Software Design, Modelling and Analysis in UML

Lecture 19: Inheritance I

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 10)

- $R_a$?
  - $s_1, s_2$
- $R_d$?
  - $s_{10}, s_{20}$
- $A, B, C, S, Ra$?
  - $s_0, s_1, s_2, s_3, s_{10}, s_{30}$
- $A, B, S, Rd$?
  - $s_0, s_1, s_2, s_3, s_{10}, s_{30}$
- $A, B, C, D, E, Ra$?
  - $s_0, s_1, s_2, s_3, s_{10}, s_{30}, s_{40}, s_{50}$
- $A, B, C, D, Rd$?
  - $s_0, s_1, s_2, s_3, s_{10}, s_{30}, s_{40}, s_{50}$
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.

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:
  \[ \text{\includegraphics[scale=0.5]{state_machine.png}} \]
  - 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 (\( \equiv \) extend \( \varepsilon \)) or into the local state of the object (\( \equiv \) extend \( \sigma \)))
  - 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.
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?

<table>
<thead>
<tr>
<th></th>
<th>active</th>
<th>passive</th>
</tr>
</thead>
<tbody>
<tr>
<td>reactive</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>non-reactive</td>
<td>✔️</td>
<td>(X)</td>
</tr>
</tbody>
</table>

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 $u_{a} = u$.

Sending an event:
- Establish that of each signal we have a version $E_{C}$ with an association $dest : C_{0,1}, C \in C'$.
- Then $\forall E$ 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?

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

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

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**

- **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)

- 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.
Inheritance: Syntax
Inheritance: Generalisation Relation

- 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 \( C \).

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

Abstract Syntax

Recall: a signature (with signals) is a tuple \( \mathcal{S} = (\mathcal{F}, \mathcal{E}, V, atr, \Theta) \).

Now (finally): extend to

\[ \mathcal{S} = (\mathcal{F}, \mathcal{E}, V, atr, \Theta, F, mth, \triangleleft) \]

where \( F/mth \) are methods, analogously to attributes and

\[ \triangleleft \subseteq (\mathcal{E} \times \mathcal{E}) \cup (\mathcal{V} \times \mathcal{V}) \]

is a generalisation relation such that \( C \triangleleft^+ C \) for no \( C \in \mathcal{E} \) (“acyclic”).

\( C \triangleleft D \) 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 \( D \),
- \( \ldots \)
Mapping Concrete to Abstract Syntax by Example

\[ \mathcal{A} = \{ C_0 \triangleleft C_1, C_1 \triangleleft C_2, D \triangleleft C_1 \} \]

Note: we can have multiple inheritance.

Reflexive, Transitive Closure of Generalisation

Definition. Given classes \( C_0, C_1, D \in \mathcal{C} \), we say \( D \) inherits from \( C_0 \) via \( C_1 \) if and only if there are \( C_1^1, \ldots, C_1^n, C_1^0, \ldots, C_1^m \in \mathcal{C} \) such that

\[ C_0 \triangleleft C_1^0 \triangleleft \ldots C_1^n \triangleleft C_1 \triangleleft C_1^1 \triangleleft \ldots C_1^m \triangleleft D. \]

We use ‘\( \leq \)’ to denote the reflexive, transitive closure of ‘\( \triangleleft \)’.

In the following, we assume

- that all attribute (method) names are of the form \( C::v, \quad C \in \mathcal{C} \cup \mathcal{B} \) \( C::f, \quad C \in \mathcal{C} \),

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

We still want to accept “context \( C \ inv : v < 0 \)”, which \( v \) is meant? Later!
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).)

“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 $T$,

\[ \text{the behavior of } P \text{ is unchanged } \]

when $o_1$ is substituted for $o_2$, then $S$ is a subtype of $T$.”

\[ S \text{ sub-type of } T : \iff \forall o_1 \in S \exists o_2 \in T \forall P \in \mathcal{P}_T \colon [P_T](o_1) = [P_T](o_2) \]
References
References


