Software Design, Modelling and Analysis in UML

Lecture X: Active vs. Passive Objects

2015-01-27

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

Albert-Ludwigs-Universität Freiburg, Germany

Active and Passive Objects [Harel and Gery, 1997]

What about non-Active Objects?

Recall:

- We're **still** working under the assumption that all classes in the class diagram (and thus all objects) are **active**.
- That is, each object has its own thread of control and is (if stable) at any time ready to process an event from the ether.

But the world doesn't consist of only active objects.

For instance, in the crossing controller from the exercises we could wish to have the whole system live in one thread of control.

So we have to address questions like:

- Can we send events to a non-active object?
- And if so, when are these events processed?
- etc.

Active and Passive Objects: Nomenclature

[Harel and Gery, 1997] propose the following (orthogonal!) notions:

- A class (and thus the instances of this class) is either **active** or **passive** as declared in the class diagram.
  - An **active** object has (in the operating system sense) an own thread: an own program counter, an own stack, etc.
  - A **passive** object doesn't.
- A class is either **reactive** or **non-reactive**.
  - A **reactive** class has a (non-trivial) state machine.
  - A **non-reactive** one hasn't.

Which combinations do we understand?

	active	passive
reactive	✓	?
non-reactive	(✓)	(✓)

Passive and Reactive

- So why don't we understand passive/reactive?
- Assume passive objects u_1 and u_2, and active object u, and that there are events in the ether for all three.

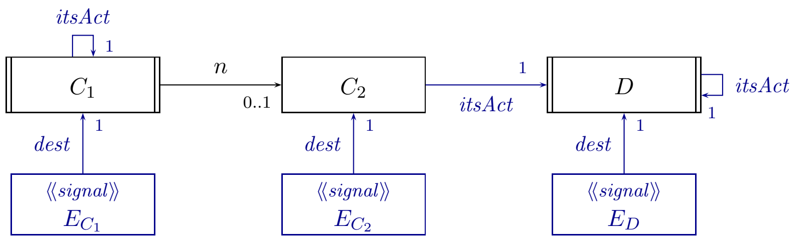
  Which of them (can) start a run-to-completion step...?
  Do run-to-completion steps still interleave...?

Reasonable Approaches:

- **Avoid** — for instance, by
  - require that **reactive implies active** for model well-formedness.
  - requiring for model well-formedness that events are **never sent** to instances of non-reactive classes.
- **Explain** — here: (following [Harel and Gery, 1997])
  - Delegate all dispatching of events to the active objects.

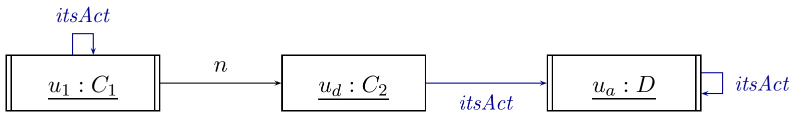
Passive Reactive Classes

- Firstly, establish that each object u knows, via (implicit) link $itsAct$, **the active object** u_{act} which is responsible for dispatching events to u.
- If u is an instance of an active class, then $u_a = u$.



Passive Reactive Classes

- Firstly, establish that each object u knows, via (implicit) link $itsAct$, **the active object** u_{act} which is responsible for dispatching events to u.
- If u is an instance of an active class, then $u_a = u$.



Sending an event:

- Establish that of each signal we have a version E_C with an association $dest : C_{0,1}$, $C \in \mathscr{C}$.
- Then $n!E$ in $u_1 : C_1$ becomes:
- Create an instance u_e of E_{C_2} and set u_e's $dest$ to $u_d := \sigma(u_1)(n)$.
- Send to $u_a := \sigma(\sigma(u_1)(n))(itsAct)$, i.e., $\varepsilon' = \varepsilon \oplus (u_a, u_e)$.

Dispatching an event:

- Observation: the ether only has events for active objects.
- Say u_e is ready in the ether for u_a.
- Then u_a asks $\sigma(u_e)(dest) = u_d$ to process u_e — and waits until completion of corresponding RTC.
- u_d may in particular discard event.

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