

Software Design, Modelling and Analysis in UML

Lecture 14: Hierarchical State Machines I

2012-12-19

Prof. Dr. Andreas Podolski, Dr. Bernd Westphal
Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

- RTC-Rules: Dispatch, Commence.

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
 - What does this State Machine mean? What happens if I reject this event?
 - Can you please model the following behaviour.
 - What is: initial state.
 - What does this **hierarchical** State Machine mean? What **may** happen if I reject this event?
 - What is: AND-State, OR-State, pseudo-state, entry/exit, do, final state, ...

Content:

- Step, RTC Divergence
- Putting It All Together
- Recursive Demo
- Hierarchical State Machines Syntax

Step and Run-to-completion Step

Notions of Steps: The Step

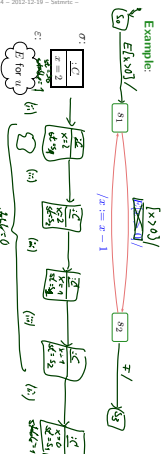
Note: we call one evolution $(\sigma, \delta) \xrightarrow{(Comm, Sml)} (e', \delta')$ a **step**.

Thus in our setting, a **step directly corresponds** to one object (namely v), takes a **simple transition** between regular states. (We have to extend the concept of "single transition" for hierarchical state machines.)
That is: We're going for an interleaving semantics without true parallelism.

Notions of Steps: The Run-to-Completion Step

What is a **run-to-completion step**...?

- **Intuition:** a maximal sequence of steps, where the first step is a **dispatch** step and all later steps are **commence** steps.
 - **Note:** one step corresponds to one transition in the state machine.
- A run-to-completion step is in general not syntactically definable — one transition may be taken multiple times during an RTC-step.



Notions of Steps: The Run-to-Completion Step Cont'd

Proposal: Let

$$(\sigma_0, \delta_0) \xrightarrow{u_0} (\sigma_1, \delta_1) \xrightarrow{u_1} \dots \xrightarrow{u_{n-1}} (\sigma_n, \delta_n), \quad n > 0,$$

be a finite (!) non-empty, maximal, consecutive sequence such that

- object v is alive in σ_0
- $u_0 = u$ and $(Comm, Sml)$ indicates dispatching to v , i.e. $Comm = \{(v, \delta^i - d^i)\}$
- there are no receptions by v in between, i.e.
 - $\sigma_{i-1} = \sigma$ and v is stable only in σ_i and σ_{i+1} , i.e.
 - $\sigma_i \cap \{v\} \times Dom(\delta^i, \mathcal{D}) = \emptyset, i > 1$.

Let $0 = k_1 < k_2 < \dots < k_N = n$ be the maximal sequence of indices such that $u_{k_j} = u$ for $1 \leq j \leq N$. Then we call the sequence

$$(\sigma^i(v)) = (\sigma_{k_1}(v), \sigma_{k_2}(v), \dots, \sigma_{k_N}(v)) \quad (= \sigma_{n_{i-1}}(v))$$

a (!) **run-to-completion computation** of v (from (local) configuration $\sigma_0(v)$) δ_{u_0}

We say, object u can **diverge** on reception *cons* from (local) configuration $\sigma_0(u)$ if and only if there is an infinite, consecutive sequence

$$(\sigma_0, s_0) \xrightarrow{v}^{(cons, S_{in}(u))} (\sigma_1, s_1) \xrightarrow{v}^{(cons, S_{in}(u))} \dots$$

such that u doesn't become stable again.

- **Note:** disappearance of object not considered in the definitions. By the current definitions, it's **not** divergence **but** an RT-Cstep.

What people may **like** on our definition of RT-C-step is that it takes a **global** and **non-compositional** view. That is:

- In the projection onto a single object we still **see** the effect of interaction with other objects
- Adding classes (or even objects) may change the divergence behaviour of existing ones.
- Compositional would be: the behaviour of a set of objects is determined by the behaviour of each object in isolation.

Our semantics and notion of RT-Cstep doesn't have this (often desired) property. Can we give (syntactical) criteria such that any global run-to-completion step is an interleaving of local ones?

Maybe: Strict interfaces (Prove it as exercise...)

- (A): Refer to private features only via "self"
- (B): Let objects only communicate by events, i.e. don't let them modify each other's local state via links at all

Recall: a labelled transition system is (S, \rightarrow, S_0) . We have

- S : system configurations (σ, ε)
- \rightarrow : labelled transition relation $(\sigma, \varepsilon) \xrightarrow{u}^{(cons, S_{in}(u))} (\sigma', \varepsilon')$
- **Wanted:** initial states S_0

Proposal: Require a (finite) set of **object diagrams** $OD \in \mathcal{O}\mathcal{D}$, ε empty).

$$(\mathcal{G}\mathcal{D}, \mathcal{S}\mathcal{M}, \mathcal{O}\mathcal{D})$$

And set $S_0 = \{(\sigma, \varepsilon) \mid \sigma \in C^{-1}(OD), OD \in \mathcal{O}\mathcal{D}, \varepsilon \text{ empty}\}$.

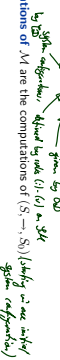
Other Approach: (used by Rhapsody tool) multiplicity of classes. We can read that as an abbreviation for an object diagram.

The semantics of the UML model

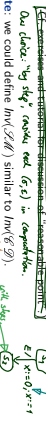
$$\mathcal{M} = (\mathcal{G}\mathcal{D}, \mathcal{S}\mathcal{M}, \mathcal{O}\mathcal{D})$$

- some classes in $\mathcal{G}\mathcal{D}$ are stereotyped as 'signal' (standard), some signals and attributes are stereotyped as 'external' (non-standard)
- there is a 1-to-1 relation between classes and state machines.
- $\mathcal{O}\mathcal{D}$ is a set of object diagrams over $\mathcal{G}\mathcal{D}$.

is the transition system (S, \rightarrow, S_0) constructed on the previous slide



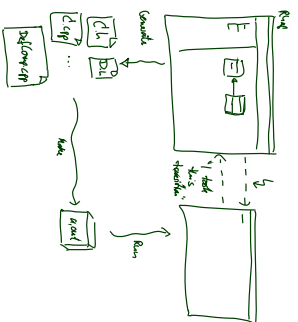
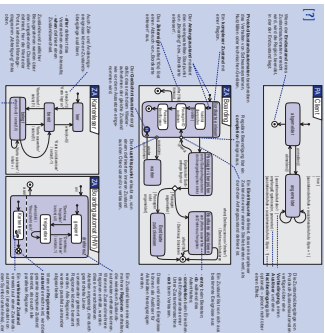
- Let $\mathcal{M} = (\mathcal{G}\mathcal{D}, \mathcal{S}\mathcal{M}, \mathcal{O}\mathcal{D})$ be a UML model.
- We call \mathcal{M} **consistent** iff, for each OCL constraint $expr \in Inv(\mathcal{G}\mathcal{D})$, $\sigma \models expr$ for each "reasonable point" (σ, ε) of computations of \mathcal{M} .



Note: we could define $Inv(\mathcal{S}\mathcal{M})$ similar to $Inv(\mathcal{G}\mathcal{D})$.

Pragmatics: In UML-as-blueprint mode, if $\mathcal{S}\mathcal{M}$ doesn't exist yet, then $\mathcal{M} = (\mathcal{G}\mathcal{D}, \mathcal{S}\mathcal{M}, \mathcal{O}\mathcal{D})$ is consistent.

If the developer makes a mistake, then \mathcal{M} is inconsistent. **Not common:** if $\mathcal{S}\mathcal{M}$ is given, then constraints are also considered when choosing transitions in the RT-C-algorithm. In other words, even in presence of mistakes, the $\mathcal{S}\mathcal{M}$ never move to inconsistent configurations.



UML State-Machines: What do we have to cover?

The Full Story

UML distinguishes the following kinds of states:

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

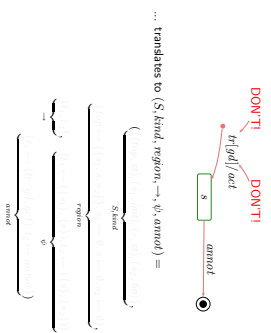
Representing All Kinds of States

- **Until now:** $(S, s_0, \rightarrow), s_0 \in S, \rightarrow \subseteq S \times (\mathcal{P} \cup \{\bullet\}) \times \text{Expr}_s \times \text{Act}_s \times S$

- **Until now:** $(S, s_0 \rightarrow)$, $s_0 \in S \rightarrow \subseteq S \times (\mathcal{P} \cup \{\perp\}) \times \text{Exp}_{\mathcal{F}} \times \text{Act}_{\mathcal{F}} \times S$
- **From now on: (hierarchical) state machines** $(S, \text{kind}, \text{region}, \rightarrow, \psi, \text{annot})$

- **Until now:** $(S, s_0 \rightarrow)$, $s_0 \in S \rightarrow \subseteq S \times (\mathcal{P} \cup \{\perp\}) \times \text{Exp}_{\mathcal{F}} \times \text{Act}_{\mathcal{F}} \times S$
 - **From now on: (hierarchical) state machines** $(S, \text{kind}, \text{region}, \rightarrow, \psi, \text{annot})$
- where
- $S \supseteq \{\text{top}\}$ is a finite set of states
 - $\text{kind} : S \rightarrow \{\text{st}, \text{fin}, \text{sub}, \text{obj}, \text{fork}, \text{join}, \text{jump}, \text{chrt}, \text{ent}, \text{exit}, \text{term}\}$ is a function which labels states with their kind
 - $\text{region} : S \rightarrow 2^S$ is a function which characterises the regions of a state.
 - \rightarrow is a set of transitions.
 - $\psi : (-) \rightarrow 2^S \times 2^S$ is an **inference function** and **(change)**
 - $\text{annot} : (-) \rightarrow (\mathcal{P} \cup \{\perp\}) \times \text{Exp}_{\mathcal{F}} \times \text{Act}_{\mathcal{F}}$ provides an annotation for each transition **(new)**
 - (s_0) is then redundant — replaced by proper state (!) of kind **init**. **(new)**

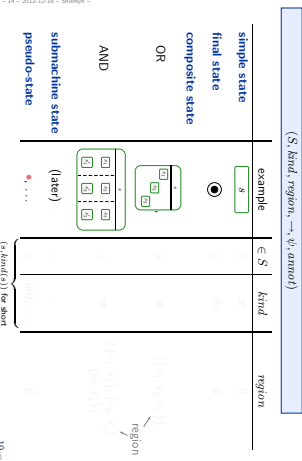
From UML to Hierarchical State Machines: By Example



Well-Formedness: Regions (follows from diagram)

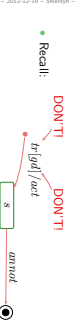
	$\in S$	kind	region $\subseteq 2^S, S_i \subseteq S$	child $\subseteq S$
simple state	s	st	\emptyset	\emptyset
final state	s	fin	\emptyset	\emptyset
composite state	s	st	$\{S_1, \dots, S_n\}, n \geq 1$	$S_1 \cup \dots \cup S_n$
pseudo-state	s	init, ...	\emptyset	\emptyset
implicit top state	top	st	$\{S\}$	S_i

- Each state (except for *top*) lies in exactly one region.
- States $s \in S$ with $\text{kind}(s) = \text{st}$ may comprise regions.
 - No region: simple state.
 - One region: OR-state.
 - Two or more regions: AND-state.
- Final and pseudo states don't comprise regions.
- The region function induces a **child** function.



Well-Formedness: Initial State (requirement on diagram)

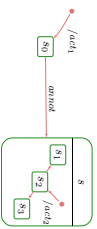
- Each non-empty region has a reasonable initial state and at least one transition from there, i.e.
 - for each $s \in S$ with $\text{region}(s) = \{S_1, \dots, S_n\}$, $n \geq 1$, for each $1 \leq i \leq n$,
 - there exists exactly one initial pseudo-state $(s_i, \text{init}) \in S$ and at least one transition $t \in \rightarrow$ with s_i^s as source, and such transition's target s_i^t is in S_i , and **(for simplicity)** $\text{kind}(s_i^t) = \text{st}$, and $\text{annot}(t) = (\perp, \text{true}, \text{act})$.
- No ongoing transitions to initial states.
- No outgoing transitions from final states.



	example	initial pseudostate	example
simple state		initial (shallow) history	
final state		deep history	
composite state		fork/join	
OR		junction, choice	
AND		entry point	
		exit point	
		terminate	
		alternation state	

- Initial pseudostate, final state.
- Composite states.
- Entry /do, exit actions, internal transitions.
- History and other pseudostates, the rest.

Initial Pseudostate



- Principle:**
- when entering a region **without** a specific destination state,
 - then go to a state which is destination of an initiation transition,
 - execute the action of the chosen initiation transitions **between** exit and entry actions.

Special case: the region of *top*

- If class *C* has a state-machine, then “create-*C* transformer” is the constructor of *C*.
- the transformer of the “constructors” of *C* (here not introduced explicitly) and
- a transformer corresponding to one initiation transition of the top region.

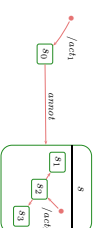
Initial Pseudostates and Final States



- Transitions without trigger can **conceptually** be viewed as being sensitive for the “completion event”.
- Dispatching (here: *E*) can then **alternatively** be viewed as

Towards Final States: Completion of States

Initial Pseudostate



- Principle:**
- when entering a region **without** a specific destination state,
 - then go to a state which is destination of an initiation transition,
 - execute the action of the chosen initiation transitions **between** exit and entry actions.

Towards Final States: Completion of States



- Transitions without trigger can **conceptually** be viewed as being sensitive for the “completion event”.
- Dispatching (here: *E*) can then **alternatively** be viewed as
- (!) fetch event (here: *E*) from the ether.

Towards Final States: Completion of States



- Transitions without trigger can **conceptually** be viewed as being sensitive for the “completion event”.
- Dispatching (here: E) can then **alternatively** be viewed as
 - (i) fetch event (here: E) from the ether,
 - (ii) take an enabled transition (here: to s_2).

Towards Final States: Completion of States



- Transitions without trigger can **conceptually** be viewed as being sensitive for the “completion event”.
- Dispatching (here: E) can then **alternatively** be viewed as
 - (i) fetch event (here: E) from the ether,
 - (ii) take an enabled transition (here: to s_2).
 - (iii) remove event from the ether.

Towards Final States: Completion of States



- Transitions without trigger can **conceptually** be viewed as being sensitive for the “completion event”.
- Dispatching (here: E) can then **alternatively** be viewed as
 - (i) fetch event (here: E) from the ether,
 - (ii) take an enabled transition (here: to s_2).
 - (iii) remove event from the ether,
 - (iv) after having finished entry and do action of current state (here: s_2) — the state is then called **completed** —.

Towards Final States: Completion of States



- Transitions without trigger can **conceptually** be viewed as being sensitive for the “completion event”.
- Dispatching (here: E) can then **alternatively** be viewed as
 - (i) fetch event (here: E) from the ether,
 - (ii) take an enabled transition (here: to s_2).
 - (iii) remove event from the ether,
 - (iv) after having finished entry and do action of current state (here: s_2) — the state is then called **completed** —.
 - (v) raise a **completion event** — with strict priority over events from ether!

Towards Final States: Completion of States



- Transitions without trigger can **conceptually** be viewed as being sensitive for the “completion event”.
- Dispatching (here: E) can then **alternatively** be viewed as
 - (i) fetch event (here: E) from the ether,
 - (ii) take an enabled transition (here: to s_2).
 - (iii) remove event from the ether,
 - (iv) after having finished entry and do action of current state (here: s_2) — the state is then called **completed** —.
 - (v) raise a **completion event** — with strict priority over events from ether!
 - then take it (here: (s_2, s_2)).
 - otherwise become stable.

Final States



- If
 - a step of object u moves u into a final state (s, fn) , and
 - all sibling regions are in a final state,then (conceptually) a completion event for the current composite state s is raised.

Final States



- If
 - a step of object u moves u into a final state (s, fn), and
 - all sibling regions are in a final state,
 then (conceptionally) a completion event for the current composite state s is raised.
- If there is a transition of a parent state ($t.e.$, inverse of $child$) of s enabled which is sensitive for the completion event,
 - then take that transition,
 - otherwise kill u
- adjust (2.) and (3.) in the semantics accordingly

Final States



- If
 - a step of object u moves u into a final state (s, fn), and
 - all sibling regions are in a final state,
 then (conceptionally) a completion event for the current composite state s is raised.
- If there is a transition of a parent state ($t.e.$, inverse of $child$) of s enabled which is sensitive for the completion event,
 - then take that transition,
 - otherwise kill u
- adjust (2.) and (3.) in the semantics accordingly
- **One consequence:** u never survives reaching a state (s, fn) with $s \in child(top)$.

Final States

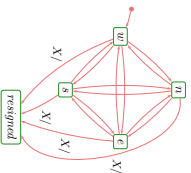


- If
 - a step of object u moves u into a final state (s, fn), and
 - all sibling regions are in a final state,
 then (conceptionally) a completion event for the current composite state s is raised.
- If there is a transition of a parent state ($t.e.$, inverse of $child$) of s enabled which is sensitive for the completion event,
 - then take that transition,
 - otherwise kill u
- adjust (2.) and (3.) in the semantics accordingly
- **One consequence:** u never survives reaching a state (s, fn) with $s \in child(top)$.
- **Now:** in Core State Machines, there is no parent state
- **Later:** in Hierarchical ones, there may be one.

Composite States (omission, follows (?))

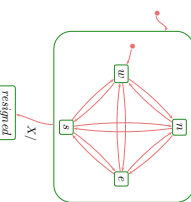
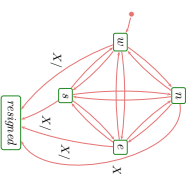
Composite States

- In a sense, composite states are about **abbreviation, structuring, and avoiding redundancy.**
- Idea: in Ton, for the Player's StateMachine, instead of



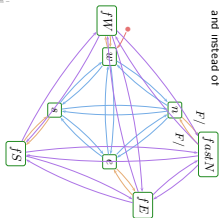
Composite States

- In a sense, composite states are about **abbreviation, structuring, and avoiding redundancy.**
- Idea: in Ton, for the Player's StateMachine, instead of



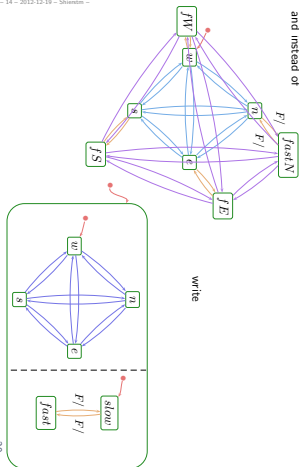
Composite States

and instead of



Composite States

and instead of

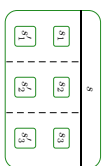


Recall: Syntax

translates to

$$\left\{ \left(\text{top}, \text{st} \right), \left(s, \text{st} \right), \left(s_1, \text{st} \right), \left(s_1, \text{st} \right), \left(s_2, \text{st} \right), \left(s_2, \text{st} \right), \left(s_2, \text{st} \right), \left(s_3, \text{st} \right), \left(s_3, \text{st} \right), \left(s_3, \text{st} \right), \left(s_4, \text{st} \right), \left(s_4, \text{st} \right), \left(s_4, \text{st} \right), \left(s_5, \text{st} \right), \left(s_5, \text{st} \right), \left(s_5, \text{st} \right), \left(s_6, \text{st} \right), \left(s_6, \text{st} \right), \left(s_6, \text{st} \right), \left(s_7, \text{st} \right), \left(s_7, \text{st} \right), \left(s_7, \text{st} \right), \left(s_8, \text{st} \right), \left(s_8, \text{st} \right), \left(s_8, \text{st} \right), \left(s_9, \text{st} \right), \left(s_9, \text{st} \right), \left(s_9, \text{st} \right), \left(s_{10}, \text{st} \right), \left(s_{10}, \text{st} \right), \left(s_{10}, \text{st} \right) \right\}$$

$\xrightarrow{\psi, \text{anmod}}$

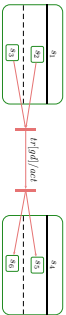


Syntax: ForkJoin

- For brevity, we always consider transitions with (possibly) multiple sources and targets, i.e.

$$\psi : (-) \rightarrow (2^S \setminus \emptyset) \times (2^S \setminus \emptyset)$$

- For instance,

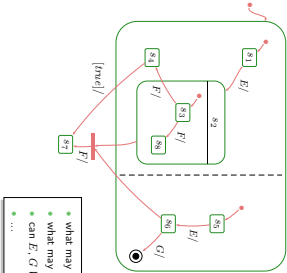


translates to

$$(S, \text{kind}, \text{region}, \{s_1\}, \{s_1 \mapsto \{s_{21}, s_{22}\}, s_2 \mapsto \{s_{31}, s_{32}\}\}, \{s_4 \mapsto \{r_1, \text{g}1, \text{a}1\}\})$$

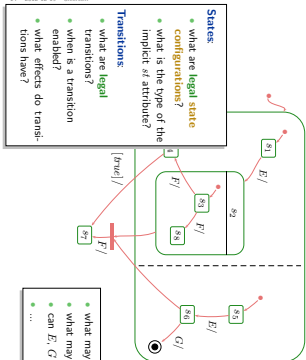
- Naming convention: $\psi(t) = (\text{source}(t), \text{target}(t))$.

Composite States: Blessing or Curse?



- what may happen on $E?$?
- what may happen on E , F ?
- can E , C kill the object?
- ...

Composite States: Blessing or Curse?

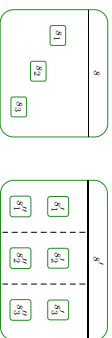


- what may happen on $E?$?
- what may happen on E , F ?
- can E , C kill the object?
- ...

State Configuration

- The type of s_i is from now on a **set of states**, i.e. $s_i : 2^S$
- A set $S_1 \subseteq S$ is called (**legal**) **state configurations** if and only if
 - $top \in S_1$, and
 - well: each state $s \in S_1$ that has a non-empty region $\theta \neq R \in region(s)$, exactly one (non pseudo state) child of θ is in S_1 , i.e.

$$|\{s \in R \mid kind(s) \in \{st, fn\}\} \cap S_1| = 1$$



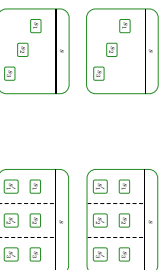
- The **substate-** (or **child**-) **relation induces a partial order on states**:
- $top \leq s_i$, for all $s \in S_1$
 - $s \leq s'$, for all $s' \in child(s)$,
 - transitive, reflexive, antisymmetric,
 - $s' \leq s$ and $s'' \leq s$ implies $s' \leq s''$ or $s'' \leq s'$.

State Configuration

- The type of s_i is from now on a **set of states**, i.e. $s_i : 2^S$
- A set $S_1 \subseteq S$ is called (**legal**) **state configurations** if and only if
 - $top \in S_1$, and
 - well: each state $s \in S_1$ that has a non-empty region $\theta \neq R \in region(s)$, exactly one (non pseudo state) child of θ is in S_1 , i.e.

$$|\{s \in R \mid kind(s) \in \{st, fn\}\} \cap S_1| = 1$$

Examples:



Least Common Ancestor and Top

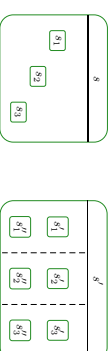
- The **least common ancestor** is the function $lca : 2^S \rightarrow S$ such that
 - The states in S_1 are (transitive) children of $lca(S_1)$, i.e.

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

A Partial Order on States

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

- $top \leq s_i$, for all $s \in S_1$
- $s \leq s'$, for all $s' \in child(s)$,
- transitive, reflexive, antisymmetric,
- $s' \leq s$ and $s'' \leq s$ implies $s' \leq s''$ or $s'' \leq s'$.

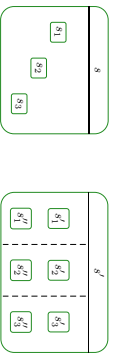


- The **least common ancestor** is the function $lca : 2^S \rightarrow S$ such that
 - The states in S_1 are (transitive) children of $lca(S_1)$, i.e.

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

Least Common Ancestor and Thing

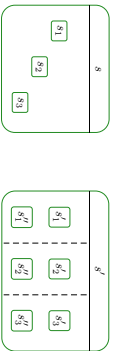
- Two states $s_1, s_2 \in S$ are called **orthogonal**, denoted $s_1 \perp s_2$, if and only if
 - they are unordered, i.e. $s_1 \not\leq s_2$ and $s_2 \not\leq s_1$, and
 - they live in different regions of an AND-state, i.e.
 - $\exists s_1, \text{region}(s) = \{s_1, \dots, s_n\}, 1 \leq i \neq j \leq n : s_i \in \text{child}(S_1) \wedge s_j \in \text{child}(S_2)$



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

Least Common Ancestor and Thing

- 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.
 - $\exists s_1, \text{region}(s) = \{s_1, \dots, s_n\}, 1 \leq i \neq j \leq n : s_i \in \text{child}(S_1) \wedge s_j \in \text{child}(S_2)$



- A hierarchical state-machine $(S, \text{kind}, \text{region}, \rightarrow, \psi, \text{anno})$ is called **well-formed** if and only if for all transitions $t \in \rightarrow$,
 - source and destination are consistent, i.e. $\perp \text{source}(t)$ and $\perp \text{target}(t)$,
 - source (and destination) states are pairwise unordered, i.e.
 - forall $s_i, s'_i \in \text{source}(t) (\in \text{target}(t)), s_i \perp s'_i$,
 - the top state is neither source nor destination, i.e.
 - $\text{top} \notin \text{source}(t) \cup \text{target}(t)$
- Recall: final states are not sources of transitions.

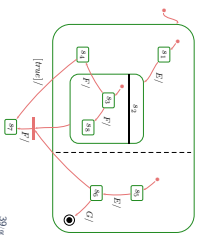
Least Common Ancestor and Thing

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

Legal Transitions

- A hierarchical state-machine $(S, \text{kind}, \text{region}, \rightarrow, \psi, \text{anno})$ is called **well-formed** if and only if for all transitions $t \in \rightarrow$,
 - source and destination are consistent, i.e. $\perp \text{source}(t)$ and $\perp \text{target}(t)$,
 - source (and destination) states are pairwise unordered, i.e.
 - forall $s_i, s'_i \in \text{source}(t) (\in \text{target}(t)), s_i \perp s'_i$,
 - the top state is neither source nor destination, i.e.
 - $\text{top} \notin \text{source}(t) \cup \text{target}(t)$
- Recall: final states are not sources of transitions.

Example:



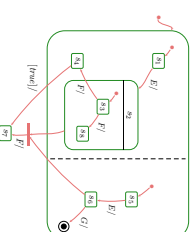
The Depth of States

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

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

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

Enabledness in Hierarchical State-Machines

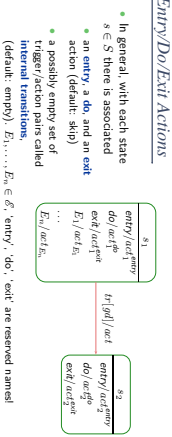
- The **scope** ("set of possibly affected states") of a transition t is the **least common region** of $source(t) \cup target(t)$.
- Two transitions t_1, t_2 are called **consistent** if and only if their scopes are orthogonal (i.e. states in scopes pairwise orthogonal).
- The **priority** of transition t is the depth of its innermost source state, i.e. $priority := \max\{depth(s) \mid s \in source(t)\}$.
- 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. $source(t) \subseteq \sigma(u)(s) \subseteq S$.

- Let T be a set of transitions enabled in u .
- Then $(\sigma, \varepsilon) \xrightarrow{(entry, SInid)} (\sigma', \varepsilon')$ if
 - $\sigma'(\mathcal{U}(S))$ consists of the target states of T , i.e. for simple states the simple states themselves, for composite states the initial states.
 - $\sigma', \varepsilon', SInid$ and $SInid'$ 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.

→ adjust (2), (3), (5) accordingly

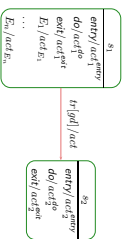
42₃₆

Entry/Do/Exit Actions, Internal Transitions



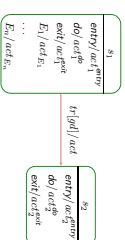
- In general, with each state $s \in S$ there is associated
 - an entry, a do, and an exit action (default: skip)
 - a possibly empty set of trigger/action pairs called **internal transitions**, (default: empty). $E_1, \dots, E_n \in \mathcal{E}^T$, 'entry', 'do', 'exit' are reserved names!
 - Recall: each action's supposed to have a transformer. Here: $f_{act1^entry}, f_{act1^do}, \dots$
 - Taking the transition above then amounts to applying

$$f_{act1^entry} \circ f_{act1^do} \circ f_{act1^exit}$$
 instead of only f_{act1}
- adjust (2), (3) accordingly
- 44₃₆

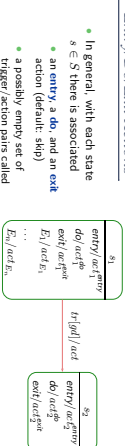


- For **internal transitions**, taking the one for E_1 , for instance, still amounts to taking **only** $f_{act1^E_1}$.
 - Intuition: The state is neither left nor entered, so: no exit, no entry.
 - Note: internal transitions also start a run-to-completion step.
- adjust (2) accordingly
- 45₃₆

Internal Transitions



- For **internal transitions**, taking the one for E_1 , for instance, still amounts to taking **only** $f_{act1^E_1}$.
 - Intuition: The state is neither left nor entered, so: no exit, no entry.
 - Note: internal transitions also start a run-to-completion step.
- adjust (2) accordingly
- Note: the standard seems not to clarify whether internal transitions have **priority** over regular transitions with the same trigger at the same state. Some code generators assume that internal transitions have priority!
- 45₃₆



- In general, with each state $s \in S$ there is associated
 - an entry, a do, and an exit action (default: skip)
 - a possibly empty set of trigger/action pairs called **internal transitions**, (default: empty). $E_1, \dots, E_n \in \mathcal{E}^T$, 'entry', 'do', 'exit' are reserved names!
- 44₃₆

Junction and Choice

- Junction ("static conditional branch"):



- Choice: ("dynamic conditional branch")



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

50^{ks}

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 enablement
- 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...

50^{ks}

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 enablement
- at best, start with "trigger", branch into conditions, then apply actions

- Choice: ("dynamic conditional branch")



- **red!** 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...

50^{ks}

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

- 14 - 2012-12-19 - 5h41 -

51^{ks}

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 entering state,
 - then the actions of the transition,
 - then the entry actions of the entered state,
 - then the action of the transition from the entry point to an internal state,
 - and then that internal state's entry action.

S : s



- 14 - 2012-12-19 - 5h41 -

51^{ks}

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 entering state,
 - then the actions of the transition,
 - then the entry actions of the entered state,
 - then the 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.

S : s



- 14 - 2012-12-19 - 5h41 -

51^{ks}

Deferred Events in State-Machines

- 14 - 2012-12-19 - main -

52/66

Active and Passive Objects [?]

- 14 - 2012-12-19 - main -

53/66

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.

- 14 - 2012-12-19 - Section -

54/66

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.

- 14 - 2012-12-19 - Section -

54/66

Active and Passive Objects: Nomenclature

[?] 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.

- 14 - 2012-12-19 - Section -

55/66

Active and Passive Objects: Nomenclature

[?] 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.

- 14 - 2012-12-19 - Section -

55/66

Active and Passive Objects: Nomenclature

- [?] 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 v_1 and v_2 and active object v_3 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...?

Passive and Reactive

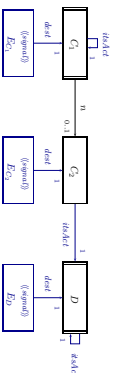
- So why don't we understand passive/reactive?
 - Assume passive objects v_1 and v_2 and active object v_3 and that there are events in the ether for all three.
 - Which of them (can) start a run-to-completion step...? Do run-to-completion steps still interleave...?

Reasonable Approaches:

- Avoid** — for instance, by
 - requiring 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 [?])
 - Delegate all dispatching of events to the active objects.

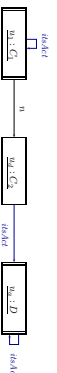
Passive Reactive Classes

- Firstly, establish that each object v_i knows, via (implicit) link $link_i^k$, the **active object** v_{act} , which is responsible for dispatching events to v_i .
- If v_i is an instance of an active class, then $v_{act} = v_i$.



Passive Reactive Classes

- Firstly, establish that each object v_i knows, via (implicit) link $link_i^k$, the **active object** v_{act} , which is responsible for dispatching events to v_i .
- If v_i is an instance of an active class, then $v_{act} = v_i$.



Sending an event:

- Establish that of each signal we have a version E_C with an association $dest : C \rightarrow E$.
- Then $n \in E$ in $v_i : C_i$ becomes:
 - Create an instance u_C of E_C and set u_C 's $dest$ to $v_{act} := \sigma(v_i)(v)$.
 - Send to $u_C := \sigma(v_i)(v)(link_i^k)$, i.e., $e^* = e \oplus (u_C, u_C)$.

Passive Reactive Classes

- Firstly, establish that each object v_i knows, via (implicit) link $link_i^k$, the **active object** v_{act} , which is responsible for dispatching events to v_i .
- If v_i is an instance of an active class, then $v_{act} = v_i$.



Sending an event:

- Establish that of each signal we have a version E_C with an association $dest : C \rightarrow E$.
- Then $n \in E$ in $v_i : C_i$ becomes:
 - Say u_C is ready in the ether for u_C .
 - Then u_C asks $\sigma(v_i)(dest) = u_C$ to process u_C — and waits until completion of corresponding RTC.
 - Send to $u_C := \sigma(v_i)(v)(link_i^k)$, i.e., $e^* = e \oplus (u_C, u_C)$.

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.

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.

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.

C
$\{ f: (r_1, \dots, r_n) \rightarrow r, R \}$
$\{ s: f: (r_1, \dots, r_n) \rightarrow r, R \}$
$\langle \text{signal} \rangle E$

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
$\{ f: (r_1, \dots, r_n) \rightarrow r, R \}$
$\{ s: f: (r_1, \dots, r_n) \rightarrow r, R \}$
$\langle \text{signal} \rangle E$

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

- The class' **state-machine** ("triggered operation").

Behavioural Features

C
$\{ f: (r_1, \dots, r_n) \rightarrow r, R \}$
$\{ s: f: (r_1, \dots, r_n) \rightarrow r, R \}$
$\langle \text{signal} \rangle E$

- 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").

Behavioural Features

C
$\{ f: (r_1, \dots, r_n) \rightarrow r, R \}$
$\{ s: f: (r_1, \dots, r_n) \rightarrow r, R \}$
$\langle \text{signal} \rangle E$

- 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 r_i parameters for a stable instance of C
 - Creates an auxiliary event f' and dispatches it (bypassing the ether).
 - Transition acting on the region f' are:
 - Operation f on the region f' .
 - Operation f on the region f' .
 - For a non-stable instance, the call or blocks until stability is reached again.

\mathcal{C}
$\mathcal{C} = (F, \text{In}, \dots, \text{Out}, \dots)$
$\mathcal{C} = (F, \text{In}, \dots, \text{Out}, \dots)$
$\text{In} = \{i_1, i_2, \dots, i_n\}$
$\text{Out} = \{o_1, o_2, \dots, o_m\}$

- **Visibility:**
 - Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods.

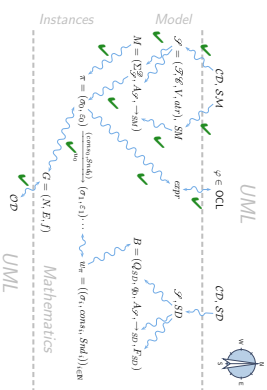
\mathcal{C}
$\mathcal{C} = (F, \text{In}, \dots, \text{Out}, \dots)$
$\mathcal{C} = (F, \text{In}, \dots, \text{Out}, \dots)$
$\text{In} = \{i_1, i_2, \dots, i_n\}$
$\text{Out} = \{o_1, o_2, \dots, o_m\}$

- **Visibility:**
 - Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods.
- **Useful properties:**
 - **concurrency** — is the read safe
 - **contention** — some synchronization 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.

You are here.

Course Map



References

