Software Design, Modelling and Analysis in UML

Lecture 15: Hierarchical State Machines I

or: Core State Machines V

2015-01-08

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

- RTC-Rules: Discard, Dispatch, Commence, ~~Item~~ Step, RTC

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 is: initial state.
  - 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:**
  - Transformer: Create and Destroy, Divergence
  - Putting It All Together
  - Hierarchical State Machines Syntax

Missing Transformers: Create and Destroy

Transformer: Create

abstract syntax	**concrete syntax**
$\texttt{create}(C, expr, v)$	expr.v := 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	
$expr : \tau_D,\ v \in atr(D)$, $atr(C) = \{\langle v_i : \tau_i, expr_i^0\rangle \mid 1 \leq i \leq n\}$	
semantics	
...	
observables	
...	
(error) conditions	
$I[\![expr_i^0]\!](\sigma, u_x)$ not defined for some i.	

SO NOT: x := (new C).x + (new C).y;

IF NEEDED: tmp₁ := new C;
tmp₂ := new C;
x := tmp₁.x + tmp₂.y

SO NOT: new Circle(0.5);

IF NEEDED: tmp := new Circle;
tmp.init(0.5);

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

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

semantics

...

observables

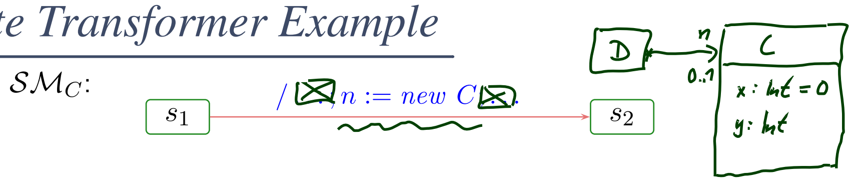
...

(error) conditions

$I[\![expr_i^0]\!](\sigma, u_x)$ not defined for some i.

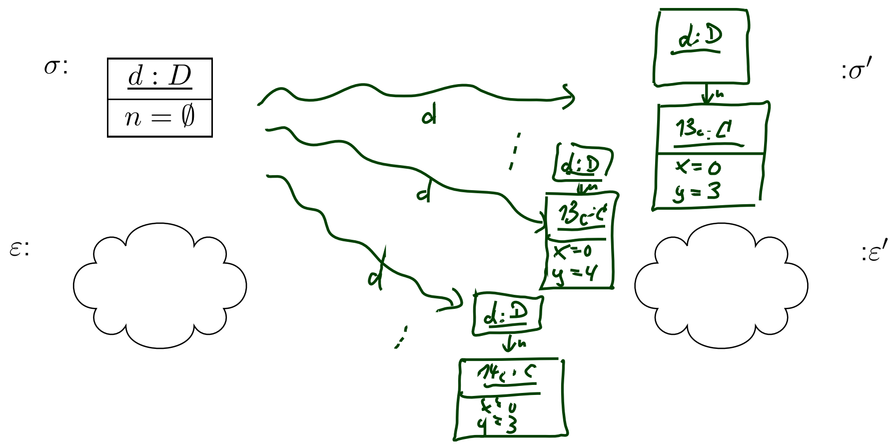
- We use an "and assign"-action for simplicity — it doesn't add or remove expressive power, but moving creation to the expression language raises all kinds of other problems such as order of evaluation (and thus creation).
- Also for simplicity: no parameters to construction ($\sim$ parameters of constructor).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

Create Transformer Example

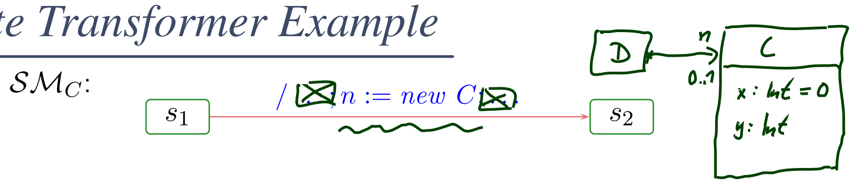


$\texttt{create}(C, expr, v)$

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

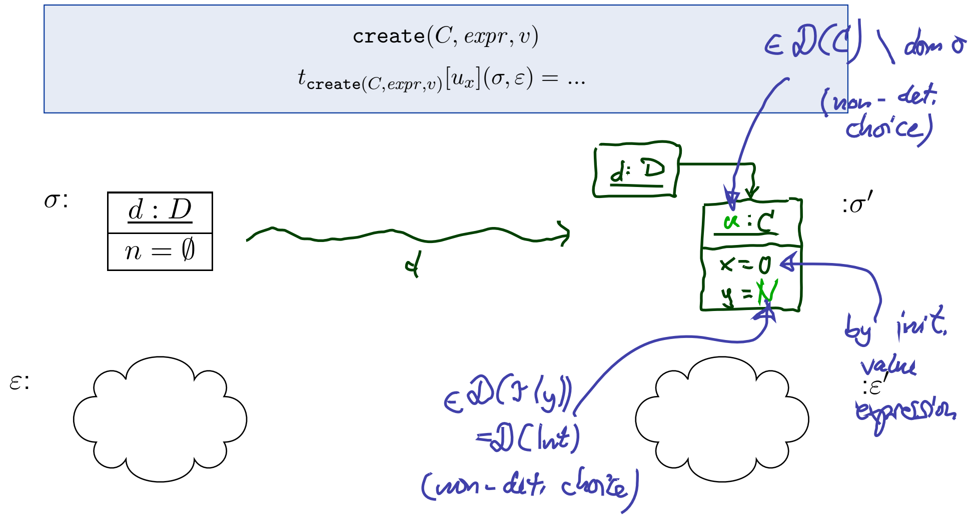


Create Transformer Example



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

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



How To Choose New Identities?

- **Re-use**: choose any identity that is not alive **now**, i.e. not in $\mathrm{dom}(\sigma)$. X OUR CHOICE
  - 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**

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

intuitive semantics

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

well-typedness

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

semantics

similar to update

id of new object

new object similar to send

$$((\sigma, \varepsilon), (\sigma', \varepsilon')) \in t$$

iff

$$\sigma' = \sigma[u_0 \mapsto \sigma(u_0)[v \mapsto u]] \cup \{u \mapsto \{v_i \mapsto d_i \mid 1 \le i \le n\}\},$$

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

$u_0 = I[\![expr]\!](\sigma, u_x)$; $d_i = I[\![expr_i^0]\!](\sigma, u_x)$ if $expr_i^0 \neq$ ‘$\star$’ and $d_i \in \mathscr{D}(\tau_i)$ otherwise (non-determinism).

object, whose v attr. points to new object

observables

$$Obs_{\mathtt{create}}[u_x] = \{(u_x, \bot, (*, \emptyset), u)\}$$

creation...

... of u

(error) conditions

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

clean ether

Transformer: Destroy

abstract syntax **concrete syntax**

$\mathtt{destroy}(expr)$ delete expr;

intuitive semantics

Destroy the object denoted by expression $expr$.

well-typedness

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

semantics

...

observables

destruction...

$$Obs_{\mathtt{destroy}}[u_x] = \{(u_x, \bot, (+, \emptyset), u)\}$$

... of u

(error) conditions

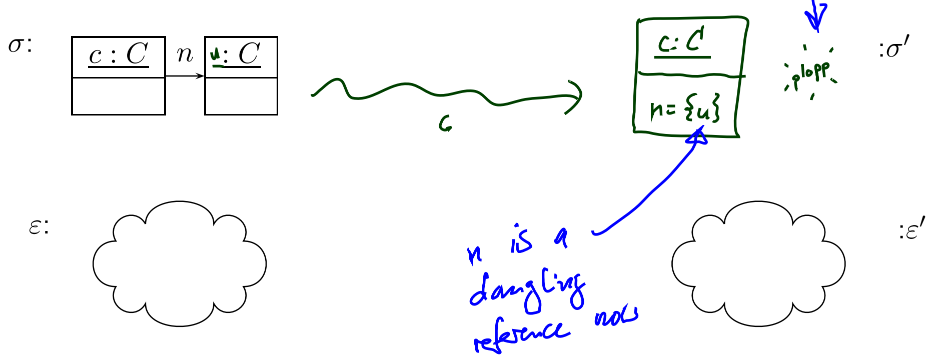
$I[\![expr]\!](\sigma, u_x)$ not defined.

Destroy Transformer Example

destroy($expr$)

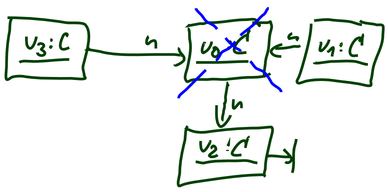
$t_{\texttt{destroy}(expr)}[u_x](\sigma, \varepsilon) = \ldots$

u is gone



What to Do With the Remaining Objects?

Assume object u_0 is destroyed. by u_3 ..



- object u_1 may still refer to it via association n:
  - allow dangling references?
  - or remove u_0 from $\sigma(u_1)(n)$?
- object u_0 may have been the last one linking to object u_2:
  - leave u_2 alone?
  - or remove u_2 also?
- 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**

$\mathtt{destroy}(expr)$

intuitive semantics

Destroy the object denoted by expression $expr$.

well-typedness

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

semantics

$$t[u_x](\sigma, \varepsilon) = (\sigma', \varepsilon)$$

function restriction

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

observables

$$Obs_{\mathtt{destroy}}[u_x] = \{(u_x, \perp, (+, \emptyset), u)\}$$

(error) conditions

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

Step and Run-to-completion Step

Notions of Steps: The Step

Note: we call one evolution $(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon')$ a **step**.

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

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

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

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

Remark: With only methods (later), the notion of step is not so clear.
For example, consider

- c_1 calls `f()` at c_2, which calls `g()` at c_1 which in turn calls `h()` for c_2.

- Is the completion of `h()` a step?
- Or the completion of `f()`?
- Or doesn't it play a role?

It does play a role, because **constraints**/**invariants** are typically (= by convention) assumed to be evaluated at step boundaries, and sometimes the convention is meant to admit (temporary) violation in between steps.

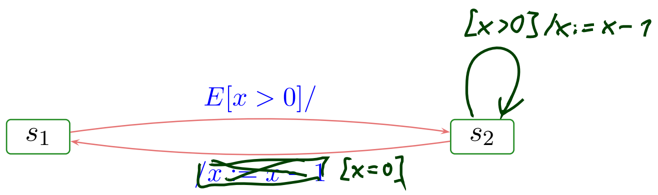
Notions of Steps: The Run-to-Completion Step

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

- **Intuition**: a maximal sequence of steps, where the first step is a **dispatch** step and all later steps are **commence** steps.
- **Note**: one step corresponds to one transition in the state machine.

  A run-to-completion step is in general not syntacically definable — one transition may be taken multiple times during an RTC-step.

Example:



σ: $\underline{:C}$, $x = 2$

ε: E for u

Notions of Steps: The RTC 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

- object u is alive in σ_0,
- $u_0 = u$ and $(cons_0, Snd_0)$ indicates dispatching to u, i.e. $cons = \{(u, \vec{v} \mapsto \vec{d})\}$,
- there are no receptions by u in between, i.e.

$$cons_i \cap \{u\} \times Evs(\mathscr{E}, \mathscr{D}) = \emptyset, i > 1,$$

- $u_{n-1} = u$ and u is stable only in σ_0 and σ_n, i.e.

$$\sigma_0(u)(stable) = \sigma_n(u)(stable) = 1 \text{ and } \sigma_i(u)(stable) = 0 \text{ for } 0 < i < n,$$

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

$$(\sigma_0(u) =) \quad \sigma_{k_1}(u), \sigma_{k_2}(u) \dots, \sigma_{k_N}(u) \quad (= \sigma_{n-1}(u))$$

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

Divergence

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

$$(\sigma_0, \varepsilon_0) \xrightarrow{(cons_0, Snd_0)} (\sigma_1, \varepsilon_1) \xrightarrow{(cons_1, Snd_1)} \dots$$

such that u doesn't become stable again.

- **Note**: disappearance of object not considered in the definitions.
  By the current definitions, it's neither divergence nor an RTC-step.

Run-to-Completion Step: Discussion.

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

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

References

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

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

[Fecher and Schönborn, 2007] Fecher, H. and Schönborn, J. (2007). UML 2.0 state machines: Complete formal semantics via core state machines. In Brim, L., Haverkort, B. R., Leucker, M., and van de Pol, J., editors, *FMICS/PDMC*, volume 4346 of *LNCS*, pages 244–260. Springer.

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

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

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