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

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

- **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

$$I[\cdot](\cdot, \cdot) : Expr \rightarrow ((\Sigma F \times (\{\text{ref}\} \rightarrow D(C))) \rightarrow \mathbb{B})$$

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

$$t_{act} \subseteq D(C) \times (\Sigma F \times Eth) \times (\Sigma F \times Eth).$$
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

$$\text{Act}_p := \{ \text{skip} \}
\cup \{ \text{update}(\text{expr}_1, v, \text{expr}_2) \mid \text{expr}_1, \text{expr}_2 \in \text{OCLE}\text{Expr}\}
\cup \{ \text{send}(\text{expr}_1, E, \text{expr}_2) \mid \text{expr}_1, \text{expr}_2 \in \text{OCLE}\text{Expr}, E \in E_{(p)}\}
\cup \{ \text{create}(C, \text{expr}, v) \mid \text{expr} \in \text{OCLE}\text{Expr}. C \in C, v \in V\}
\cup \{ \text{destroy}(\text{expr}) \mid \text{expr} \in \text{OCLE}\text{Expr}\}$$
<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
</tr>
<tr>
<td>intuitive semantics</td>
<td>...</td>
</tr>
<tr>
<td>well-typedness</td>
<td>...</td>
</tr>
<tr>
<td>semantics</td>
<td>[(\sigma, \varepsilon), (\sigma', \varepsilon') \in t_{op}[u_x] ] iff ...</td>
</tr>
<tr>
<td></td>
<td>or [t_{op}[u_x](\sigma, \varepsilon) = ((\sigma', \varepsilon'))] where ...</td>
</tr>
</tbody>
</table>
| observables     | \[
Obs_{op}[u_x](\sigma, \varepsilon) = \{\ldots\}, \text{ not a relation, depends on choice}
\] |
| (error) conditions | Not defined if ... |
# Transformer: Skip

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>\texttt{skip}</td>
</tr>
</tbody>
</table>

## Intuitive semantics

\textit{do nothing}

## Well-typedness

\.\/.\.

## Semantics

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

## Observables

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

## (Error) conditions
### Transformer: Update

**abstract syntax**

| update\( (expr_1, v, expr_2) \) |

**concrete syntax**

\( expr_1, v := expr_2 \)

**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 \( v : \tau \in atr(C) \); \( expr_2 : \tau \);

\( expr_1, expr_2 \) obey visibility and navigability

**semantics**

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

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

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

**observables**

\[
Obs_{update}(expr_1, v, expr_2)[u_x] = \emptyset
\]

**error conditions**

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

**SM_C:**

\[ s_1 \xrightarrow{\{x := x + 1\}} s_2 \]

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

<table>
<thead>
<tr>
<th>(\sigma:)</th>
<th>(u_1 : C)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(x = 4)</td>
</tr>
<tr>
<td></td>
<td>(y = 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(\varepsilon:)</th>
<th>(u_1 : C)</th>
<th>(\sigma')</th>
</tr>
</thead>
<tbody>
<tr>
<td>only (x) changes</td>
<td>(x = 5)</td>
<td></td>
</tr>
<tr>
<td>(t_{\text{update}}[u_x](\sigma, \varepsilon) = (\sigma[u \mapsto \sigma(u)]v \mapsto I[\text{expr}_2](\sigma, \beta)], \varepsilon))</td>
<td>(u_1 : C)</td>
<td></td>
</tr>
<tr>
<td>(\varepsilon')</td>
<td>(\sigma')</td>
<td></td>
</tr>
</tbody>
</table>

\[ I[I[\text{self} + 1D(\sigma, \varepsilon)](u_x)] = 5 \]
Transformer: Send

**Abstract Syntax**

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

**Concrete Syntax**

\[
expr_{dst} \odot E(\ldots)
\]

**Intuitive Semantics**

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

**Well-Typedness**

\( expr_{dst} : \tau_D, \; C, D \in \mathcal{C}, \; E \in \mathcal{E}, \; \text{atr}(E) = \{v_1 : \tau_1, \ldots, v_n : \tau_n\}; \)

\( expr_i : \tau_i, \; 1 \leq i \leq n; \)

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

**Semantics**

\[
t_{\text{send}}(E(expr_1, \ldots, expr_n), expr_{dst})[u_x](\sigma, \varepsilon) \triangleright (\sigma', \varepsilon')
\]

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

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

\( u \in \mathcal{D}(E) \) a fresh identity, i.e. \( u \not\in \text{dom}(\sigma), \)

and where \( (\sigma', \varepsilon') = (\sigma, \varepsilon) \) if \( u_{dst} \not\in \text{dom}(\sigma); \; \beta = \{v_x \mapsto u_x\} \)

**Observables**

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

**Error Conditions**

\[
I[expr](\sigma, \beta) \text{ not defined for any } expr \in \{expr_{dst}, expr_1, \ldots, expr_n\}
\]
Send Transformer Example

$SM_C$:

\[ s_1 \xrightarrow{\text{\ldots, } n} ! F(x + 1); \ldots \xrightarrow{} s_2 \]

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

\[
t_{\text{send}}(expr_{src}, E(expr_1, \ldots, expr_n), expr_{dst})[u_x](\sigma, \varepsilon) = \ldots
\]
### 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.v := 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

$expr : \tau_D$, $v \in \text{atr}(D)$, \( \text{atr}(C) = \{ (v_i : \tau_i, expr_i) \mid 1 \leq i \leq n \} \)

#### Semantics

\[ \ldots \]

#### Observables

\[ \ldots \]

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

$SM_C:\$

\[ s_1 \rightarrow / \ldots; n := \text{new } C; \ldots \rightarrow s_2 \]

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

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

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

\[
\varepsilon: 
\]

\[
\sigma': 
\]

\[
\varepsilon': 
\]
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

**abstract syntax**

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

**intuitive semantics**

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

**well-typedness**

\[ \text{expr} : \tau_D, \; v \in \text{atr}(D), \; \text{atr}(C) = \{ \langle v_1 : \tau_1, \text{expr}^0_i \rangle \mid 1 \leq i \leq n \} \]

**semantics**

\[
((\sigma, \varepsilon), (\sigma', \varepsilon')) \in t
\]

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

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

\(u_0 = I[\text{expr}] (\sigma, \beta); \; d_i = I[\text{expr}^0_i] (\sigma, \beta)\) if \(\text{expr}^0_i \neq ""\) and arbitrary value from \(\mathcal{D}(\tau_i)\) otherwise; \(\beta = \{ \text{this} \mapsto u_x \} \).

**observables**

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

**(error) conditions**

\(I[\text{expr}] (\sigma)\) 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></td>
</tr>
</tbody>
</table>

### Intuitive Semantics

*Destroy the object denoted by expression* `expr`.

### Well-Typedness

`expr : \tau_C, C \in \mathcal{C}`

### Semantics

...  

### Observables

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

### (Error) Conditions

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

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

\[
destroy(expr)
\]

\[
t_{\text{destroy}(expr)}[u_x](\sigma, \varepsilon) = \ldots
\]
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.
**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 : τ_C, C ∈ C`

**semantics**

\[ t[u_x](σ, ε) = (σ', ε) \]

where \( σ' = σ|\{\text{dom}(σ) \backslash u}\} \) 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

$$ (t_2 \circ t_1)[u_x](\sigma, \varepsilon) = t_2[u_x](t_1[u_x](\sigma, \varepsilon)) $$

with observation

$$ Obs(t_2 \circ t_1)[u_x](\sigma, \varepsilon) = Obs_{t_1}[u_x](\sigma, \varepsilon) \cup Obs_{t_2}[u_x](t_1(\sigma, \varepsilon)). $$

- **Clear**: not defined if one of 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.
Run-to-completion Step
**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.
Definition. Let $I_0 = (T_0, C_0, V_0, atr_0)$ be a signature with signals (all classes active), $D_0$ a structure of $I_0$, and $(Eth, ready, \oplus, \ominus, [\cdot])$ an ether over $I_0$ and $D_0$. Assume there is one core state machine $M_C$ per class $C \in C$.

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

$$(\sigma, \varepsilon) \xrightarrow{cons,Snd}_{u} (\sigma', \varepsilon')$$

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 $u$,

$$(cons, \emptyset) \xrightarrow{s} \#$$

if and only if
(v) $s = \#$ and $cons = \emptyset$, or an error condition occurs during consumption of $cons$. 

From Core State Machines to LTS
(i) Discarding An Event

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

if

- an \(E\)-event (instance of signal \(E\)) is ready in \(\varepsilon\) for \(\exists\) object of a class \(C\), i.e.
  \[\exists u \in \text{dom(}\sigma) \cap D(C) \exists u_E \in D(\varepsilon) : u_E \in \text{ready}(\varepsilon, u)\]

- \(u\) 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)

\[\forall (s, F, expr, act, s') \in \rightarrow (SM_C) : F \neq E \lor I[expr](\sigma) = 0\]

and

- the system configuration doesn’t change, i.e. \(\sigma' = \sigma\)

- the event \(u_E\) is removed from the ether, i.e.
  \[\varepsilon' = \varepsilon \ominus u_E,\]

- consumption of \(u_E\) is observed, i.e.
  \[cons = \{(u, (E, \sigma(u_E)))\}, Snd = \emptyset.\]
**Example: Discard**

\[ x > 0 \] / \( x := x - 1 \); \( n! J \)

\( SM_C: \)

\[ S_1 \] \hspace{1cm} \[ G [ x > 0 ] / x := y \] \hspace{1cm} \[ S_2 \]

\( H / z := y / x \)

\( \sigma: \)

\[ c : C \]

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

\[ st = s_1 \]

\[ stable = 1 \]

\( \varepsilon: \)

- \( J \) for \( c \), \( G \) for \( c \)

- \( \exists u \in \text{dom}(\sigma) \cap \mathcal{D}(C) \)
  - \( \exists u_E \in \mathcal{D}(\varepsilon') : u_E \in \text{ready}(\varepsilon, u) \)

- \( \forall (s, F, expr, act, s') \in \rightarrow (SM_C) : \)
  - \( F \neq E \lor I[expr](\sigma) = 0 \)

- \( \sigma(u)(stable) = 1, \sigma(u)(st) = s, \)

- \( \sigma' = \sigma, \varepsilon' = \varepsilon \oplus u_E \)

- \( \text{cons} = \{(u, (E, \sigma(u_E)))\}, \text{Snd} = \emptyset \)
(ii) **Dispatch**

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

- \(u \in \text{dom}(\sigma) \cap \mathcal{D}(C) \) \(\exists u_E \in \mathcal{D}(\mathcal{E}) : u_E \in \text{ready}(\varepsilon, u)\)
- \(u\) is stable and in state machine state \(s\), i.e. \(\sigma(u)(\text{stable}) = 1\) and \(\sigma(u)(\text{st}) = s\),
- a transition is enabled, i.e.

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

where \(\tilde{\sigma} = \sigma[u.params_E \mapsto u_e]\).

and

- \((\sigma', \varepsilon')\) results from applying \(t_{act}\) to \((\sigma, \varepsilon)\) and removing \(u_E\) from the ether, i.e.

\[(\sigma'', \varepsilon') = t_{act}(\tilde{\sigma}, \varepsilon \uplus u_E),\]

\[\sigma' = (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset])|\mathcal{D}(\mathcal{E})\setminus\{u_E\}\]

where \(b\) depends:

- If \(u\) becomes stable in \(s'\), then \(b = 1\). It **does** become stable if and only if there is no transition **without trigger** enabled for \(u\) in \((\sigma', \varepsilon')\).
- Otherwise \(b = 0\).

- Consumption of \(u_E\) and the side effects of the action are observed, i.e.

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

$SM_C$: 

$x > 0 \triangleright x := x - 1; n ! J$

$H / z := y / x$

$[x > 0] / x := y$

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

$H \triangleright n$

$s_1 \to s_2$

$\rho$

$\sigma$

$x = 1, z = 0, y = 2$

$st = s_1$

$stable = 1$

$\varepsilon$

$G$ for $c$

$\exists u \in \text{dom}(\sigma) \cap D(C)$

$\exists u_E \in D(\varepsilon') : u_E \in \text{ready}(\varepsilon, u)$

$\exists (s, F, expr, act, s') \in (SM_C)$

$F = E \land I[expr](\tilde{\sigma}) = 1$

$\tilde{\sigma} = \sigma[u.params_E \mapsto u_e]$

$\sigma'(stable) = 1, \sigma'(st) = s$,

$(\sigma'', \varepsilon') = \tau_{act}(\tilde{\sigma}, \varepsilon \oplus u_E)$

$\sigma' = (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset])|D(\varepsilon) \{u_E\}$

$\text{cons} = \{(u, (E, \sigma(u_E)))\}$, $\text{Snd} = \text{Obs}_{act}(\tilde{\sigma}, \varepsilon \oplus u_E)$

$\langle \langle \text{signal, env} \rangle \rangle$

$H$

$\langle \langle \text{signal} \rangle \rangle$

$G, J$

$C$

$x, z : \text{Int}$

$y : \text{Int} \langle \langle \text{env} \rangle \rangle$

$\sigma : c : C$

$\varepsilon : c / C$

$\langle \langle \text{signal} \rangle \rangle$

$\varepsilon'$

$\langle \langle \text{signal} \rangle \rangle$
(iii) Commence Run-to-Completion

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

if

- there is an unstable object of a class $\mathcal{C}$, i.e.
  \[\exists u \in \text{dom}(\sigma) \cap \mathcal{D}(C) : \sigma(u)(\text{stable}) = 0\]

- there is a transition without guard enabled from the current state $s = \sigma(u)(st)$, i.e.
  \[\exists (s, \_, \text{expr}, \text{act}, s') \in \rightarrow (\mathcal{SM}_C) : I[expr](\sigma) = 1\]

and

- $(\sigma', \varepsilon')$ results from applying $t_{\text{act}}$ to $(\sigma, \varepsilon)$, i.e.
  \[(\sigma'', \varepsilon') \in t_{\text{act}}(\sigma, \varepsilon), \quad \sigma' = \sigma''[u.st \mapsto s', u.\text{stable} \mapsto b]\]

  where $b$ depends as before.

- Only the side effects of the action are observed, i.e.
  \[\text{cons} = \emptyset, \text{Snd} = \text{Obs}_{t_{\text{act}}}(\sigma, \varepsilon)\]
Example: Commence

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

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

\[ \langle \langle \text{signal}, \text{env} \rangle \rangle \]

\[ \langle \langle \text{signal} \rangle \rangle \]

\[ \langle \langle \text{env} \rangle \rangle \]

\[ \sigma: \]

\[ c : C \]

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

\[ st = s_2 \]

\[ stable = 0 \]

\[ \varepsilon: \]

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

\[ \exists (s, \omega, \text{expr}, \text{act}, s') \in (\mathcal{SM}_C) : I[\text{expr}](\sigma) = 1 \]

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

\[ (\sigma'', \varepsilon') = t_{act}(\sigma, \varepsilon), \]

\[ \sigma' = \sigma''[u.st \mapsto s', u.stable \mapsto b] \]

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

Assume that a set $E_{env} \subseteq E$ is designated as environment events and a set of attributes $v_{env} \subseteq V$ is designated as input attributes.

Then

$$\left(\sigma, \varepsilon\right) \xrightarrow{\left(cons, Snd\right)}_{env} \left(\sigma', \varepsilon'\right)$$

if

- an environment event $E \in E_{env}$ is spontaneously sent to an alive object $u \in D(\sigma)$, i.e.

  $$\sigma' = \sigma \cup \{u_E \mapsto \{v_i \mapsto d_i \mid 1 \leq i \leq n\}, \quad \varepsilon' = \varepsilon \oplus u_E$$

  where $u_E \notin \text{dom}(\sigma)$ and $\text{atr}(E) = \{v_1, \ldots, v_n\}$.

- Sending of the event is observed, i.e. $cons = \emptyset$, $Snd = \{(env, E(\vec{d}))\}$.

or

- Values of input attributes change freely in alive objects, i.e.

  $$\forall v \in V \forall u \in \text{dom}(\sigma) : \sigma'(u)(v) \neq \sigma(u)(v) \implies v \in V_{env}.$$ 

  and no objects appear or disappear, i.e. $\text{dom}(\sigma') = \text{dom}(\sigma)$.

- $\varepsilon' = \varepsilon$. 
Example: Environment

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

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

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

\[ \langle \langle \text{signal, env} \rangle \rangle \]

\[ \langle \langle \text{signal} \rangle \rangle \]

\[ C \]

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

\[ y : \text{Int} \]

\[ \langle \langle \text{env} \rangle \rangle \]

\( \sigma : C \)

\( x = 0, z = 0, y = 2 \)

\( \text{st} = s_2 \)

\( \text{stable} = 1 \)

\( \epsilon : \)

\[ \cdot \sigma' = \sigma \cup \{ u_E \mapsto \{ v_i \mapsto d_i \mid 1 \leq i \leq n \} \} \]

\[ \cdot \epsilon' = \epsilon \oplus u_E \text{ where } u_E \notin \text{dom}(\sigma) \]

\[ \text{and } atr(E) = \{ v_1, \ldots, v_n \}. \]

\[ u \in \text{dom}(\sigma) \]

\[ \cdot \text{cons} = \emptyset, \text{Snd} = \{ (\text{env}, E(\vec{d})) \}. \]
(v) Error Conditions

\[ s \xrightarrow{(\text{cons}, \text{Snd})} u \to \# \]

if, in (ii) or (iii),

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

and

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

Examples:

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

$SM_C$: 

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

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

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

$C:\; x, z: \text{Int}$

\[ \langle \langle \text{signal} \rangle \rangle \]

\[ y: \text{Int} \]

\[ \langle \langle \text{env} \rangle \rangle \]

\[ 0, 1 \]

\[ n \]

$\sigma$: 

\[ c: C \]

\[ x = 0, z = 0, y = 27 \]

\[ st = s_2 \]

\[ stable = 1 \]

$\varepsilon$: 

\[ H \text{ for } c \]

\[ (H, \emptyset) \]

\[ \# \]

- $I[expr]$ not defined for $\sigma$, or
- $t_{act}$ is not defined for $(\sigma, \varepsilon)$
- consumption according to (ii) or (iii)
- $Snd = \emptyset$
Notions of Steps: The Step

**Note:** we call one evolution \((\sigma, \varepsilon) \xrightarrow{(\text{cons}, \text{Snd})} u \xrightarrow{\text{a step}} (\sigma', \varepsilon')\) 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.
Notions of Steps: The Run-to-Completion Step

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:**

\[
E[x > 0]/ \quad /x := x - 1
\]

\[
\sigma:
\begin{array}{|c|}
\hline
C \\
\hline
x = 2 \\
\hline
\end{array}
\]

\[
\varepsilon:
\begin{array}{|c|}
\hline
E \text{ for } u \\
\hline
\end{array}
\]
Notions of Steps: The Run-to-Completion Step

What about this Example:

\( SM_C: \)

\[
\begin{align*}
\sigma: & \quad : C \\
& \quad x = 2
\end{align*}
\]

\( SM_D: \)

\[
\begin{align*}
\varepsilon: & \quad E \text{ for } c \\
& \quad F \text{ for } d
\end{align*}
\]
Proposal: Let

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

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.
  \[\sigma_0(u)(\text{stable}) = \sigma_n(u)(\text{stable}) = 1 \quad \text{and} \quad \sigma_i(u)(\text{stable}) = 0 \quad \text{for} \quad 0 < i < n,\]

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

\[(\sigma_0(u) =) \quad \sigma_{k_1}(u), \sigma_{k_2}(u) \ldots, \sigma_{k_N}(u) \quad (= \sigma_{n-1}(u))\]
a (!) run-to-completion computation of \(u\) (from (local) configuration \(\sigma_0(u)\)).
Divergence

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

\[
(\sigma_0, \varepsilon_0) \xrightarrow{(cons_0, Snd_0)} (\sigma_1, \varepsilon_1) \xrightarrow{(cons_1, Snd_1)} \ldots
\]

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. 

- (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
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 \((\sigma, \varepsilon) \xrightarrow{\text{(cons, Snd)}} u (\sigma', \varepsilon')\).

**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. We can read that as an abbreviation for an object diagram.
The **semantics** of the **UML model**

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

where

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

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

The **computations of** \( M \) are the computations of \((S, \rightarrow, S_0)\).
OCL Constraints and Behaviour

- Let $M = (CD, SM, OD)$ be a UML model.
- We call $M$ consistent iff, for each OCL constraint $expr \in Inv(CD)$,
  $$\sigma \models expr$$
  for each “reasonable point” $(\sigma, \varepsilon)$ 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.
References
References

