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
Core State Machine

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

\[ \rightarrow \subseteq S \times (\mathcal{E} \cup \{\} \times \text{Expr}_\mathcal{I} \times \text{Act}_\mathcal{I} \times S) \]

is a labelled transition relation.

We assume a set \( \text{Expr}_\mathcal{I} \) of boolean expressions over \( \mathcal{I} \) (for instance OCL, may be something else) and a set \( \text{Act}_\mathcal{I} \) of **actions**.
From UML to Core State Machines: By Example

UML state machine diagram $SM$:

$annot := [ [⟨event⟩[':'⟨event⟩]* ] [ ['⟨guard⟩'] ] [ '/⟨action⟩'] ]$

with

- $event ∈ E$,
- $guard ∈ Expr_{SP}$
- $action ∈ Act_{SP}$

maps to

$M(SM) = (\{s_1, s_2\}, s_1, (s_1, event, guard, action, s_2))$
\[ Y = \{ \text{let}, \{ (x, 0, 0), (E, \text{signal}, 0, 0), \langle T, \text{signal}, 0, 0 \rangle \}, \{ x: \text{let}, y: \text{let}, n: \text{C}, n \rangle \}, \{ C \rightarrow x; n, E \rightarrow y; n \} \} \]

\[ M = \{ s_1, s_2, s_3, s_4, s_5, \]

\[ \{ s_1, \ldots, \text{true}, s_3, \}

\[ (s_1, E, \text{not occlude}(C)),

\[ n, \text{true}, s_2) \} \]

\[ \text{UML} \]

\[ \text{CD:} \]

\[ \text{Expr: OCL over Y} \]

\[ \text{Acc: \{} \text{skip}, x++, \} \]

\[ \text{C} \]

\[ E \]

\[ y: \text{let} \]

\[ \text{F} \]

\[ \text{MATH} \]
The Basic Causality Model
6.2.3 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."
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 \textit{run-to-completion assumption}, interpreted as \textit{run-to-completion processing}.

\textbf{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 \textbf{run-to-completion step}.

Before commencing on a \textbf{run-to-completion step}, a state machine is in a \textbf{stable state} configuration with all entry/exit/internal-activities (but not necessarily do-activities) completed.

The same conditions apply after the \textbf{run-to-completion step} is completed.

Thus, an event occurrence will never be processed [...] in some intermediate and inconsistent situation.

[IOW,] The \textbf{run-to-completion step} is the passage between two state configurations of the state machine.

The \textbf{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 \textbf{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?

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

\[ F/x := 0 \]

\[ s_1 \quad s_2 \]

\[ s_3 \]

\[ /n := \emptyset \]

...:

- 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
**Definition.** Let $\mathcal{I} = (\mathcal{I}, \mathcal{C}, V, \text{attr})$ 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.

  $$\text{ready}: \text{Eth} \times \mathcal{D}(\mathcal{C}) \rightarrow 2^{\mathcal{D}(\mathcal{E})}$$

- a operation to **insert** an event destined for a given object, i.e.

  $$\oplus: \text{Eth} \times \mathcal{D}(\mathcal{C}) \times \mathcal{D}(\mathcal{E}) \rightarrow \text{Eth}$$

- a operation to **remove** an event, i.e.

  $$\ominus: \text{Eth} \times \mathcal{D}(\mathcal{E}) \rightarrow \text{Eth}$$

- an operation to clear the ether for a given object, i.e.

  $$[\cdot]: \text{Eth} \times \mathcal{D}(\mathcal{C}) \rightarrow \text{Eth}.$$
Ether: Examples

- A (single, global, shared, reliable) FIFO queue is an ether:
  - \( Eth: \)
    - the set of finite sequence of pairs \((u, e), u \in D(E), e \in D(E)\)
  - \(ready\)\( (u, e, u) = \{ (u, e) \}\), \(ready\)\( (v, e, u) = \emptyset\) if \(v \neq u\), \(ready\)\( (\cdot, u) = \emptyset\)
  - \(\oplus\)\( (e, u, e) = \_\)\( (u, e)\)
  - \(\ominus\)\( (u, e, \_e) = \_\), \(\Theta\)\( (u, e, e, e') = \_\) if \(e' \neq e\), \(\Theta\)\( (\cdot, e) = \cdot\)
  - \([\_e](u)\): remove all \((u, e)\) pairs from \(e\)

- 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  \[\text{OMG, 2007b, 563}\]

- 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**. [...]
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.
References
References

