

Software Design, Modelling and Analysis in UML

Lecture 16: Hierarchical State Machines I

2014-01-15

Prof. Dr. Andreas Podolski, 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

2/9

OCL Constraints and Behaviour

- Let $M = (\mathcal{Q}, \mathcal{S}, M, \theta)$ be a UML model.
 - We call M consistent iff, for each OCL constraint $expr \in Inv(\mathcal{Q})$,
 - $\theta \models expr$ for each "reasonable point" (σ, ε) of computations of M .
- Note: we could define $Inv(\mathcal{S}, M)$ similar to $Inv(\mathcal{Q})$.
- Pragmatics: \mathcal{S} Composite \mathcal{C} Inv. $\mathcal{S} = \{s\}$ implies $x \geq 2t$ **also OCL required** **no OCL required**
- In UML-as-blueprint mode, if \mathcal{S}, M doesn't exist yet, then $M = (\mathcal{Q}, \mathcal{N}, \theta)$ is typically asking the developer to provide \mathcal{S}, M such that $M' = (\mathcal{Q}, \mathcal{S}, M, \theta)$ is consistent.
- If the developer makes a mistake, then M' is inconsistent.
- Not common:** if \mathcal{S}, M is given, then constraints are also considered when closing transitions in the RT-Algorithm. In other words, even in presence of mistakes, the \mathcal{S}, M never move to inconsistent configurations.

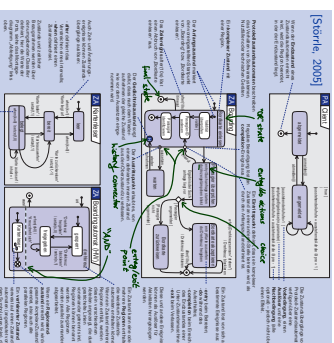
4/9

Hierarchical State Machines

5/9

State Machines and OCL

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



6/9

3/9

UML distinguishes the following kinds of states:

reserved keywords, not visible as signal values	example	pseudo-state	example
simple state		initial (shallow) history	
composite state		deep history	
final state		fork/join	
OR		junction choice	
AND		entry point	
		exit point	
		terminate	
		submachine state	

Representing All Kinds of States

- Until now, state $s_k \in S$ is a finite set of states S .
- Now, state $s_k \in S$ is a function which labels states with their kind: $S \rightarrow \{st, int, fm, stst, chst, fork, join, junic, choic, ent, exit, term\}$.
- Example: $s_1 \in \{comp, ter, sub\}$ is a function which labels states with their kind: $\{s_1, s_2, s_3\} \rightarrow \{st, fm, st\}$.
- Example: $s_1 \in \{comp, ter, sub\}$ is a function which labels states with their kind: $\{s_1, s_2, s_3\} \rightarrow \{st, fm, st\}$.

Representing All Kinds of States

- Until now: (S, s_0, \rightarrow) , $s_0 \in S \rightarrow \subseteq S \times (\mathcal{P}(U \cup \{ \cdot \})) \times Expr \mathcal{C} \times Act \mathcal{C} \times S$.
- From now on: (hierarchical) state machines $(S, kind, region, \rightarrow, \psi, annot)$.
- where $(S, kind, region, \rightarrow, \psi, annot)$ is a finite set of states S .
- $kind : S \rightarrow \{st, int, fm, stst, chst, fork, join, junic, choic, ent, exit, term\}$ is a function which labels states with their kind.
- $region : S \rightarrow \mathcal{P}(S)$ is a function which aggregates the regions of a state.
- \rightarrow is a set of transitions, (ψ transition values) (changed) (new)
- $\psi : \rightarrow \rightarrow \mathcal{P}(S) \times \mathcal{P}(S)$ is an incidence function and (changed) (new)
- $annot : \rightarrow \rightarrow (\mathcal{P}(U \cup \{ \cdot \})) \times Expr \mathcal{C} \times Act \mathcal{C}$ provides an annotation for each transition. (new)
- (s_0) is then redundant — replaced by proper state (!) of kind 'int'.

From UML to Hierarchical State Machines: By Example

	$(S, kind, region, \rightarrow, \psi, annot)$	example	kind	region
simple state (making model writer's final state)	$s \in S$		st	\emptyset
composite state	$s \in S$		fm	\emptyset
OR	$s \in S$		st	$\{s_1, s_2, s_3\}$
AND	$s \in S$		st	$\{s_1, s_2, s_3\}$
submachine state	$s \in S$		st	\emptyset
pseudo-state	$s \in S$		st	\emptyset

From UML to Hierarchical State Machines: By Example

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

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

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

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

Well-Formedness: Regions (follows from diagram)

	$\in S$	kind	region $\subseteq \mathcal{P}(S)$, $S \subseteq S$	child $\subseteq S$
simple state	s	st	\emptyset	\emptyset
final state	s	fm	\emptyset	\emptyset
composite state	s	st	$\{S_1, \dots, S_n\}$, $n \geq 1$	$S_1 \cup \dots \cup S_n$
pseudo-state	s	int, ...	\emptyset	\emptyset
implicit top state	top	st	$\{S\}$	$\{S\}$



child (S_1, S_2, S_3)

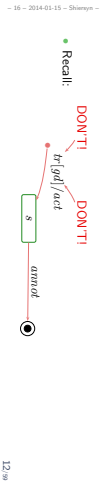
$\{S_1, S_2, S_3\}$

$\{S_1, S_2, S_3, S_4\}$

$\{S_1, S_2, S_3, S_4\}$














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 $region(s) = \{S_1, \dots, S_n\}$, $n \geq 1$, for each $1 \leq i \leq n$,
 - there exists exactly one initial pseudo-state $(s_i, init) \in S_i'$ and at least one transition $t \in \rightarrow$ with s_i' as source,
 - and such transition's target s_j' is in S_i' , and
 - (for simplicity) $kind(s_j') = st$, and $arrow(t) = \langle true, act \rangle$.
- No outgoing transitions to initial states.
 
- No outgoing transitions from final states.
 

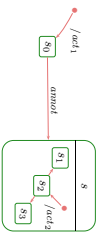


Plan

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

	example	practical state	example
simple state		initial state (initial) history	
final state		deep history	
composite state		junction, choice	
AND		entry point	
OR		exit point	
AND		terminate	
		history/other states	

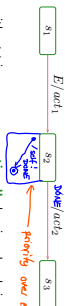
Initial Pseudostate



- Principle:**
- when entering a region **without** a specific destination state, i.e. **from (s, st)**
 - then go to a state which is destination of an **initial transition**,
 - execute the action of the chosen initiation transitions **between** exit and entry actions! (see **14.4.4**).

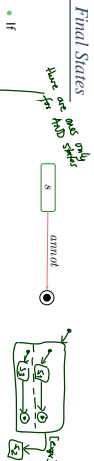
- Special case:** the region of *top*.
- If class C has a state-machine, then "create- C transformer" is the concatenation of
 - the transformer of the "constructor" of C (here not introduced explicitly) and
 - a transformer corresponding to one initiation transition of the top region.

Towards Final States: Completion of States



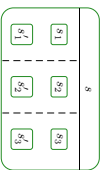
- Transitions without trigger can conceptually be viewed as being sensitive for the "completion event".
- Dispatching (here: E) can then **alternatively** be viewed as
 - fetch event (here: E) from the ether
 - take an enabled transition (here: to s_2)
 - remove event from the ether,
 - after having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

Initial Pseudostates and Final States



- If
 - a step of object u moves u into a final state (s, fn) , and
 - all sibling objects are in a final state,
 then (conceptually) 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 .
- One consequence: u never survives reaching a state (s, fn) with $s \in child(top)$.
 - adjust (2.) and (3.) in the semantics accordingly

Recall: Syntax



translates to

$$\{(top, st), (s, st), (s_1, st), (s_2, st), (s_3, st), (s_1, st), (s_2, st), (s_3, st), (s_1, st), (s_2, st), (s_3, st), \dots\}$$

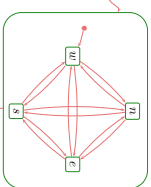
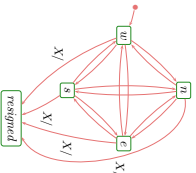
$$\xrightarrow{S.kind} \{(top, st), s, \dots, \{(s_1, st), (s_2, st), (s_3, st)\}, s_1, \dots, \emptyset, s_1', \dots, \emptyset, \dots\}$$

$$\xrightarrow{region} \dots$$

$$\xrightarrow{\psi, \text{atomid}} \dots$$

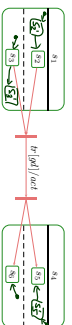
Composite States

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



Syntax: Fork/Join

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



translates to

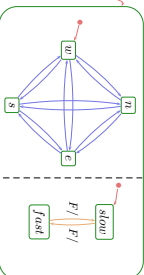
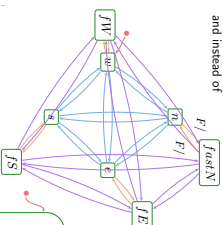
$$(S.kind, region, \{s_1\}, \{s_2\}, \{(s_2, s_1), (s_1, s_2)\}, \{s_1, s_2\})$$

$$\xrightarrow{\psi} (S.kind, region, \{s_1\}, \{s_2\}, \{(s_2, s_1), (s_1, s_2)\}, \{s_1, s_2\})$$

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

Composite States

and instead of



Composite States: Blessing or Curse?

States:

- what are legal state configurations?
- what is the type of the implicit $\&$ attribute?

Transitions:

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

A diagram showing a transition from state s_1 to state s_2 via a transition labeled $F_1/G_1/act_1$.

- what may happen on E_1 ?
- what may happen on E_2 ?
- can E_2 , G kill the object?
- ...

- 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. $\{s_0 \in R \mid kind(s_0) \in \{st, fn\}\} \cap S_1 = 1$.

- 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. $\{s_0 \in R \mid kind(s_0) \in \{st, fn\}\} \cap S_1 = 1$.

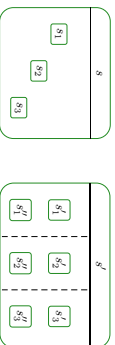


- The type of st is from now on a set of states, i.e. $st : 2^S$
- A set $S_1 \subseteq S$ is called **(legal) state configurations** if and only if
 - $top \in S_1$, and
 - for each state $s \in S_1$, for each non-empty region $\emptyset \neq R \in region(s)$, exactly one (non pseudo-state) child of s (from R) is in S_1 , i.e. $\{s_0 \in R \mid kind(s_0) \in \{st, fn\}\} \cap S_1 = 1$.



- The substrate- (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 $s'' \leq s'$.

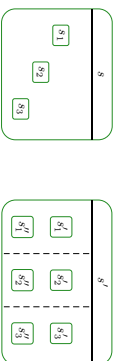
- The substrate- (or child-) relation induces a **partial order on states**:
- $top \leq s$, for all $s \in S$.
 - $s \leq s'$, for all $s' \in child(s)$.
 - transitive, reflexive, antisymmetric.
 - $s' \leq s$ and $s'' \leq s$ implies $s' \leq s''$ or $s'' \leq s'$.



- The **least common ancestor** is the function $lca : 2^S \setminus \{\emptyset\} \rightarrow S$ such that
 - The states in S_1 are (transitive) children of $lca(S_1)$, i.e. $lca(S_1) \leq s$, 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 $lca: 2^S \setminus \{\emptyset\} \rightarrow S$ such that
 - The states in S_1 are (transitive) children of $lca(S_1)$, i.e. $lca(S_1) \leq s_1$ for all $s_1 \in S_1 \subseteq S$.
 - $lca(S_1)$ is minimal, i.e. if $\delta \leq s$ for all $s \in S_1$, then $\delta \leq lca(S_1)$.
- **Note:** $lca(S_1)$ exists for all $S_1 \subseteq S$ (last candidate: top).



26/09

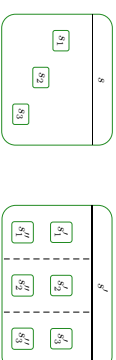
Least Common Ancestor and Ting

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

27/09

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



27/09

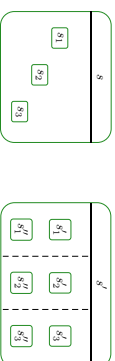
Least Common Ancestor and Ting

- A set of states $S_1 \subseteq S$ is called **consistent**, denoted by $\perp S_1$, if and only if for each $s_1, s_1' \in S_1$,
 - $s_1 \leq s_1'$, or
 - $s_1' \leq s_1$, or
 - $s_1 \perp s_1'$.

28/09

Least Common Ancestor and Ting

- A set of states $S_1 \subseteq S$ is called **consistent**, denoted by $\perp S_1$, if and only if for each $s_1, s_1' \in S_1$,
 - $s_1 \leq s_1'$, or
 - $s_1' \leq s_1$, or
 - $s_1 \perp s_1'$.



28/09

Legal Transitions

- A hierarchical state-machine $(S, \text{final}, \text{region}, \rightarrow, \psi, \text{anno})$ 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 orthogonal, i.e.
 - $\forall s_1, s_1' \in \text{source}(t) (\in \text{target}(t)), s_1 \perp s_1'$.
 - the top state is neither source nor destination, i.e.
 - $top \notin \text{source}(t) \cup \text{target}(t)$.
- Recall: final states are not sources of transitions.

29/09

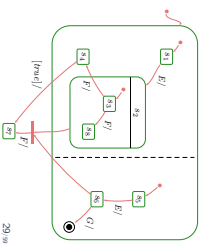
Legal Transitions

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

- (i) source and destination are consistent, i.e. $\perp source(t)$ and $\perp target(t)$,
- (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:



29/99

Embeddness in Hierarchical State-Machines

- The **scope** ("set of possibly affected states") of a transition t is the **least common region** of $source(t) \cup target(t)$.

The Depth of States

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

Embeddness in Hierarchical State-Machines

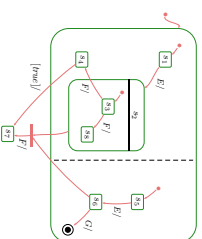
- The **scope** ("set of possibly affected states") of a transition t is the **least common region** of $source(t) \cup target(t)$.

- 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 Depth of States

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

Example:



Embeddness in Hierarchical State-Machines

- The **scope** ("set of possibly affected states") of a transition t is the **least common region** of $source(t) \cup target(t)$.

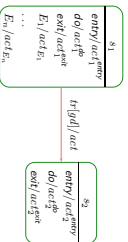
- Two transitions t_1, t_2 are called **consistent** if and only if their scopes are orthogonal (i.e. states in scopes pairwise orthogonal).
- The **priority** of transition t is the depth of its innermost source state, i.e. $priority := \max\{depth(s) \mid s \in source(t)\}$

- 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. $priority(t) := \max\{depth(s) \mid s \in source(t)\}$
- A set of transitions $T \subseteq \dots$ 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. $source(t) \subseteq \sigma(u)(\mathcal{A}) \subseteq S$.

- Let T be a set of transitions enabled in u .
- Then $(\alpha, \beta) \xrightarrow{Trans, S, \text{Incl}} (\sigma', \sigma')$ if
 - $\sigma'(u)(s)$ consists of the target states of t , i.e. for simple states the simple states themselves, for composite states the initial states,
 - $\sigma', \sigma', \text{cons}$, and S, Incl 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.

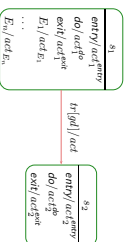
Entry/Do/Exit Actions

- 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 **trigger transitions** (default: empty), $E_1, \dots, E_n \in \mathcal{E}, \text{'entry'}$, 'do' , 'exit' are reserved names!



Entry/Do/Exit Actions

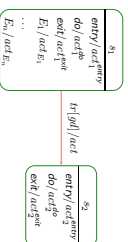
- In general, with each state $s \in S$ there is associated
 - an **entry**, a **do**, and an **exit** action (default: skip)
 - a possibly empty set of trigger/action pairs called **trigger transitions** (default: empty), $E_1, \dots, E_n \in \mathcal{E}, \text{'entry'}$, 'do' , 'exit' are reserved names!



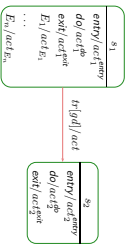
- Recall: each action's supposed to have a transformer. Here: $Kact1p, Kact1e, \dots, Kact2p \circ Kact \circ Kact1e$
- Taking the transition above then amounts to applying $Kact$ instead of only $Kact$
- \rightsquigarrow adjust (2), (3) accordingly

Entry/Do/Exit Actions, Internal Transitions

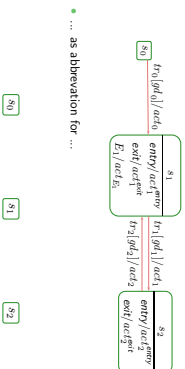
- For **internal transitions**, taking the one for E_1 , for instance, still amounts to taking **only** $KactE_1$.
- Intuition: The state is neither left nor entered, so: no exit, no entry.
- \rightsquigarrow adjust (2) accordingly.
- Note: internal transitions also start a run-to-completion step.



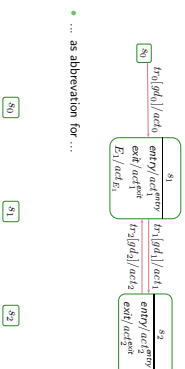
Internal Transitions



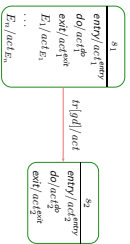
- For **internal transitions**, taking the one for E_1 , for instance, still amounts to taking **only** act_1 .
- **Intuition:** The state is neither left nor entered, so: no exit, no entry, --> adjust (2.) accordingly.
- Note: internal transitions also start a run-to-completion step.
- Note: the standard 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!



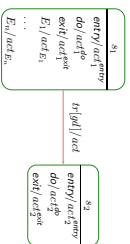
- ... as abbreviation for ...



- ... as abbreviation for ...



- **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.

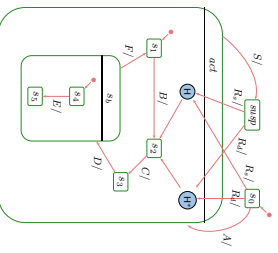


- **Intuition:** after entering a state, start its do-action.
- If the do-action terminates,
- then the state is considered **completed**,
- otherwise,
- if the state is left before termination, the do-action is stopped.

- 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...
- R1?
 - R2?
 - R3?
 - A,B,C,S,R,2
 - A,B,S,R,2?
 - A,B,C,D,E,R,2
 - A,B,C,D,R,2?

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. S : s



- 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.



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

Entry and Exit Point, Submachine State, Terminate

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



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 “**rolled**” for readability. (but: this can also hinder readability)
- Can even be taken from a different state machine for re-use. S : x
- **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.

X

41/00

– 16 – 2014-01-15 – Slide –

Deferred Events in State-Machines


– 16 – 2014-01-15 – main –

42/00

Deferred Events: Idea

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

The idea is as follows:

- Consider the following state machine:
- 
- ```
graph LR; s1[s1] -- E/ --> s2[s2]; s2[s2] -- F/ --> s3[s3];
```
- 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.


43/00

– 16 – 2014-01-15 – Slide –

### Deferred Events: Idea

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

The idea is as follows:

- Consider the following state machine:
- 
- ```
graph LR; s1[s1] -- E/ --> s2[s2]; s2[s2] -- F/ --> s3[s3];
```
- 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.


43/00

– 16 – 2014-01-15 – Slide –

Deferred Events: Idea

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

The idea is as follows:

- Consider the following state machine:
- 
- ```
graph LR; s1[s1] -- E/ --> s2[s2]; s2[s2] -- F/ --> s3[s3];
```
- Assume we’re stable in  $s_1$ , and  $F$  is ready in the ether.
  - **In the framework of the course,  $F$  is discarded.**


43/00

– 16 – 2014-01-15 – Slide –

### Deferred Events: Idea

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

The idea is as follows:

- Consider the following state machine:
- 
- ```
graph LR; s1[s1] -- E/ --> s2[s2]; s2[s2] -- F/ --> s3[s3];
```
- 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. (– OMC’s choice)

43/00

– 16 – 2014-01-15 – Slide –

- **Syntactically,**
- Each state has (in addition to the name) a set of deferred events.
- **Default:** the empty set.

44/99

- **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_i is dispatched,
 - and there is no transition enabled to consume E_i ,
 - and E_i is in the deferred set of the current state configuration,
 - then stuff E_i into some “deferred events space” of the object. (e.g. into the ether (= extend \Rightarrow) or into the local state of the object (= extend \Rightarrow))
 - and turn attention to the next event.

44/99

- **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_i is dispatched,
 - and there is no transition enabled to consume E_i ,
 - and E_i is in the deferred set of the current state configuration,
 - then stuff E_i into some “deferred events space” of the object. (e.g. into the ether (= extend \Rightarrow) or into the local state of the object (= extend \Rightarrow))
 - 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 Schenborn, 2007], e.g.: claim to provide semantics for the complete Hierarchical State Machine language, including deferred events.

44/99

Active and Passive Objects [Harel and Gery, 1997]

45/99

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.

46/99

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 exercise 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.

46/99

[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.

[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.

[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	✓	✓

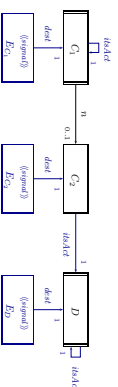
- So why don't we understand passive/reactive?
- Assume passive objects u_1 and u_2 , and active object u_3 , and that there are events in the ether for all three.
- Which of them (can) start a run-to-completion step...?
- Do run-to-completion steps still interleave...?

- So why don't we understand passive/reactive?
- Assume passive objects u_1 and u_2 , and active object u_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...?

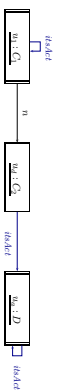
Reasonable Approaches:

- **Avoid** — for instance, by
- 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.

- Firstly, establish that each object u knows, via (implicit) link $u.kAct$, the **active object** $u.act$ which is responsible for dispatching events to u .
- If u is an instance of an active class, then $u.u = u$.



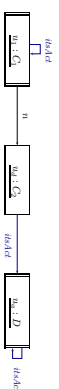
- Firstly, establish that each object u knows, via (implicit) link $u.kdA$, the active object u_{act} , which is responsible for dispatching events to u .
- If u is an instance of an active class, then $u_{act} = u$.



Sending an event:

- Establish that of each signal we have a version E_C with an association $kdA : C_{A1}, C \in \mathcal{C}$.
- Then nE in $v_1 : C_1$ becomes:
- Create an instance u_2 of E_C and set u_2 's kdA to $u_1 := \sigma(v_1)(v)$
- Send to $u_2 := \sigma(\sigma(v_1)(v))(kdA)$, i.e., $\sigma \in \sigma(v_1, u_2)$.

- Firstly, establish that each object u knows, via (implicit) link $u.kdA$, the active object u_{act} , which is responsible for dispatching events to u .
- If u is an instance of an active class, then $u_{act} = u$.



Sending an event:

- Establish that of each signal we have a version E_C with an association $kdA : C_{A1}, C \in \mathcal{C}$.
- Then nE in $v_1 : C_1$ becomes:
- Create an instance u_2 of E_C and set u_2 's kdA to $u_1 := \sigma(v_1)(v)$
- Send to $u_2 := \sigma(\sigma(v_1)(v))(kdA)$, i.e., $\sigma \in \sigma(v_1, u_2)$.

Dispatching an event:

- Observation: the ether only has events for active objects.
- Say u_2 is ready in the ether for u_2 .
- Then u_2 asks $\sigma(v_2)(kdA) = u_2$ to process u_2 's pending events in completion of corresponding RTC.
- Send to $u_2 := \sigma(\sigma(v_2)(v))(kdA)$, u_2 may in particular discard event.
- i.e., $\sigma \in \sigma(v_2, u_2)$.

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.

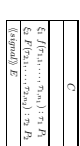
And What About Methods?

- 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.



Behavioural Features

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



- The class **state-machine** ("triggered operator")

Behavioural Features

C
$f_1(f_1, \dots, f_n, a_1) \text{ or } R$
$f_2(f_2, \dots, f_n, a_2) \text{ or } R$
\dots
$f_m(f_m, \dots, f_n, a_m) \text{ or } R$
<i>(optional) E</i>

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
$f_1(f_1, \dots, f_n, a_1) \text{ or } R$
$f_2(f_2, \dots, f_n, a_2) \text{ or } R$
\dots
$f_m(f_m, \dots, f_n, a_m) \text{ or } R$
<i>(optional) E</i>

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 m_2 parameters for a stable instance of C creates an auxiliary event F and dispatches it (bypassing the ether).
- Transition actions may fill in the return value.
- On completion of the RTC step, the call returns.
- For a non-stable instance, the caller blocks until stability is reached again.

Behavioural Features: Visibility and Properties

C
$f_1(f_1, \dots, f_n, a_1) \text{ or } R$
$f_2(f_2, \dots, f_n, a_2) \text{ or } R$
\dots
$f_m(f_m, \dots, f_n, a_m) \text{ or } R$
<i>(optional) E</i>

- **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
$f_1(f_1, \dots, f_n, a_1) \text{ or } R$
$f_2(f_2, \dots, f_n, a_2) \text{ or } R$
\dots
$f_m(f_m, \dots, f_n, a_m) \text{ or } R$
<i>(optional) E</i>

- **Visibility**
 - Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods.
- **Useful properties**:
 - **concurrency** — is thread safe
 - **concurrent** — some mechanism ensures/should ensure mutual exclusion
 - **guarded** — is not thread safe, users have to ensure mutual exclusion
 - **sequential** — doesn't modify the state space (thus thread safe)
 - **isQuery** — doesn't modify the notion of steps untouched, we construct our semantics around state machines.
- 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 substrate; 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".

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)

Pessimistic view: They are legion...

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

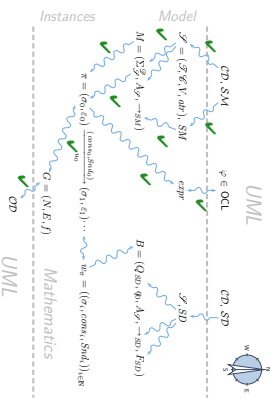
Exercise: Search the standard for “semantical variation point”.

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

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

You are here.

Course Map



References

[Crane and Dingel, 2007] Crane, M. L. and Dingel, J. (2007). UML vs. classical vs. rhapsody statecharts: not all models are created equal. *Software and Systems Modeling*, 6(4):415–435.

[Damm et al., 2003] Damm, W., Josko, B., Vainitsen, A., and Pruvil, A. (2003). A formal semantics for a UML kernel language 1.2. IST/33822/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., Havelort, 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., Gole-Rhodes, M., Reif, W., Schneider, 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.

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