Software Design, Modelling and Analysis in UML

Lecture 19: Hierarchical State Machines III

2015-01-29

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

- Initial and Final State
- Composite State Semantics started

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 does this **hierarchical** State Machine mean? What **may happen** if I inject this event?
  - What is: AND-State, OR-State, pseudo-state, entry/exit/do, final state, ...
- **Content:**
  - Composite State Semantics cont'd
  - The Rest

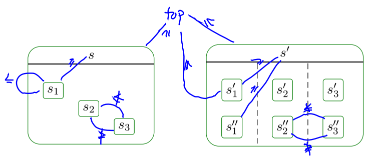
Composite States

(formalisation follows [Damm et al., 2003])

A Partial Order on States

The substate- (or **child-**) relation **induces** a **partial order on states**:

- $top \leq s$, for all $s \in S$,
- $s \leq s'$, for all $s' \in child(s)$,
- transitive, reflexive, antisymmetric,
- $s' \leq s$ and $s'' \leq s$ implies $s' \leq s''$ or $s'' \leq s'$.

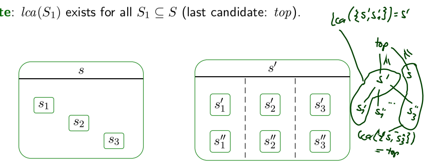


Least Common Ancestor and Ting

- The **least common ancestor** is the function $lca : 2^S \setminus \{\emptyset\} \to S$ such that
  - The states in S_1 are (transitive) children of $lca(S_1)$, i.e.

$$lca(S_1) \leq s, \text{ for all} s \in S_1 \subseteq S,$$

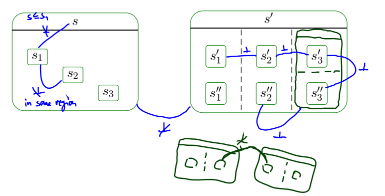
  - $lca(S_1)$ is minimal, i.e. if $\hat{s} \leq s$ for all $s \in S_1$, then $\hat{s} \leq lca(S_1)$
- **Note**: $lca(S_1)$ exists for all $S_1 \subseteq S$ (last candidate: top).



Least Common Ancestor and Ting

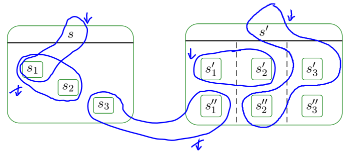
- Two states $s_1, s_2 \in S$ are called **orthogonal**, denoted $s_1 \perp s_2$, if and only if
  - they are unordered, i.e. $s_1 \not\leq s_2$ and $s_2 \not\leq s_1$, and
  - they "live" in different regions of an AND-state, i.e.

$$\exists s, region(s) = \{S_1, \ldots, S_n\} \, \exists 1 \leq i \neq j \leq n : s_1 \in child^*(S_i) \wedge s_2 \in child^*(S_j),$$



Least Common Ancestor and Ting

- A set of states $S_1 \subseteq S$ is called **consistent**, denoted by $\downarrow S_1$, if and only if for each $s, s' \in S_1$,
  - $s \leq s'$, or
  - $s' \leq s$, or
  - $s \perp s'$.



Legal Transitions

A hiearchical state-machine $(S, kind, region, \rightarrow, \psi, annot)$ is called **well-formed** if and only if for all transitions $t \in \rightarrow$,

(i) source and destination are consistent, i.e. $\downarrow source(t)$ and $\downarrow target(t)$,

(ii) source (and destination) states are pairwise orthogonal, i.e.

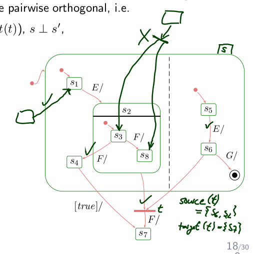
- forall $s \neq s' \in source(t)$ ($\in target(t)$), $s \perp s'$,

(iii) the top state is neither source nor destination, i.e.

- $top \notin source(t) \cup source(t)$.

- Recall: final states are not sources of transitions.

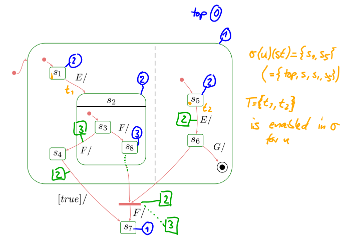
Example:



The Depth of States

- $depth(top) = 0$,
- $depth(s') = depth(s) + 1$, for all $s' \in child(s)$

Example:



Enabledness in Hierarchical State-Machines

- The **scope** ("set of possibly affected states") of a transition t is the **least common region** of

$$source(t) \cup target(t).$$

- Two transitions t_1, t_2 are called **consistent** if and only if their scopes are orthogonal (i.e. states in scopes pairwise orthogonal).

Enabledness in Hierarchical State-Machines

- The **scope** ("set of possibly affected states") of a transition t is the **least common region** of

$$source(t) \cup target(t).$$

- Two transitions t_1, t_2 are called **consistent** if and only if their scopes are orthogonal (i.e. states in scopes pairwise orthogonal).
- The **priority** of transition t is the depth of its innermost source state, i.e.

$$prio(t) := \max\{depth(s) \mid s \in source(t)\}$$

- A set of transitions $T \subseteq \rightarrow$ is **enabled** in an object u if and only if
  - T is consistent,
  - T is maximal wrt. priority,
  - all transitions in T share the same trigger,
  - all guards are satisfied by $\sigma(u)$, and
  - for all $t \in T$, the source states are active, i.e.

$$source(t) \subseteq \sigma(u)(st) \ (\subseteq S).$$

Transitions in Hierarchical State-Machines

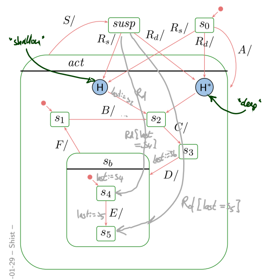
- Let T be a set of transitions enabled in u.
- Then $(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon')$ if
  - $\sigma'(u)(st)$ consists of the target states of t,

    i.e. for simple states the simple states themselves, for composite states the initial states,
  - σ', ε', $cons$, and Snd are the effect of firing each transition $t \in T$ **one by one**, **in any order**, i.e. for each $t \in T$,
    - the exit transformer of all affected states, highest depth first,
    - the transformer of t,
    - the entry transformer of all affected states, lowest depth first.

$\rightsquigarrow$ adjust (2.), (3.), (5.) accordingly.

The Concept of History, and Other Pseudo-States

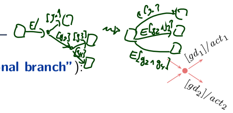
History and Deep History: By Example



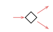
What happens on... (right after creation)

- 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, C, S, R_d?
  s_0, s_1, s_2, s_3, susp, s_3
- A, B, C, D, E, S, R_s?
  s_0, s_1, s_2, s_3, s_4, s_5, susp, s_4
- A, B, C, D, E, S, R_d?

Junction and Choice

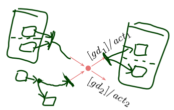


- Junction (**"static conditional branch"**):
  - **good**: abbreviation
  - unfolds to so many similar transitions with different guards, the unfolded transitions are then checked for enabledness
  - at best, start with trigger, branch into conditions, then apply actions
- Choice: (**"dynamic conditional branch"**)



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

Junction and Choice



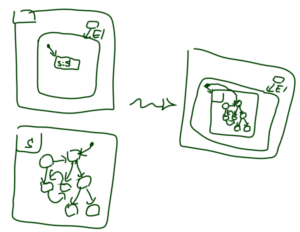
- Junction (**"static conditional branch"**):
  - **good**: abbreviation
  - unfolds to so many similar transitions with different guards, the unfolded transitions are then checked for enabledness
  - at best, start with trigger, branch into conditions, then apply actions
- Choice: (**"dynamic conditional 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... ;-)

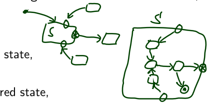
Entry and Exit Point, Submachine State, Terminate

- 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. $S : s$



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. $S : s$
- **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.

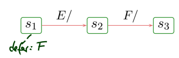
Deferred Events in State-Machines

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.

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

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

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

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

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

- **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...

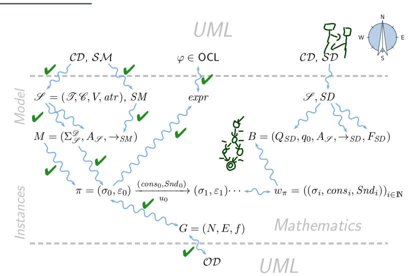
- **For instance**,
  - allow **absence of initial pseudo-states**
    can then "be" in enclosing state without being in any substate; or assume one of the children states non-deterministically
  - (implicitly) **enforce determinism**, e.g.
    by considering the order in which things have been added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

You are here.

Course Map



References

[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.

[Damm et al., 2003] Damm, W., Josko, B., Votintseva, A., and Pnueli, A. (2003). A formal semantics for a UML kernel language 1.2. IST/33522/WP 1.1/D1.1.2-Part1, Version 1.2.

[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.

[Harel and Kugler, 2004] Harel, D. and Kugler, H. (2004). The rhapsody semantics of statecharts. In Ehrig, H., Damm, W., Große-Rhode, M., Reif, W., Schnieder, E., and Westkämper, E., editors, *Integration of Software Specification Techniques for Applications in Engineering*, number 3147 in LNCS, pages 325–354. Springer-Verlag.

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