Contents & Goals

Last Lecture:
- State machine syntax
- Core state machines

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:
  - The basic causality model
  - Ether
  - System Configuration, Transformer
  - Examples for transformer
  - Run-to-completion Step
Recall: Core State Machines
Definition.
A core state machine over signature $\mathfrak{S} = (T, C, V, atr, E)$ 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

$$\rightarrow \subseteq S \times (E \cup \{\_\}) \times Expr_{\mathfrak{S}} \times Act_{\mathfrak{S}} \times S$$

is a labelled transition relation.

We assume a set $Expr_{\mathfrak{S}}$ of boolean expressions (may be OCL, may be something else) and a set $Act_{\mathfrak{S}}$ of actions over $\mathfrak{S}$. 
UML state machine diagram $SM$:

$annot ::= [\langle event\rangle[\ '.']\langle event\rangle^* [\ ['']\langle guard\rangle [''] ] [\ '/']\langle action\rangle ] ]$

with

- $\text{event} \in \mathcal{E}$,
- $\text{guard} \in \text{Expr}$, $\text{guard} \text{ default: true, assumed to be in Expr}$
- $\text{action} \in \text{Act}$, $\text{action} \text{ default: skip, assumed to be in Act}$

maps to

$$M(SM) = (\{s_1, s_2\}, s_0, (s_1, \text{event}, \text{guard}, \text{action}, s_2))$$
Reconsider the syntax of transition annotations:

\[
\text{annot} ::= \left[ \langle\text{event}\rangle[\ '.'] \langle\text{event}\rangle]^* \left[ \left[ '\langle\text{guard}\rangle' \right] \right] \left[ '/' \langle\text{action}\rangle\right] \right]
\]

and let's play a bit with the defaults:

\[
\begin{align*}
E / & \rightsquigarrow E \text{ true / skip} \\
E / \text{ act} & \rightsquigarrow E \text{ true / act} \\
\text{[exp]} / & \rightsquigarrow \text{[exp]} \text{ skip} \\
\text{[exp]} / \text{ act} & \rightsquigarrow \text{[exp]} \text{ act}
\end{align*}
\]

In the standard, the syntax is even more elaborate:

- \(E(v)\) — when consuming \(E\) in object \(u\), attribute \(v\) of \(u\) is assigned the corresponding attribute of \(E\).
- \(E(v : \tau)\) — similar, but \(v\) is a local variable, scope is the transition
What is that useful for?

- **No Event:**

- **No annotation:**
State-Machines belong to Classes

- In the following, we assume that a UML models consists of a set $\mathcal{CD}$ of class diagrams and a set $\mathcal{SM}$ of state chart diagrams (each comprising one state machines $SM$).

- Furthermore, we assume that each state machine $SM \in \mathcal{SM}$ is associated with a class $C_{SM} \in \mathcal{C}(\mathcal{I})$.

- For simplicity, we even assume a bijection, i.e. we assume that each class $C \in \mathcal{C}(\mathcal{I})$ has a state machine $SM_C$ and that its class $C_{SM_C}$ is $C$.

  If not explicitly given, then this one:

  $$SM_0 := (\{s_0\}, s_0, \emptyset).$$

  We’ll see later that, semantically, this choice does no harm.

- Intuition 1: $SM_C$ describes the behaviour of the instances of class $C$.

- Intuition 2: Each instance of $C$ executes $SM_C$ with own “program counter”.

Note: we don’t consider multiple state machines per class.

(Because later (when we have AND-states) we’ll see that this case can be viewed as a single state machine with as many AND-states.)
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.”
6.2.3 The Basic Causality Model \[? , 12\]

- 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.
15.3.12 StateMachine [?, 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**.
15.3.12 StateMachine \[?, 563\]

- 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. [...]
Our choice here:
implies "set all x's of all c's to -1."

rest steps the same
...:

- 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**.
System Configuration, Ether, Transformer
**Definition.** Let $\mathcal{I} = (\mathcal{T}, \mathcal{C}, V, atr, \mathcal{E})$ be a signature with signals and $\mathcal{D}$ a structure.

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

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

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

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

- A (single, global, shared, reliable) FIFO queue is an ether:
  - \( Eth = (\mathcal{D}(E) \times \mathcal{D}(E))^* \) e.g. \( E = (u,e), (v,f), (w,e_2) \)
  - the set of all finite sequences of pairs \((u,e) \in \mathcal{D}(E) \times \mathcal{D}(E)\)
  - \( \text{ready}(E, u, e, v) = \{ \{(u,e)\} \) if \( v = u \) \( \emptyset \) otherwise
  - \( \oplus(E, u, e) = E \cup \{u,e\} \)
  - \( \ominus(E, u, e, f) = \{e\) if \( f = e \)
  - \( \ominus(E, u, e, f) = \{e\) otherwise
  - \([\cdot]\): remove all \((u,e)\) pairs from a given sequence

- One FIFO queue per active object is an ether.

- Lossy queue (\( \oplus \) becomes a relation then).

- One-place buffer.

- Priority queue.

- Multi-queues (one per sender).

- Trivial example: sink, “black hole”.

- …
15.3.12 StateMachine [?, 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. [...]
