Software Design, Modelling and Analysis in UML

Lecture 16: Hierarchical State Machines I

2014-01-15

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

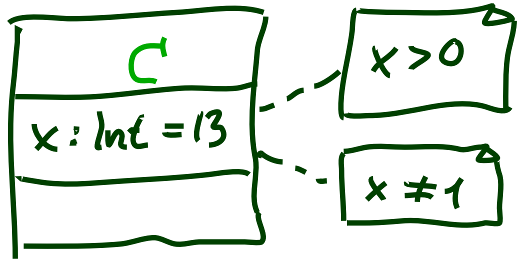
- Putting it all together: UML model semantics (so far)
- Rhapsody demo, code generation

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:**
  - State Machines and OCL
  - Hierarchical State Machines Syntax
  - Initial and Final State
  - Composite State Semantics
  - The Rest

State Machines and OCL

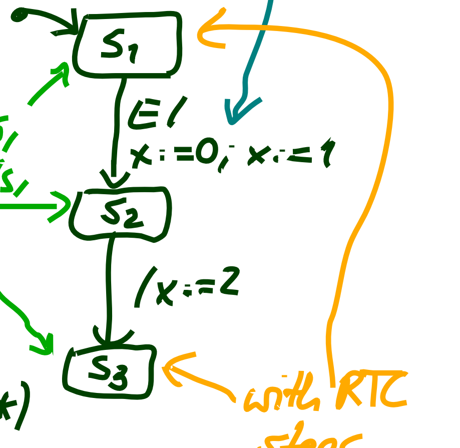
OCL Constraints and Behaviour



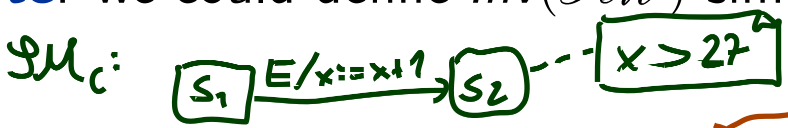
- Let $\mathcal{M} = (\mathscr{CD}, \mathscr{SM}, \mathscr{OD})$ be a UML model.
- We call $\mathcal{M}$ **consistent** iff, for each OCL constraint $expr \in Inv(\mathscr{CD})$,

  $\sigma \models expr$ for each "reasonable point" (σ, ε) of computations of $\mathcal{M}$.

  (~~Cf. exercises and tutorial for~~ discussion of "reasonable point".)



Note: we could define $Inv(\mathscr{SM})$ similar to $Inv(\mathscr{CD})$.



Pragmatics: context C inv: st=s2 implies x>27

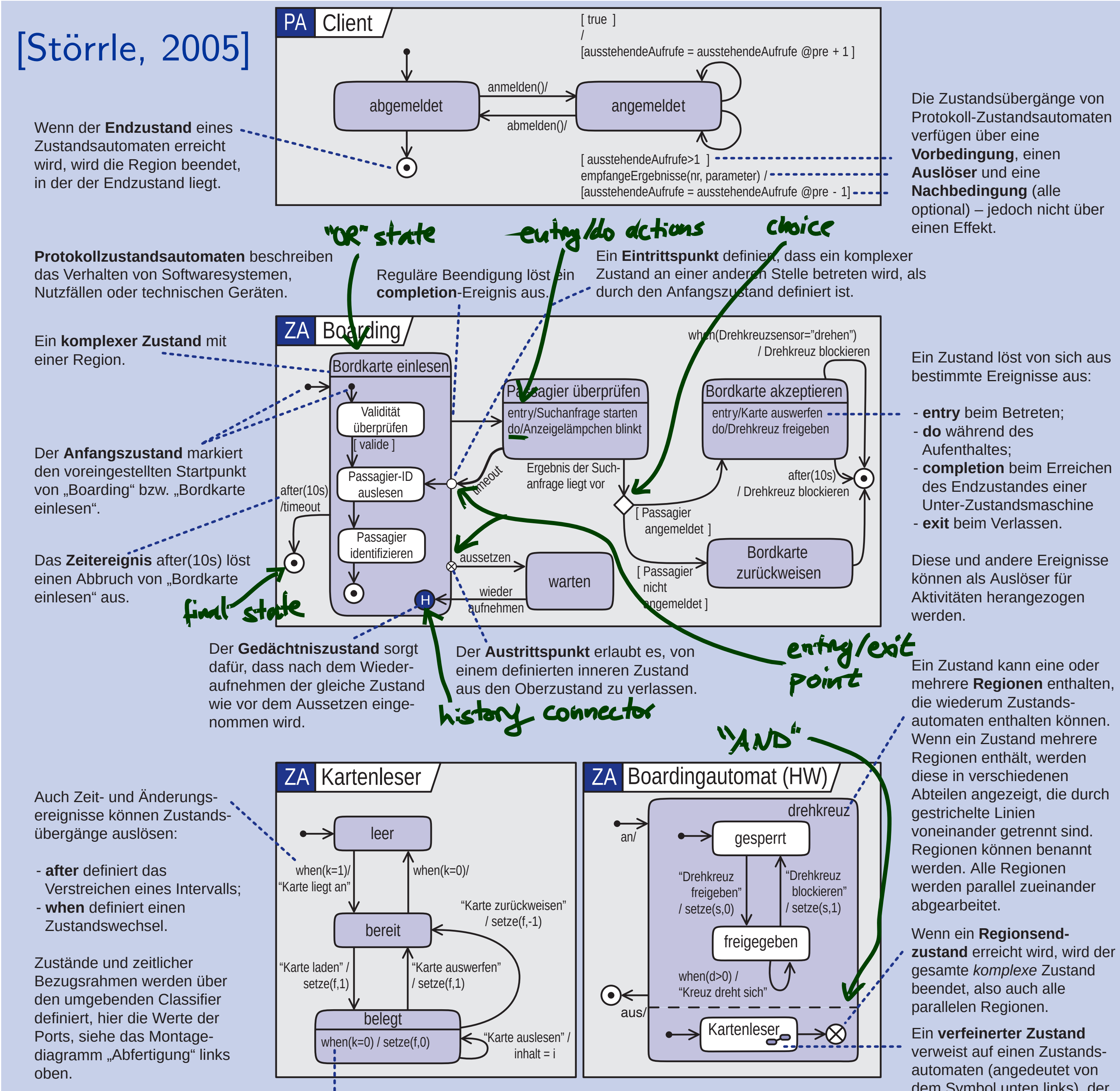
- In **UML-as-blueprint mode**, if $\mathscr{SM}$ doesn't exist yet, then $\mathcal{M} = (\mathscr{CD}, \emptyset, \mathscr{OD})$ is typically asking the developer to provide $\mathscr{SM}$ such that $\mathcal{M}' = (\mathscr{CD}, \mathscr{SM}, \mathscr{OD})$ is consistent.

  If the developer makes a mistake, then $\mathcal{M}'$ is inconsistent.

- **Not common**: if $\mathscr{SM}$ is given, then constraints are also considered when choosing transitions in the RTC-algorithm. In other words: even in presence of mistakes, the $\mathscr{SM}$ never move to inconsistent configurations.

Hierarchical State Machines

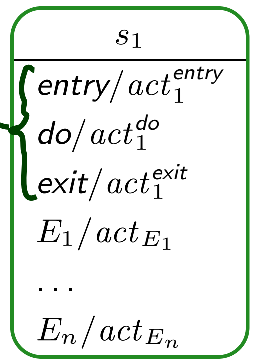
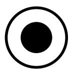
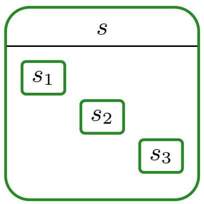
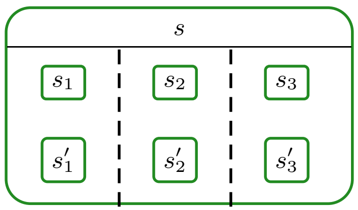
UML State-Machines: What do we have to cover?

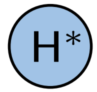
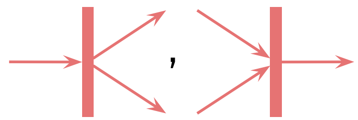
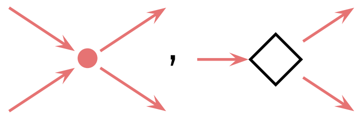
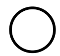


The Full Story

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

reserved keywords, not usable as signal name

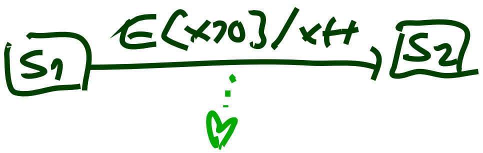
	example
simple state	
final state	
composite state	
OR	
AND	

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

Representing All Kinds of States

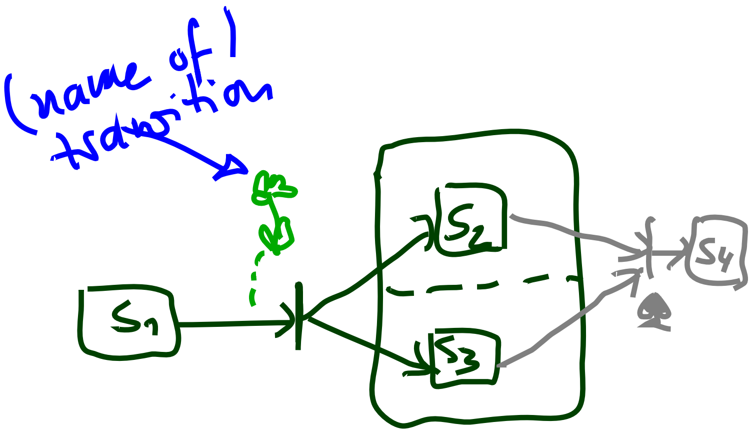
- **Until now**: $(S, s_0, \to)$, $\quad s_0 \in S, \to \;\subseteq S \times (\mathscr{E} \cup \{\_\}) \times Expr_{\mathscr{S}} \times Act_{\mathscr{S}} \times S$

  set of states, init. state, transitions; source, trigger, guard, action, dest



$\rightsquigarrow (s_1, E, x>0, x{+}{+}, s_2)$

- $(\{s_1, s_2\}, \{\heartsuit\}, \{\heartsuit \mapsto (\{s_1\}, \{s_2\})\})$



- (states, transitions, incidence) $(\{s_1, s_2, s_3\}, \{✿\}, \{✿ \mapsto (\{s_1\}, \{s_2, s_3\})\})$, ♠ $\mapsto (\{\{s_2, s_3\}, \{s_4\}\})$

Representing All Kinds of States

- **Until now**:

$$(S, s_0, \rightarrow), \quad s_0 \in S, \rightarrow \ \subseteq S \times (\mathscr{E} \cup \{\_\}) \times Expr_{\mathscr{S}} \times Act_{\mathscr{S}} \times S$$

- **From now on**: (**hierarchical**) **state machines**

$$(S, kind, region, \rightarrow, \psi, annot)$$

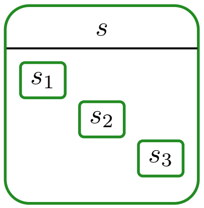
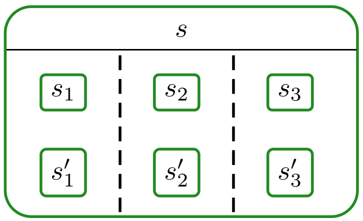
where

- $S \supseteq \{top\}$ is a finite set of (state machine) states **(as before)**,
- $kind : S \rightarrow \{st, init, fin, shist, dhist, fork, join, junc, choi, ent, exi, term\}$ is a function which labels states with their **kind**, **(new)**
- $region : S \rightarrow 2^{2^S}$ is a function which characterises the **regions** of a state, **(new)**
  sets of sets of states
- $\rightarrow$ is a set of transitions, (or transition names) **(changed)**
- $\psi : (\rightarrow) \rightarrow 2^S \times 2^S$ is an **incidence function**, and **(new)**
- $annot : (\rightarrow) \rightarrow (\mathscr{E} \cup \{\_\}) \times Expr_{\mathscr{S}} \times Act_{\mathscr{S}}$ provides an annotation for each transition. **(new)**

(s_0 is then redundant — replaced by proper state (!) of kind '*init*'.)

From UML to Hierarchical State Machines: By Example

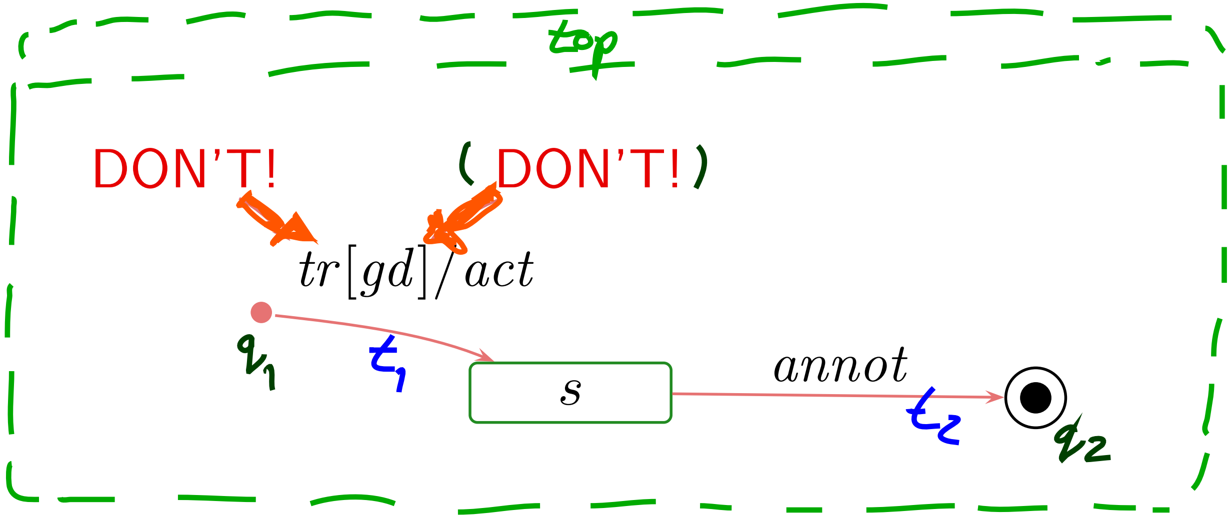
$$(S, kind, region, \rightarrow, \psi, annot)$$

	example	$\in S$	$kind$	$region$
simple state (nothing nested within)	s	s	st	$\emptyset$
final state	◉ fresh name →	q	fin	$\emptyset$
composite state				
OR	,	s	st	$\{\{s_1, s_2, s_3\}\}$
AND		s	st	$\{\{s_1, s_1'\}, \{s_2, s_2'\}, \{s_3, s_3'\}\}$
submachine state	(later)			
pseudo-state	●, Ⓗ, ...	q	$init, shist, \ldots$	$\emptyset$

region

$(s, kind(s))$ for short

From UML to Hierarchical State Machines: By Example



... translates to $(S, kind, region, \to, \psi, annot) =$

$$(\underbrace{\{(top, st), (s, st), (q_1, init), (q_2, fin)\}}_{S, kind},$$

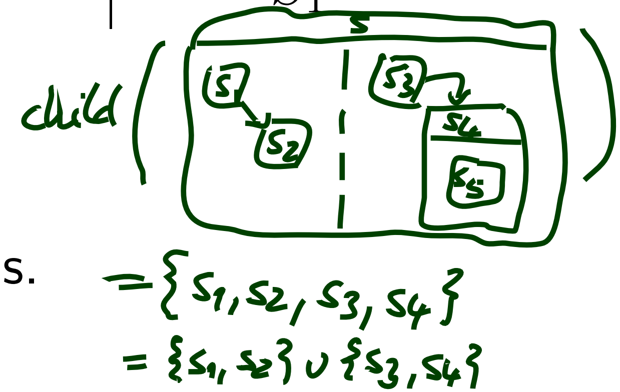
$$\underbrace{\{top \mapsto \{\{q_1, s, q_2\}\}, s \mapsto \emptyset, q_1 \mapsto \emptyset, q_2 \mapsto \emptyset\}}_{region},$$

$$\underbrace{\{t_1, t_2\}}_{\to}, \quad \underbrace{\{t_1 \mapsto (\{q_1\}, \{s\}), t_2 \mapsto (\{s\}, \{q_2\})\}}_{\psi},$$

$$\underbrace{\{t_1 \mapsto (tr, gd, act), t_2 \mapsto annot\}}_{annot})$$

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*	$\{S_1, \ldots, S_n\}, n \geq 1$	$S_1 \cup \cdots \cup S_n$
pseudo-state	s	*init*, ...	$\emptyset$	$\emptyset$
implicit top state	top	*st*	$\{S_1\}$	S_1

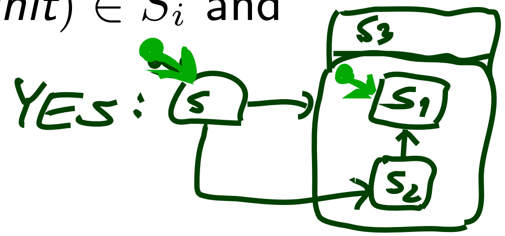


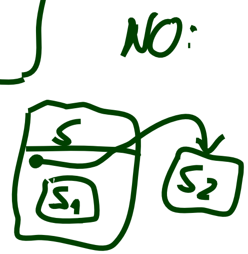
- Each state (except for top) lies in exactly one region,
- States $s \in S$ with $kind(s) =$ *st* **may comprise** regions.
  - No region: simple state.
  - One region: OR-state.
  - Two or more regions: AND-state.
- Final and pseudo states **don't comprise** regions.

- The region function induces a **child** function.

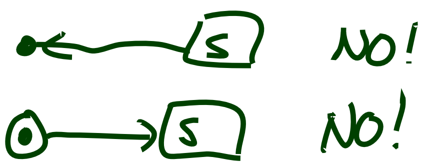
Well-Formedness: Initial State (requirement on diagram)

- Each non-empty region has a reasonable initial state and at least one transition from there, i.e.
  - for each $s \in S$ with $region(s) = \{S_1, \ldots, S_n\}$, $n \geq 1$, for each $1 \leq i \leq n$,
  - there exists exactly one initial pseudo-state $(s_1^i, \mathit{init}) \in S_i$ and at least one transition $t \in \longrightarrow$ with s_1^i as source,
  - and such transition's target s_2^i is in S_i, and (**for simplicity!**) $kind(s_2^i) = \mathit{st}$, and $annot(t) = (\_, \mathit{true}, act)$.

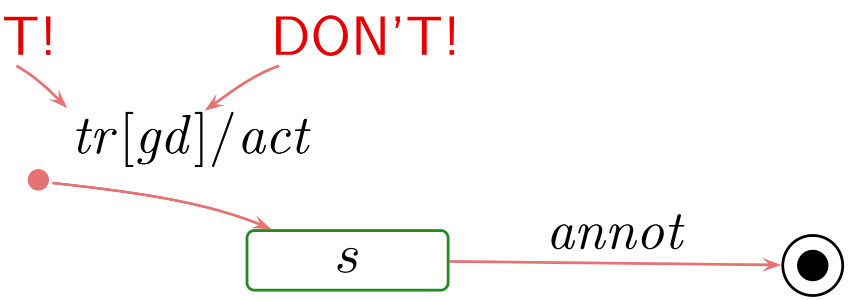




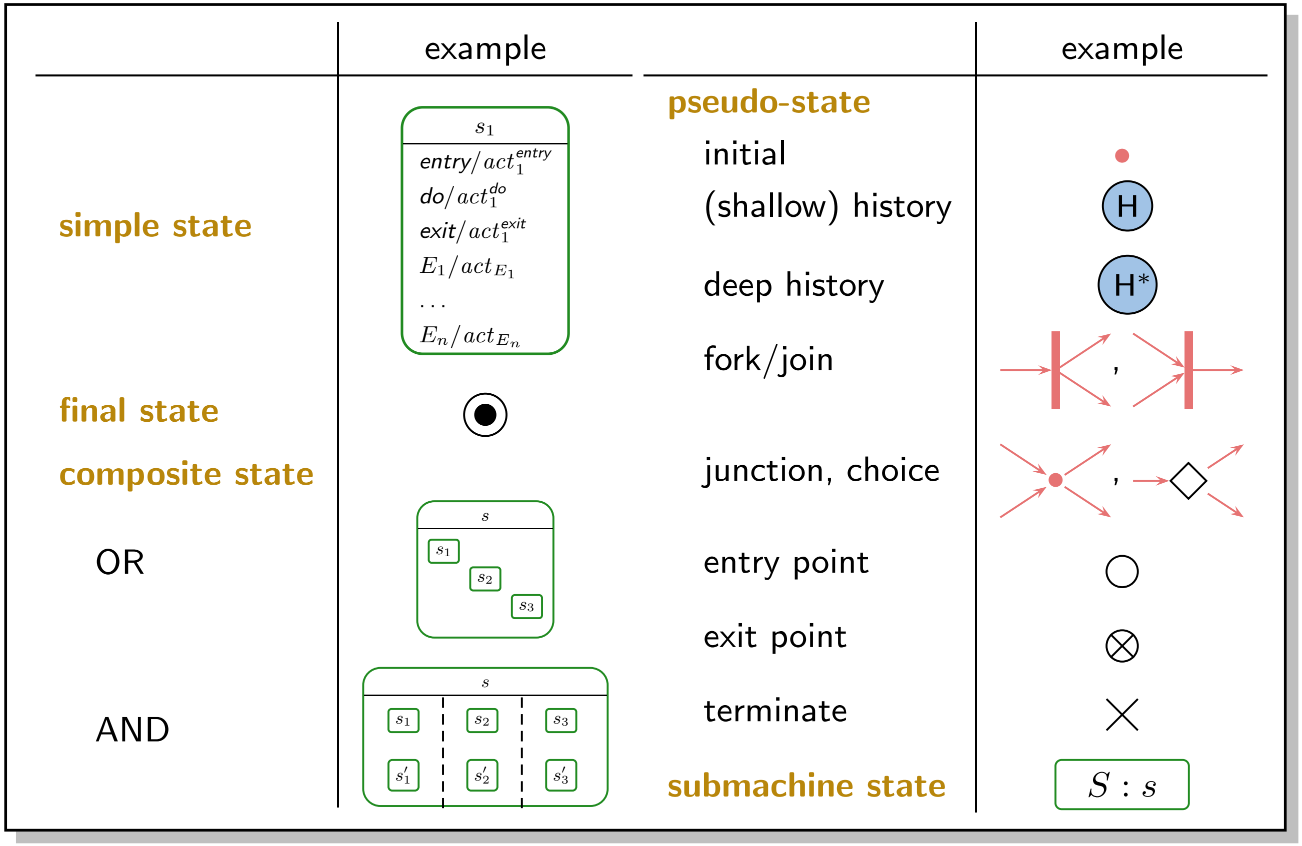
- No ingoing transitions to initial states.
- No outgoing transitions from final states.



- Recall:



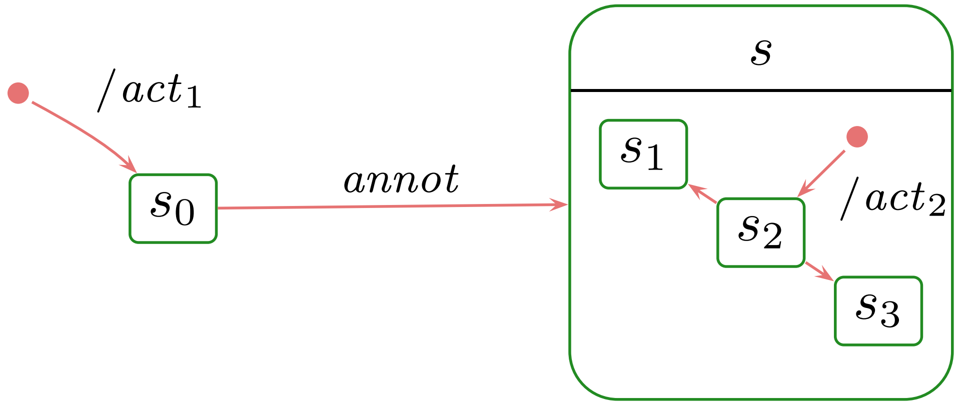
Plan



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

Initial Pseudostates and Final States

Initial Pseudostate



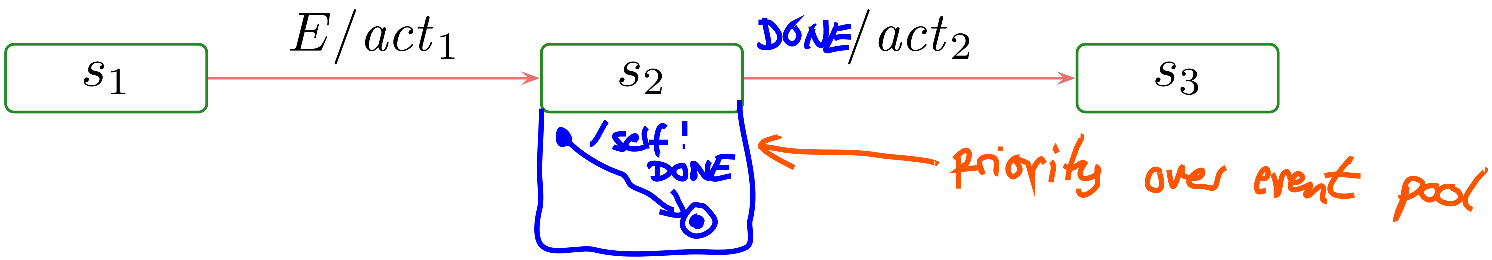
Principle:

- when entering a region **without** a specific destination state, i.e. from (s, init) to some (s′, st)
- 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 (see later).

Special case: the region of top.

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

Towards Final States: Completion of States

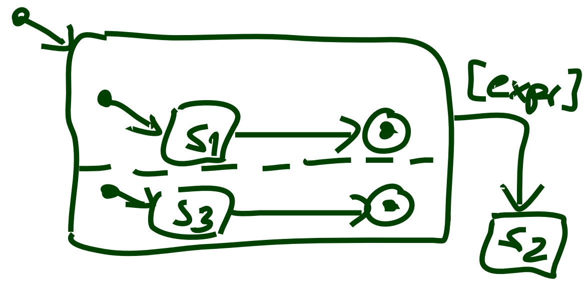


- Transitions without trigger can **conceptionally** be viewed as being sensitive for the "completion event".

- 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** —, e.g. "DONE"
  - (v) raise a **completion event** — with strict priority over events from ether!
  - (vi) if there is a transition enabled which is sensitive for the completion event,
    - then take it (here: (s_2, s_3)).
    - otherwise become stable.

Final States

there are ones only for AND states



- If
  - a step of object u moves u into a final state (s, fin), 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** (i.e., inverse of child) of s enabled which is sensitive for the completion event,
  - then take that transition,
  - otherwise kill u

  $\rightsquigarrow$ adjust $(2.)$ and $(3.)$ in the semantics accordingly

- **One consequence**: u never survives reaching a state (s, fin) with $s \in \mathit{child}(\mathit{top})$.

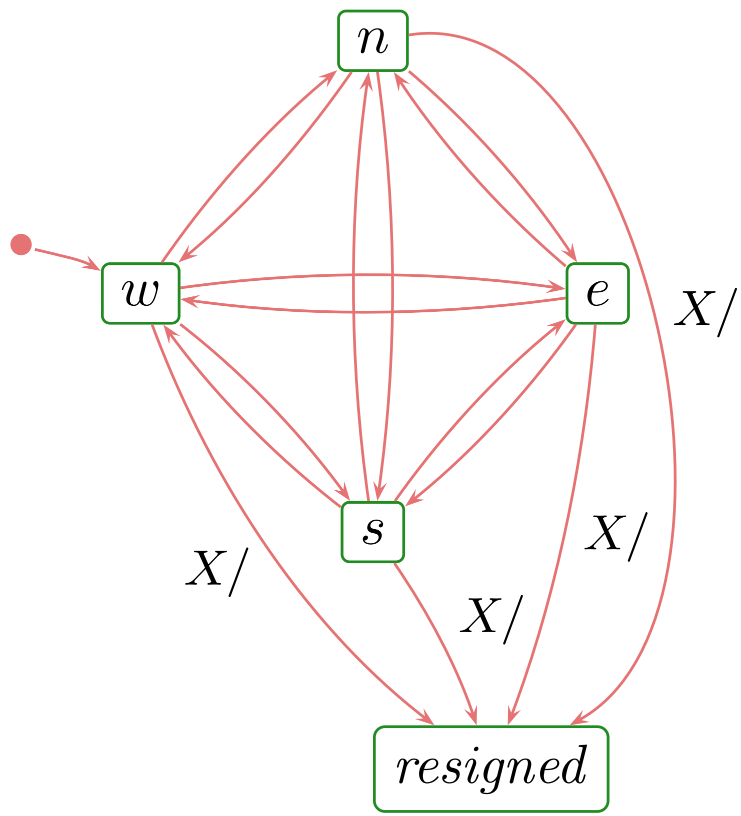
Composite States

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

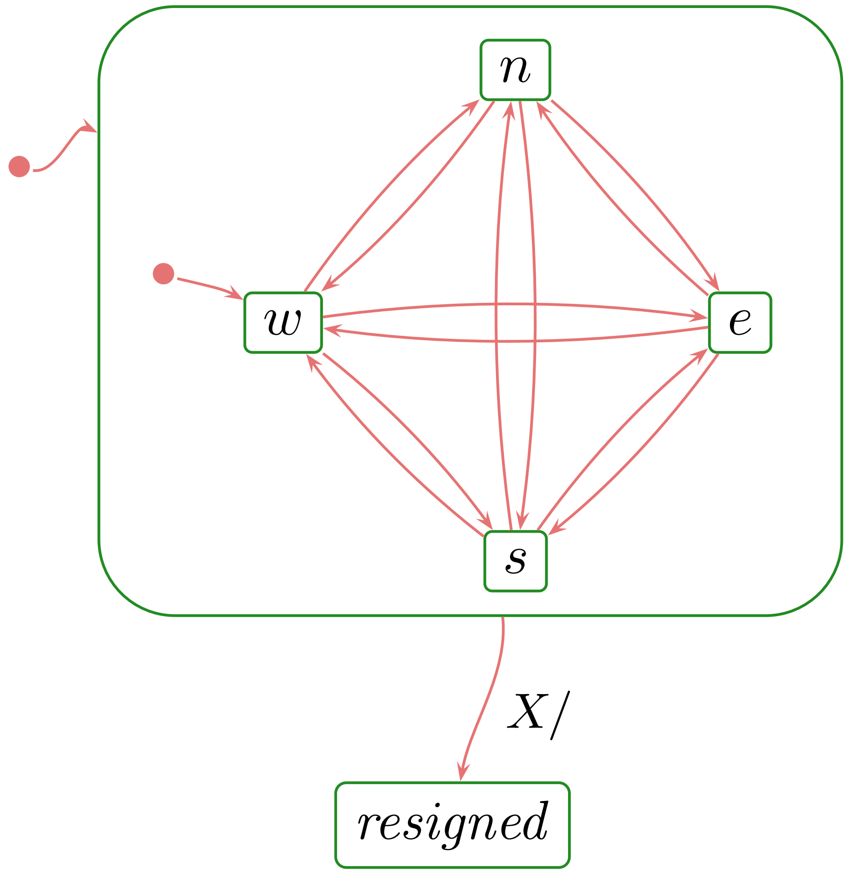
Composite States

- In a sense, composite states are about **abbreviation**, **structuring**, and **avoiding redundancy**.

- Idea: in Tron, for the Player's Statemachine, instead of

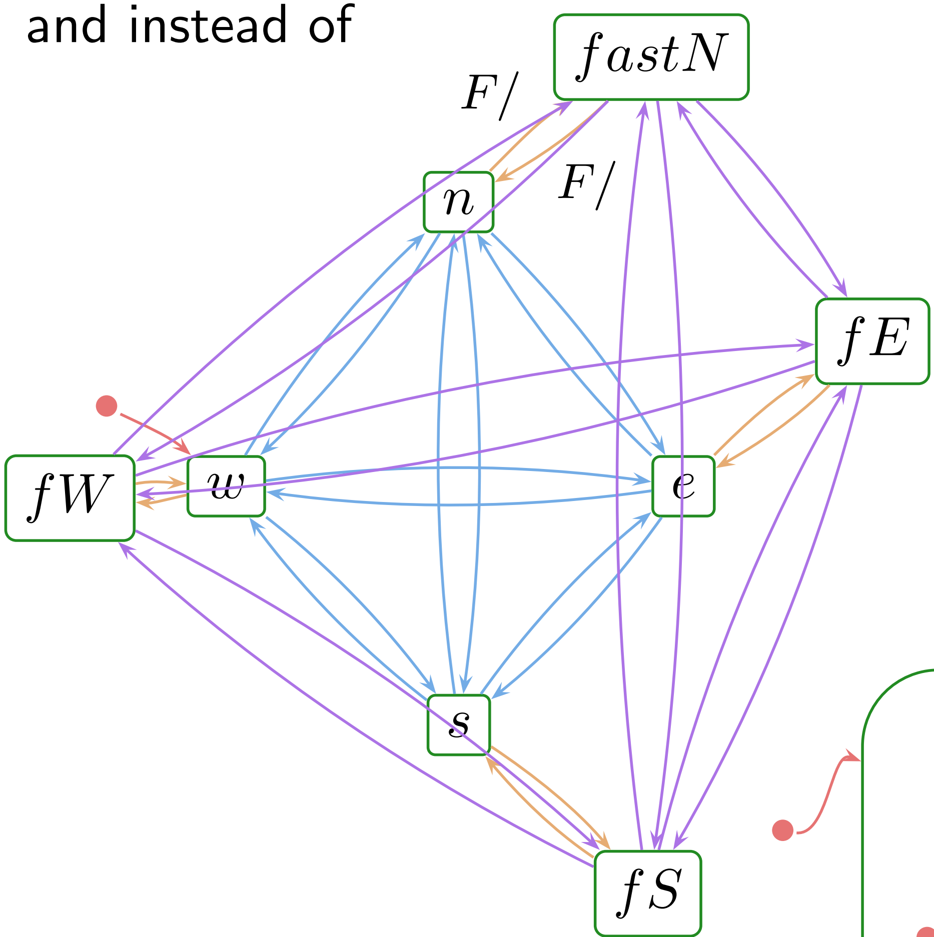


write

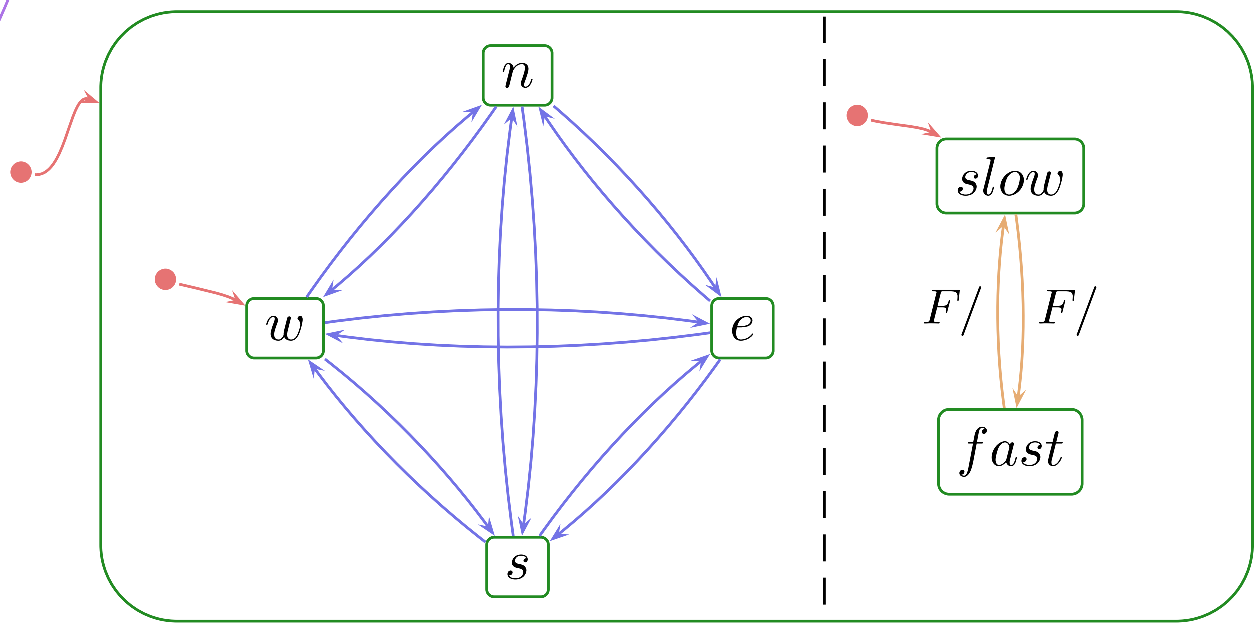


Composite States

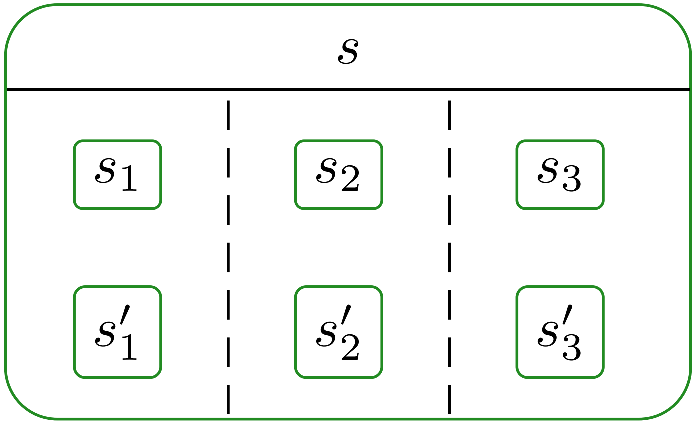
and instead of



write



Recall: Syntax



translates to

$$(\underbrace{\{(top, \mathsf{st}), (s, \mathsf{st}), (s_1, \mathsf{st})(s'_1, \mathsf{st})(s_2, \mathsf{st})(s'_2, \mathsf{st})(s_3, \mathsf{st})(s'_3, \mathsf{st})\}}_{S, kind},$$

$$\underbrace{\{top \mapsto \{s\}, s \mapsto \{\{s_1, s'_1\}, \{s_2, s'_2\}, \{s_3, s'_3\}\}, s_1 \mapsto \emptyset, s'_1 \mapsto \emptyset, \ldots\}}_{region},$$

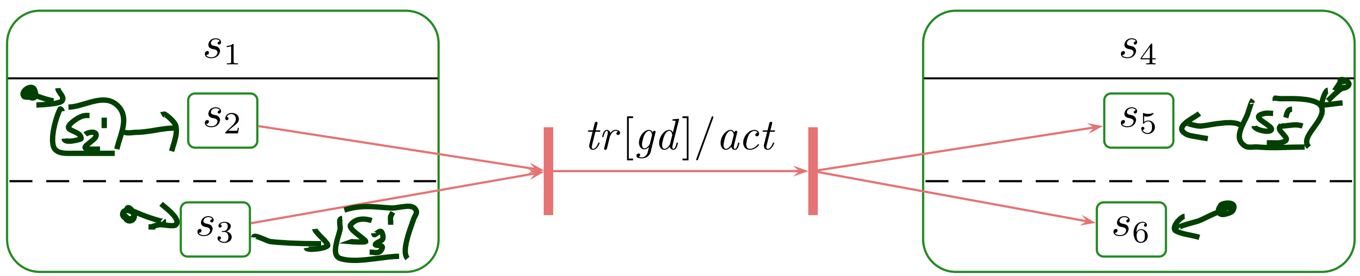
$$\to, \psi, annot)$$

Syntax: Fork/Join

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

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

- For instance,

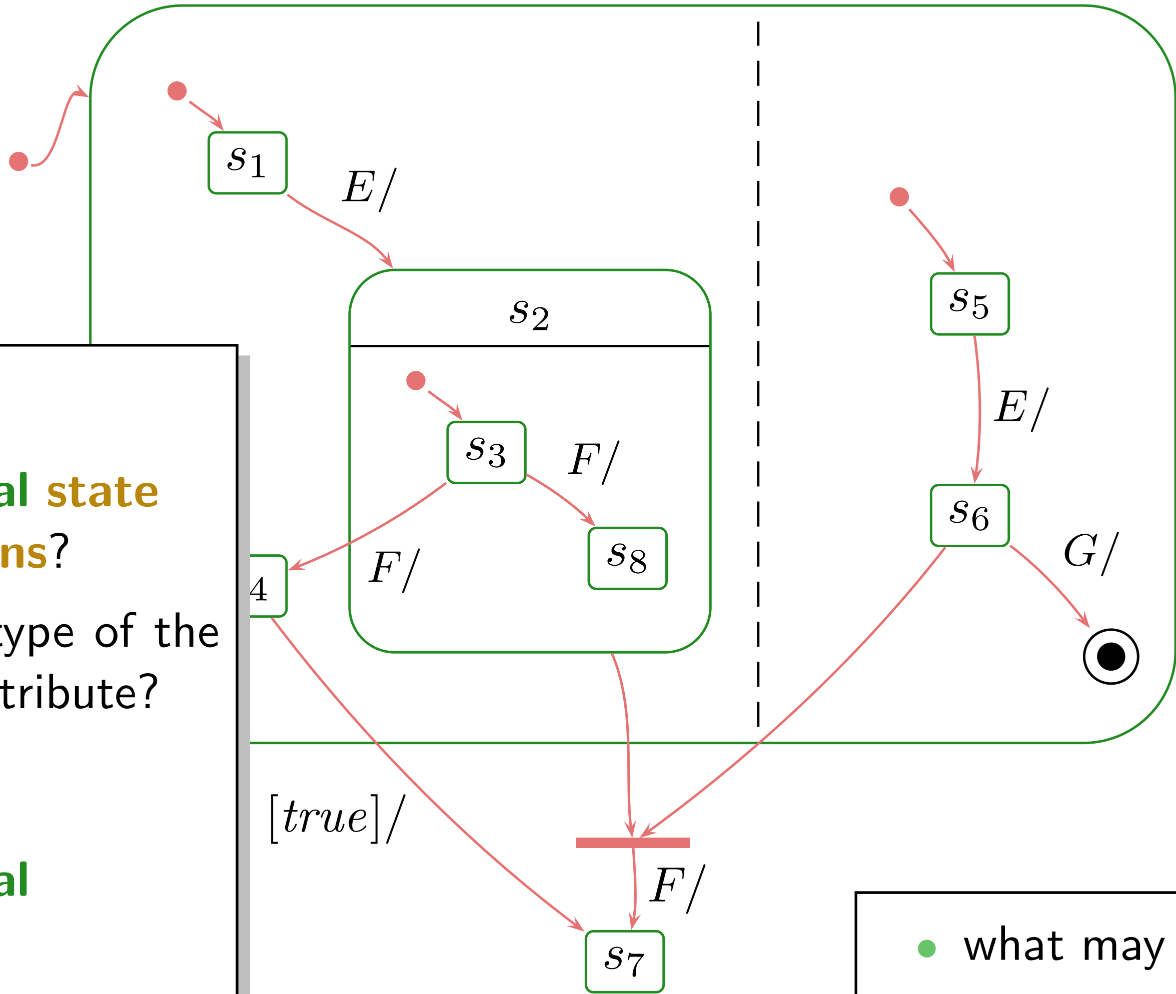


translates to

$$(S, kind, region, \underbrace{\{t_1\}}_{\rightarrow}, \underbrace{\{t_1 \mapsto (\{s_2, s_3\}, \{s_5, s_6\})\}}_{\psi}, \underbrace{\{t_1 \mapsto (tr, gd, act)\}}_{annot})$$

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

Composite States: Blessing or Curse?



States:

- what are **legal state configurations**?
- what is the type of the implicit st attribute?

Transitions:

- what are **legal** transitions?
- when is a transition enabled?
- what effects do transitions have?

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

State Configuration

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

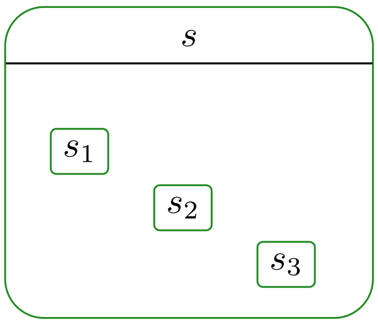
$$|\{s_0 \in R \mid kind(s_0) \in \{\textit{st}, \textit{fin}\}\} \cap S_1| = 1.$$

State Configuration

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

$$|\{s_0 \in R \mid kind(s_0) \in \{\mathit{st}, \mathit{fin}\}\} \cap S_1| = 1.$$

- **Examples**:

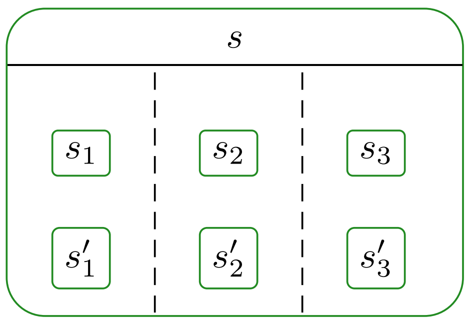


State Configuration

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

$$|\{s_0 \in R \mid kind(s_0) \in \{\mathit{st}, \mathit{fin}\}\} \cap S_1| = 1.$$

- **Examples**:



A Partial Order on States

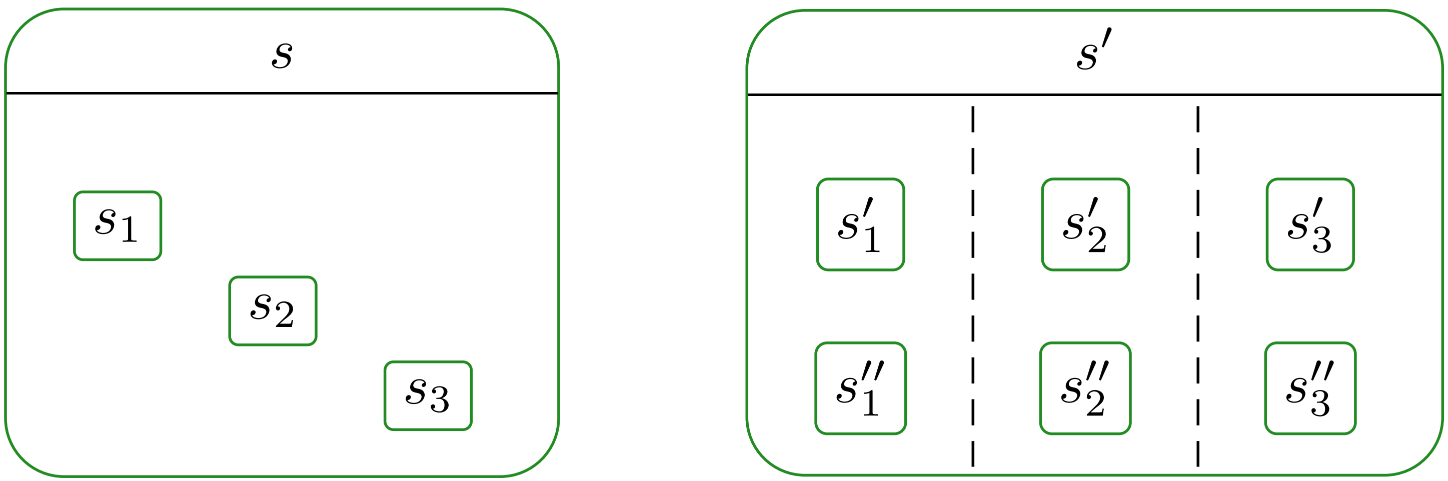
The substate- (or **child-**) relation **induces** a **partial order on states**:

- $top \leq s$, 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 $s'' \leq s'$.

A Partial Order on States

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

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



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.

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

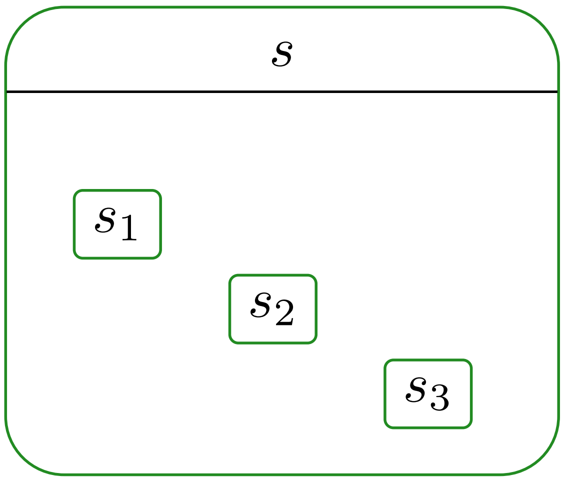
  - $lca(S_1)$ is minimal, i.e. if $\hat{s} \leq s$ for all $s \in S_1$, then $\hat{s} \leq lca(S_1)$
- **Note**: $lca(S_1)$ exists for all $S_1 \subseteq S$ (last candidate: top).

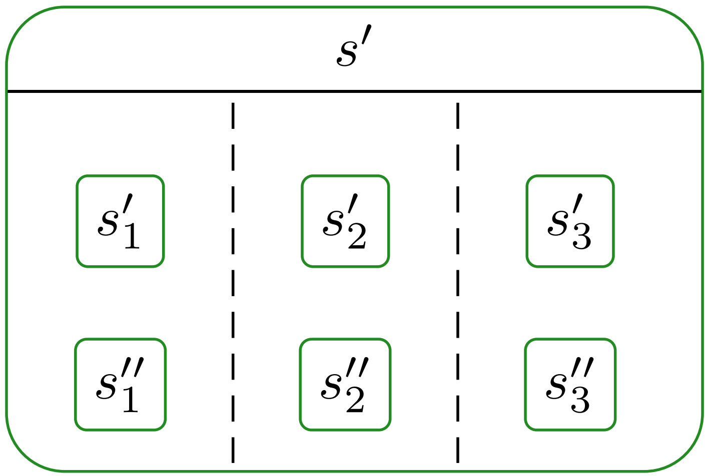
Least Common Ancestor and Ting

- 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

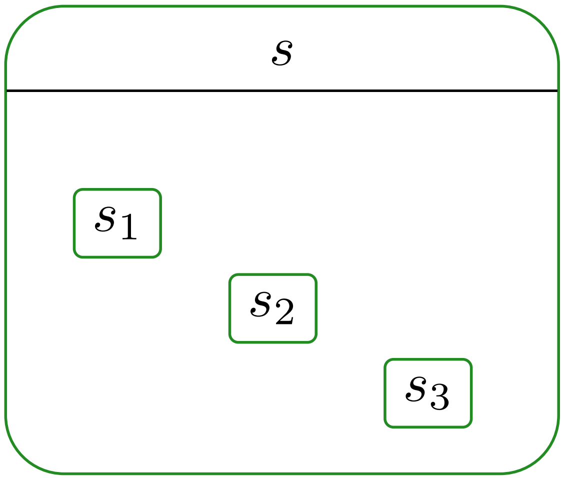
- Two states $s_1, s_2 \in S$ are called **orthogonal**, denoted $s_1 \perp s_2$, if and only if
  - they are unordered, i.e. $s_1 \not\leq s_2$ and $s_2 \not\leq s_1$, and
  - they "live" in different regions of an AND-state, i.e.

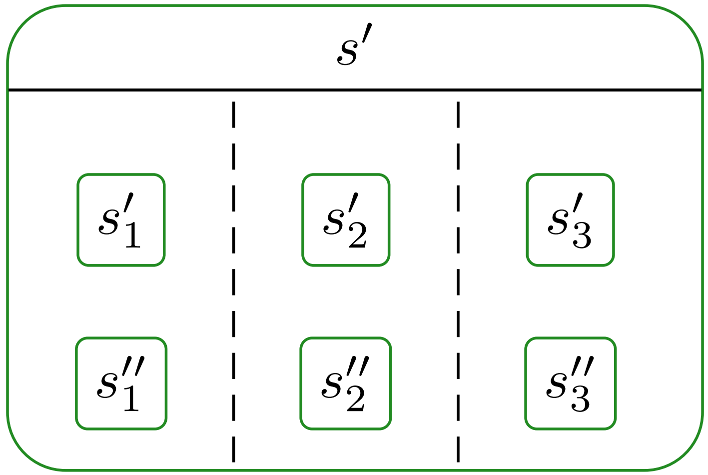
$$\exists\, s, \mathit{region}(s) = \{S_1, \dots, S_n\}\ \exists\, 1 \leq i \neq j \leq n : s_1 \in \mathit{child}^*(S_i) \wedge s_2 \in \mathit{child}^*(S_j),$$

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.

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



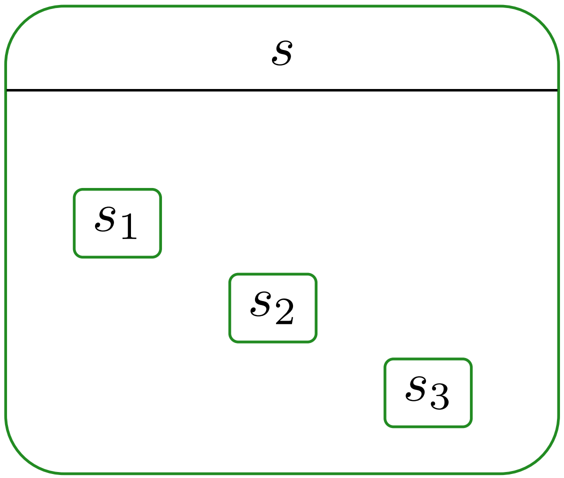


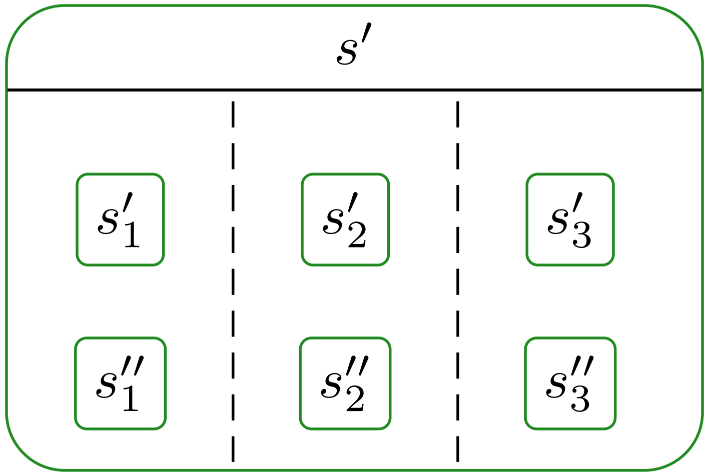
Least Common Ancestor and Ting

- 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
  - $s \perp s'$.

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

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

(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.

Legal Transitions

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

(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.

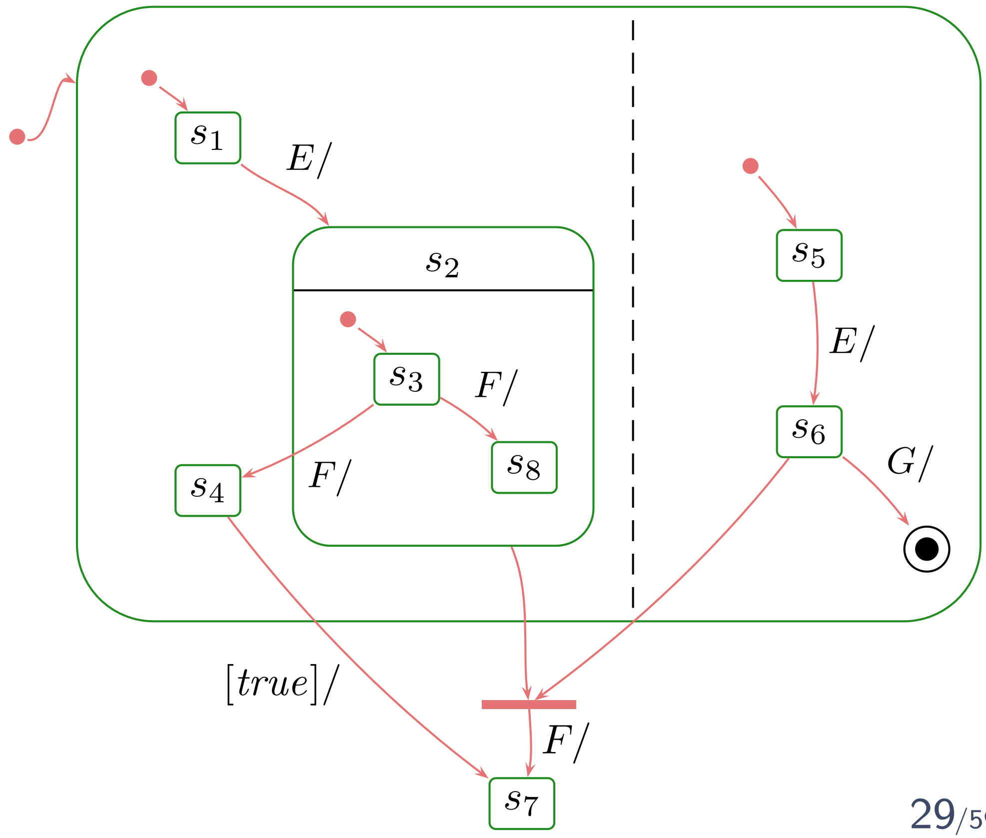
- forall $s, s' \in source(t)$ ($\in target(t)$), $s \perp s'$,

(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:



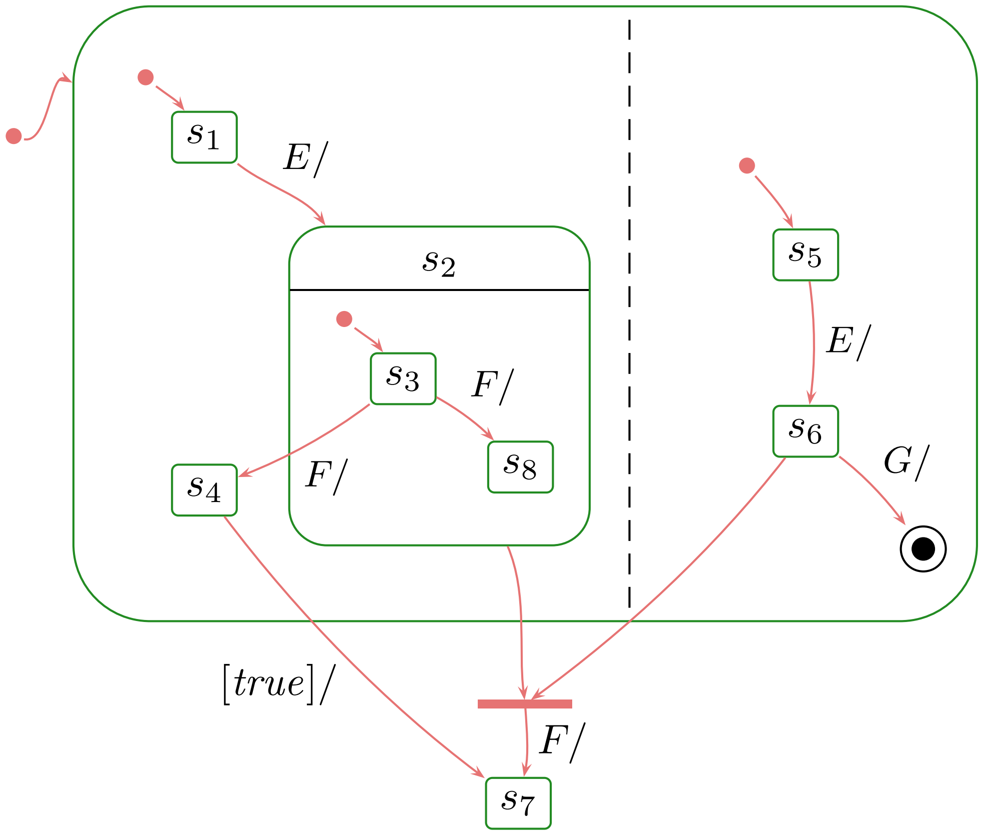
The Depth of States

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

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

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

- 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

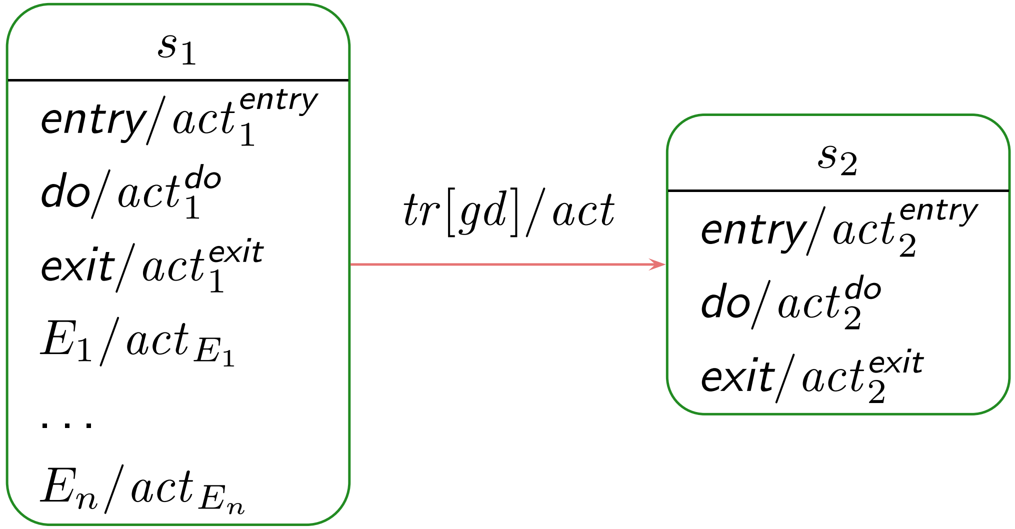
- Let T be a set of transitions enabled in u.
- Then $(\sigma, \varepsilon) \xrightarrow{(cons, Snd)} (\sigma', \varepsilon')$ if
  - $\sigma'(u)(st)$ consists of the target states of t,

    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.

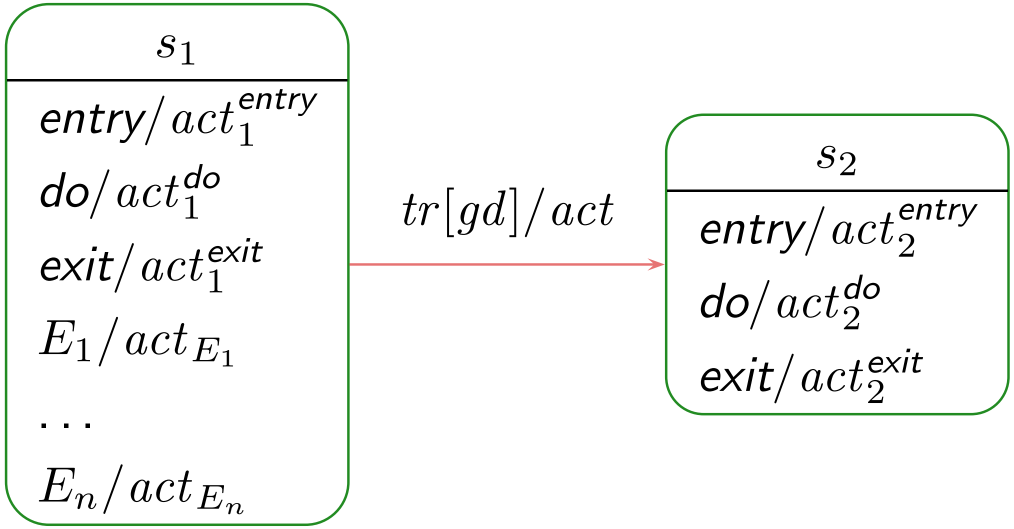
Entry/Do/Exit Actions, Internal Transitions

Entry/Do/Exit Actions



- 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). $E_1, \dots, E_n \in \mathscr{E}$, 'entry', 'do', 'exit' are reserved names!

Entry/Do/Exit Actions



- 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). $E_1, \dots, E_n \in \mathscr{E}$, 'entry', 'do', 'exit' are reserved names!

- Recall: each action's supposed to have a transformer. Here: $t_{act_1^{entry}}$, $t_{act_1^{exit}}$, ...
- Taking the transition above then amounts to applying

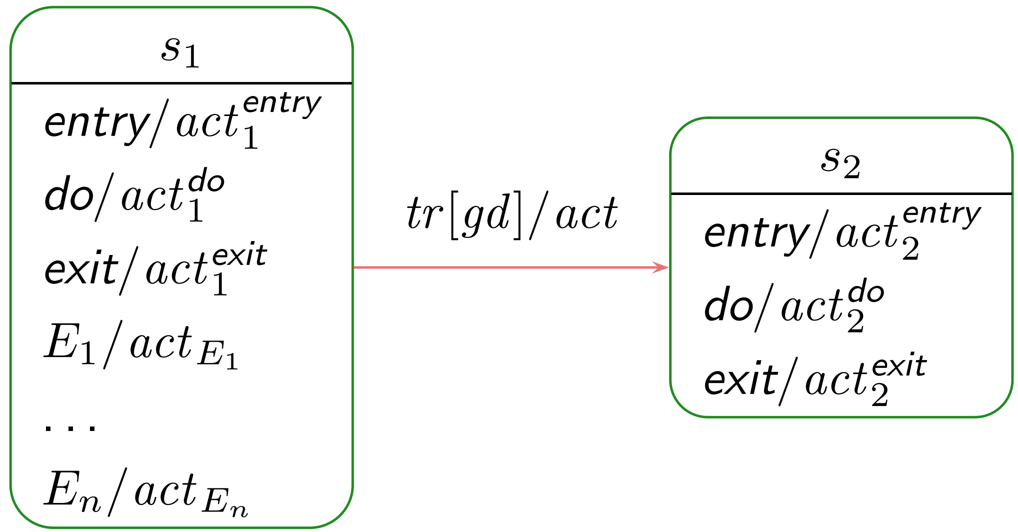
$$t_{act_{s_2}^{entry}} \circ t_{act} \circ t_{act_{s_1}^{exit}}$$

instead of only

$$t_{act}$$

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

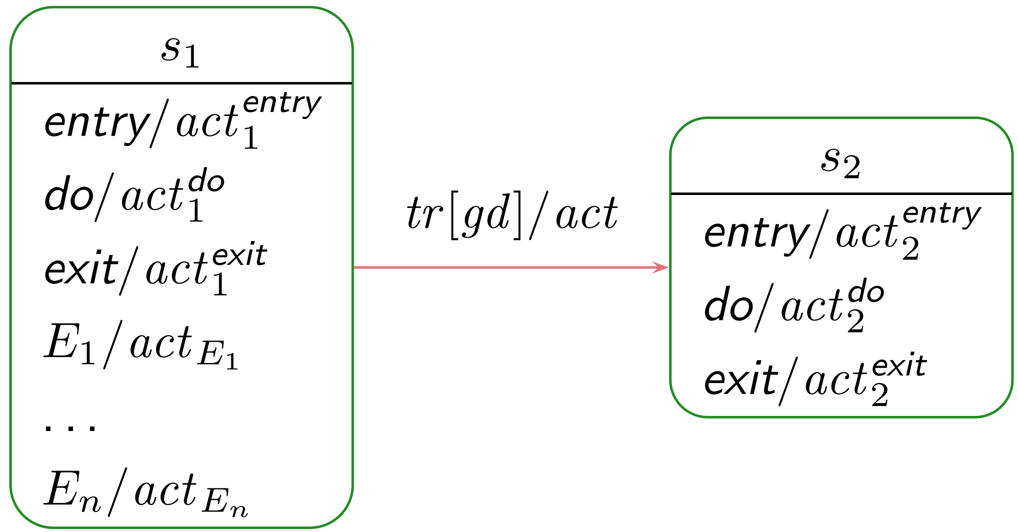
Internal Transitions



- For **internal transitions**, taking the one for E_1, for instance, still amounts to taking **only** $t_{act_{E_1}}$.
- Intuition: The state is neither left nor entered, so: no exit, no entry.

  $\rightsquigarrow$ adjust $(2.)$ accordingly.
- Note: internal transitions also start a run-to-completion step.

Internal Transitions

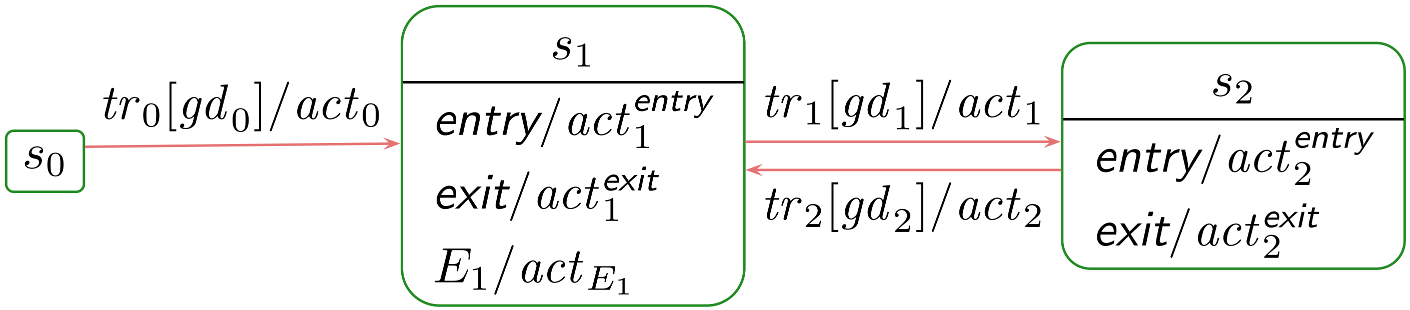


- For **internal transitions**, taking the one for E_1, for instance, still amounts to taking **only** $t_{act_{E_1}}$.
- Intuition: The state is neither left nor entered, so: no exit, no entry.

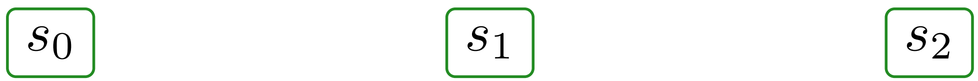
  $\rightsquigarrow$ adjust $(2.)$ accordingly.
- Note: internal transitions also start a run-to-completion step.
- Note: the standard seems not to clarify whether internal transitions have **priority** over regular transitions with the same trigger at the same state.

  Some code generators assume that internal transitions have priority!

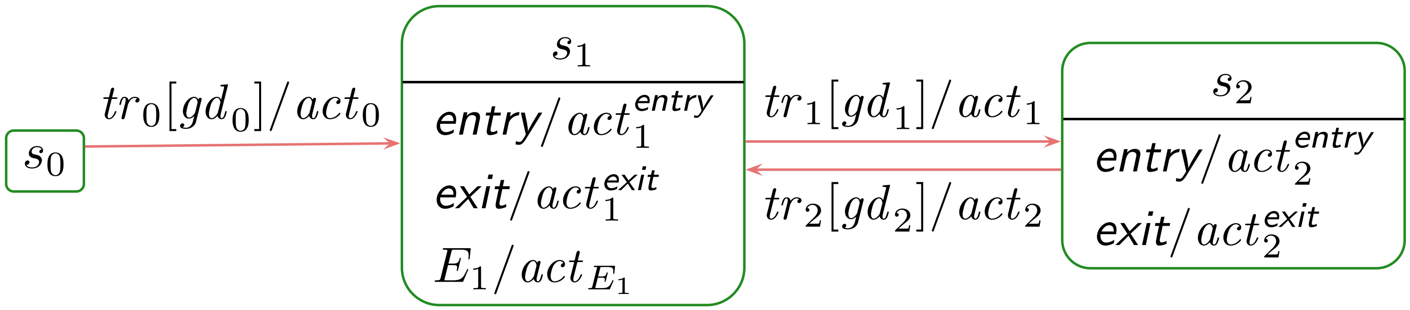
Alternative View: Entry/Exit/Internal as Abbreviations



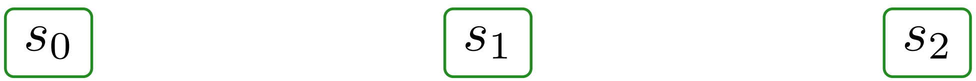
- ... as abbrevation for ...



Alternative View: Entry/Exit/Internal as Abbreviations

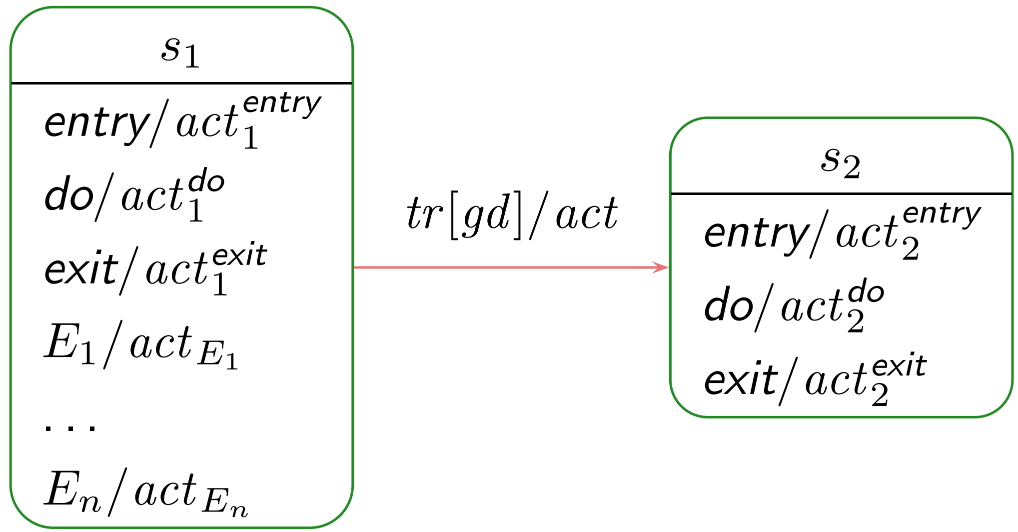


- ... as abbrevation for ...



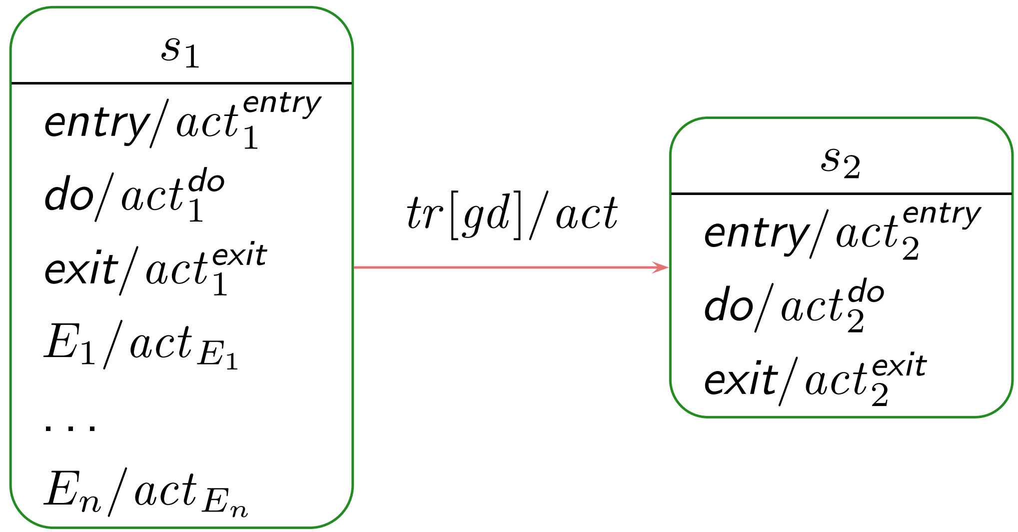
- That is: Entry/Internal/Exit don't add expressive power to Core State Machines. If internal actions should have priority, s_1 can be embedded into an OR-state (see later).
- Abbreviation may avoid confusion in context of hierarchical states (see later).

Do Actions



- **Intuition**: after entering a state, start its do-action.
- If the do-action terminates,
  - then the state is considered **completed**,
- otherwise,
  - if the state is left before termination, the do-action is stopped.

Do Actions



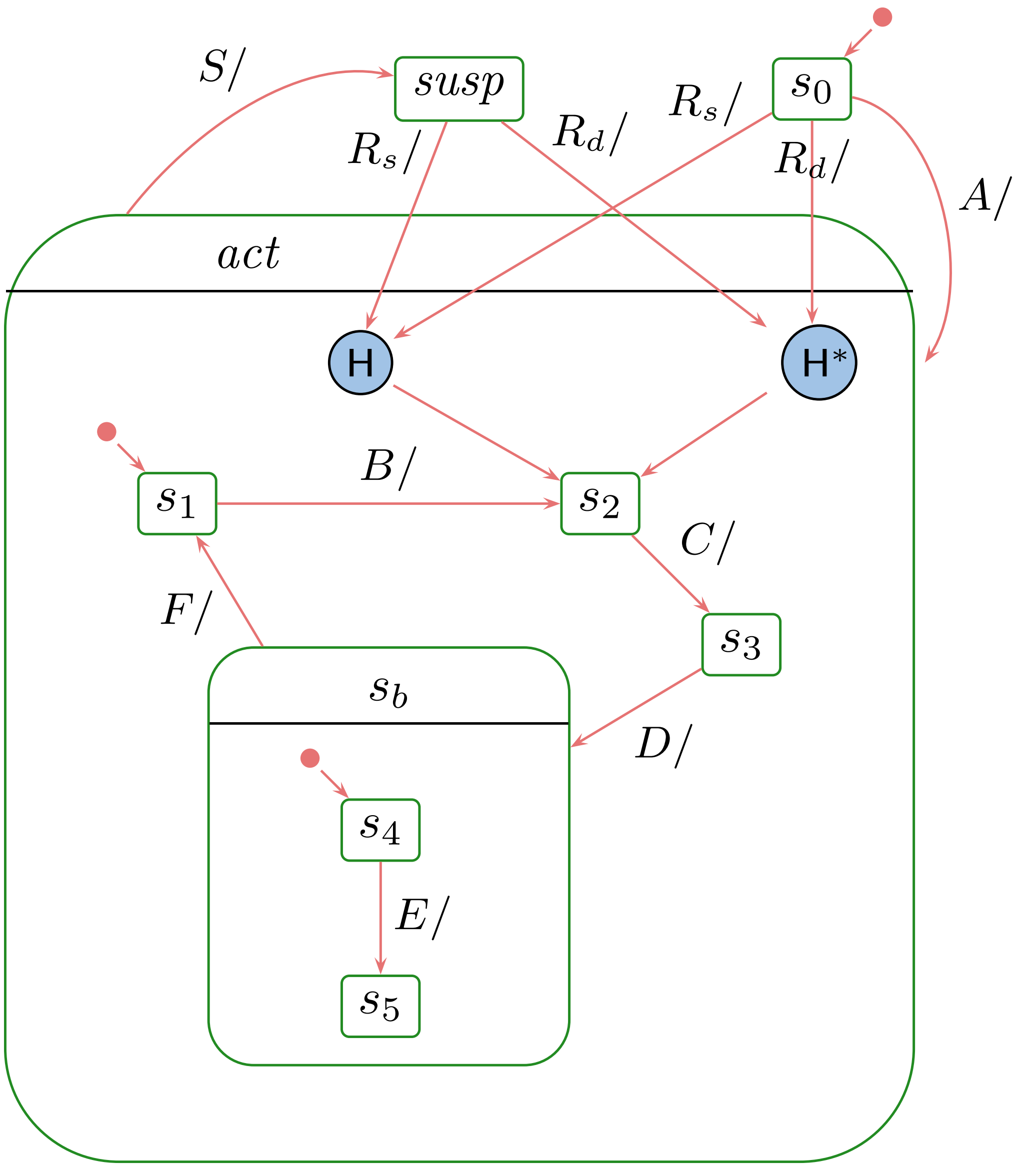
- **Intuition**: after entering a state, start its do-action.
- If the do-action terminates,
  - then the state is considered **completed**,
- otherwise,
  - if the state is left before termination, the do-action is stopped.

- Recall the overall UML State Machine philosophy:

  "An object is either idle or doing a run-to-completion step."
- Now, what is it exactly while the do action is executing...?

The Concept of History, and Other Pseudo-States

History and Deep History: By Example



What happens on...

- 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?

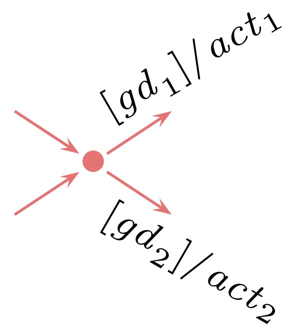
  $s_0, s_1, s_2, s_3, s_4, s_5, susp, s_3$

- A, B, C, D, R_d?

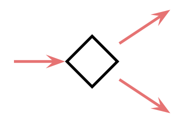
  $s_0, s_1, s_2, s_3, s_4, s_5, susp, s_5$

Junction and Choice

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



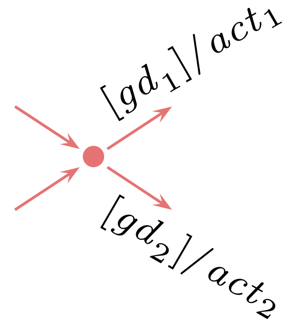
- Choice: (**"dynamic conditional branch"**)



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

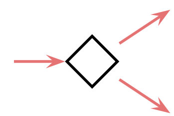
Junction and Choice

- 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

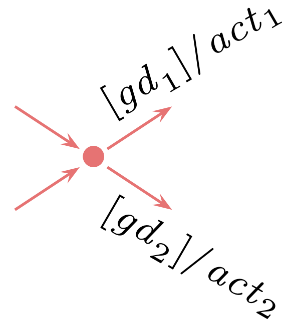
- Choice: (**"dynamic conditional branch"**)



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

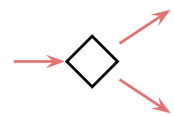
Junction and Choice

- 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.

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.

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.

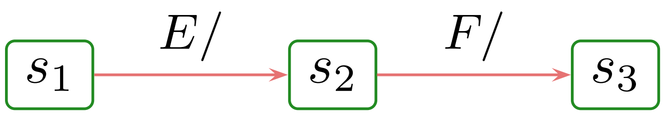
Deferred Events in State-Machines

Deferred Events: Idea

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

The idea is as follows:

- Consider the following state machine:



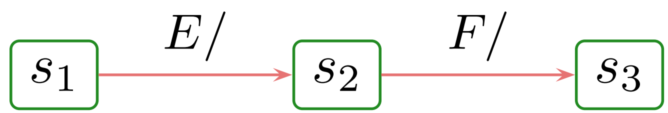
- Assume we're stable in s_1, and F is ready in the ether.
- In **the framework of the course**, F is **discarded**.

Deferred Events: Idea

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

The idea is as follows:

- Consider the following state machine:



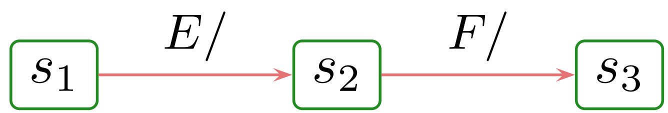
- Assume we're stable in s_1, and F is ready in the ether.
- In **the framework of the course**, F is **discarded**.
- But we **may** find it a pity to discard the poor event and **may** want to remember it for later processing, e.g. in s_2, in other words, **defer** it.

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:

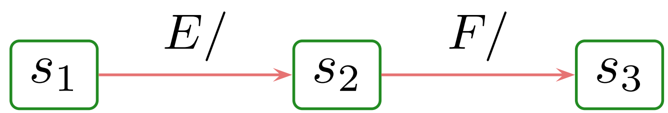
- Provide a pattern how to "program" this (use self-loops and helper attributes).
- Turn it into an original language concept.

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.

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.

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.

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.

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.

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...?

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.

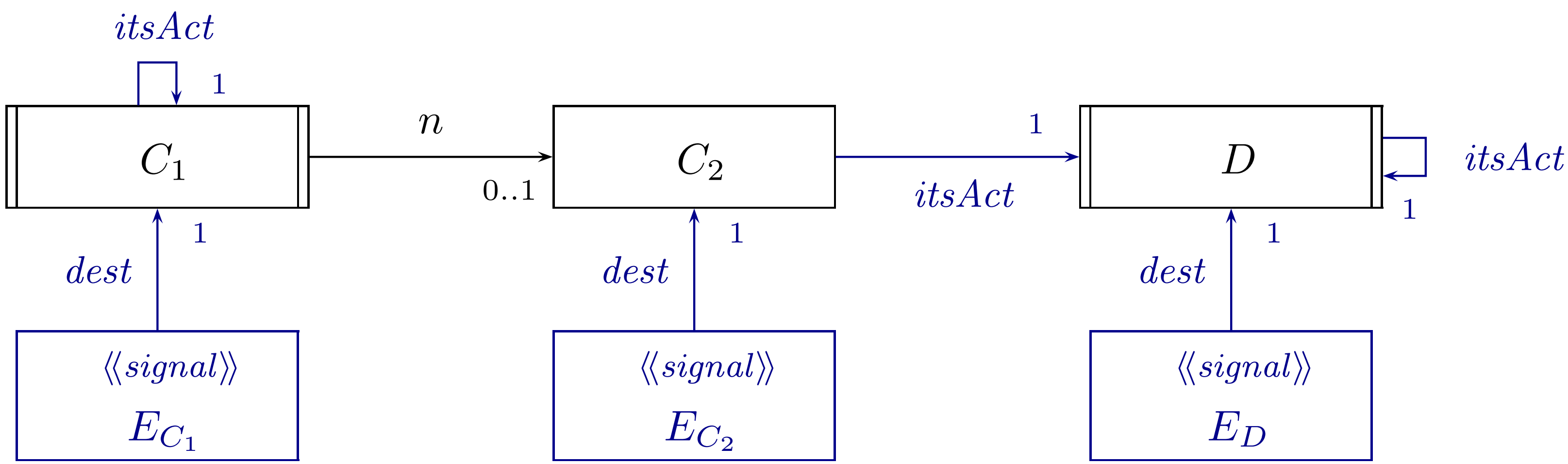
  Which of them (can) start a run-to-completion step...?
  Do run-to-completion steps still interleave...?

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.

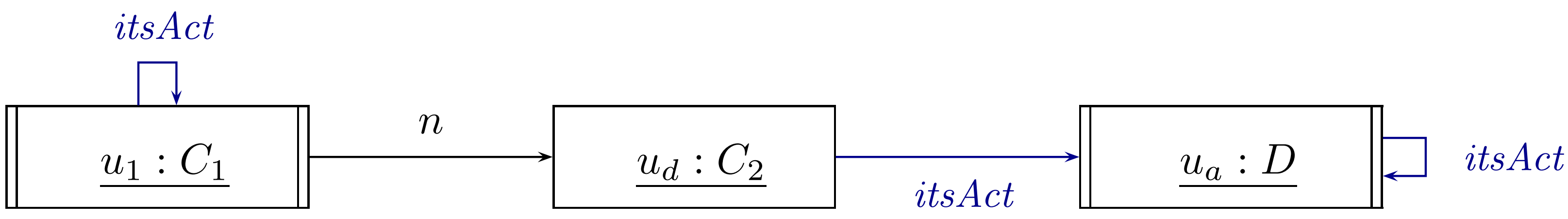
Passive Reactive Classes

- 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 $u_a = u$.

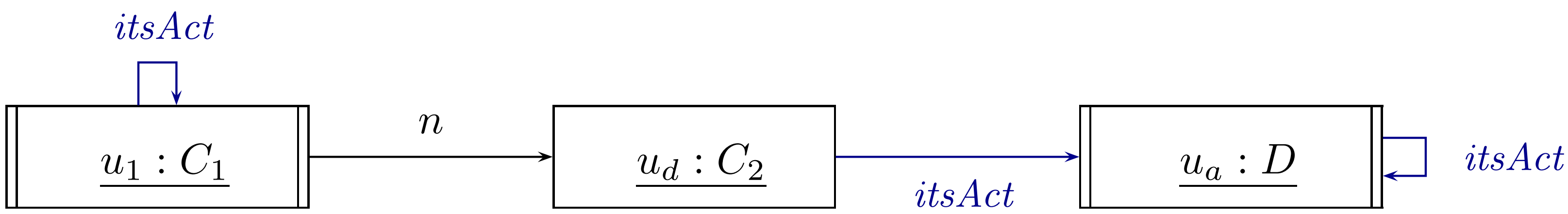


Sending an event:

- 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., $\varepsilon' = \varepsilon \oplus (u_a, u_e)$.

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.

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

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

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").

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:

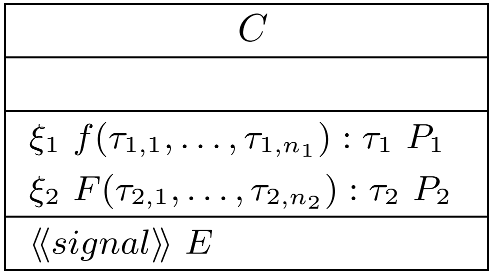
- The **implementation** of a behavioural feature can be provided by:
  - An **operation**.

    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



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

Behavioural Features: Visibility and Properties

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

- **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".

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)

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**

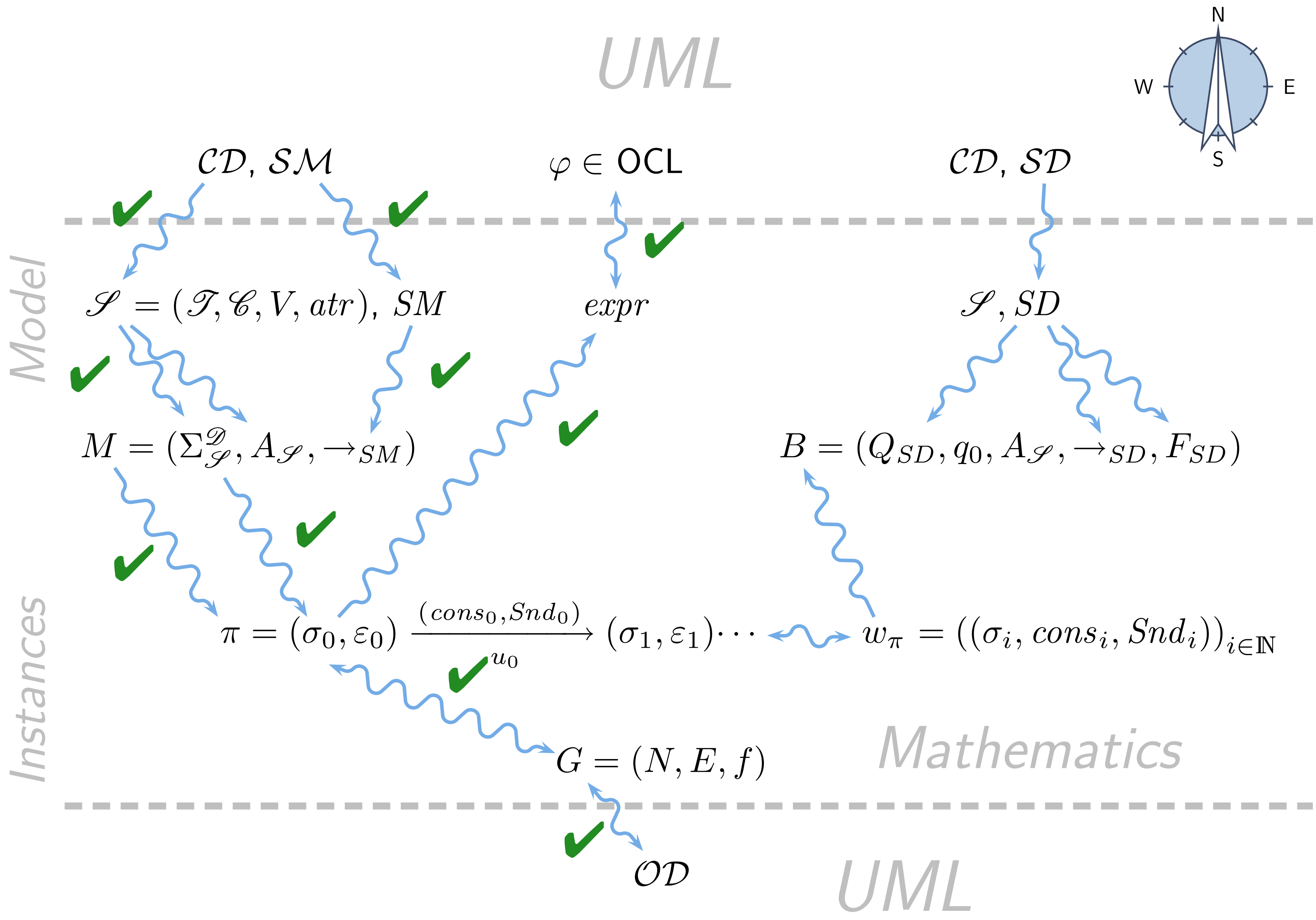
  Exercise: Search the standard for "semantical variation point".

- [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



UML
N
W
E
S
$\mathcal{CD}$, $\mathcal{SM}$
$\varphi \in$ OCL
$\mathcal{CD}$, $\mathcal{SD}$
Model
$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr)$, SM
$expr$
$\mathscr{S}$, SD
$M = (\Sigma_{\mathscr{S}}^{\mathscr{D}}, A_{\mathscr{S}}, \to_{SM})$
$B = (Q_{SD}, q_0, A_{\mathscr{S}}, \to_{SD}, F_{SD})$
Instances
$\pi = (\sigma_0, \varepsilon_0) \xrightarrow[u_0]{(cons_0, Snd_0)} (\sigma_1, \varepsilon_1) \cdots$
$w_\pi = ((\sigma_i, cons_i, Snd_i))_{i \in \mathbb{N}}$
$G = (N, E, f)$
Mathematics
$\mathcal{OD}$
UML

References

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

[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.

[Störrle, 2005] Störrle, H. (2005). *UML 2 für Studenten*. Pearson Studium.