Software Design, Modelling and Analysis in UML

Lecture 12: Core State Machines III

2011-12-11

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal
Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:
- The basic causality model
- Ether

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:
  - System Configuration, Transformer
  - Examples for transformer
  - Run-to-completion Step
  - Putting It All Together
And?

- We have to formally define what event occurrence is.
- We have to define where events are stored — what the event pool is.
- We have to explain how transitions are chosen — “matching”.
- We have to explain what the effect of actions is — on state and event pool.
- We have to decide on the granularity — micro-steps, steps, run-to-completion steps (aka. super-steps)?
- We have to formally define a notion of stability and RTC-step completion.

- And then: hierarchical state machines.
**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.

---

**System Configuration, Ether, Transformer**
Ether aka. Event Pool

Definition. Let \( \mathcal{S} = (\mathcal{T}, \mathcal{C}, V, at) \) be a signature with signals and \( \mathcal{D} \) a structure.

We call a structure \( (\text{Eth}, \text{ready}, \oplus, \ominus, [\cdot]) \) an ether over \( \mathcal{S} \) and \( \mathcal{D} \) if and only if it provides

- a ready operation which yields a set of events that are ready for a given object, i.e. \( \text{ready} : \text{Eth} \times \mathcal{D}(\mathcal{C}) \to 2^\mathcal{D}(\mathcal{C}) \)

- a operation to insert an event destined for a given object, i.e. \( \oplus : \text{Eth} \times \mathcal{D}(\mathcal{C}) \times \mathcal{D}(\mathcal{E}) \to \text{Eth} \)

- a operation to remove an event, i.e. \( \ominus : \text{Eth} \times \mathcal{D}(\mathcal{E}) \to \text{Eth} \)

- an operation to clear the ether for a given object, i.e. \( [\cdot] : \text{Eth} \times \mathcal{D}(\mathcal{C}) \to \text{Eth} \).

Ether and [OMG, 2007b]

The standard distinguishes (among others)

- SignalEvent [OMG, 2007b, 450] and Reception [OMG, 2007b, 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, 2007b, 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.) [OMG, 2007b, 450]

Our ether is a general representation of the possible choices.

Often seen minimal requirement: order of sending by one object is preserved.

But: we’ll later briefly discuss “discarding” of events.
**System Configuration**

**Definition.** Let $\mathcal{S} = (\mathcal{P}, \mathcal{C}, V_0, \text{atr}_0, \mathcal{E})$ be a signature with signals, $\mathcal{P}$ a structure of $\mathcal{S}$, $\mathcal{E}$ an ether over $\mathcal{S}$ and $\mathcal{P}$.

Furthermore assume there is one core state machine $M_C$ per class $C \in \mathcal{C}$.

A **system configuration** over $\mathcal{S}$, $\mathcal{P}$, and $\mathcal{E}$ is a pair

\[
(\sigma, \epsilon) \in \Sigma^{\mathcal{P}} \times \mathcal{E}
\]

where

- $\mathcal{P} = (\mathcal{P} \cup \{S_{MC} \mid C \in \mathcal{C}\}, \mathcal{C}_0$, $V_0 \cup \{\text{stable} : \text{Bool}, _-, _{\text{true}}, \emptyset\}$
- $\cup \{\text{st}_C : S_{MC}, _+, \mathcal{E}_0, \emptyset \mid C \in \mathcal{C}\}
- \cup \{\text{params}_E : E_0, _+, \mathcal{E}_0, \emptyset \mid E \in \mathcal{E}_0\}$

$\sigma(u)(r) \cap \mathcal{D}(\mathcal{E}_0) = \emptyset$ for each $u \in \text{dom}(\sigma)$ and $r \in V_0, \mathcal{E}_0$. This holds to signal freshness (of these are not a trade in order to access their attributes.)
System Configuration Step-by-Step

- We start with some signature with signals \( S_0 = (S_0, E_0, V_0, atr_0, \mathcal{E}) \).
- A **system configuration** is a pair \((\sigma, \varepsilon)\) which comprises a system state \(\sigma\) wrt. \(\mathcal{S}\) (not wrt. \(S_0\)).
- Such a **system state** \(\sigma\) wrt. \(\mathcal{S}\) provides, for each object \(u \in \text{dom}(\sigma)\),
  - values for the **explicit attributes** in \(V_0\),
  - values for a number of **implicit attributes**, namely
    - a **stability flag**, i.e. \(\sigma(u)(\text{stable})\) is a boolean value,
    - a **current (state machine) state**, i.e. \(\sigma(u)(\text{st})\) denotes one of the states of core state machine \(M_C\),
    - a temporary association to access **event parameters** for each class, i.e. \(\sigma(u)(\text{params}_E)\) is defined for each \(E \in \mathcal{E}\).
- For convenience require: there is no link to an event except for \(\text{params}_E\).

**Stability**

**Definition.**
Let \((\sigma, \varepsilon)\) be a system configuration over some \(S_0, E_0, Eth\).
We call an object \(u \in \text{dom}(\sigma) \cap \mathcal{P}(E_0)\) **stable in \(\sigma\)** if and only if
\[
\sigma(u)(\text{stable}) = true.
\]
**Events Are Instances of Signals**

**Definition.** Let $\mathcal{D}_0$ be a structure of the signature with signals $\mathcal{S}_0 = (\mathcal{S}_0, \mathcal{C}_0, V_0, atr_0, \mathcal{S})$ and let $E \in \mathcal{D}_0$ be a signal.

Let $atr(E) = \{v_1, \ldots, v_n\}$. We call

$$e = (E, \{v_1 \mapsto d_1, \ldots, v_n \mapsto d_n\}),$$

or shorter (if mapping is clear from context)

$$(E, (d_1, \ldots, d_n)) \text{ or } (E, \vec{d}),$$

an event (or an instance) of signal $E$ (if type-consistent).

We use $\text{Evs}(\mathcal{D}_0, \mathcal{D}_0)$ to denote the set of all events of all signals in $\mathcal{S}_0$ wrt. $\mathcal{D}_0$.

As we always try to maximize confusion...:

- By our existing naming convention, $u \in \mathcal{D}(E)$ is also called instance of the (signal) class $E$ in system configuration $(\sigma, \epsilon)$ if $u \in \text{dom}(\sigma)$.
- The corresponding event is then $(E, \sigma(u))$.

---

**Signals? Events...? Ether...?!**

The idea is the following:

- **Signals** are types (classes).
- **Instances of signals** (in the standard sense) are kept in the system state component $\sigma$ of system configurations $(\sigma, \epsilon)$.
- **Identities** of signal instances are kept in the ether.
- Each signal instance is in particular an event — somehow “a recording that this signal occurred” (without caring for its identity)
- The main difference between signal instance and event: Events don’t have an identity.
- Why is this useful? In particular for reflective descriptions of behaviour, we are typically not interested in the identity of a signal instance, but only whether it is an “$E$” or “$F$”, and which parameters it carries.
**Where are we?**

- **Wanted:** a labelled transition relation

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

on system configuration, labelled with the **consumed** and **sent** events, 
\((\sigma', \varepsilon')\) being the result (or effect) of one object \(u_x\) taking a transition of its state machine from the current state mach. state \(\sigma(u_x)(stC)\).

- **Have:** system configuration \((\sigma, \varepsilon)\) comprising current state machine state and stability flag for each object, and the ether.

- **Plan:**
  1. Introduce **transformer** as the semantics of action annotations.
     Intuitively, \((\sigma', \varepsilon')\) is the effect of applying the transformer of the taken transition.
  2. Explain how to choose transitions depending on \(\varepsilon\) and when to stop taking transitions — the run-to-completion "algorithm".

---

**Transformer**

**Definition.**
Let \(\SigmaP\) the set of system configurations over some \(\mathcal{X}_0, \mathcal{P}_0, Eth\).

We call a relation

\[t \subseteq \mathcal{P}(\mathcal{E}) \times (\SigmaP \times Eth) \times (\SigmaP \times Eth)\]

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**

\[\text{Obs}_t[u_x](\sigma, \varepsilon) \in 2^{\mathcal{P}(\mathcal{E})} \times \mathcal{E}_0(\mathcal{E}) \cup \{\ast, \dagger\} \times \mathcal{P}(\mathcal{E})\]

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

**Special cases:** creation/destruction.
Why Transformers?

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

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

- **Clear**: \(\langle \text{event} \rangle\) is from \(\sigma\) 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++, \ldots expressions
    - \ldots
  - **Action Language**:
    - UML Action Semantics, “Executable UML”
    - Java, C++, \ldots statements (plus some event send action)
    - \ldots

In the following, we consider:

\[
\text{Act}_\phi \ ::= \{ \text{skip} \}
\]

\[
\begin{align*}
\text{Act}_\phi \ ::= \& \text{update}(\text{expr}_1, \text{expr}_2) \mid \text{expr}_1, \text{expr}_2 \in \text{OCLExp}_v, \text{vec}_v \\
\text{Act}_\phi \ ::= \& \text{send}(\text{expr}_1, \text{expr}_2) \mid \text{expr}_1, \text{expr}_2 \in \text{OCLExp}_E, \text{vec}_E \\
\text{Act}_\phi \ ::= \& \text{create}(\text{expr}_1, \text{vec}_v) \mid \text{expr}_1 \in \text{OCLExp}_v, \text{vec}_E, \text{vec}_v \\
\text{Act}_\phi \ ::= \& \text{destroy}(\text{expr}_1) \mid \text{expr}_1 \in \text{OCLExp}_v
\end{align*}
\]

\[
\text{Empty}_\phi \ ::= \\text{OCL expressions over } \varphi
\]
Transformers as Abstract Actions!

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
  \[
  I[\cdot](\cdot, \cdot) : \text{Expr} \to (\Sigma_{\text{BW}} \times (\text{BW})) \to B
  \]
  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 $\text{act} \in \text{Act}$, we assume to have
  \[
  \tau_{\text{act}} \subseteq \mathcal{P}(\mathcal{E}) \times (\Sigma_{\text{BW}} \times \text{Eth}) \times (\Sigma_{\text{BW}} \times \text{Eth})
  \]

### 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
### Transformer Examples: Presentation

<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>object <em>exactly</em> acme op</td>
</tr>
<tr>
<td></td>
<td>((\sigma, \varepsilon), (\sigma', \varepsilon') \in t_{op}[u_x] \text{ iff } ... ) or (t_{op}[u_x](\sigma, \varepsilon) = {(\sigma', \varepsilon')} \text{ where } ... )</td>
</tr>
<tr>
<td>observables</td>
<td>(\text{Obs}_{op}[u_x] = {\ldots}, \text{ not a relation, depends on choice} )</td>
</tr>
<tr>
<td>(error) conditions</td>
<td>Not defined if ...</td>
</tr>
</tbody>
</table>

### Transformer: Skip

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>skip</td>
</tr>
<tr>
<td>intuitive semantics</td>
<td>do nothing</td>
</tr>
<tr>
<td>well-typedness</td>
<td>./</td>
</tr>
<tr>
<td>semantics</td>
<td>(t[u_x](\sigma, \varepsilon) = {(\sigma, \varepsilon)} )</td>
</tr>
<tr>
<td>observables</td>
<td>(\text{Obs}_{\text{skip}}[u_x](\sigma, \varepsilon) = \emptyset )</td>
</tr>
<tr>
<td>(error) conditions</td>
<td></td>
</tr>
</tbody>
</table>
Transformer: Update

**Intuitive Semantics**

Update attribute \( v \) in the object denoted by \( \text{expr}_1 \) to the value denoted by \( \text{expr}_2 \).

**Well-typedness**

\( \text{expr}_1 : \tau \) and \( v : \tau \in \text{attr}(C) \);

\( \text{expr}_1, \text{expr}_2 \) obey visibility and navigability.

**Semantics**

\[
\text{update(}\text{expr}_1, v, \text{expr}_2)\left[\sigma, \varepsilon\right] = \{ (\sigma', \varepsilon) \}
\]

where \( \sigma' = \sigma[u \mapsto I[\text{expr}_1]_C](\sigma, \beta) \).

**Observables**

\[ \text{Obs}\text{update(}\text{expr}_1, v, \text{expr}_2\left[\sigma, \varepsilon\right] = \emptyset\text{.} \]

**Error Conditions**

Not defined if \( I[\text{expr}_1]_C(\sigma, \beta) \) or \( I[\text{expr}_2]_C(\sigma, \beta) \) not defined.

---

**Update Transformer Example**

**SM_C:**

\[
\begin{array}{c}
S_1 \xrightarrow{\text{update}} S_2 \\
/ x := x + 1 \\
\end{array}
\]

\[ S_1: x = 4 \quad y = 0 \]

\[ S_2: x = 5 \quad y = 0 \]

**Observables**

\[ \text{Obs}_{\text{update(}x, x + 1\left[\sigma, \varepsilon\right] = \emptyset.} \]

---

**Concrete Syntax**

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

\[
\begin{array}{c}
\text{update} \left( \text{expr}_1, v, \text{expr}_2 \right) \left[\sigma, \varepsilon\right] = \{ (\sigma', \varepsilon) \}
\end{array}
\]

where \( \sigma' = \sigma[u \mapsto I[\text{expr}_1]_C](\sigma, \beta) \).

**Error Conditions**

Not defined if \( I[\text{expr}_1]_C(\sigma, \beta) \) or \( I[\text{expr}_2]_C(\sigma, \beta) \) not defined.
References

