Software Design, Modelling and Analysis in UML

Lecture 12: Core State Machines III

2011-12-11

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

Albert-Ludwigs-Universität Freiburg, Germany

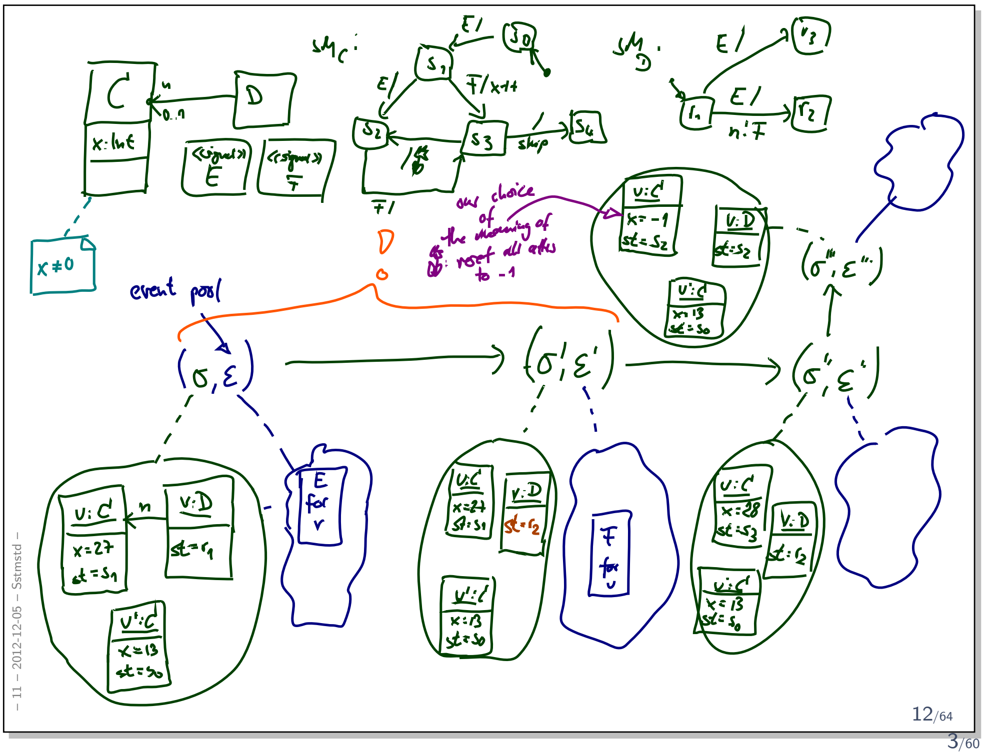
Contents & Goals

Last Lecture:

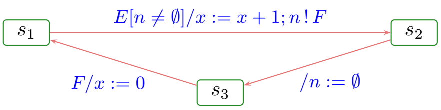
- The basic causality model
- Ether

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: Signal, Event, Ether, Transformer, Step, RTC.
- **Content:**
  - System Configuration, Transformer
  - Examples for transformer
  - Run-to-completion Step
  - Putting It All Together

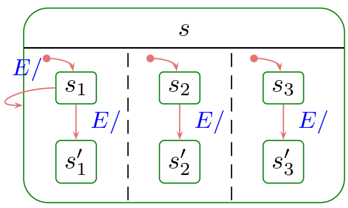


And?



- **...**:
  - We have to formally define what **event occurrence** is.
  - We have to define where events **are stored** – what the event pool is.
  - We have to explain how **transitions are chosen** – "matching".
  - We have to explain what the **effect of actions** is – on state and event pool.
  - We have to decide on the **granularity** — micro-steps, steps, run-to-completion steps (aka. super-steps)?
  - We have to formally define a notion of **stability** and RTC-step **completion**.

  - And then: hierarchical state machines.



Roadmap: Chronologically

(i) What do we (have to) cover?
UML State Machine Diagrams **Syntax**.

(ii) Def.: Signature with **signals**.

(iii) Def.: **Core state machine**.

(iv) Map UML State Machine Diagrams ✓ to core state machines.

Semantics:
The Basic Causality Model

✓ (v) Def.: **Ether** (aka. event pool)

✓ (vi) Def.: **System configuration**.

(vii) Def.: **Event**.

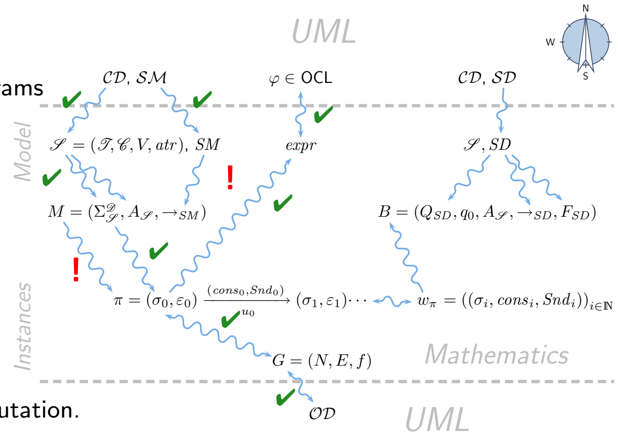
(viii) Def.: **Transformer**.

(ix) Def.: **Transition system**, computation.

(x) Transition relation induced by core state machine.

(xi) Def.: **step**, **run-to-completion step**.

(xii) Later: Hierarchical state machines.



System Configuration, Ether, Transformer

Ether aka. Event Pool

Definition. Let $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$ be a signature with signals and $\mathscr{D}$ a structure.

We call a ~~structure~~ tuple $(Eth, ready, \oplus, \ominus, [\,\cdot\,])$ an ether over $\mathscr{S}$ and $\mathscr{D}$ if and only if it provides

- a **ready** operation which yields a set of events that are ready for a given object, i.e. for an event pool ε and an object identity u obtain a set of signal-instance identities

$$ready : \underset{\ni\,\varepsilon}{Eth} \times \underset{\ni\,u}{\mathscr{D}(\mathscr{C})} \to 2^{\mathscr{D}(\mathscr{E})}$$

- a operation to **insert** an event destined for a given object, i.e.

$$\oplus : \underset{\ni\,\varepsilon}{Eth} \times \underset{\text{destination id}\,\ni\,u}{\mathscr{D}(\mathscr{C})} \times \underset{\text{event id}\,\ni\,u}{\mathscr{D}(\mathscr{E})} \to \underset{\ni\,\varepsilon'}{Eth}$$

obtain another event pool

- a operation to **remove** an event, i.e.

$$\ominus : Eth \times \mathscr{D}(\mathscr{E}) \to Eth$$

- an operation to clear the ether for a given object, i.e.

$$[\,\cdot\,] : Eth \times \mathscr{D}(\mathscr{C}) \to Eth.$$

Ether and [OMG, 2007b]

The standard distinguishes (among others)

- **SignalEvent** [OMG, 2007b, 450] and **Reception** [OMG, 2007b, 447].

On **SignalEvents**, it says

"reception takes place", more conceptual; for us: discard/dispatch

for us: event

> *A signal event represents the receipt of an asynchronous signal instance. A signal event may, for example, cause a state machine to trigger a transition.* [OMG, 2007b, 449]
>
> [...]
>
> **Semantic Variation Points**
>
> = messages
>
> *The means by which requests are transported to their target depend on the type of requesting action, the target, the properties of the communication medium, and numerous other factors.*
>
> *In some cases, this is instantaneous and completely reliable while in others it may involve transmission delays of variable duration, loss of requests, reordering, or duplication.*
>
> *(See also the discussion on page 421.)* [OMG, 2007b, 450]

Our **ether** is a general representation of the possible choices.

Often seen minimal requirement: order of sending **by one object** is preserved.
But: we'll later briefly discuss "discarding" of events.

System Configuration

Definition. Let $\mathscr{S}_0 = (\mathscr{T}_0, \mathscr{C}_0, V_0, atr_0, \mathscr{E})$ be a signature with signals, $\mathscr{D}_0$ a structure of $\mathscr{S}_0$, $(Eth, ready, \oplus, \ominus, [\,\cdot\,])$ an ether over $\mathscr{S}_0$ and $\mathscr{D}_0$. Furthermore assume there is one core state machine M_C per class $C \in \mathscr{C}$.

A system configuration over $\mathscr{S}_0$, $\mathscr{D}_0$, and Eth is a pair

a type name for the set of states in C's state machine

$$(\sigma, \varepsilon) \in \Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth$$

if $Bool \notin \mathscr{T}_0$ then add it here, and have $\mathscr{D}(Bool) = \mathbb{B}$

where

- $\mathscr{S} = (\mathscr{T}_0 \,\dot\cup\, \{S_{M_C} \mid C \in \mathscr{C}\}, \quad \mathscr{C}_0,$

  initial state of C's state machine

  $V_0 \,\dot\cup\, \{\langle stable : Bool, -, true, \emptyset\rangle\}$
  $\dot\cup\, \{\langle st_C : S_{M_C}, +, s_0, \emptyset\rangle \mid C \in \mathscr{C}\}$
  $\dot\cup\, \{\langle params_E : E_{0,1}, +, \emptyset, \emptyset\rangle \mid E \in \mathscr{E}_0\},$

  each object can refer to signal instances (at most one at a time) in order to access signal attributes

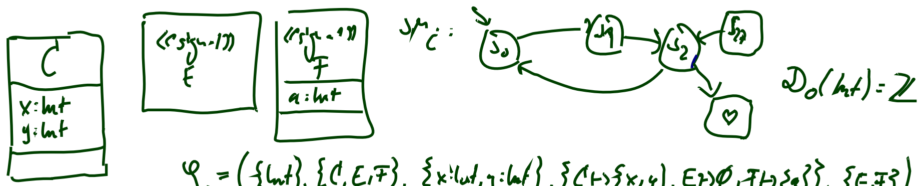
  $\{C \mapsto atr_0(C)$
  $\cup\, \{stable, st_C\} \cup \{params_E \mid E \in \mathscr{E}_0\} \mid C \in \mathscr{C}\}, \quad \mathscr{E}_0)$

  $\in 2^{\mathscr{D}(C)}$, if $r : C_{0,1}$ or $r : C_*$

- $\mathscr{D} = \mathscr{D}_0 \,\dot\cup\, \{S_{M_C} \mapsto S(M_C) \mid C \in \mathscr{C}\}$, and

  states of state machine M_C of C

- $\sigma(u)(r) \cap \mathscr{D}(\mathscr{E}_0) = \emptyset$ for each $u \in \mathrm{dom}(\sigma)$ and $r \in V_{0_{0,1,*}}$ (e.g. $r : C_{0,1}$)

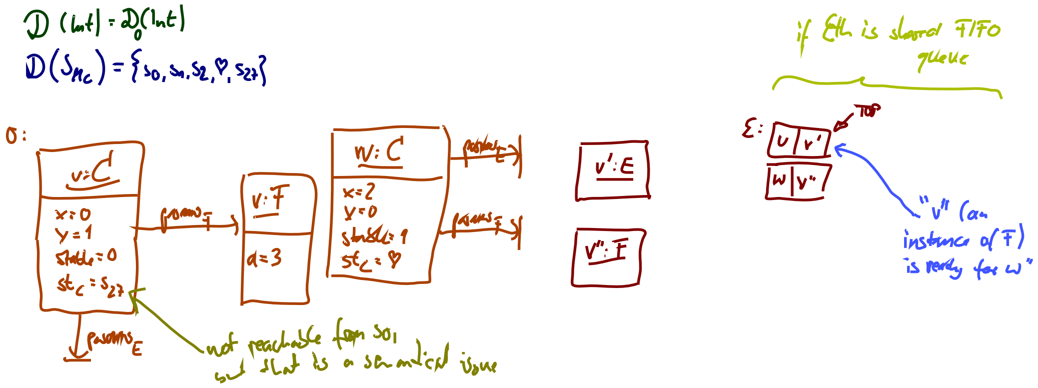


$\mathscr{S}_0 = (\{Int\}, \{C, E, F\}, \{x : Int, y : Int, a : Int\}, \{C \mapsto \{x, y\}, E \mapsto \emptyset, F \mapsto \{a\}\}, \{E, F\})$

$\mathscr{S} = (\{Int, S_{M_C}, Bool\}, \{C, E, F\}, \{x, y, a : Int, stable : Bool, st_C : S_{M_C}, params_E : E_{0,1}, params_F : F_{0,1}\},$
$\{C \mapsto \{x, y, stable, st_C, params_E, params_F\}, E \mapsto \emptyset, F \mapsto \{a\}\}, \{E, F\})$

$\mathscr{D}(Int) = \mathscr{D}_0(Int)$

$\mathscr{D}(S_{M_C}) = \{s_0, s_1, s_2, \heartsuit, s_{23}\}$



System Configuration Step-by-Step

- We start with some signature with signals $\mathscr{S}_0 = (\mathscr{T}_0, \mathscr{C}_0, V_0, atr_0, \mathscr{E})$.
- A **system configuration** is a pair (σ, ε) which comprises a system state σ wrt. $\mathscr{S}$ (not wrt. $\mathscr{S}_0$).
- Such a **system state** σ wrt. $\mathscr{S}$ provides, for each object $u \in \mathrm{dom}(\sigma)$,
  - values for the **explicit attributes** in V_0,
  - values for a number of **implicit attributes**, namely
    - a **stability flag**, i.e. $\sigma(u)(stable)$ is a boolean value,
    - a **current (state machine) state**, i.e. $\sigma(u)(st)$ denotes one of the states of core state machine M_C,
    - a temporary association to access **event parameters** for each class, i.e. $\sigma(u)(params_E)$ is defined for each $E \in \mathscr{E}$.
- For convenience require: there is **no link to an event** except for $params_E$.

Stability

Definition.
Let (σ, ε) be a system configuration over some $\mathscr{S}_0$, $\mathscr{D}_0$, Eth.

We call an object $u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(\mathscr{C}_0)$ stable in σ if and only if

$$\sigma(u)(stable) = \mathit{true}.$$

Events Are Instances of Signals

Definition. Let $\mathscr{D}_0$ be a structure of the signature with signals $\mathscr{S}_0 = (\mathscr{T}_0, \mathscr{C}_0, V_0, atr_0, \mathscr{E})$ and let $E \in \mathscr{E}_0$ be a **signal**.

Let $atr(E) = \{v_1, \dots, v_n\}$. We call

$$e = (E, \{v_1 \mapsto d_1, \dots, v_n \mapsto d_n\}),$$

or shorter (if mapping is clear from context)

$$(E, (d_1, \dots, d_n)) \text{ or } (E, \vec{d}),$$

an event (or an instance) of signal E (if type-consistent).

We use $Evs(\mathscr{E}_0, \mathscr{D}_0)$ to denote the set of all events of all signals in $\mathscr{S}_0$ wrt. $\mathscr{D}_0$.

As we always try to maximize confusion...:

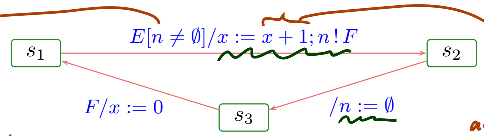
- By our existing naming convention, $u \in \mathscr{D}(E)$ is also called **instance** of the (signal) class E in system configuration (σ, ε) if $u \in \text{dom}(\sigma)$.
- The corresponding event is then $(E, \sigma(u))$.

Signals? Events...? Ether...?!

The idea is the following:

- **Signals** are **types** (classes).
- **Instances of signals** (in the standard sense) are kept in the **system state** component σ of system configurations (σ, ε).
- **Identities** of signal instances are kept in the **ether**.
- Each signal instance is in particular an **event** — somehow "a recording that this signal occurred" (without caring for its identity)
- The main difference between **signal instance** and **event**:
  Events don't have an identity.
- Why is this useful? In particular for **reflective** descriptions of behaviour, we are typically not interested in the identity of a signal instance, but only whether it is an "E" or "F", and which parameters it carries.

Where are we?



abbrev. for this.n ≠ ∅ (same rules as with "self")

abbrev. this.x + 1

- **Wanted**: a labelled transition relation

here

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

on system configuration, labelled with the **consumed** and **sent** events, (σ', ε') being the result (or effect) of **one object** u_x taking a transition of **its** state machine from the current state mach. state $\sigma(u_x)(st_C)$.

- **Have**: system configuration (σ, ε) comprising current state machine state and stability flag for each object, and the ether.

- **Plan**:
  (i) Introduce **transformer** as the semantics of action annotions.
  Intuitively, (σ', ε') is the effect of applying the transformer of the taken transition.
  (ii) Explain how to choose transitions depending on ε and when to stop taking transitions — the **run-to-completion "algorithm"**.

– 12 – 2011-12-11 – Sstmsem –

Transformer

because of non-determinism

Definition.
Let $\Sigma_{\mathscr{S}}^{\mathscr{D}}$ the set of system configurations over some $\mathscr{S}_0$, $\mathscr{D}_0$, Eth.

We call a relation

the object "executing" the action

sys. config after

$$t \subseteq \mathscr{D}(\mathscr{C}) \times (\Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth) \times (\Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth)$$

a (system configuration) **transformer**. system configuration before

- In the following, we assume that each application of a transformer t to some system configuration (σ, ε) for object u_x is associated with a set of **observations**

sender $\cup\{*,+\}$ events without identity   special symbol for create/destroy

$$Obs_t[u_x](\sigma, \varepsilon) \in 2^{\mathscr{D}(\mathscr{C}) \times \mathscr{D}(\mathscr{E}) \times Evs(\mathscr{E} \,\dot\cup\, \{*,+\}, \mathscr{D}) \times \mathscr{D}(\mathscr{C})}.$$

signal instance   receiver or destination

- An observation $(u_{src}, u_e, (E, \vec{d}), u_{dst}) \in Obs_t[u_x](\sigma, \varepsilon)$ represents the information that, as a "side effect" of u_x executing t, an event (!) $(E, \vec{d})$ has been sent from u_{src} to u_{dst}.

  Special cases: creation/destruction.

– 12 – 2011-12-11 – Sstmsem –

Why Transformers?

- **Recall** the (simplified) syntax of transition annotations:

$$annot ::= [\ \ \langle event \rangle\ \ [\ \text{`['}\ \langle guard \rangle\ \text{`]'}\ ]\ \ [\ \text{`/'}\ \langle action \rangle]\ \ ]$$

- **Clear**: $\langle event \rangle$ is from $\mathscr{E}$ of the corresponding signature.

- **But:** What are $\langle guard \rangle$ and $\langle action \rangle$?
  - UML can be viewed as being **parameterized** in **expression language** (providing $\langle guard \rangle$) and **action language** (providing $\langle action \rangle$).
  - **Examples**:
    - **Expression Language**:
      - OCL
      - Java, C++, . . . expressions
      - . . .
    - **Action Language**:
      - UML Action Semantics, "Executable UML"
      - Java, C++, . . . statements (plus some event send action)
      - . . .

In the following, we consider:

$$Act_{\mathscr{S}} ::= \{skip\}$$
$$\cup \{update(expr_1, v, expr_2) \mid expr_1, expr_2 \in OCLExpr, v \in V\}$$
$$\cup \{send(expr_1, E, expr_2) \mid expr_1, expr_2 \in OCLExpr, E \in \mathscr{E}\}$$
$$\cup \{create(C, expr, v) \mid expr \in OCLExpr, C \in \mathscr{C}, v \in V\}$$
$$\cup \{destroy(expr) \mid expr \in OCLExpr\}$$

$Expr_{\mathscr{S}}$: OCL expressions over $\mathscr{S}$

Transformers as Abstract Actions!

example OCL:

$$I[\![expr]\!](\sigma, u) = \begin{cases} true, & \text{if } I[\![expr]\!](\sigma, \{this \mapsto u\}) = true \\ false, & \text{if } I[\![expr]\!](\sigma, \{this \mapsto u\}) = false \\ \text{undefined otherwise} \end{cases}$$

In the following, we assume that we're **given**

- an **expression language** $Expr$ for guards, and
- an **action language** Act for actions,

and that we're **given**

- a **semantics** for boolean expressions in form of a partial function

$$I[\![\,\cdot\,]\!](\,\cdot\,,\,\cdot\,) : Expr \to ((\Sigma_{\mathscr{S}}^{\mathscr{D}} \times (\cancel{\{this\} \to} \mathscr{D}(\mathscr{C}))) \nrightarrow \mathbb{B})$$

  which evaluates expressions in a given system configuration,

  Assuming I to be partial is a way to treat "undefined" during runtime. If I is not defined (for instance because of dangling-reference navigation or division-by-zero), we want to go to a designated "error" system configuration.

- a **transformer** for each action: for each $act \in Act$, we assume to have

$$t_{act} \subseteq \mathscr{D}(\mathscr{C}) \times (\Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth) \times (\Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth)$$

Expression/Action Language Examples

We can make the assumptions from the previous slide because **instances exist**:

- for OCL, we have the OCL semantics from Lecture 03. Simply remove the pre-images which map to "$\perp$".
- for Java, the operational semantics of the SWT lecture uniquely defines transformers for sequences of Java statements.

We distinguish the following kinds of transformers:

- **skip**: do nothing — recall: this is the default action
- **send**: modifies ε — interesting, because state machines are built around sending/consuming events
- **create**/**destroy**: modify domain of σ — not specific to state machines, but let's discuss them here as we're at it
- **update**: modify own or other objects' local state — boring

Transformer Examples: Presentation

abstract syntax **concrete syntax**

`op`

intuitive semantics

. . .

well-typedness

. . .

semantics

object "execution" action op

$$((\sigma, \varepsilon), (\sigma', \varepsilon')) \in t_{\texttt{op}}[u_x] \text{ iff } \dots$$

or

$$t_{\texttt{op}}[u_x](\sigma, \varepsilon) = \{(\sigma', \varepsilon')\} \text{ where } \dots$$

observables

$Obs_{\texttt{op}}[u_x] = \{\dots\}$, not a relation, depends on choice

(error) conditions

Not defined if . . .

– 12 – 2011-12-11 – Sstmsem –

Transformer: Skip

abstract syntax **concrete syntax**

`skip` skip

intuitive semantics

do nothing

well-typedness

./.

semantics

"if u_x executes skip on (σ, ε), this results in (σ, ε)"

$$t[u_x](\sigma, \varepsilon) = \{(\sigma, \varepsilon)\}$$

observables

$$Obs_{\texttt{skip}}[u_x](\sigma, \varepsilon) = \emptyset$$

(error) conditions

– 12 – 2011-12-11 – Sstmsem –

Transformer: Update

abstract syntax — **concrete syntax**

$\texttt{update}(expr_1, v, expr_2)$ — $expr_1 . v := expr_2$

$v := expr_2$ (abbr. $this.v := expr_2$)

intuitive semantics

Update attribute v in the object denoted by $expr_1$ to the value denoted by $expr_2$.

well-typedness

$expr_1 : \tau_C$ and $v : \tau \in atr(C)$; $expr_2 : \tau$;
$expr_1, expr_2$ obey visibility and navigability

semantics

$$t_{\texttt{update}(expr_1,v,expr_2)}[u_x](\sigma, \varepsilon) = \{(\sigma', \varepsilon)\}$$

either doesn't change

semantics of Expr (in our case OCL)

where $\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[\![expr_2]\!](\sigma, \beta)]]$ with
$u = I[\![expr_1]\!](\sigma, \beta)$, $\beta = \{\texttt{this} \mapsto u_x\}$.

changing the value of v...

... in u

observables

$$Obs_{\texttt{update}(expr_1,v,expr_2)}[u_x] = \emptyset$$

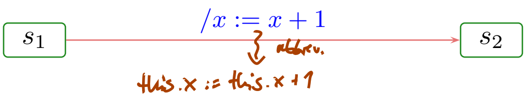
(error) conditions

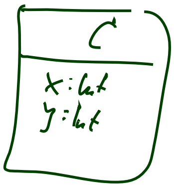
Not defined if $I[\![expr_1]\!](\sigma, \beta)$ or $I[\![expr_2]\!](\sigma, \beta)$ not defined.

i.e. $t_{update(\ldots)}[u_x](\sigma,\varepsilon) = \emptyset$

Update Transformer Example

$\mathcal{SM}_C$:





$$\texttt{update}(expr_1, v, expr_2)$$

$$t_{\texttt{update}(expr_1,v,expr_2)}[u_x](\sigma, \varepsilon) = (\sigma[u \mapsto \sigma(u)[v \mapsto I[\![expr_2]\!](\sigma, \beta)]], \varepsilon),$$
$$u = I[\![expr_1]\!](\sigma, \beta)$$

$u_x = u_1$

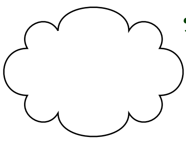
σ: $\underline{u_1 : C}$, $x = 4$, $y = 0$

$\underline{u_1 : C}$, $x = 5$, $y = 0$:σ'

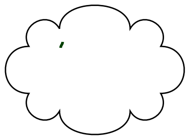
$t_{upd}[u_1](\sigma,\varepsilon) =$

$(\sigma[u_1 \mapsto \sigma(u_1)[x \mapsto \underbrace{I[\![this.x + 1]\!](\sigma, \underbrace{\{this \mapsto u_1\}}_{=:\beta})}_{=5}]], \varepsilon)$

ε:



$\underbrace{\underbrace{I[\![this.x]\!](\sigma,\beta)}_{=4} + \underbrace{I[\![1]\!](\sigma,\beta)}_{=1}}_{=5}$



:ε'

References

References

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

[OMG, 2007a] OMG (2007a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-04.

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