Roadmap: Chronologically

(i) What do we (have to) cover?
UML State Machine Diagram Syntax

(ii) Def.: Signature with guards.
(iii) Def.: Core state machine.
(iv) Map UML State Machine Diagrams to core state machines.

Semantic: 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 machines.
(xi) Def.: step, run-to-completion step.
(xii) Later: Hierarchical state machines.

System Configuration, Ether, Transformer

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

Why Transformers?

- Recall the (simplified) syntax of transition annotations:
  - \( \text{action} ::= [\langle \text{guard} \rangle \cdot \langle \text{action}\rangle ] \)
- Clear: \( \langle \text{guard} \rangle \) is from \( \Phi \) 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, C++, ...
  - Action Language:
    - UML Action Semantics, “Executable UML”
    - Java, C++, ...

Transformer

- In the following, we assume that each application of a transformer \( T \) to some system configuration \( \langle s, e \rangle \) for object \( s_0 \) is associated with a set of observations:

\[
\text{Obs}(s_0, \langle s, e \rangle, E) \in E^{\text{states}} \times E^{\text{events}} \times E^{\text{system configurations}}
\]

- An observation \( \text{Obs}(s_0, \langle s, e \rangle, E) \) represents the information that, as a “side effect” of \( s_0 \) executing \( t \), an event \( \langle E, \delta \rangle \) has been sent from object \( s_0 \) to object \( s_1 \).
- Special case: creation/destruction.

Putting It All Together

Recall the (simplified) syntax of transition annotations:
- \( \text{action} ::= [\langle \text{guard} \rangle \cdot \langle \text{action}\rangle ] \)
- Clear: \( \langle \text{guard} \rangle \) is from \( \Phi \) 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 \( \langle \text{action} \rangle \)).
In the following, we assume that we're given:

- an expression language \( \text{Expr} \) for guards, and
- an action language \( \text{Act} \) for actions,

and that we're given:

- a semantics for boolean expressions in form of a partial function
  \( I : \text{Expr} \to ((\Sigma \times \text{BW} \times \text{CB}) \times \{\text{this}\} \to \text{BW} \times \text{BV}) \to \text{BW}) \) 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 goto a designated "error" system configuration.

- a transformer for each action:
  \[ \text{Foreach } \text{act} \in \text{Act}, \text{we assume to have } t_{\text{act}} \subseteq (\Sigma \times \text{BW} \times \text{CB} \times \text{Eth}) \times (\Sigma \times \text{BW} \times \text{CB} \times \text{Eth}). \]

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 "\( \perp \)".
- for Java, the operations' semantic 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 \( \epsilon \) — interesting, because state machines are built around sending/consuming events
- create/destroy: modifies domain of \( \sigma \) — not specific to state machines, but let's discuss them here as we're at it
- update: modifies own or other objects' local state — boring

### Transformer Examples: Presentation

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
<th>skipping</th>
<th>intuitive semantics</th>
<th>well-typedness</th>
<th>observables</th>
<th>(error) conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>skip</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>update</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Adding them is straightforward (but somewhat tedious).

Also for simplicity: no parameter to construction.

We use an "and assign" action for simplicity—it doesn't add or remove expression parts.

The history of creation raises all kinds of other problems such as order of evaluation and history creation.

Intuitive semantics:

Create an object of class `expr` and assign it to attribute `v` of the object denoted by expression `expr`: `v = expr`

Well-formedness:

- `expr` must denote an object (e.g., `v`).
- `v` must be an attribute of the object denoted by `expr`.

Weakening:

- `(v = expr)` is `true`.

If `expr` is not defined for any `v`, `s` then `false`.

• Re-use: choose any identity that is not alive now, i.e., not in class `v`.
• 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 class `v` and any predecessor in current run.
• Depends on history.
• Dangling references remain dangling — could make "dirty" effects of platform.
What to Do With the Remaining Objects?

Assume object \( o_1 \) is destroyed:

- Object \( o_2 \) may still refer to it via association \( r \):
  - Allow dangling references?
  - Or remove \( o_2 \) from \( r(\cdot) \)?
- Object \( o_3 \) may have been the last one linking to object \( o_2 \):
  - Leave \( o_3 \) alone?
  - Or remove \( o_3 \) 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.

Observables:
\[
O_{\text{msem}}[x_1] = [\{x_1, +r, o\}]
\]

(a) conditions: \( f(x_1, o) \) not defined.

Sequential Composition of Transformers

- Sequential composition \( t_2 \circ t_1 \) of transformers \( t_1 \) and \( t_2 \) is canonically defined as:
\[
\{t_2 \circ t_1\}[x][r](x) = t_2[t_1[x][r]]
\]

with observables:
\[
O_{\text{msem}}[[x_1][r]](x) = O_{\text{msem}}[x_1][r](x) = O_{\text{msem}}[x_1][r](x).
\]

- Clear: not defined if one the two intermediate "micro-steps" is not defined.

\[
\begin{align*}
\forall x \in x' \Rightarrow f(x, o) &= x' \land g \subseteq F
\end{align*}
\]
Transformers And Denotational Semantics

Observation: our transformers are in particular the denotational semantics of the actions/action sequences. The trivial case, to be precise.

Note: with the previous example, 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 atomic language is, e.g., Java, then (syntactically) holed loops and calls of recursive functions.

Other Approach: see 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

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 [Harlow and Gray, 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

(i) Discarding An Event

\[
\begin{align*}
\text{Definition. Let } A & \text{ be a set of actions and } s \text{ a (not necessarily finite) set of states.} \\
\text{We call} \quad & \mathcal{L} \subseteq A \times A \times s \times s \\
\text{a (labelled) transition relation.} \\
\text{Let } S_0 \subseteq S & \text{ be a set of initial states. A sequence} \\
\sigma & = (s_0, s_1, s_2, \ldots) \\
\text{with } s_i & \in S, \ s_i \in A \text{ is called computation of the labelled transition system } (S, \sigma, S_0) \text{ if and only if} \\
& \sigma \text{ initiation: } s_0 \in S_0 \\
& \text{communication: } \{s_i, s_{i+1}\} \subseteq \mathcal{L} \text{ for } i \in \mathbb{N}.
\end{align*}
\]

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 [Harlow and Gray, 1997] (i.e. the one realised by the Rhapsody code generation) where the standard is ambiguous or leaves choices.
Example: Discard

\[
\begin{array}{c}
\text{SM}_{c_1} \xrightarrow{I \sigma} \text{Int} \xrightarrow{I \sigma, \epsilon} \text{Snd} \xrightarrow{I \sigma} \text{network} \xrightarrow{I \sigma} c_1
\end{array}
\]

(ii) Dispatch

\[
\begin{array}{c}
\text{SM}_{c_2} \xrightarrow{I \sigma} \text{Int} \xrightarrow{I \sigma, \epsilon} \text{Snd} \xrightarrow{I \sigma} \text{network} \xrightarrow{I \sigma} c_2
\end{array}
\]

Example: Discard

\[
\begin{array}{c}
\text{SM}_{c_2} \xrightarrow{I \sigma} \text{Int} \xrightarrow{I \sigma, \epsilon} \text{Snd} \xrightarrow{I \sigma} \text{network} \xrightarrow{I \sigma} c_2
\end{array}
\]

(iii) Commence Run-to-Completion

\[
\begin{array}{c}
\text{SM}_{c_3} \xrightarrow{I \sigma} \text{Int} \xrightarrow{I \sigma, \epsilon} \text{Snd} \xrightarrow{I \sigma} \text{network} \xrightarrow{I \sigma} c_3
\end{array}
\]

Example: Commence

\[
\begin{array}{c}
\text{SM}_{c_3} \xrightarrow{I \sigma} \text{Int} \xrightarrow{I \sigma, \epsilon} \text{Snd} \xrightarrow{I \sigma} \text{network} \xrightarrow{I \sigma} c_3
\end{array}
\]

(iv) Environment Interaction

Assume that a set \( E_{\text{in}} \subseteq E \) is designated an environment events and a set of attributes \( \sigma_{\text{in}} \subseteq \sigma \) is designated an input attributes.

Then

\[
\begin{array}{c}
\text{SM}_{c_3} \xrightarrow{I \sigma} \text{Int} \xrightarrow{I \sigma, \epsilon} \text{Snd} \xrightarrow{I \sigma} \text{network} \xrightarrow{I \sigma} c_3
\end{array}
\]
### Notions of Steps: The Step

**Note:** We call one evolution \((n, r) \xrightarrow{\text{compu}(\Delta t)} (n', r')\) a step.

Thus in our setting, a step directly corresponds to:

- one object (namely, \(a\)) takes a single transition between regular states.

*We have to extend the concept of "single transition" for hierarchical state machines.*

**That is:** We are going for an interleaving semantics without true parallelism.

**Remark:** With only methods (tales), the notion of step is not so clear.

For example, consider:

- \(c_1\) calls \(f()\) at \(c_2\), which calls \(g()\) at \(c_3\) which in turn calls \(h()\) for \(c_4\).
- 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/invaders are typically (by convention) assumed to be evaluated at step boundaries, and sometimes this 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
  
  - A run-to-completion step is in general not syntactically definable — one transition may be taken multiple times during an RTC-step.

**Example:**

\[
\delta x > \delta y \\
(\text{act } x = a + 1)
\]

### Notions of Steps: The Run-to-Completion Step

**What about this Example?**

\[
\delta x > \delta y \\
(\text{act } x = a + 1)
\]
Notions of Steps: The Run-to-Completion Step Cont’d

Proposal: Let
\[ (n_k, u) \xrightarrow{\text{cons}(\text{transition}(u), \sigma))} n_{k+1} \]
be a finite (f), non-empty, maximal, consecutive sequence such that
- object \( u \) is alive in \( n_{k+1} \)
- \( u \) is seen \( \sigma \) and \( \text{cons}(\text{transition}(u), \sigma) \) indicates dispatching to \( u \).
- \( u \) in \( n_{k+1} \) is stable.
- \( u \) is stable in \( n_{k+1} \).

We say object \( u \) can diverge or react as \( \sigma \) from (local) configuration \( n_k(u) \) if and only if there is an infinite, consecutive sequence
\[ (n_k(u), u \xrightarrow{\text{cons}(\text{transition}(u), \sigma))} n_{k+1}(u)) \]
such that it doesn’t become stable again.

Note: disappearance of object not considered in the definitions.

Divergence

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 use the effect of interaction with other objects.
- Adding classes (or even objects) may change the divergence behaviors of existing ones.
- Compositional would be: the behavior of a set of objects is determined by the behavior of each object. Is that right?

Note: Our semantics and notion of RTC-step doesn’t have this (often desired) property.
Can we give (syntactical) criteria such that any global non-completion step in a interleaving of local ones?

Maybe, Strict interfaces:
- \( A \): 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

The Missing Piece: Initial States

Recall a labelled transition system is \( \langle S, \rightarrow, S_0 \rangle \). We have
- \( S \): system configuration \( \langle n, r \rangle \)
- \( \rightarrow \): labelled transition relation \( \langle n, r \rangle \xrightarrow{\text{cons}(\text{transition}(u), \sigma))} \langle n', r' \rangle \).

Wanted: initial states \( S_0 \).

Proposal: Require \( \langle n, r \rangle \) belongs to \( \text{CDP} \) as part of a UML model \( \langle \text{UMLmodel}, \text{CDP} \rangle \).

And set
\[ S_0 = \{ \langle n, r \rangle \mid \langle n, r \rangle \in \text{CDP} \} \]
OCL Constraints and Behaviour

Let \( M = (\mathcal{P}, \mathcal{R}, \mathcal{E}) \) be a UML model.

We call \( M \) consistent \( \mathcal{R} \), for each OCL constraint \( \mathcal{E} \in \text{Inv}(\mathcal{P}) \).

\( \mathcal{E} \models \mathcal{E} \) for each “reasonable point” \( (\mathcal{S}, \epsilon) \) of computations of \( M \).

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

Note: we could define \( \text{Inv}(\mathcal{P}) \) similar to \( \text{Inv}(\mathcal{R}) \).

Pragmatics:

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

- If the developer makes a mistake, then \( M' \) is inconsistent.

- Not common: if \( \mathcal{P}, \mathcal{R} \) is given, then constraints are also considered when choosing transitions in the RTC-algorithm. In other words: even in presence of mistakes, the \( \mathcal{P}, \mathcal{R} \) move even to inconsistent configurations.

References