Software Design, Modelling and Analysis in UML

Lecture 13: Core State Machines III

2013-12-16

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

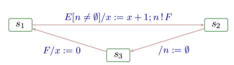
- Ether
- System configuration

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:**
  - Transformer
  - Examples for transformer
  - Run-to-completion Step
  - Putting It All Together

System Configuration, Ether, Transformer

Where are we?



- **Wanted**: a labelled transition relation

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

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 $\overset{\sigma,}{}\varepsilon$ and when to stop taking transitions — the **run-to-completion "algorithm"**.

Transformer

not a function, to model non-determinism

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

We call a relation

the identity of the object which "executes" the action

system configuration after.

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

a (system configuration) **transformer**.

system configuration before executing the action

- 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**

id of sender — maybe none $\dot{\cup}\{\bot\}$ — events without id — id of receiver (or destination)

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

id of event — special symbols for create and destroy

- 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.

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)
      - ...

Transformers as Abstract Actions!

example OCL:

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

OCL interpretation

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

the same

object id to evaluate for

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

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 "$\bot$".
- 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   only skip
- **send**: modifies ε — interesting, because state machines are built around sending/consuming events   e.g. $n!F$
- **create/destroy**: modify domain of σ — not specific to state machines, but let's discuss them here as we're at it   e.g. new C, delete n
- **update**: modify own or other objects' local state — boring   e.g. $x := x+1$

Action Language

In the following we consider

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

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

Transformer Examples: Presentation

abstract syntax	**concrete syntax**
op update(e_1, v, e_2)	$e_1.v := e_2$
intuitive semantics	
...	
well-typedness	
...	
semantics	
$((\sigma, \varepsilon), (\sigma', \varepsilon')) \in t_{\texttt{op}}[u_x]$ iff ... or $t_{\texttt{op}}[u_x](\sigma, \varepsilon) = \{(\sigma', \varepsilon')\}$ where ...	
observables	
$Obs_{\texttt{op}}[u_x] = \{\dots\}$, not a relation, depends on choice	
(error) conditions	
Not defined if ...	

Transformer: Skip

abstract syntax	**concrete syntax**
skip	skip
intuitive semantics	
do nothing	
well-typedness	
./.	
semantics	
$t[u_x](\sigma, \varepsilon) = \{(\sigma, \varepsilon)\}$	"if u_x executes skip on (σ, ε), then the result is (σ, ε)"
observables	
$Obs_{\texttt{skip}}[u_x](\sigma, \varepsilon) = \emptyset$	
(error) conditions	

Transformer: Update

abstract syntax	**concrete syntax**
update$(expr_1, v, expr_2)$	$expr_1.v := expr_2$
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	
$t_{\texttt{update}(expr_1, v, expr_2)}[u_x](\sigma, \varepsilon) = \{(\sigma', \varepsilon)\}$ where $\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[\![expr_2]\!](\sigma, u_x)]]$ with $u = I[\![expr_1]\!](\sigma, u_x)$	
observables	
$Obs_{\texttt{update}(expr_1, v, expr_2)}[u_x] = \emptyset$	
(error) conditions	
Not defined if $I[\![expr_1]\!](\sigma, u_x)$ or $I[\![expr_2]\!](\sigma, u_x)$ not defined.	

either does not change

value denoted by $expr_2$ in σ for object u

change local state of object u

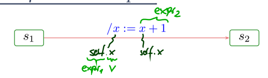
change value of v in $\sigma(u)$

object denoted by $expr_1$ (relative to u_x)

i.e. $t_{update(e_1, v, e_2)}[u_x](\sigma, \varepsilon) = \emptyset$

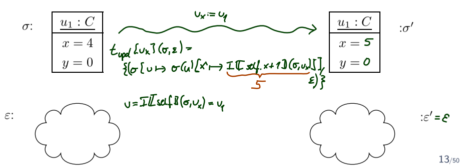
Update Transformer Example

$\mathcal{SM}_C$:



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

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



Transformer: Send

abstract syntax — $\texttt{send}(E(expr_1, ..., expr_n), expr_{dst})$

concrete syntax — $expr_{dst}\,!\,E(expr_1, \ldots, expr_n)$

intuitive semantics

Object $u_x : C$ sends event E to object $expr_{dst}$, i.e. create a fresh signal instance, fill in its attributes, and place it in the ether.

well-typedness — do not send to signal instances

$$expr_{dst} : \tau_D,\ C, D \in \mathscr{C} \setminus \mathscr{E};\ E \in \mathscr{E};$$
$$atr(E) = \{v_1 : \tau_1, \ldots, v_n : \tau_n\};\ expr_i : \tau_i,\ 1 \le i \le n;$$
all expressions obey visibility and navigability in C

semantics

$$t_{\texttt{send}(E(expr_1,\ldots,expr_n),expr_{dst})}[u_x](\sigma, \varepsilon) \ni (\sigma', \varepsilon')$$

disjoint union

where $\sigma' = \sigma \,\dot\cup\, \{u \mapsto \{v_i \mapsto d_i \mid 1 \le i \le n\}\}$; $\varepsilon' = \varepsilon \oplus (u_{dst}, u)$;

id of destination

id of new signal inst.

if $u_{dst} = I[\![expr_{dst}]\!](\sigma, u_x) \in \mathrm{dom}(\sigma)$; $d_i = I[\![expr_i]\!](\sigma, u_x)$ for $1 \le i \le n$;

$u \in \mathscr{D}(E)$ a fresh identity, i.e. $u \notin \mathrm{dom}(\sigma)$,

our choice – we could also consider it to be an error

and where $(\sigma', \varepsilon') = (\sigma, \varepsilon)$ if $u_{dst} \notin \mathrm{dom}(\sigma)$

do nothing if destination not alive in σ

observables

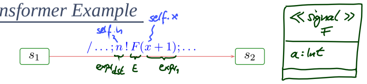
$$Obs_{\texttt{send}}[u_x] = \{(u_x, u, (E, d_1, \ldots, d_n), u_{dst})\}$$

(error) conditions

$I[\![expr]\!](\sigma, u_x)$ not defined for any $expr \in \{expr_{dst}, expr_1, \ldots, expr_n\}$

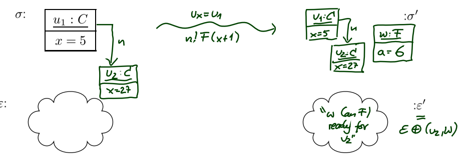
Send Transformer Example

$\mathcal{SM}_C$:



$\texttt{send}(E(expr_1, ..., expr_n), expr_{dst})$

$t_{\texttt{send}(expr_{src}, E(expr_1, ..., expr_n), expr_{dst})}[u_x](\sigma, \varepsilon) = \ldots$



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.