Software Design, Modelling and Analysis in UML

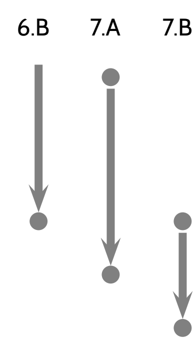
Lecture 17: Live Sequence Charts I

2017-01-17

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

Albert-Ludwigs-Universität Freiburg, Germany

The Plan



- Thu, 19. 1.: **Live Sequence Charts I**
  Firstly: State-Machines Rest, Code Generation
- Tue, 24. 1.: **Live Sequence Charts II**
- Thu, 26. 1.: **Live Sequence Charts III**
- Tue, 31. 1.: **Tutorial 7**
- Thu, 2. 2.: **Model Based/Driven SW Engineering**
- **Mon**, 6. 2.: **Inheritance**
- Tue, 7. 2.: **Meta-Modelling** + Questions

February, 17th: **The Exam.**

Content

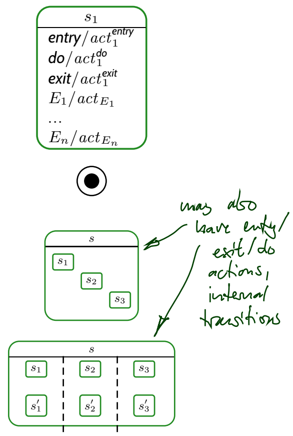
- (Hierarchical) State Machines
  - Active vs. Passive Objects
  - Methods / Behavioural Features
  - Code Generation
  - Discussion
- Reflective Descriptions of Behaviour
  - Interactions
  - A Brief History of Sequence Diagrams
- Live Sequence Charts
  - Abstract Syntax
  - Well-Formedness

constructive descriptions of behaviour

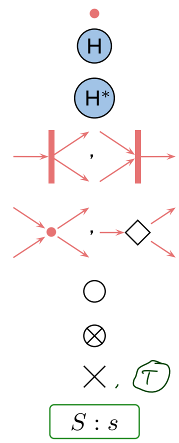
Hierarchical State Machines: Retrospective

UML distinguishes the following **kinds of states**:

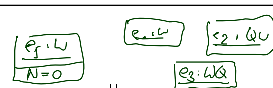
	example
simple state	
final state	
composite state	
OR	
AND	



	example
pseudo-state	
initial	
(shallow) history	
deep history	
fork/join	
junction, choice	
entry point	
exit point	
terminate	
submachine state	



Exercise 4.(i)



	u_1				u_2					
nr.	$wbtn$	dsp	st	$stable$	$lock$	wis	st	$stable$	ε	rule
0	0	−1	s_1	1	1	0	s_1	1	$(u_2, e_1).(u_1, e_2)$	
1	0	-1	s_1	1	1	0	s_1	1	(u_1, e_2)	(i)
2	1	-1	s_2	0	1	0	s_1	1	ϵ	(ii)
3	1	-1	s_3	1	1	0	s_1	1	(u_2, e_3)	(iii)
4	1	-1	s_3	1	0	0	s_2	0	ϵ	(ii)
5a	1	-1	s_3	1	0	0	s_1	1	$(u_1, e_4 \cdot EMPTY)$	(iii)
5b	1	-1	s_3	1	0	0	s_3	0	(u_1, e_5)	(iii)
6a	0	-1	s_1	1	0	0	s_1	1	ϵ	(ii)
6b1	1	-1	s_3	1	1	0	s_1	1	(u_1, e_5)	(iii)
6b2	1	0	s_4	0	0	0	s_3	0	ϵ	(ii)
7b1	1	0	s_4	0	1	0	s_1	1	ϵ	(ii)
8b1	0	-1	s_1	1	1	0	s_1	1	ϵ	(iv)
7b2i	0	-1	s_1	1	0	0	s_3	0	ϵ	(ii)
7b2ii	1	0	s_4	0	1	0	s_1	1	ϵ	(iii)
8b2i	0	-1	s_1	1	1	0	s_1	1	ϵ	(iii)

RTC
RTC
RTC

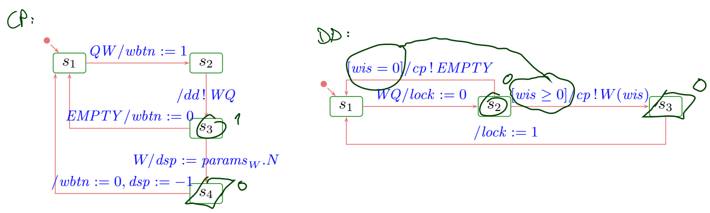
Exercise 4 (ii)-(v)

(ii) Explain the difference between step and RTC-step.

(iii) Is it possible to reach $\#$?

(iv) Considering the rule for environment interaction, how does the possible behaviour change?

(v) How does the behaviour simulated with Rhapsody compare to the results from Task (i)?



– 05 – 2016-12-20 – main –

Active and Passive Objects

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.
  $\rightarrow$ steps of active objects can **interleave**.

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

For instance, in the Vending Machine 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

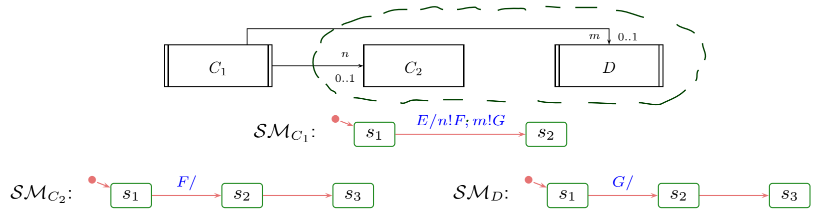
Harel and Gery (1997) propose the following (**orthogonal!**) notions:

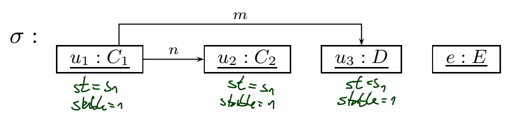
- A class (and thus the instances of this class) is either **active** or **passive** as defined by the class diagram. (not active)
  - An **active** object has (in the operating system sense) an own thread: an own program counter, an own stack, etc.
  - A **passive** object doesn't.
- A class is either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

Which combinations do we (not) understand yet?

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

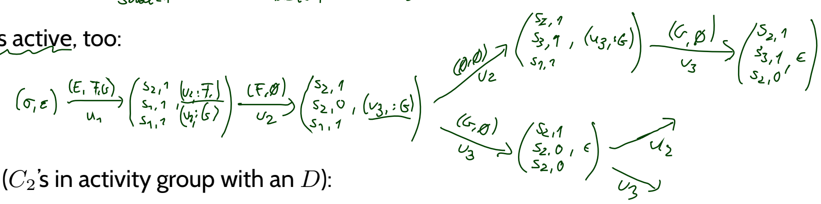
Passive and Reactive / Rhapsody Style: Example



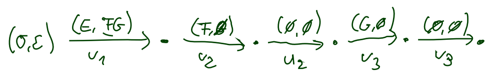


$\varepsilon : (u_1, E_1)$

If C_2 was active, too:

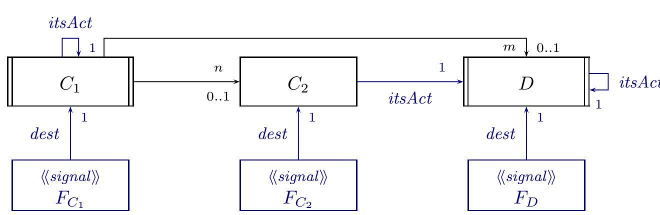


Wanted (C_2's in activity group with an D):



Passive Reactive / Rhapsody Style

- In each class, add (implicit) link $itsAct$ and use it to make each object u **know the active object** u_a which is responsible for dispatching events to u.
  If u is an instance of an active class, then $u_a = u$.
- Equip all signals with (implicit) association $dest$ and use it to point to the destination object.
  For each signal F, have a version F_C with an association $dest : C_{0,1}, C \in \mathscr{C}$ (no inheritance yet).

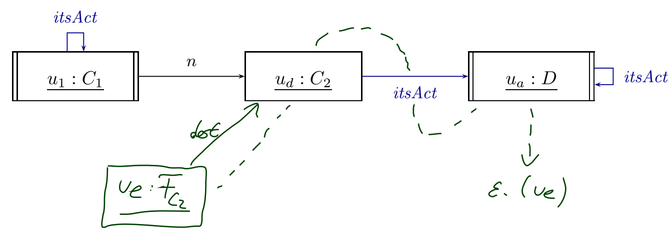


Passive Reactive / Rhapsody Style

- In each class, add (implicit) link $itsAct$ and use it to make each object u **know the active object** u_a which is responsible for dispatching events to u.

  If u is an instance of an active class, then $u_a = u$.
- Equip all signals with (implicit) association $dest$ and use it to point to the destination object.

  For each signal F, have a version F_C with an association $dest : C_{0,1}, C \in \mathscr{C}$ (no inheritance yet).



Sending an event:

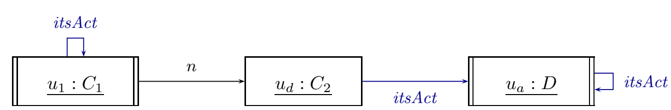
- $n!F$ in $u_1 : C_1$ becomes:
- Create an instance u_e of F_{C_2} and set u_e's $dest$ to $u_d := \sigma(u_1)(n)$. ✓
- Send to $u_a := \sigma(\sigma(u_1)(n))(itsAct)$, i.e., $\varepsilon' = \varepsilon \oplus (u_a, u_e)$.

Passive Reactive / Rhapsody Style

- In each class, add (implicit) link $itsAct$ and use it to make each object u **know the active object** u_a which is responsible for dispatching events to u.

  If u is an instance of an active class, then $u_a = u$.
- Equip all signals with (implicit) association $dest$ and use it to point to the destination object.

  For each signal F, have a version F_C with an association $dest : C_{0,1}, C \in \mathscr{C}$ (no inheritance yet).



Sending an event:

- $n!F$ in $u_1 : C_1$ becomes:
- Create an instance u_e of F_{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.

And What About Methods?

And What About Methods?

- In the current setting, the (local) state of objects is **only** modified by actions of transitions, which we abstract to transformers.
- In general, there are also **methods**.
- UML follows an approach to separate
  - the **interface declaration** from
  - the **implementation**.

  In C++-lingo: distinguish **declaration** and **definition** of method.

- In UML, the former is called **behavioural feature** and can (roughly) be
  - a **call interface** $f(T_{1_1}, \dots, T_{n_1}) : T_1$
  - a **signal name** E

C
$\xi_1\ f(T_{1,1}, \dots, T_{1,n_1}) : T_1\ P_1$ $\xi_2\ F(T_{2,1}, \dots, T_{2,n_2}) : T_2\ P_2$
$\langle\langle signal \rangle\rangle\ E$

Note: The signal list can be seen as redundant (can be looked up in the state machine) of the class. But: certainly useful for documentation (or sanity check).

Behavioural Features

C
$\xi_1\ f(T_{1,1}, \dots, T_{1,n_1}) : T_1\ P_1$ $\xi_2\ F(T_{2,1}, \dots, T_{2,n_2}) : T_2\ P_2$
$\langle\langle signal \rangle\rangle\ E$

Semantics:

- The **implementation** of a behavioural feature can be provided by:
  - An **operation**.

    In our setting, we simply assume a transformer like T_f.

    It is then, e.g. clear how to admit method calls as actions on transitions: function composition of transformers (clear but tedious: non-termination).

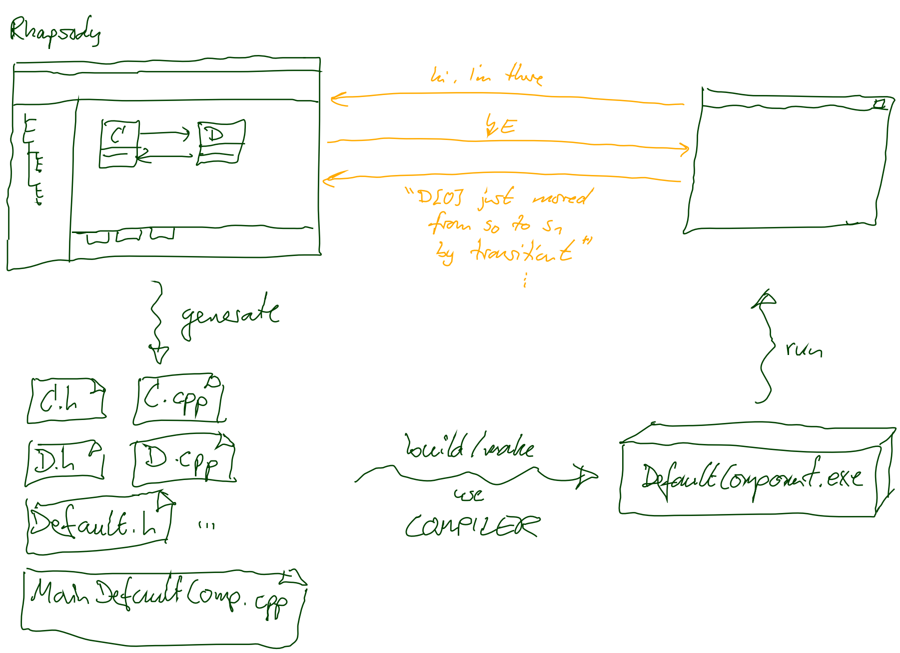
    In a setting with Java as action language: operation is a method body.
  - The class' **state-machine** ("triggered operation").
    - Calling F with n_2 parameters for a stable instance of C creates an auxiliary event F and dispatches it (bypassing the ether).
    - Transition actions may fill in the return value.
    - On completion of the RTC step, the call returns.
    - For a non-stable instance, the caller blocks until stability is reached again.

Behavioural Features: Visibility and Properties

C
$\xi_1\ f(T_{1,1}, \dots, T_{1,n_1}) : T_1\ P_1$ $\xi_2\ F(T_{2,1}, \dots, T_{2,n_2}) : T_2\ P_2$
$\langle\langle signal \rangle\rangle\ E$

- **Visibility**:
  - Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods.
- **Useful properties**:
  - **concurrency**
    - **concurrent** – is thread safe
    - **guarded** – some mechanism ensures/should ensure mutual exclusion
    - **sequential** – is not thread safe, users have to ensure mutual exclusion
  - **isQuery** – doesn't modify the state space (thus thread safe)

A Closer Look to Rhapsody Code Generation



- Rhapsody also supports **non-active objects** – their instances share an event pool with an **active object**.
- Behavioural Features: exist.
- **Semantic Variation Points** are **legion** – but manageable, e.g. by appropriate modelling guidelines (stick to "the beaten track").

- Interactions can be used for **reflective** descriptions of behaviour, i.e.
  - describe **what** behaviour is (un)desired,
    without (yet) defining **how** to realise it.
- One visual formalism for interactions: **Live Sequence Charts**
  - partially ordered locations,
  - instantaneous and aynchronous messages,
  - conditions and local invariants

  Later: pre-charts.

References

References

Crane, M. L. and Dingel, J. (2007). UML vs. classical vs. rhapsody statecharts: not all models are created equal. *Software and Systems Modeling*, 6(4):415–435.

Damm, W. and Harel, D. (2001). LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80.

Harel, D. (1997). Some thoughts on statecharts, 13 years later. In Grumberg, O., editor, *CAV*, volume 1254 of *LNCS*, pages 226–231. Springer-Verlag.

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

Harel, D. and Maoz, S. (2007). Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling (SoSyM)*. To appear. (Early version in SCESM'06, 2006, pp. 13-20).

Harel, D. and Marelly, R. (2003). *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag.

Klose, J. (2003). *LSCs: A Graphical Formalism for the Specification of Communication Behavior*. PhD thesis, Carl von Ossietzky Universität Oldenburg.

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

OMG (2011a). Unified modeling language: Infrastructure, version 2.4.1. Technical Report formal/2011-08-05.

OMG (2011b). Unified modeling language: Superstructure, version 2.4.1. Technical Report formal/2011-08-06.

Störrle, H. (2003). Assert, negate and refinement in UML-2 interactions. In Jürjens, J., Rumpe, B., France, R., and Fernandez, E. B., editors, *CSDUML 2003*, number TUM-I0323. Technische Universität München.