Software Design, Modelling and Analysis in UML

Lecture 19: Inheritance I

~~2012-02-01~~
2013-01-30

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

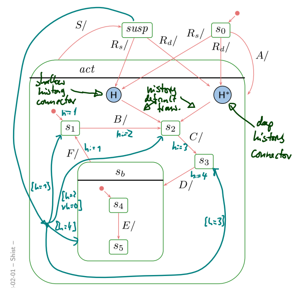
- Live Sequence Charts Semantics

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - What's the Liskov Substitution Principle?
  - What is late/early binding?
  - What is the subset, what the uplink semantics of inheritance?
  - What's the effect of inheritance on LSCs, State Machines, System States?
  - What's the idea of Meta-Modelling?
- **Content:**
  - Quickly complete State Machine semantics
  - Inheritance in UML: concrete syntax
  - Liskov Substitution Principle — desired semantics
  - Two approaches to obtain desired semantics

The Concept of History, and Other Pseudo-States

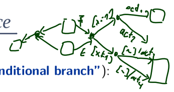
History and Deep History: By Example



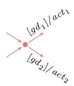
What happens on... (in s_0)

- R_s?
  s_0, s_2
- R_d?
  s_0, s_2
- A, B, C, S, R_s?
  $s_0, s_1, s_2, s_3, susp, s_3$
- A, B, S, R_d?
  $s_0, s_1, s_2, s_3, susp, s_3$
- A, B, C, D, E, R_s?
  $s_0, s_1, s_2, s_3, s_4, s_5, susp, s_4$ shallow w.
- A, B, C, D, R_d?
  $s_0, s_1, s_2, s_3, s_4, s_5, susp, s_5$ deep history

Junction and Choice



- Junction (**"static conditional branch"**):



  - **good**: abbreviation
  - unfolds to so many similar transitions with different guards, the unfolded transitions are then checked for enabledness
  - at best, start with trigger, branch into conditions, then apply actions

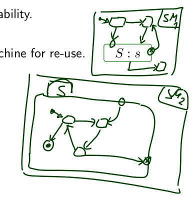
- Choice: (**"dynamic conditional branch"**)

  - **evil**: may get stuck
  - enters the transition **without knowing** whether there's an enabled path
  - at best, use "else" and convince yourself that it cannot get stuck
  - maybe even better: **avoid**

Note: not so sure about naming and symbols, e.g.,
I'd guessed it was just the other way round...

Entry and Exit Point, Submachine State, Terminate

- Hierarchical states can be **"folded"** for readability.
  (but: this can also hinder readability.)
- Can even be taken from a different state-machine for re-use.



Entry and Exit Point, Submachine State, Terminate

- Hierarchical states can be **"folded"** for readability.
  (but: this can also hinder readability.)
- Can even be taken from a different state-machine for re-use. $\boxed{S : s}$
- **Entry/exit points**
  - Provide connection points for finer integration into the current level, than just via initial state.
  - Semantically a bit tricky:
    - **First** the exit action of the exiting state,
    - **then** the actions of the transition,
    - **then** the entry actions of the entered state,
    - **then** action of the transition from the entry point to an internal state,
    - and **then** that internal state's entry action.
- **Terminate Pseudo-State**
  - When a terminate pseudo-state is reached, the object taking the transition is immediately killed.

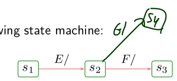
Deferred Events in State-Machines

Deferred Events: Idea

For ages, UML state machines comprises the feature of **deferred events**.

The idea is as follows:

- Consider the following state machine:



- Assume we're stable in s_1, and F is ready in the ether.
- In **the framework of the course**, F is **discarded**.
- But we **may** find it a pity to discard the poor event and **may** want to remember it for later processing, e.g. in s_2, in other words, **defer** it.

General options to satisfy such needs:

- Provide a pattern how to "program" this (use self-loops and helper attributes).
- Turn it into an original language concept. **(← OMG's choice)**

Deferred Events: Syntax and Semantics

- **Syntactically**,
  - Each state has (in addition to the name) a set of deferred events.
  - **Default**: the empty set.
- The **semantics** is a bit intricate, something like
  - if an event E is dispatched,
  - and there is no transition enabled to consume E,
  - and E is in the deferred set of the current state configuration,
  - then stuff E into some "deferred events space" of the object, (e.g. into the ether (= extend ε) or into the local state of the object (= extend σ))
  - and turn attention to the next event.
- **Not so obvious**:
  - Is there a priority between deferred and regular events?
  - Is the order of deferred events preserved?
  - ...

  [Fecher and Schönborn, 2007], e.g., claim to provide semantics for the complete Hierarchical State Machine language, including deferred events.

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.

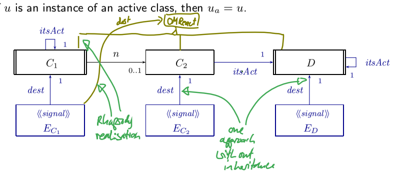
  Which of them (can) start a run-to-completion step...?
  Do run-to-completion steps still interleave...?

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.

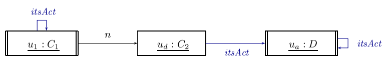
Passive Reactive Classes

- 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.

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** $f(\tau_{1_1}, \ldots, \tau_{n_1}) : \tau_1$
  - a **signal name** E

signal name

C
$\xi_1\ f(\tau_{1,1}, \ldots, \tau_{1,n_1}) : \tau_1\ P_1$
$\xi_2\ F(\tau_{2,1}, \ldots, \tau_{2,n_2}) : \tau_2\ P_2$
$\langle\langle signal\rangle\rangle\ E$

Note: The signal list is redundant as it can be looked up in the state machine of the class. But: certainly useful for documentation.

Behavioural Features

C
$\xi_1\ f(\tau_{1,1}, \dots, \tau_{1,n_1}) : \tau_1\ P_1$ $\xi_2\ F(\tau_{2,1}, \dots, \tau_{2,n_2}) : \tau_2\ P_2$
$\langle\langle signal \rangle\rangle\ E$

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

context C::f
pre: x>0
post: ret<7

C
$\xi_1\ f(\tau_{1,1}, \dots, \tau_{1,n_1}) : \tau_1\ P_1$ $\xi_2\ F(\tau_{2,1}, \dots, \tau_{2,n_2}) : \tau_2\ P_2$
$\langle\langle signal \rangle\rangle\ E$

$S_1 \xrightarrow{E/x:=0;\ n.F();\ y:=1} S_2$

- **Visibility**:
  - Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods. $(\sigma,\varepsilon) \longrightarrow (\sigma',\varepsilon')$
- **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)
- For simplicity, we leave the notion of steps untouched, we construct our semantics around state machines.
  Yet we could explain pre/post in OCL (if we wanted to).

Discussion.

Semantic Variation Points

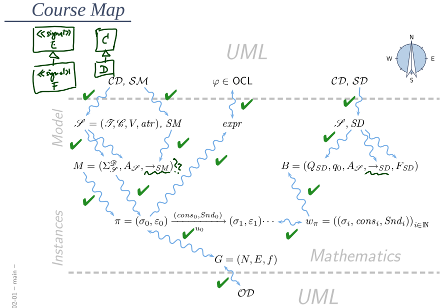
Pessimistic view: They are legion...

- **For instance**,
  - allow **absence of initial pseudo-states**
    can then "be" in enclosing state without being in any substate; or assume one of the children states non-deterministically
  - (implicitly) **enforce determinism**, e.g.
    by considering the order in which things have been added to the CASE tool's repository, or graphical order
  - allow **true concurrency**

  Exercise: Search the standard for "semantical variation point".
- [Crane and Dingel, 2007], e.g., provide an in-depth comparison of Statemate, UML, and Rhapsody state machines — the bottom line is:
  - **the intersection is not empty**
    (i.e. there are pictures that mean the same thing to all three communities)
  - **none is the subset of another**
    (i.e. for each pair of communities exist pictures meaning different things)

Optimistic view: tools exist with complete and consistent code generation.

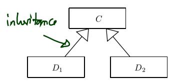
Course Map



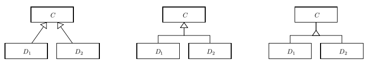
Inheritance: Syntax

Inheritance: Generalisation Relation

inheritance



- **Alternative renderings**:



- **Read**:
  - C **generalises** D_1 and D_2; C is a **generalisation** of D_1 and D_2,
  - D_1 and D_2 **specialise** C; D_1 **is a** (specialisation of) C,
  - D_1 **is a** C; D_2 **is a** C.

NOT:

- **Well-formedness rule**: No **cycles** in the generalisation relation.

Abstract Syntax

Recall: a signature (with signals) is a tuple $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$.

Now (finally): extend to

behavioural features $mth: \mathscr{C} \to 2^F$

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E}, F, mth, \lhd)$$

where F/mth are methods, analogously to attributes and

$$\lhd \subseteq \cancel{(\mathscr{C} \times \mathscr{C}) \cup} (\mathscr{E} \times \mathscr{E}) \cup (\mathscr{C} \setminus \mathscr{E} \times \mathscr{C} \setminus \mathscr{E})$$

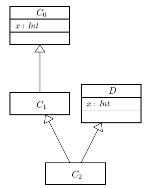
NOT: C «signal» E

is a **generalisation** relation such that $C \lhd^+ C$ for **no** $C \in \mathscr{C}$ ("acyclic").

$C \lhd D$ reads as

- C is a generalisation of D,
- D is a specialisation of C,
- D inherits from C,
- D is a sub-class of C,
- C is a super-class of D,
- ...

Mapping Concrete to Abstract Syntax by Example



$\lhd = \{ C_0 \lhd C_1, C_1 \lhd C_2, D \lhd C_2 \}$

Note: we can have **multiple inheritance**.

Reflexive, Transitive Closure of Generalisation

Definition. Given classes $C_0, C_1, D \in \mathscr{C}$, we say D inherits from C_0 **via** C_1 if and only if there are $C_0^1, \dots C_0^n, C_1^1, \dots C_1^m \in \mathscr{C}$ such that

$$C_0 \lhd C_0^1 \lhd \dots C_0^n \lhd C_1 \lhd C_1^1 \lhd \dots C_1^m \lhd D.$$

We use '$\preceq$' to denote the reflexive, transitive closure of '$\lhd$'.

In the following, we assume

- that all attribute (method) names are of the form

$$C{::}v, \quad C \in \mathscr{C} \cup \mathscr{E} \qquad (C{::}f, \quad C \in \mathscr{C}),$$

- that we have $C{::}v \in atr(C)$ resp. $C{::}f \in mth(C)$ **if and only if** v (f) appears in an attribute (method) compartment of C in a class diagram.

C x : Int
D x : Int

We still want to accept "context C inv : $v < 0$", which v is meant? Later!

Inheritance: Desired Semantics

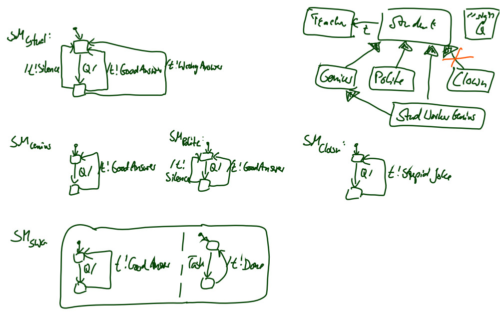
Desired Semantics of Specialisation: Subtyping

There is a classical description of what one **expects** from **sub-types**, which in the OO domain is closely related to inheritance:

The principle of type substitutability [Liskov, 1988, Liskov and Wing, 1994].
(**Liskov Substitution Principle** (LSP).)

P T ? S

"If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T,
the behavior of P is unchanged when o_1 is substituted for o_2
then S is a **subtype** of T."



References

References

[Buschermöhle and Oelerink, 2008] Buschermöhle, R. and Oelerink, J. (2008). Rich meta object facility. In Proc. 1st IEEE Int'l workshop UML and Formal Methods.

[Crane and Dingel, 2007] 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.

[Fecher and Schönborn, 2007] Fecher, H. and Schönborn, J. (2007). UML 2.0 state machines: Complete formal semantics via core state machines. In Brim, L., Haverkort, B. R., Leucker, M., and van de Pol, J., editors, FMICS/PDMC, volume 4346 of LNCS, pages 244–260. Springer.

[Fischer and Wehrheim, 2000] Fischer, C. and Wehrheim, H. (2000). Behavioural subtyping relations for object-oriented formalisms. In Rus, T., editor, AMAST, number 1816 in Lecture Notes in Computer Science. Springer-Verlag.

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

[Liskov, 1988] Liskov, B. (1988). Data abstraction and hierarchy. SIGPLAN Not., 23(5):17–34.

[Liskov and Wing, 1994] Liskov, B. H. and Wing, J. M. (1994). A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(6):1811–1841.

[OMG, 2003] OMG (2003). Uml 2.0 proposal of the 2U group, version 0.2, `http://www.2uworks.org/uml2submission`.

[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.

[Stahl and Völter, 2005] Stahl, T. and Völter, M. (2005). Modellgetriebene Softwareentwicklung. dpunkt.verlag, Heidelberg.