Software Design, Modelling and Analysis in UML

Lecture 12: Core State Machines II

2014-12-09

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

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 Machine

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

$$\rightarrow \subseteq S \times \underbrace{(\mathscr{E} \dot{\cup} \{\_\})}_{\text{trigger}} \times \underbrace{Expr_{\mathscr{S}}}_{\text{guard}} \times \underbrace{Act_{\mathscr{S}}}_{\text{action}} \times S$$

  is a labelled transition relation.

We assume a set $Expr_{\mathscr{S}}$ of boolean expressions (may be OCL, may be something else) and a set $Act_{\mathscr{S}}$ of **actions** over $\mathscr{S}$.

From UML to Core State Machines: By Example

UML state machine diagram $\mathcal{SM}$:

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

with

- $event \in \mathscr{E}$,
- $guard \in Expr_{\mathscr{S}}$ (default: $true$, assumed to be in $Expr_{\mathscr{S}}$)
- $action \in Act_{\mathscr{S}}$ (default: skip, assumed to be in $Act_{\mathscr{S}}$)

maps to

$$M(\mathcal{SM}) = (\underbrace{\{s_1, s_2\}}_{S}, \underbrace{s_1}_{s_0}, \underbrace{(s_1, event, guard, action, s_2)}_{\rightarrow})$$

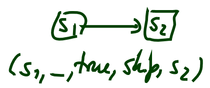
Annotations and Defaults in the Standard

Reconsider the syntax of transition annotations:

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

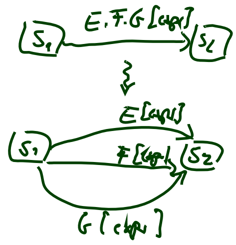
and let's play a bit with the defaults:

	$\rightsquigarrow$	_ [true] / skip
/	$\rightsquigarrow$	_ [true] / skip
E /	$\rightsquigarrow$	E [true] / skip
/ act	$\rightsquigarrow$	_ [true] / act
E / act	$\rightsquigarrow$	E [true] / act



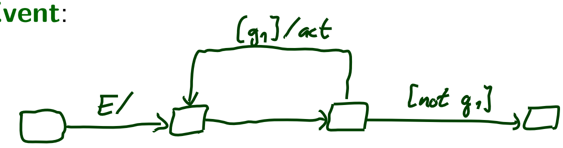
In the standard, the syntax is even more elaborate:

- $E(v)$ — 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

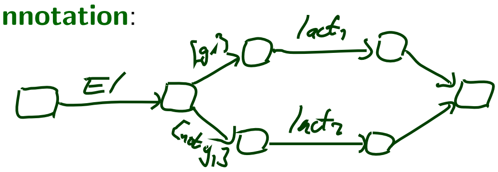


What is that useful for?

- **No Event**:



- **No annotation**:



State-Machines belong to Classes

- In the following, we assume that a UML models consists of a set $\mathscr{CD}$ of class diagrams and a set $\mathscr{SM}$ of **state chart diagrams** (each comprising one **state machines** $\mathcal{SM}$).
- Furthermore, we assume that **each state machine** $\mathcal{SM} \in \mathscr{SM}$ is **associated** with **a class** $C_{\mathcal{SM}} \in \mathscr{C}(\mathscr{S})$.
- For simplicity, we even assume a bijection, i.e. we assume that each class $C \in \mathscr{C}(\mathscr{S})$ has a state machine $\mathcal{SM}_C$ and that its class $C_{\mathcal{SM}_C}$ is C.

  If not explicitly given, then this one:

$$\mathcal{SM}_0 := (\{s_0\}, s_0, \emptyset).$$

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

- **Intuition 1**: $\mathcal{SM}_C$ describes the behaviour of **the instances** of class C.
  Intuition 2: Each instance of C executes $\mathcal{SM}_C$ 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

6.2.3 The Basic Causality Model [?, 12]

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

The causality model is quite straightforward:

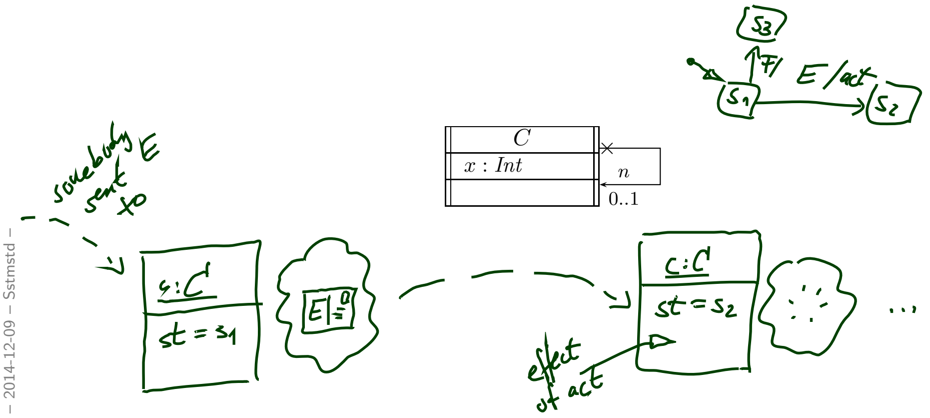
- *Objects respond to **messages** that are generated by objects executing communication actions.*
- *When these messages arrive, the receiving objects eventually respond by executing the behavior that is **matched** to that message.*
- *The dispatching method by which a particular behavior is associated 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."

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.*

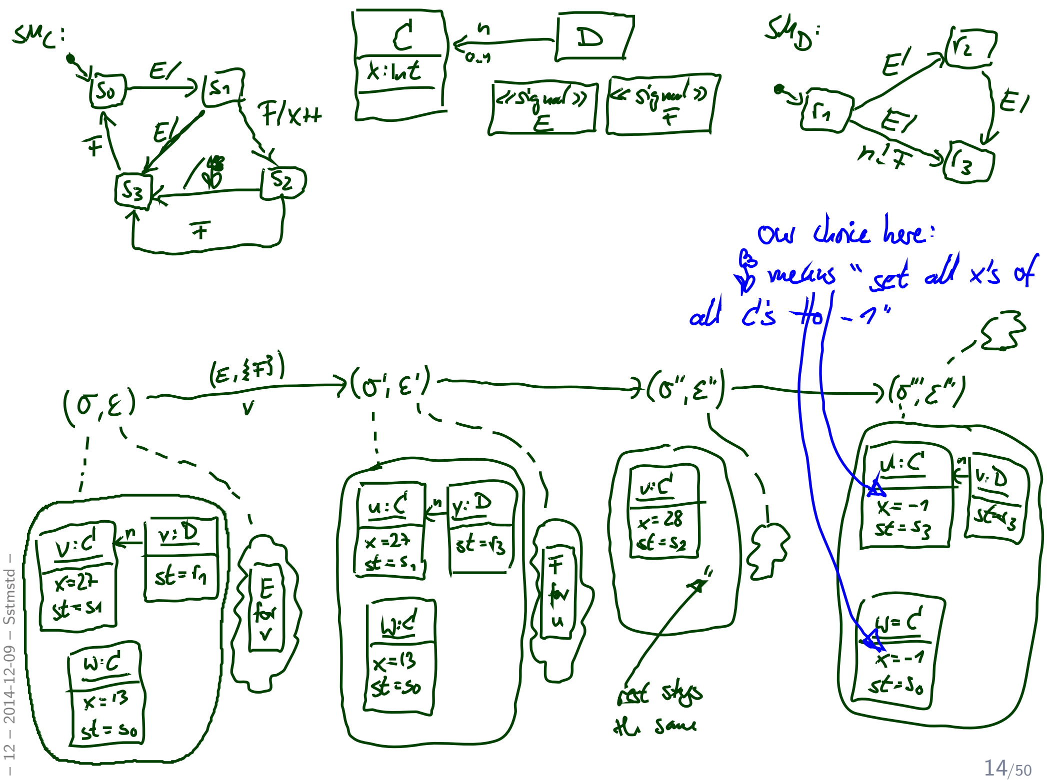


15.3.12 StateMachine [?, 563]

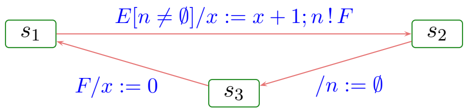
- Event occurrences are detected, dispatched, and then processed by the state machine, one at a time.
- The semantics of event occurrence processing is based on the **run-to- completion assumption**, interpreted as **run-to-completion processing**.
- **Run-to-completion processing** means that an event [...] 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**. [...]

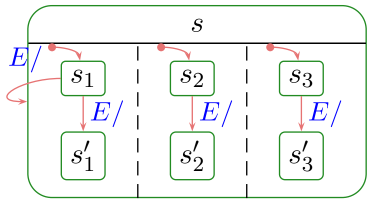


And?



- **...**:
  - 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.



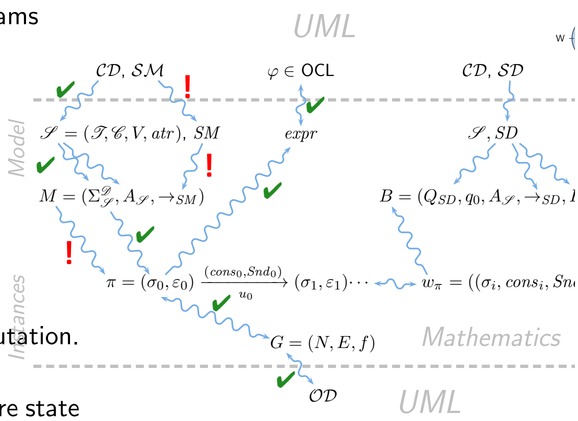
Roadmap: Chronologically

(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**.



System Configuration, Ether, Transformer

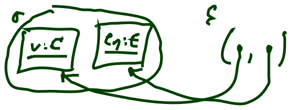
Ether aka. Event Pool

Definition. Let $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$ be a signature with signals and $\mathscr{D}$ a structure.

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

- a **ready** operation which yields a set of events that are ready for a given object, i.e.
$$ready : Eth \times \mathscr{D}(\mathscr{C}) \to 2^{\mathscr{D}(\mathscr{E})}$$
- a operation to **insert** an event destined for a given object, i.e.
$$\oplus : Eth \times \mathscr{D}(\mathscr{C}) \times \mathscr{D}(\mathscr{E}) \to Eth$$
- a operation to **remove** an event, i.e.
$$\ominus : Eth \times \mathscr{D}(\mathscr{E}) \to Eth$$
- an operation to clear the ether for a given object, i.e.
$$[\,\cdot\,] : Eth \times \mathscr{D}(\mathscr{C}) \to Eth.$$

Ether: Examples



- A (single, global, shared, reliable) FIFO queue is an ether:
  - $Eth := (\mathscr{D}(\mathscr{C}) \times \mathscr{D}(\mathscr{E}))^*$ e.g. $\varepsilon = (u,e_1),(v,f_1),(w,e_2)$
    the set of all finite sequences of pairs $(u,e) \in \mathscr{D}(\mathscr{C}) \times \mathscr{D}(\mathscr{E})$
  - $ready((u,e).\varepsilon, v) = \begin{cases} \{(u,e)\} & \text{if } v = u \\ \emptyset & \text{otherwise} \end{cases}$ $ready(\epsilon, v) = \emptyset$
  - $\oplus(\varepsilon, u, e) = \varepsilon.(u,e)$
  - $\ominus((u,e).\varepsilon, f) = \begin{cases} \varepsilon & \text{if } f = e \\ (u,e).\varepsilon & \text{otherwise} \end{cases}$ $\ominus(\epsilon, f) = \epsilon$ empty seq.
  - $[\cdot]$: 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**. [...]