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.
Definition.
A core state machine over signature $\mathcal{S} = (T, C, V, atr, \mathcal{E})$ 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

$$\rightarrow \subseteq S \times (\mathcal{E} \cup \{\emptyset\}) \times Expr_T \times Act_T \times S$$

is a labelled transition relation.

We assume a set $Expr_T$ of boolean expressions over $\mathcal{S}$ (for instance OCL, may be something else) and a set $Act_T$ of actions.
From UML to Core State Machines: By Example

UML state machine diagram $SM$:

\[
\text{annot} := [ (\text{event})[\cdot \langle \text{event} \rangle]^* ] [ [ \langle \text{guard} \rangle ] ] [ [ / \langle \text{action} \rangle ] ] \\
\]

with

- $\text{event} \in \mathcal{E}$,
- $\text{guard} \in \text{Expr}_{\mathcal{F}}$
- $\text{action} \in \text{Act}_{\mathcal{F}}$

maps to

\[
M(SM) = (\{s_1, s_2\}, s_1, (s_1, \text{event, guard, action, s}_2))
\]

initial state

\[
S \xrightarrow{s_0}
\]
\[ Y = \left\{ \text{Int}, \right\} + \left\{ (C, 0, 0), \text{Int}, y = 0 \right\} + \left\{ (C, \text{Int}, x = x), \text{Int}, y = y \right\} + \left\{ (C, \text{Int}, x = x), \text{Int}, y = y \right\} + \left\{ (C, \text{Int}, x = x), \text{Int}, y = y \right\} \]

\[ \varepsilon(Y) = \{ E, \text{Int} \} \]
The Basic Causality Model
“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 state 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.
• 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**. [...]

**OMG, 2007b, 563**
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.
**Definition.** Let \( \mathcal{I} = (\mathcal{T}, \mathcal{C}, V, \text{atr}) \) be a signature with signals and \( \mathcal{D} \) a structure.

We call a structure \((\text{Eth}, \text{ready}, \oplus, \ominus, [\cdot])\) an ether over \(\mathcal{I}\) 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.
  \[
  \text{ready} : \text{Eth} \times \mathcal{D}(\mathcal{C}) \to 2^{\mathcal{D}(E(S))}
  \]

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

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

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

- A (single, global, shared, reliable) FIFO queue is an ether:
  - Eth:
    - the set of finite sequences of \((v,e)\)-pairs \(v \in D(e), e \in D(ready)\)
    
    \[
    \text{ready}(v,e) = \begin{cases} 
    \varepsilon & \text{ready}(v,e,v) = \emptyset, v \neq v, \Theta(\emptyset,v) = \emptyset \\
    (\varepsilon,v,e) & \text{ready}(\emptyset,v) = \emptyset \\
    (\varepsilon,v,e) & \Theta(\emptyset,v) = \emptyset \\
    (\varepsilon,v,e) & \emptyset \neq v, \Theta(\emptyset,v) = \emptyset \\
    \end{cases}
    \]
  - \([\cdot]:\cdot\)

- One FIFO queue per active object is an ether.
  - Lossy queue. (would need \(\Theta\) to yield sets of ethers)

- 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.
**Definition.** Let $\mathcal{I}_0 = (\mathcal{I}_0, \mathcal{C}_0, V_0, atr_0, \mathcal{D})$ be a signature with signals, $\mathcal{D}_0$ a structure of $\mathcal{I}_0$, $(Eth, ready, \oplus, \ominus, [\cdot])$ an ether over $\mathcal{I}_0$ and $\mathcal{D}_0$. Furthermore assume there is one core state machine $M_C$ per class $C \in \mathcal{C}$.

A system configuration over $\mathcal{I}_0$, $\mathcal{D}_0$, and $Eth$ is a pair $(\sigma, \varepsilon) \in \Sigma_{\mathcal{D}} \times Eth$ where

- $\mathcal{I} = (\mathcal{I}_0 \cup \{S_{MC} \mid C \in \mathcal{C}\}, \mathcal{C}_0,$
  
  $V_0 \cup \{(\text{stable} : \text{Bool}, -, \text{true}, \emptyset)\}$
  
  $\cup \{\langle stC : S_{MC}, +, s_0, \emptyset \rangle \mid C \in \mathcal{C}\}$
  
  $\cup \{\langle \text{params}_E : E_{0,1}, +, \emptyset, \emptyset \rangle \mid E \in \mathcal{E}\}$,
  
  $\{C \mapsto atr_0(C)$
  
  $\cup \{\text{stable}, stC\} \cup \{\text{params}_E \mid E \in \mathcal{E}\} \mid C \in \mathcal{C}\}$

- $\mathcal{D} = \mathcal{D}_0 \cup \{S_{MC} \mapsto S(M_C) \mid C \in \mathcal{C}\}$, and

- $\sigma(u)(r) \cap \mathcal{D}(\mathcal{E}_{\mathcal{D}}) = \emptyset$ for each $u \in \text{dom}(\sigma)$ and $r \in V_0$. 

(x)
System Configuration Step-by-Step

- We start with some signature with signals $S_0 = (T_0, C_0, V_0, atr_0, )$.

- A **system configuration** is a pair $(\sigma, \varepsilon)$ which comprises a system state $\sigma$ wrt. $S$ (not wrt. $S_0$).

- Such a **system state** $\sigma$ wrt. $S$ 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. $\sigma(u)(stable)$ is a boolean value,
    - a **current (state machine) state**, i.e. $\sigma(u)(st)$ denotes one of the states of core state machine $M_C$,
    - a temporary association to access **event parameters** for each class, i.e. $\sigma(u)(params_E)$ is defined for each $E \in E$.

- For convenience require: there is **no link to an event** except for $params_E$. 
Definition.
Let \((\sigma, \varepsilon)\) be a system configuration over some \(\mathcal{I}_0, \mathcal{D}_0, \text{Eth}\).
We call an object \(u \in \text{dom}(\sigma) \cap \mathcal{D}(\mathcal{C}_0)\) stable in \(\sigma\) if and only if
\[
\sigma(u)(\text{stable}) = \text{true}.
\]
**Definition.** Let $\mathcal{D}_0$ be a structure of the signature with signals $\mathcal{S}_0 = (\mathcal{I}_0, \mathcal{C}_0, V_0, atr_0)$ and let $E \in \mathcal{I}_0$ be a signal. Let $atr(E) = \{v_1, \ldots, v_n\}$. We call

$$e = (E, \{v_1 \mapsto d_1, \ldots, v_n \mapsto d_n\}), \quad E \in \mathcal{S}_0 \times (V_0 \cup \mathcal{D}(E)) \times (\mathcal{D}(E) \cup V_0)$$

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 $Evs(\mathcal{S}_0, \mathcal{D}_0)$ 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 $(\sigma, \varepsilon)$ if $u \in \text{dom}(\sigma)$.
- The corresponding event is then $(E, \sigma(u))$. 


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**.\(\varepsilon\).

- 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

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

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. From the current state \(\sigma(v)(st_e)\).

- **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”.

\[E[n \neq \emptyset]/x := x + 1; n \! F\]

\[F/x := 0\]

\[/n := \emptyset\]

\[s_1 \quad s_2 \quad s_3\]
Definition.
Let $\Sigma$ the set of system configurations over some $\mathcal{I}_0, \mathcal{D}_0, \mathcal{E}_0$.
We call a partial function
\[
t : \Sigma \to \Sigma \times \mathcal{E}_0
\]
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
  \[
  \text{Obs}_t(\sigma, \varepsilon) \in 2^{\mathcal{D}(\mathcal{C}) \times \text{Evs}(\mathcal{E} \cup \{\ast, +\}, \mathcal{D}) \times \mathcal{D}(\mathcal{C})}.
  \]
- An observation
  \[
  (u_{\text{src}}, (E, \vec{d}), u_{\text{dst}}) \in \text{Obs}_t(\sigma, \varepsilon)
  \]
  represents the information that, as a “side effect” of $t$, an event $(E, \vec{d})$ has been sent from $u_{\text{src}}$ to $u_{\text{dst}}$. 
Why Transformers?

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

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

- **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++, ... 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

$$I[\cdot](\cdot) : Expr \rightarrow (\Sigma_{\mathcal{G}} \rightarrow \mathbb{B})$$

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.
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
References
References

