Software Design, Modelling and Analysis in UML

Lecture 11: Core State Machines I

2015-12-10

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

- What makes a class diagram a good class diagram?
- Core State Machine syntax

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:
  - UML standard: basic causality model
  - Ether
  - Transformers
  - Step, Run-to-Completion Step
6.2.3 The Basic Causality Model (OMG, 2011b, 11)

“The 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).
6.2.3 The Basic Causality Model (OMG, 2011b, 11)

“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, 2011b, 574)

- 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. [...]

5/34
Example

$E[n \neq 0]/x := x + 1; n/F$

$F/x := 0$

$(\{E\}, \{F\}) \to (\sigma_1, \varepsilon_1) \to (\sigma_2, \varepsilon_2) \to (\sigma_3, \varepsilon_3) \to (\{F\}, \emptyset) \to (\sigma_4, \varepsilon_4)$

$u_1$

$C$

$E$

$F$

$D$

$n$

Example
Recall: **15.3.12 StateMachine** *(OMG, 2011b, 563)*

- The order of dequeuing is **not defined**, leaving open the possibility of modeling different priority-based schemes.
Ether and OMG (2011b)

The standard distinguishes (among others)

- **SignalEvent** (OMG, 2011b, 450) and **Reception** (OMG, 2011b, 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, 2011b, 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, 2011b, 450)

Our **ether** (→ in a minute) is a general representation of many possible choices.

Often seen minimal requirement: order of sending by one object is preserved.

---

**Ether aka. Event Pool**

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

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

- a **ready** operation which yields a set of events (i.e., signal instances) that are ready for a given object, i.e.

\[
\text{ready} : \text{Eth} \times \mathcal{D}(\mathcal{C}) \to 2^{\mathcal{D}(\mathcal{E})}
\]

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

\[
\oplus : \text{Eth} \times \mathcal{D}(\mathcal{C}) \times \mathcal{D}(\mathcal{E}) \to \text{Eth}
\]

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

\[
\ominus : \text{Eth} \times \mathcal{D}(\mathcal{E}) \to \text{Eth}
\]

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

\[
[\cdot] : \text{Eth} \times \mathcal{D}(\mathcal{C}) \to \text{Eth}.
\]
**Example: FIFO Queue**

A (single, global, shared, reliable) FIFO queue is an ether:

- \( Eth = (\mathcal{D}(E) \times \mathcal{D}(E))^* \)

  The set of finite sequences of pairs \( (u,e) \in \mathcal{D}(E) \times \mathcal{D}(E) \)

- \( ready : Eth \times \mathcal{P}(E) \rightarrow 2^{\mathcal{P}(E)} \)

  \((u,e) \mapsto \{ (u,e) \} \) if \( u = u \)

  \( \emptyset \) otherwise (also if \( E \) is empty)

- \( \oplus : Eth \times \mathcal{P}(E) \times \mathcal{P}(E) \rightarrow Eth \)

  \((u, u, e) \mapsto E, (u, e) \)

- \( \ominus : Eth \times \mathcal{P}(E) \rightarrow Eth \)

  \((u, e) \mapsto E \) if \( e = e \)

  \( \emptyset \) otherwise (also if \( E \) is empty)

- \([\cdot] : Eth \times \mathcal{P}(E) \rightarrow Eth \)

  remove all pairs \((u,e)\) from \( E \)

*Other Examples*

- One FIFO queue per active object is an ether.
  \[ Eth_a = \mathcal{D}(E) \rightarrow (\mathcal{D}(E) \times \mathcal{D}(E))^* \]

- One-place buffer.
  \[ Eth_b = E \cup (\mathcal{D}(E) \times \mathcal{D}(E)) \]

- Priority queue.

- Multi-queues (one per sender).

- Trivial example: sink, “black hole”.

- Lossy queue \( \oplus \) needs to become a relation then.

- ...
**System Configuration**

**Definition.** Let \( \mathcal{S}_0 = (T_0, C_0, V_0, \text{atr}_0, \mathcal{E}) \) be a signature with signals, \( \mathcal{D}_0 \) a structure of \( \mathcal{S}_0 \), \((\text{Eth, ready, } \oplus, \ominus, [\cdot])\) an ether over \( \mathcal{S}_0 \) and \( \mathcal{D}_0 \).

Furthermore assume there is one core state machine \( M_C \) per class \( C \in \mathcal{C} \).

A **system configuration** over \( \mathcal{S}_0, \mathcal{D}_0 \), and \( \mathcal{E} \) is a pair \( (\sigma, \varepsilon) \in \Sigma_{\mathcal{D}} \times \mathcal{E} \) where

- \( \sigma = (\mathcal{S}_0 \cup \{ S_M \mid C \in \mathcal{C} \}, \mathcal{D}_0 \)
- \( \mathcal{D} = \mathcal{D}_0 \cup \{ S_M \mapsto S(M_C) \mid C \in \mathcal{C} \} \)
- \( \sigma(u)(r) \cap \mathcal{D}(\delta_0) = \emptyset \) for each \( u \in \text{dom}(\sigma) \) and \( r \in V_0 \).
References
