Software Design, Modelling and Analysis in UML

Lecture 14: Core State Machines IV

2016-01-12

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

- Transitions by Rule (i) to (v).

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - What is a step / run-to-completion step?
  - What is divergence in the context of UML models?
  - How to define what happens at "system / model startup"?
  - What are roles of OCL contraints in behavioural models?
  - Is this UML model consistent with that OCL constraint?
  - What do the actions create / destroy do? What are the options and our choices (why)?

- **Content:**
  - Step / RTC-Step revisited, Divergence
  - Initial states
  - Missing pieces: create / destroy transformer
  - A closer look onto code generation
  - Maybe: hierarchical state machines

Step and Run-to-Completion

Notions of Steps: The Step

Note: we call one evolution

$$(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon')$$

a **step**.

in case of Rules (i) + (iii)

Thus in our setting, **a step directly corresponds** to

> **one object** (namely u) taking **a single transition** between regular states.

(We will extend the concept of "single transition" for hierarchical state machines.)

That is: We're going for an interleaving semantics without true parallelism.

Notions of Steps: The Run-to-Completion Step

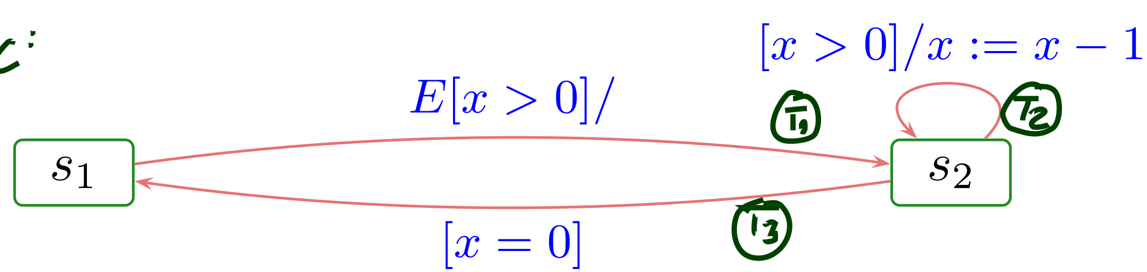
What is a **run-to-completion** step...?

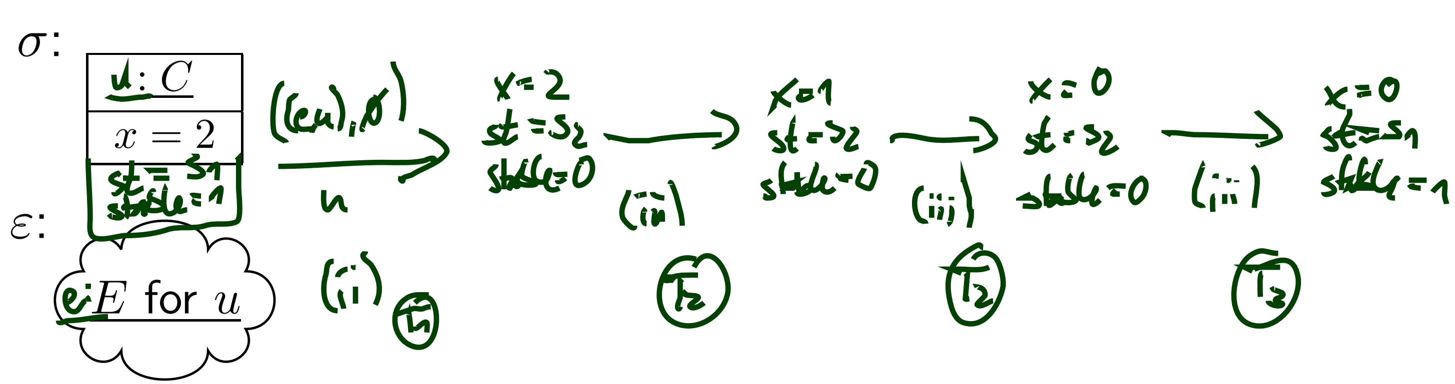
- **Intuition**: a **maximal** sequence of steps of one object, where the first step is a **dispatch** step, all later steps are **continue** steps, and the last step establishes stability (or object disappears).

Note: while one step corresponds to one transition in the state machine, a run-to-completion step is in general **not syntacically definable**:

one transition may be taken multiple times during an RTC-step.

Example: SM_C:





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

Proposal: Let

$$(\sigma_0, \varepsilon_0) \xrightarrow[u_0]{(cons_0, Snd_0)} \dots \xrightarrow[u_{n-1}]{(cons_{n-1}, Snd_{n-1})} (\sigma_n, \varepsilon_n), \quad n > 0,$$

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

- $(cons_0, Snd_0)$ indicates dispatching to $u := u_0$ (by Rule (ii)~~,~~ or (i))
  i.e. $cons = \{u_E\}$, $u_E \in \mathrm{dom}(\sigma_0) \cap \mathscr{D}(\mathscr{E})$,

- if u becomes stable or disappears, then in the last step, i.e.

$$\forall\, i > 0 \bullet (\sigma_i(u)(stable) = 1 \lor u \notin \mathrm{dom}(\sigma_i)) \implies i = n$$

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

$$(\sigma_0(u) =) \quad \sigma_{k_1}(u), \sigma_{k_2}(u) \dots, \sigma_{k_N}(u), \sigma_n(u)$$

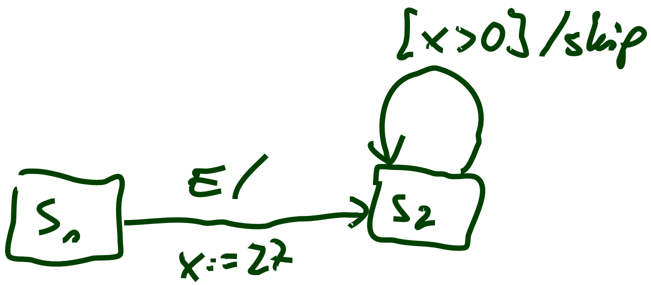
a (!) **run-to-completion step** of u (from (local) configuration $\sigma_0(u)$ to $\sigma_n(u)$).

Divergence

We say, object u **can diverge** on reception $cons_0$ from (local) configuration $\sigma_0(u)$ if and only if there is an **infinite**, consecutive sequence

$$(\sigma_0, \varepsilon_0) \xrightarrow[u_0]{(cons_0, Snd_0)} (\sigma_1, \varepsilon_1) \xrightarrow[u_1]{(cons_1, Snd_1)} \ldots$$

where $u_i = u$ for infinitely many $i \in \mathbb{N}_0$ and $\sigma_i(u)(stable) = 0$, $i > 0$, i.e. u does not become stable again.



Run-to-Completion Step: Discussion.

Our definition of RTC-step takes a **global** and **non-compositional** view, that is:

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

  Our semantics and notion of RTC-step doesn't have this (often desired) property.

Can we give (syntactical) criteria such that any (global) run-to-completion step is an interleaving of local ones?

Maybe: **Strict interfaces**. (*Proof left as exercise...*)

- **(A)**: Refer to private features only via "self".

  (Recall that other objects of the same class can modify private attributes.)
- **(B)**: Let objects only communicate by events, i.e.
  don't let them modify each other's local state via links **at all**.

Putting It All Together

Initial States

Recall: a labelled transition system is $(S, A, \to, S_0)$. We **have**

- S: system configurations (σ, ε)
- $\to$: labelled transition relation $(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon')$.

Wanted: initial states S_0.

Proposal:

Require a (finite) set of **object diagrams** $\mathcal{OD}$ as part of a UML model

$$(\mathscr{CD}, \mathscr{SM}, \mathscr{OD}).$$

And set

$$S_0 = \{(\sigma, \varepsilon) \mid \sigma \in G^{-1}(\mathcal{OD}), \quad \mathcal{OD} \in \mathscr{OD}, \quad \varepsilon \text{ empty}\}.$$

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

Semantics of UML Model (So Far)

The **semantics** of the **UML model**

$$\mathcal{M} = (\mathscr{CD}, \mathscr{SM}, \mathscr{OD})$$

where

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

is the **transition system** $(S, A, \to, S_0)$ constructed on the previous slide(s).

The **computations of** $\mathcal{M}$ are the computations of $(S, A, \to, S_0)$.

OCL Constraints and Behaviour

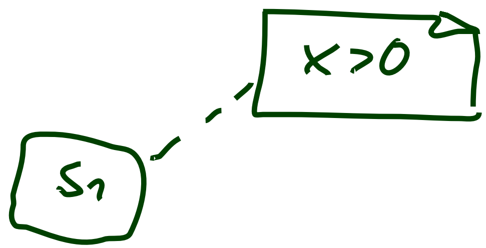
- Let $\mathcal{M} = (\mathscr{CD}, \mathscr{SM}, \mathscr{OD})$ be a UML model.
- We call $\mathcal{M}$ **consistent** iff, for each OCL constraint $expr \in \mathit{Inv}(\mathscr{CD}) \cup \mathit{Inv}(\mathscr{SM})$

$$\sigma \models expr \text{ for each “reasonable point” } (\sigma, \varepsilon) \text{ of computations of } \mathcal{M}.$$

(Cf. exercises and tutorial for discussion of “reasonable point”.)

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

SM_C:



abbrev.

context C inv : st = s1
implies
x > 0

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 $expr \in \mathit{Inv}(\mathscr{CD})$,

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

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

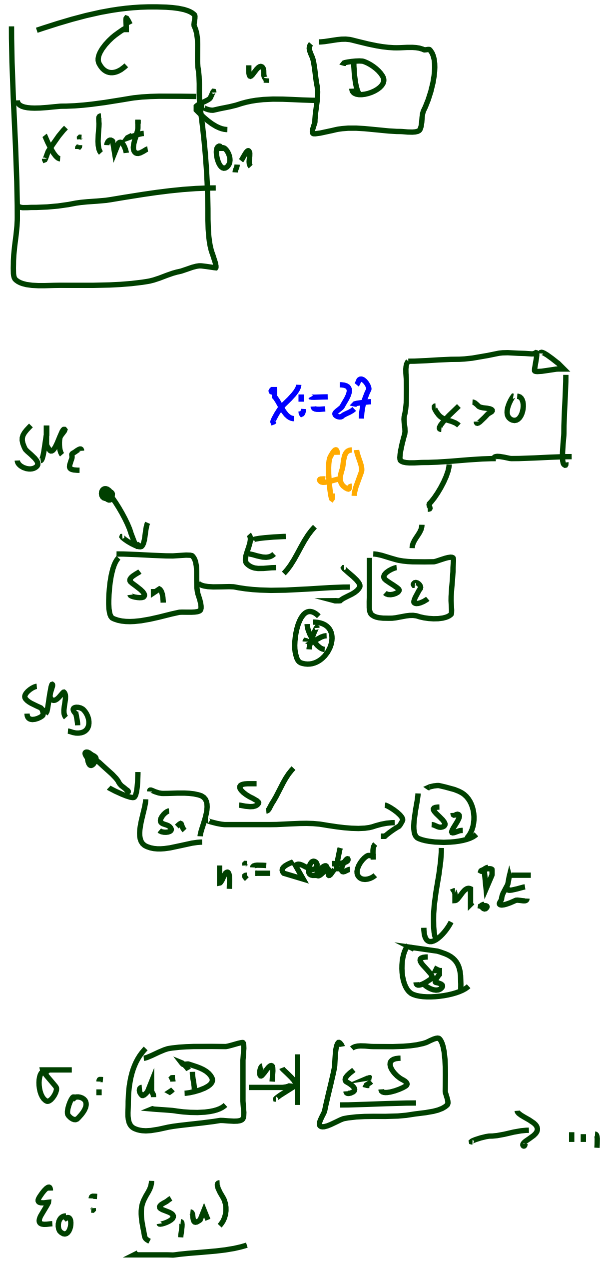
Note: we could define $\mathit{Inv}(\mathscr{SM})$ similar to $\mathit{Inv}(\mathscr{CD})$.

Pragmatics:

- 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.

- (and not completely uncommon) **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.



1. model is "broken"

2. model behaviour considers constraints, (*) would not be taken if $x \leq 0$ after creation

Last Missing Piece: Create and Destroy Transformer

Transformer: Create

abstract syntax $\texttt{create}(C, expr, v)$

concrete syntax $expr.v := \texttt{new}\ C$

intuitive semantics

Create an object of class C and assign it to attribute v of the object denoted by expression $expr$.

well-typedness

$$expr : T_D,\ v \in atr(D),\quad v : C_{0,1},$$
$$atr(C) = \{\langle v_1 : T_1, expr_i^0 \rangle \mid 1 \leq i \leq n\}$$

semantics

$$\ldots$$

observables

$$\ldots$$

(error) conditions

$I[\![expr]\!](\sigma, \beta)$ not defined.

instead

x := (new C).y + (new D).z;

write

temp₁ := new C;
temp₂ := new D;
x := temp₁.y + temp₂.z;

Transformer: Create

abstract syntax	**concrete syntax**
$\texttt{create}(C, expr, v)$	

intuitive semantics

Create an object of class C and assign it to attribute v of the object denoted by expression $expr$.

well-typedness

$$expr : T_D,\ v \in atr(D),$$
$$atr(C) = \{\langle v_1 : T_1, expr_i^0\rangle \mid 1 \le i \le n\}$$

semantics

...

observables

...

(error) conditions

$I[\![expr]\!](\sigma, \beta)$ not defined.

- We use an "and assign"-action for simplicity — it doesn't add or remove expressive power, but moving creation to the expression language raises all kinds of other problems since then expressions would need to modify the system state.
- Also for simplicity: no parameters to construction ($\sim$ parameters of constructor). Adding them is straightforward (but somewhat tedious).

How To Choose New Identities?

- **Re-use**: choose any identity that is not alive **now**, i.e. not in $\mathrm{dom}(\sigma)$.
  - Doesn't depend on history.
  - May "undangle" dangling references – may happen on some platforms.

- **Fresh**: choose any identity that has not been alive **ever**, i.e. not in $\mathrm{dom}(\sigma)$ and any predecessor in current run.
  - Depends on history.
  - Dangling references remain dangling – could mask "dirty" effects of platform.

Transformer: Create

abstract syntax **concrete syntax**

$\texttt{create}(C, expr, v)$

intuitive semantics

Create an object of class C and assign it to attribute v of the object denoted by expression $expr$.

well-typedness

$$expr : T_D,\ v \in atr(D),$$
$$atr(C) = \{\langle v_1 : T_1, expr_i^0\rangle \mid 1 \leq i \leq n\}$$

semantics

$$((\sigma, \varepsilon), (\sigma', \varepsilon')) \in t_{\texttt{create}(C,expr,v)}[u_x]$$

iff

$$\underbrace{\sigma' = \sigma[u_0 \mapsto \sigma(u_0)[v \mapsto \{u\}]]}_{\text{similar to update}} \cup \underbrace{\{u \mapsto \{v_i \mapsto d_i \mid 1 \leq i \leq n\}\}}_{\text{similar to send}},$$

$\varepsilon' = [u](\varepsilon)$; $u \in \mathscr{D}(C)$ fresh, i.e. $u \notin \mathrm{dom}(\sigma)$;

$u_0 = I[\![expr]\!](\sigma, u_x)$; $d_i = I[\![expr_i^0]\!](\sigma, \emptyset)$ if $expr_i^0 \neq \emptyset$ and arbitrary value from $\mathscr{D}(T_i)$ otherwise.

clean either (do it on slide 20!)

observables

$$Obs_{\texttt{create}}[u_x] = \{(*, u)\}$$

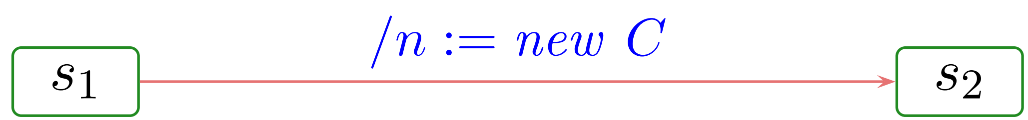
creation

(error) conditions

$I[\![expr]\!](\sigma, u_x)$ not defined.

Create Transformer Example

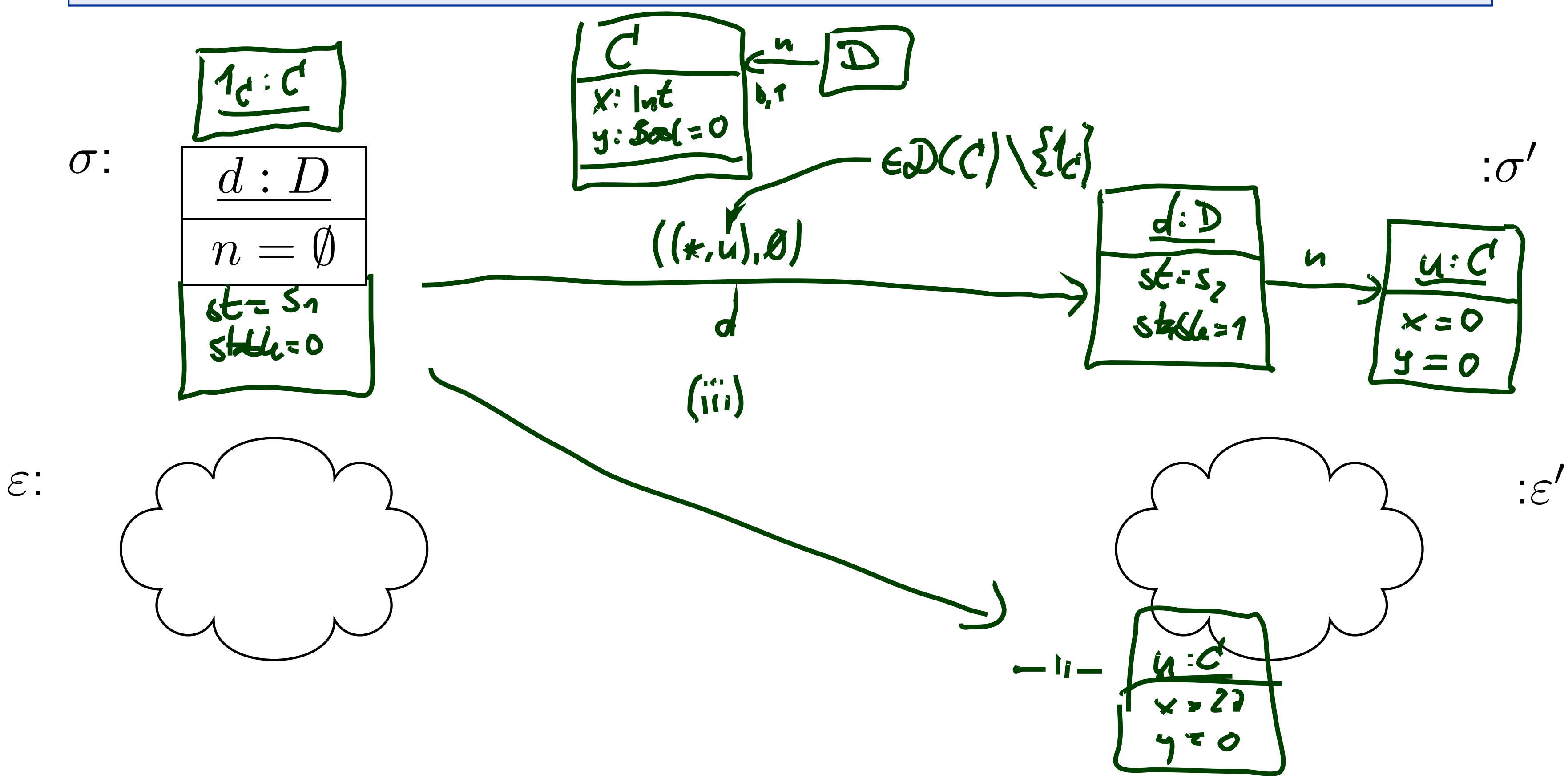
$\mathcal{SM}_{D}$:



$\texttt{create}(C, expr, v)$

$t_{\texttt{create}(C,expr,v)}[u_x](\sigma, \varepsilon) = \dots$

$\mathscr{D}(Int) = \mathbb{Z}$



Transformer: Destroy

abstract syntax	**concrete syntax**
$\texttt{destroy}(expr)$	
intuitive semantics	
Destroy the object denoted by expression $expr$.	
well-typedness	
$expr : T_C,\ C \in \mathscr{C}$	
semantics	
$\ldots$	
observables	
$Obs_{\texttt{destroy}}[u_x] = \{(u_x, \perp, (+, \emptyset), u)\}$	
(error) conditions	
$I[\![expr]\!](\sigma, \beta)$ not defined.	

What to Do With the Remaining Objects?

Assume object u_0 is destroyed...

- object u_1 may still refer to it via association r:
  - allow dangling references?
  - or remove u_0 from $\sigma(u_1)(r)$?
- object u_0 may have been the last one linking to object u_2:
  - leave u_2 alone?
  - or remove u_2 also? (garbage collection)
- Plus: (temporal extensions of) OCL may have dangling references.

Our choice: Dangling references and no garbage collection!

This is in line with "expect the worst", because there are target platforms which don't provide garbage collection — and models shall (in general) be correct without assumptions on target platform.

But: the more "dirty" effects we see in the model, the more expensive it often is to analyse. Valid proposal for simple analysis: monotone frame semantics, no destruction at all.

Transformer: Destroy

abstract syntax **concrete syntax**

$\texttt{destroy}(expr)$

intuitive semantics

Destroy the object denoted by expression $expr$.

well-typedness

$$expr : T_C,\ C \in \mathscr{C}$$

semantics

function restriction

$$t_{\texttt{destroy}(expr)}[u_x](\sigma, \varepsilon) = \{(\sigma', \varepsilon')\},$$

$\varepsilon' = [u](\varepsilon)$

where $\sigma' = \sigma|_{\mathrm{dom}(\sigma)\setminus\{u\}}$ with $u = I[\![expr]\!](\sigma, u_x)$.

observables

$$Obs_{\texttt{destroy}(expr)}[u_x] = \{(+, u)\}$$

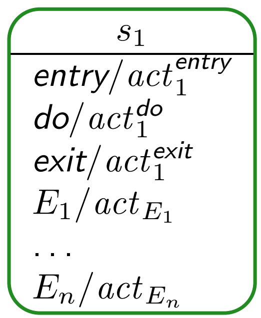
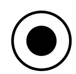
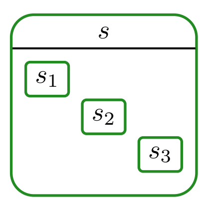
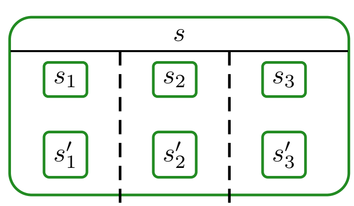
(error) conditions

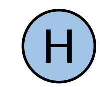
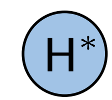
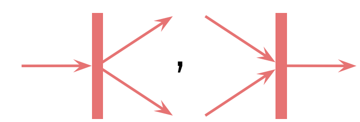
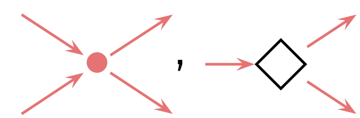
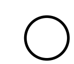
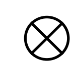
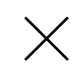
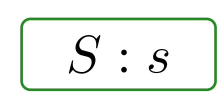
$I[\![expr]\!](\sigma, u_x)$ not defined.

Hierarchical State-Machines

The Full Story

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

	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	

References

References

OMG (2011a). Unified modeling language: Infrastructure, version 2.4.1. Technical Report formal/2011-08-05.

OMG (2011b). Unified modeling language: Superstructure, version 2.4.1. Technical Report formal/2011-08-06.