Software Design, Modelling and Analysis in UML

Lecture 11: Core State Machines II

2012-12-05

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

Albert-Ludwigs-Universität Freiburg, Germany

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

disjoint union: _ should not already be in $\mathscr{E}$ (otherwise rename first)

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

$$M = (S, s_0, \to)$$

where

- S is a non-empty, finite set of **(basic) states**,
- $s_0 \in S$ is an **initial state**,
- and

source state · signals in $\mathscr{S}$ · dest. state

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

is a labelled transition relation.

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

From UML to Core State Machines: By Example

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

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

trigger, guard, action

with

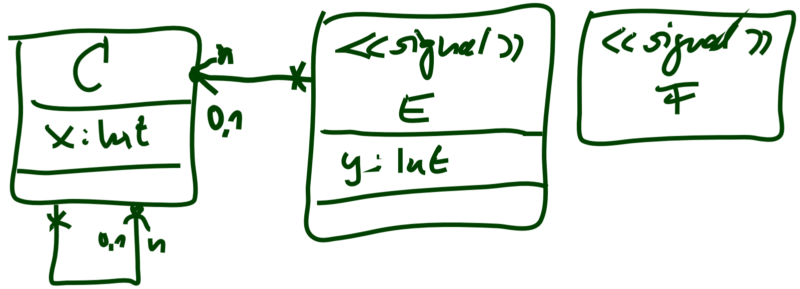
- $event \in \mathscr{E}$, (default: _ if no trigger)
- $guard \in Expr_{\mathscr{S}}$ (default: *true*, assumed to be in $Expr_{\mathscr{S}}$)
- $action \in Act_{\mathscr{S}}$ (default: `skip`, assumed to be in $Act_{\mathscr{S}}$)

maps to

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

initial state

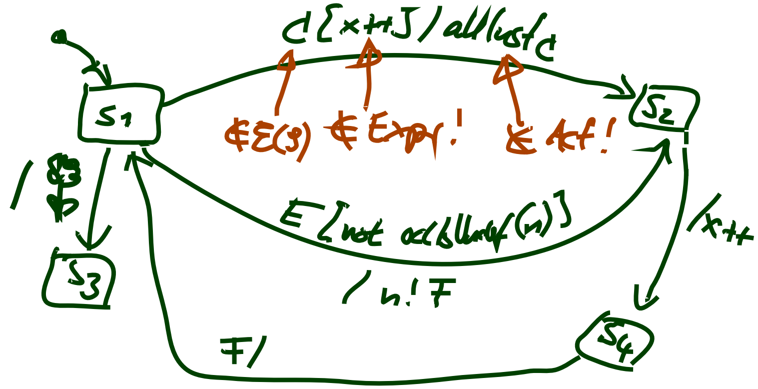
CD:



$Expr_\mathcal{S}$: OCL over $\mathcal{S}$

$Act_\mathcal{S}$: { skip, x++, n!F }

UML



$$\mathcal{S} = \Big(\{ Int \}, \{ \langle C, \emptyset, 0, 0 \rangle, \langle E, \{signal\}, 0, 0 \rangle, \langle F, \{signal\}, 0, 0 \rangle \}, \{ x : Int, y : Int, n : C_{0,1} \}, \{ C \mapsto \{x, n\}, E \mapsto \{y\} \}, \{E, F\} \Big)$$

$$M = \big(\{ s_1, s_2, s_3, s_4 \}, s_1, \{ (s_1, -, true, \ldots, s_3), (s_1, E, not\ oclIsUndef(n), n!F, s_2), \ldots \} \big)$$

MATH

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, [...].

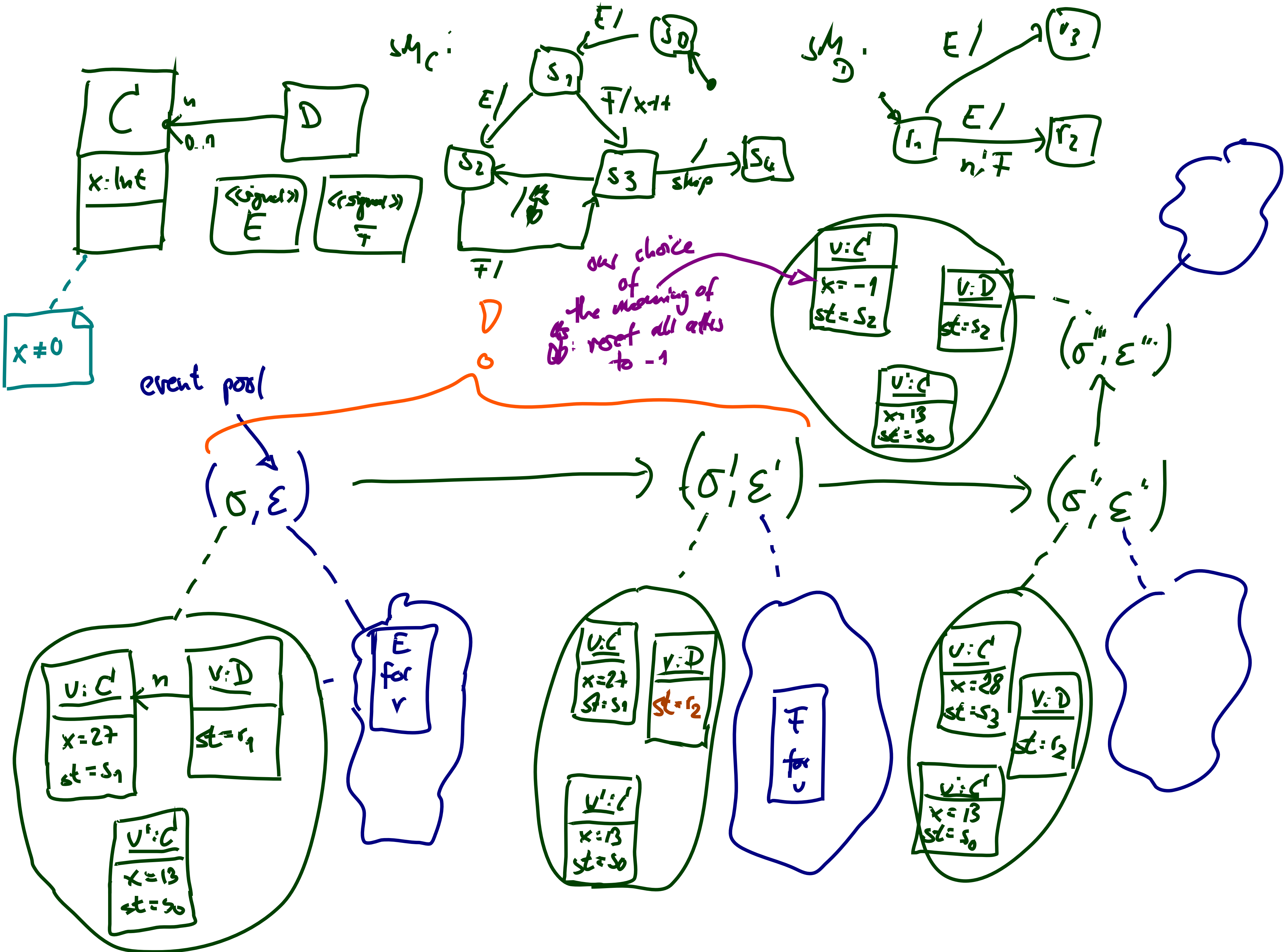
This purely 'procedural' or 'process' model can be used by itself or in conjunction with the object-oriented model of the previous example."

15.3.12 StateMachine [OMG, 2007b, 563]

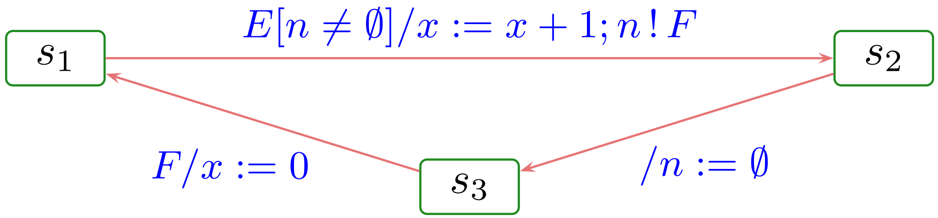
- Event occurrences are detected, dispatched, and then processed by the state machine, one at a time.
- The semantics of event occurrence processing is based on the **run-to-completion assumption**, interpreted as **run-to-completion processing**.
- **Run-to-completion processing** means that an event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

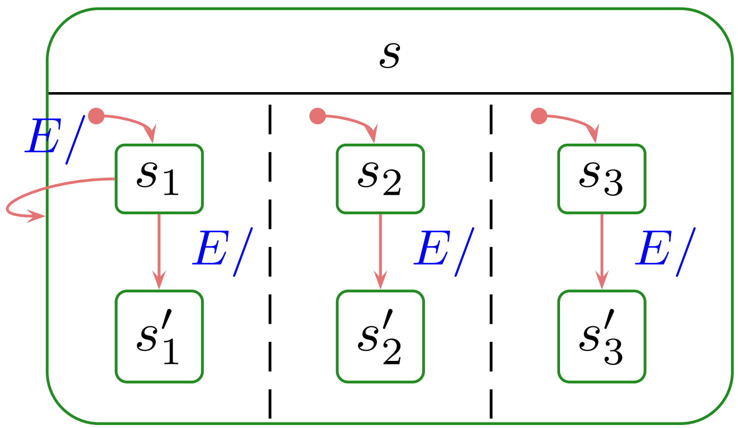

C
x : Int
D
0..1
n
x ≠ 0
«signal» E
«signal» F
SM_C :
S_0
S_1
S_2
S_3
S_4
E /
F / x++
skip
SM_D :
r_1
r_2
r_3
n ! F
D
our choice of the meaning of D: reset all attrs to -1
event pool
(σ, ε)
(σ', ε')
(σ'', ε')
(σ''', ε''')
u : C
x = 27
st = s_1
v : D
st = r_1
u' : C
x = 13
st = s_0
E for v
F for u
x = 28
st = s_3
st = r_2
x = -1
st = s_2

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**.

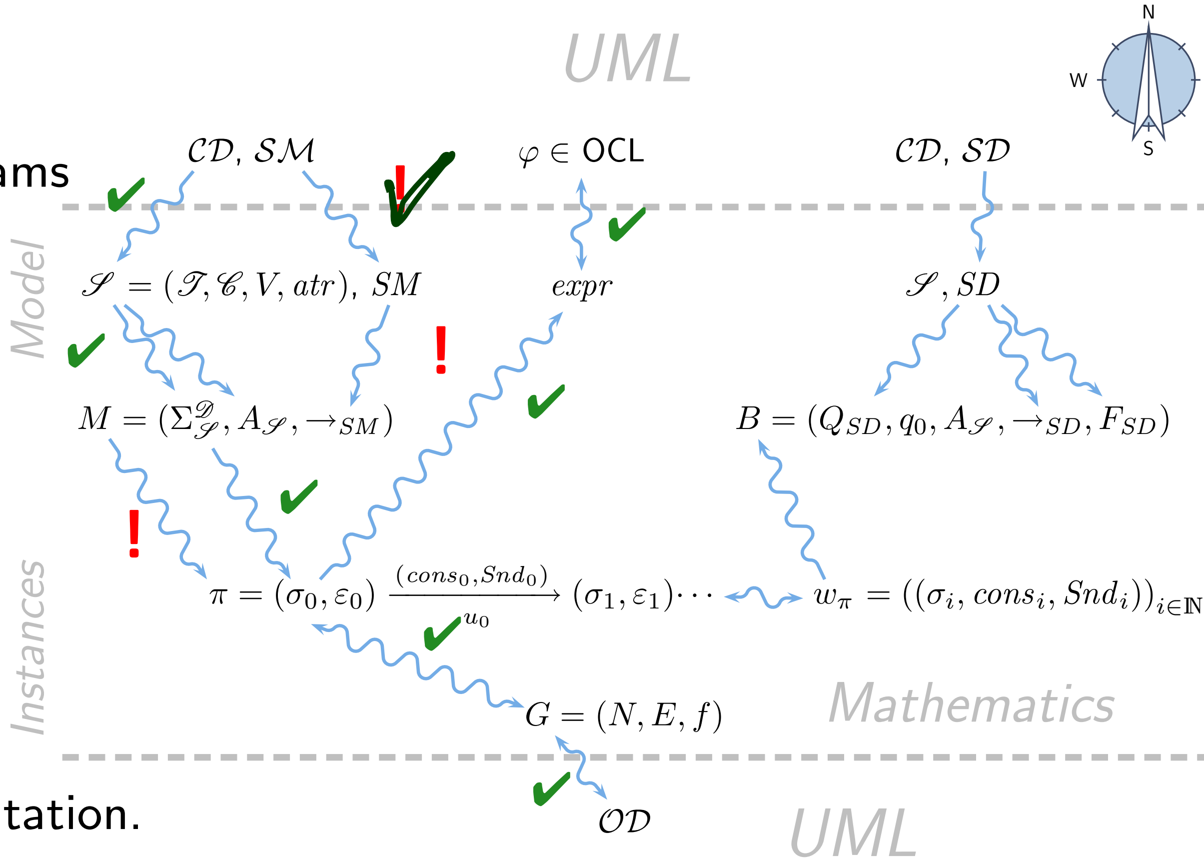
(viii) Def.: **Transformer**.

(ix) Def.: **Transition system**, computation.

(x) Transition relation induced by core state machine.

(xi) Def.: **step**, **run-to-completion step**.

(xii) Later: Hierarchical state machines.



System Configuration, Ether, Transformer

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 ~~structure~~ 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

$(Eth, ready, \oplus, \ominus, [\cdot])$
$ready : Eth \times \mathscr{D}(\mathscr{C}) \to 2^{\mathscr{D}(\mathscr{E})}$
$\oplus : Eth \times \mathscr{D}(\mathscr{C}) \times \mathscr{D}(\mathscr{E}) \to Eth$
$\vdots$

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

References

References

[Harel and Gery, 1997] Harel, D. and Gery, E. (1997). Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42.

[OMG, 2007a] OMG (2007a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-04.

[OMG, 2007b] OMG (2007b). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.