Contents & Goals

Last Lecture:
- Core State Machines
- UML State Machine syntax
- State machines belong to classes.

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:
  - Ether, System Configuration, Transformer
  - Run-to-completion Step
  - Putting It All Together
Recall: UML 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.
**Core State Machine**

**Definition.**
A core state machine over signature \( \mathcal{S} = (\mathcal{P}, \mathcal{E}, \mathcal{V}, \text{atr}, \mathcal{A}) \) is a tuple

\[
M = (S, s_0, \rightarrow)
\]

where
- \( S \) is a non-empty, finite set of **(basic) states**, 
- \( s_0 \in S \) is an **initial state**, 
and
- \( \rightarrow \subseteq S \times (\mathcal{E} \cup \{\_\}) \times \mathit{Expr}_\mathcal{S} \times \mathit{Act}_\mathcal{S} \times S \) is a labelled transition relation.

We assume a set \( \mathit{Expr}_\mathcal{S} \) of boolean expressions over \( \mathcal{S} \) (for instance OCL, may be something else) and a set \( \mathit{Act}_\mathcal{S} \) of actions.

---

**From UML to Core State Machines: By Example**

UML state machine diagram \( SM \):

\[
\text{annot} := \left[ \text{event} \mid \text{guard} \mid \text{action} \right]
\]

maps to

\[
M(SM) = \{(s_1, s_2), (s_1, (\text{event}, \text{guard}, \text{action}, s_2))\}
\]
The Basic Causality Model

6.2.3 The Basic Causality Model [OMG, 2007b, 12]

"'Causality model' is a specification of how things happen at run time [...].

The causality model is quite straightforward:

- Objects respond to messages that are generated by objects executing communication actions.
- When these messages arrive, the receiving objects eventually respond by executing the behavior that is matched to that message.
- The dispatching method by which a particular behavior is associated with a given message depends on the higher-level formalism used and is not defined in the UML specification (i.e., it is a semantic variation point).

The causality model also subsumes behaviors invoking each other and passing information to each other through arguments to parameters of the invoked behavior, [...].

This purely 'procedural' or 'process' model can be used by itself or in conjunction with the object-oriented model of the previous example."
15.3.12 StateMachine [OMG, 2007b, 563]

- Event occurrences are detected, dispatched, and then processed by the state machine, one at a time.
- The semantics of event occurrence processing is based on the **run-to-completion assumption**, interpreted as run-to-completion processing.
- **Run-to-completion processing** means that an event [...] can only be taken from the pool and dispatched if the processing of the previous [...] is fully completed.
- The processing of a single event occurrence by a state machine is known as a **run-to-completion step**.
- Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal-activities (but not necessarily do-activities) completed.
- The same conditions apply after the run-to-completion step is completed.
- Thus, an event occurrence will never be processed [...] in some intermediate and inconsistent situation.
- [IOW.] The run-to-completion step is the passage between two stable configurations of the state machine.
- The run-to-completion assumption simplifies the transition function of the StM, since concurrency conflicts are avoided during the processing of event, allowing the StM to safely complete its run-to-completion step.

15.3.12 StateMachine [OMG, 2007b, 563]

- The order of dequeuing is not defined, leaving open the possibility of modeling different priority-based schemes.
- Run-to-completion may be implemented in various ways. [...]
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.

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.

**Ether aka. Event Pool**

Definition. Let $\mathcal{S} = (\mathcal{F}, \mathcal{E}, V, \text{atr})$ be a signature with signals and $\mathcal{D}$ a structure.

We call a structure $(\mathcal{E}, \text{ready}, \oplus, \ominus, [\cdot])$ 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.
  
  $$\text{ready} : \mathcal{E} \times \mathcal{D}(\mathcal{S}) \rightarrow 2^\mathcal{D}(\mathcal{S})$$

- a operation to **insert** an event destined for a given object, i.e.
  
  $$\oplus : \mathcal{E} \times \mathcal{D}(\mathcal{S}) \times \mathcal{D}(\mathcal{S}) \rightarrow \mathcal{E}$$

- a operation to **remove** an event, i.e.
  
  $$\ominus : \mathcal{E} \rightarrow \mathcal{D}(\mathcal{S})$$

- an operation to clear the ether for a given object, i.e.
  
  $$[\cdot] : \mathcal{E} \times \mathcal{D}(\mathcal{S}) \rightarrow \mathcal{E}.$$
Ether: Examples

- A (single, global, shared, reliable) FIFO queue is an ether.
  - Eth:
    - The set of finite sequences of \((u,e)\)-pairs, \(u \in \mathcal{D}(X)^0\), \(e \in \mathcal{D}(X0)^0\)
    - \(\text{ready}(u,e,v) = \{e\}\), \(\text{ready}(u,e,v) = \emptyset, v \neq u\)
    - \(\oplus(u,e,v) = 2\), \(\oplus(u,e,v) = 0\)
  - One FIFO queue per active object is an ether.
  - Lossy queue.
  - One-place buffer.
  - Priority queue.
  - Multi-queues (one per sender).
  - Trivial example: sink, "black hole".
  - . . .

15.3.12 StateMachine [OMG, 2007b, 563]

- The order of dequeuing is not defined, leaving open the possibility of modeling different priority-based schemes.
- Run-to-completion may be implemented in various ways. [...]
**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 $\mathcal{I}_0 = (\mathcal{P}_0, \mathcal{C}_0, V_0, \text{atr}_0, \mathcal{E})$ be a signature with signals, $\mathcal{P}_0$ a structure of $\mathcal{I}_0$, (Eth, ready, ⊕, ⊖, [·]) an ether over $\mathcal{I}_0$ and $\mathcal{P}_0$. Furthermore assume there is one core state machine $M_C$ per class $C ∈ \mathcal{C}$.

A **system configuration** over $\mathcal{I}_0$, $\mathcal{P}_0$, and Eth is a pair $(σ, ε) ∈ \Sigma[\mathcal{P}_0] × Eth$

where

- $\mathcal{I} = (\mathcal{P}_0 \cup \{S_{MC} | C ∈ \mathcal{C}\}, \mathcal{C}_0,$
  - $V_0 \cup \{(\text{stable} : \text{Bool}, -, \text{true}, \emptyset)\}$
  - $\cup \{(st_C : S_{MC}, +, s_0, \emptyset) | C ∈ \mathcal{C}\}$
  - $\cup \{(\text{params}_E : E_{0,1}, +, 0, \emptyset) | E ∈ \mathcal{E}\}$,
  - $\{C \mapsto \text{atr}_0(C)\}$
  - $\cup \{(\text{stable}, st_C) \cup \{\text{params}_E | E ∈ \mathcal{E}\} | C ∈ \mathcal{C}\}$) $\mathcal{E}(\mathcal{I})$
- $\mathcal{D} = \mathcal{P}_0 \cup \{S_{MC} \mapsto S(M_C) | C ∈ \mathcal{C}\}$, and
- $σ(u)(r) \cap \mathcal{D}(\mathcal{E}(\mathcal{I})) = \emptyset$ for each $u ∈ \text{dom}(σ)$ and $r ∈ V_0$. 

(x) maybe better: no associations to signals, i.e.,

$\forall (v; C_0) ∈ V_0 \cup \{E ∈ \mathcal{E}\}$ $\mathcal{E}(\mathcal{I})$

$\forall (v; C_k) ∈ V_0 \cup C \in \mathcal{E}$ $\mathcal{E}(\mathcal{I})$
System Configuration Step-by-Step

- We start with some signature with signals $\mathcal{S}_0 = (\mathcal{R}_0, \mathcal{E}_0, V_0, atr_0)$.  
- A system configuration is a pair $(\sigma, \varepsilon)$ which comprises a system state $\sigma$ wrt. $\mathcal{S}$ (not wrt. $\mathcal{S}_0$).  
- 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 $\mathcal{S}_0, \mathcal{R}_0, Eth$.  
We call an object $u \in \text{dom}(\sigma) \cap \mathcal{R}(\mathcal{S}_0)$ stable in $\sigma$ if and only if
\[
\sigma(u)(\text{stable}) = \text{true}.
\]
**Events Are Instances of Signals**

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

Let $\text{atr}(E) = \{v_1, \ldots, v_n\}$. We call $e = (E, \{v_1 \mapsto d_1, \ldots, v_n \mapsto d_n\})$, $e \in (E \times (V_0 \rightarrow D_{\text{attr}}))$, or shorter (if mapping is clear from context) $(E, (d_1, \ldots, d_n))$ or $(E, \vec{d})$,

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

We use $\text{Evs}(\delta_0, \mathcal{D}_0)$ 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, $u \in \mathcal{R}(E)$ is also called instance of the (signal) class $E$ in system configuration $(\sigma, \varepsilon)$ 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 of system configurations. $(\sigma, \varepsilon)$,
- **Identities** of signal instances are kept in the ether $E$.
- 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
  \[
  (\sigma, \varepsilon) \xrightarrow{(\text{cons, Snd})} (\sigma', \varepsilon')
  \]
on system configuration, labelled with the **consumed** and **sent** events,
  \((\sigma', \varepsilon')\) being the result (or effect) of one object taking a transition of
  its state machine.
- **Have**: system configuration \((\sigma, \varepsilon)\) comprising current state machine state
  and stability flag for each object, and the ether.

**Plan**:

(i) Introduce transformer as the semantics of action annotations.

   **Intuitively**, \((\sigma', \varepsilon')\) is the effect of applying the transformer
   of the taken transition.

(ii) Explain how to choose transitions depending on \(\varepsilon\) and when to stop taking
    transitions — the run-to-completion "algorithm".

---

**Transformer**

**Definition.**

Let \(\Sigma_{\mathcal{P}}\) the set of system configurations over some \(\mathcal{S}_0, \mathcal{P}_0, \mathcal{E}_0\).

We call a partial function
\[
t : \Sigma_{\mathcal{P}} \xrightarrow{\mathcal{E}_0} \Sigma_{\mathcal{P}} \times \mathcal{E}_0
\]
a (system configuration) transformer.

- In the following, we assume that each application of a transformer \(t\) to
  some system configuration \((\sigma, \varepsilon)\) is associated with a set of observations
  \[
  \text{Obs}_t(\sigma, \varepsilon) \in \mathcal{P}(\mathcal{E}) \times \text{Evs}(\mathcal{E}_t \cup \{+\}) \times \mathcal{P}(\mathcal{E}_t).
  \]
- An observation
  \[
  (u_{\text{src}}, (E, \bar{d}), u_{\text{dst}}) \in \text{Obs}_t(\sigma, \varepsilon)
  \]
  represents the information that, as a "side effect" of \(t\), an event \((E, \bar{d})\) has
  been sent from \(u_{\text{src}}\) to \(u_{\text{dst}}\).
**Why Transformers?**

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

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

- **Clear**: \langle \text{event}\rangle is from \& 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

---

**Transformers as Abstract Actions!**

In the following, we assume that we’re **given**

- an **expression language** \textit{Expr} for guards, and
- an **action language** \textit{Act} for actions,

and that we’re **given**

- a **semantics** for boolean expressions in form of a partial function

  \[
  I\lbrack\cdot\rbrack\lbrack\cdot\rbrack : \text{Expr} \rightarrow (\Sigma \rightarrow \text{B})
  \]

  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.
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 $ε$ — interesting, because state machines are built around sending/consuming events
- **create/destroy**: modify domain of $σ$ — 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

References
References

