Software Design, Modelling and Analysis in UML

Lecture 11: Core State Machines I

2016-12-08

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

Albert-Ludwigs-Universität Freiburg, Germany

Content

- **Recall**: Basic Causality Model
- **Event Pool**
  - insert, remove, clear, ready.
- **System Configuration**
  - **implicit attributes**: $stable$, st, and friends.
  - **system state** plus **event pool**
- **Actions**
  - simple **action language**.
  - **transformer**: effects of actions.

Roadmap: Chronologically

Syntax:

(i) UML State Machine Diagrams. ✓

(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**.

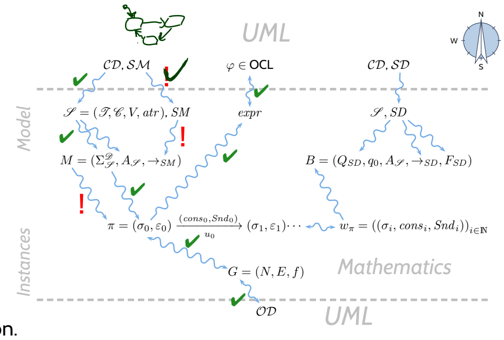
(viii) Def.: **Transformer**. x:=x+1; n!F

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

(x) Transition relation induced by core state machine.

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

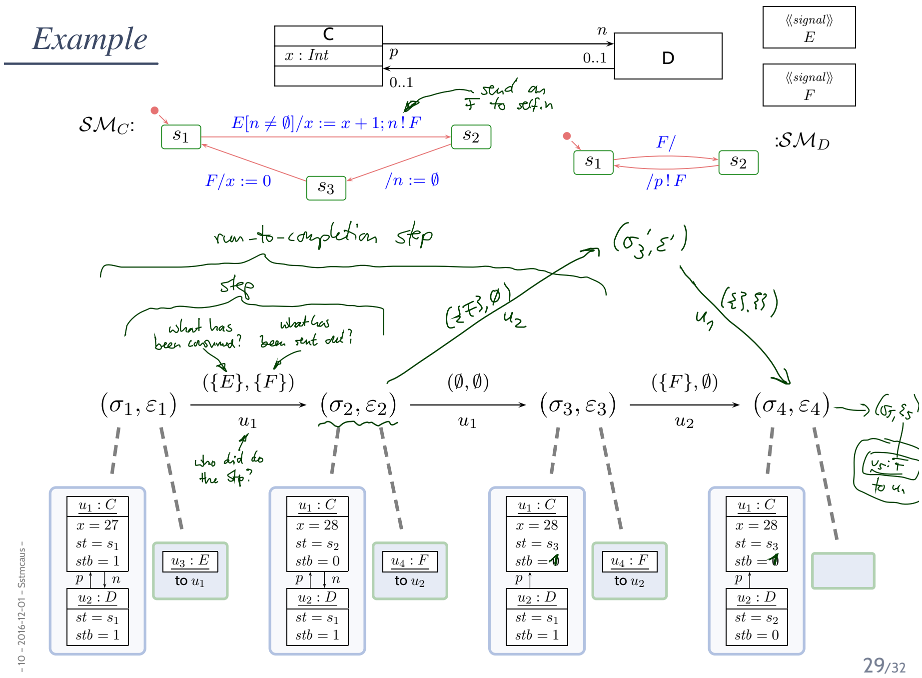
(xii) Later: Hierarchical state machines.



15.3.12 StateMachine (OMG, 2011b, 574)

- Event occurrences are detected, dispatched, and then processed by the state machine, one at a time.
- The semantics of event occurrence processing is based on the **run-to- completion assumption**, interpreted as **run-to-completion processing**.
- **Run-to-completion processing** means that an event [...] can only be taken from the pool and dispatched if the processing of the previous [...] is fully completed.
- The processing of a single event occurrence by a state machine is known as a **run-to-completion step**.
- Before commencing on a **run-to-completion step**, a state machine is in a **stable state** configuration with all entry/exit/internal-activities (but not necessarily do-activities) completed.
- The same conditions apply after the **run-to-completion step** is completed.
- Thus, an event occurrence will never be processed [...] in some intermediate and inconsistent situation.
- [IOW,] The **run-to-completion step** is the passage between two ~~state~~ stable configurations of the state machine.
- The **run-to-completion assumption** simplifies the transition function of the StM, since concurrency conflicts are avoided during the processing of event, allowing the StM to safely complete its **run-to-completion step**.
- The order of dequeuing is **not defined**, leaving open the possibility of modeling different priority-based schemes.
- Run-to-completion may be implemented in **various ways**. [...]

Example



Ether

Recall: 15.3.12 StateMachine (OMG, 2011b, 563)

- The order of dequeuing is **not defined**, leaving open the possibility of modeling different (priority-based) schemes.

Ether and OMG (2011b)

«signal»
E

The standard distinguishes (among others)

- **SignalEvent** (OMG, 2011b, 450) and **Reception** (OMG, 2011b, 447).

On **SignalEvents**, it says

> *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, 2011b, 449) [...]
>
> **Semantic Variation Points**
>
> *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, 2011b, 450)

Our **ether** ($\rightarrow$ in a minute) is a general representation of **many possible choices**.

Often seen minimal requirement: order of sending **by one object** is preserved.

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

for an event pool ε... ...and object identity u... ..yield a set of signal instances ready for consumption by u

- a ready operation which yields a set of events (i.e., signal instances) that are ready for a given object, i.e.

$$ready : Eth \times \mathscr{D}(\mathscr{C}) \to 2^{\mathscr{D}(\mathscr{E})}$$

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

destination signal instance

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

- 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.

destination

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

Example: FIFO Queue

A (single, global, shared, reliable) FIFO queue is an ether:

- $Eth = (\mathscr{D}(\mathscr{C}) \times \mathscr{D}(\mathscr{E}))^*$

  destination object id signal instance

  the set of finite sequences of pairs $(u, e) \in \mathscr{D}(\mathscr{C}) \times \mathscr{D}(\mathscr{E})$

- $ready : Eth \times \mathscr{D}(\mathscr{C}) \to 2^{\mathscr{D}(\mathscr{E})}$

$$(\varepsilon, u_2) \mapsto \begin{cases} \{(u_2, e)\}, & \text{if } \varepsilon = (u_2, e).\varepsilon' \\ \emptyset, & \text{otherwise} \end{cases}$$

- $\oplus : Eth \times \mathscr{D}(\mathscr{C}) \times \mathscr{D}(\mathscr{E}) \to Eth$

  $(\varepsilon, u, e) \mapsto \varepsilon.(u, e)$

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

$$(\varepsilon, e) \mapsto \begin{cases} \varepsilon', & \text{if } \varepsilon = (u, e).\varepsilon', \ u \in \mathscr{D}(\mathscr{C}) \\ \varepsilon, & \text{otherwise} \end{cases}$$

- $[\,\cdot\,] : Eth \times \mathscr{D}(\mathscr{C}) \to Eth$ $\quad [\cdot](\varepsilon, u)$:

  remove all (u, e) elements from the given ε, $e \in \mathscr{D}(\mathscr{E})$

Other Examples

- One FIFO queue per active object is an ether.

  $$Eth = \mathscr{D}(\mathscr{C}) \to (\mathscr{D}(\mathscr{C}) \times \mathscr{D}(\mathscr{E}))^*$$

- One-place buffer.

  $$Eth = \epsilon \,\dot{\cup}\, (\mathscr{D}(\mathscr{C}) \times \mathscr{D}(\mathscr{E}))$$

- Priority queue.

  ...

- Multi-queues (one per sender).

  ...

- Trivial example: sink, "black hole".

  ...

- Lossy queue ($\oplus$ needs to become a relation then).

- ...

System Configuration

System Configuration

Definition. Let $\mathscr{S}_0 = (\mathscr{T}_0, \mathscr{C}_0, V_0, atr_0, \mathscr{E}_0)$ 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

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

where

a new type for each class

if $Bool \notin \mathscr{T}_0$ then add it and use $\mathscr{D}(Bool) = \{0,1\}$

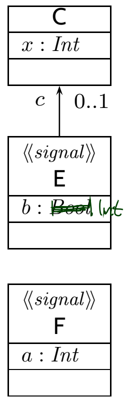
initial state of M_C

- $\mathscr{S} = (\mathscr{T}_0 \,\dot\cup\, \{S_{M_C} \mid C \in \mathscr{C}_0\}, \quad \mathscr{C}_0,$
  $V_0 \,\dot\cup\, \{\langle stable : Bool, -, \textbf{true}, \emptyset\rangle\}$
  $\quad \dot\cup\, \{\langle st_C : S_{M_C}, +, s_0, \emptyset\rangle \mid C \in \mathscr{C}\}$
  $\quad \dot\cup\, \{\langle params_E : E_{0,1}, +, \emptyset, \emptyset\rangle \mid E \in \mathscr{E}_0\},$
  $\{C \mapsto atr_0(C)$
  $\quad \cup \{stable, st_C\} \cup \{params_E \mid E \in \mathscr{E}_0\} \mid C \in \mathscr{C}\}, \quad \mathscr{E}_0)$

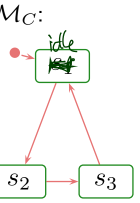
  $\cup \; atr_0|_{\mathscr{E}_0}$

- $\mathscr{D} = \mathscr{D}_0 \,\dot\cup\, \{S_{M_C} \mapsto S(M_C) \mid C \in \mathscr{C}\}$, and
- $\sigma(u)(r) \cap \mathscr{D}(\mathscr{E}_0) = \emptyset$ for each $u \in \mathrm{dom}(\sigma)$ and $r \in V_0$.

System Configuration: Example



SM_C:



$\mathscr{S}_0 = (\mathscr{T}_0, \mathscr{C}_0, V_0, atr_0, \mathscr{E}_0)$, $\mathscr{D}_0$; $\quad (\sigma, \varepsilon) \in \Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth$ where

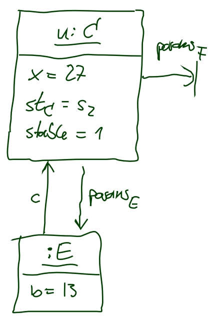
- $\mathscr{S} = (\mathscr{T}_0 \,\dot\cup\, \{S_{M_C} \mid C \in \mathscr{C}\}, \quad \mathscr{C}_0,$
  $V_0 \,\dot\cup\, \{\langle stable : Bool, -, \textbf{true}, \emptyset\rangle\} \,\dot\cup\, \{\langle st_C : S_{M_C}, +, s_0, \emptyset\rangle \mid C \in \mathscr{C}\}$
  $\quad \dot\cup\, \{\langle params_E : E_{0,1}, +, \emptyset, \emptyset\rangle \mid E \in \mathscr{E}_0\},$
  $\{C \mapsto atr_0(C) \cup \{stable, st_C\} \cup \{params_E \mid E \in \mathscr{E}_0\} \mid C \in \mathscr{C}\}, \quad \mathscr{E}_0)$
- $\mathscr{D} = \mathscr{D}_0 \,\dot\cup\, \{S_{M_C} \mapsto S(M_C) \mid C \in \mathscr{C}\}$, and
- $\sigma(u)(r) \cap \mathscr{D}(\mathscr{E}_0) = \emptyset$ for each $u \in \mathrm{dom}(\sigma)$ and $r \in V_0$.

$\mathscr{S}_0 = (\{Int\},$
$\{C\},$
$\underbrace{\{x : Int, b : Int, a : Int, c : C_{0,1}\}}_{=: V_0},$
$\{C \mapsto \{x\},$
$E \mapsto \{b, c\},$
$F \mapsto \{a\}\},$
$\{E, F\})$

$\mathscr{S} = (\{Int, Bool, S_{M_C}\},$
$\{C\},$
$V_0 \cup \{stable : Bool,$
$st_C : S_{M_C},$
$params_E : E_{0,1},$
$params_F : F_{0,1}\},$
$\{C \mapsto \{x, stable, st_C,$
$params_E, params_F\},$
$E \mapsto \{b, c\},$
$F \mapsto \{a\}\},$ $\{E, F\}$

$\mathscr{D}(S_{M_C}) = \{idle, s_2, s_3\}$

$\sigma \in \Sigma_{\mathscr{S}}^{\mathscr{D}}$:



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

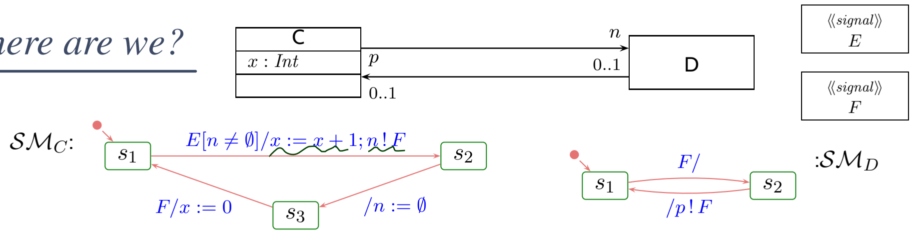
Definition.
Let (σ, ε) be a system configuration over some $\mathscr{S}_0$, $\mathscr{D}_0$, Eth.

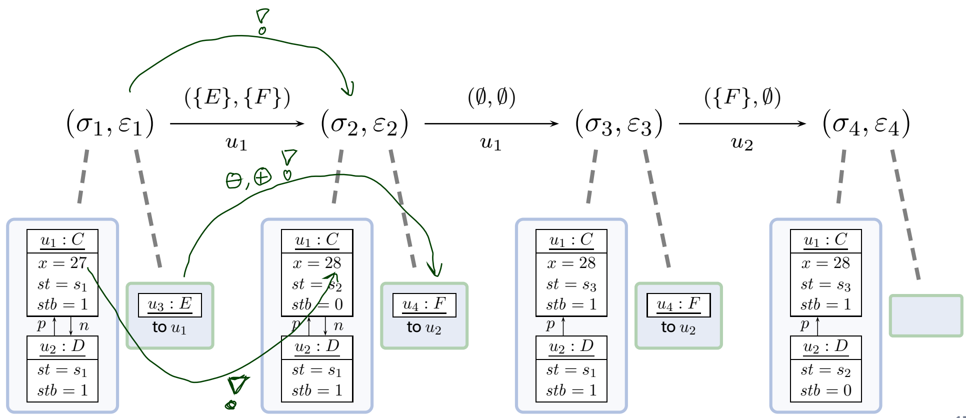
We call an object $u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(\mathscr{C}_0)$ stable in σ if and only if

$$\sigma(u)(stable) = \cancel{true}. \ 1$$

And unstable otherwise.

Where are we?





Transformer

Recall

- The (simplified) syntax of transition annotations:

$$annot ::= \big[\quad \langle event \rangle \quad [\text{‘[’} \, \langle guard \rangle \, \text{‘]’}] \quad [\text{‘/’} \, \langle action \rangle] \quad \big]$$

- **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)
      - ...
      - $\rightarrow$

Needed: Semantics

OCL:

$$I_{Expr}[\![expr]\!](\sigma, u) := \begin{cases} 1, & \text{if } I_{OCL}[\![expr]\!](\sigma, \{self \mapsto u\}) = 1 \\ 0, & \text{if } I_{OCL}[\![expr]\!](\sigma, \{self \mapsto u\}) = 0 \\ \text{undefined}, & \text{otherwise} \end{cases}$$

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

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

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

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

Transformer

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

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

a (system configuration) **transformer**.

Example:

- $t[u_x](\sigma, \varepsilon) \subseteq \Sigma_{\mathscr{S}}^{\mathscr{D}} \times Eth$ is
  - the set (!) of the **system configurations**
  - which **may** result from **object** u_x
  - **executing** transformer t.
- $t_{\mathtt{skip}}[u_x](\sigma, \varepsilon) = \{(\sigma, \varepsilon)\}$
- $t_{\mathtt{create}}[u_x](\sigma, \varepsilon)$: add a previously non-alive object to σ (id. non-det. chosen)

Observations

- 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**

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

- An observation

  $$(u_e, u_{dst}) \in Obs_t[u_x](\sigma, \varepsilon)$$

  represents the information that,
  as a "side effect" of object u_x executing t in system configuration (σ, ε),
  the event u_e has been sent to u_{dst}.

  Special cases: creation ('$*$') / destruction ('$+$').

A Simple Action Language

In the following we use

$$
\begin{aligned}
Act_{\mathscr{S}} = \{\texttt{skip}\}& \\
&\cup \{\texttt{update}(expr_1, v, expr_2) \mid expr_1, expr_2 \in Expr_{\mathscr{S}}, v \in atr\} \\
&\cup \{\texttt{send}(E(expr_1, ..., expr_n), expr_{dst}) \mid expr_i, expr_{dst} \in Expr_{\mathscr{S}}, E \in \mathscr{E}\} \\
&\cup \{\texttt{create}(C, expr, v) \mid C \in \mathscr{C}, expr \in Expr_{\mathscr{S}}, v \in V\} \\
&\cup \{\texttt{destroy}(expr) \mid expr \in Expr_{\mathscr{S}}\}
\end{aligned}
$$

and OCL expressions over $\mathscr{S}$ (with partial interpretation) as $Expr_{\mathscr{S}}$.

Transformer Examples: Presentation

abstract syntax | **concrete syntax**

`op`

intuitive semantics

...

well-typedness

...

semantics

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

or

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

transformer t_{op}

observables

$Obs_{\text{op}}[u_x] = \{\dots\}$

(error) conditions

Not defined if ...

Transformer: Skip

abstract syntax — **concrete syntax**

`skip` — skip

intuitive semantics

do nothing

well-typedness

./.

semantics

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

observables

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

(error) conditions

Transformer: Update

x := x + 1
(self.x := self.x + 1)

abstract syntax — **concrete syntax**

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

intuitive semantics

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

well-typedness

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

either does not change

semantics

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

change state of object u — local — change value of this attr.

where $\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto \underbrace{I[\![expr_2]\!](\sigma, u_x)}_{\text{new value}}]]$ with

$u = \underbrace{I[\![expr_1]\!](\sigma, u_x)}_{\text{object denoted by } expr_1 \text{ (relative to } u_x \text{ as self)}}$.

observables

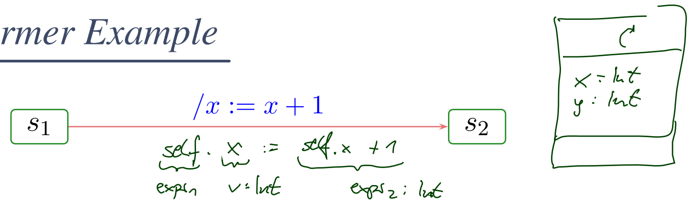
$$Obs_{\texttt{update}(expr_1,v,expr_2)}[u_x] = \emptyset$$

(error) conditions

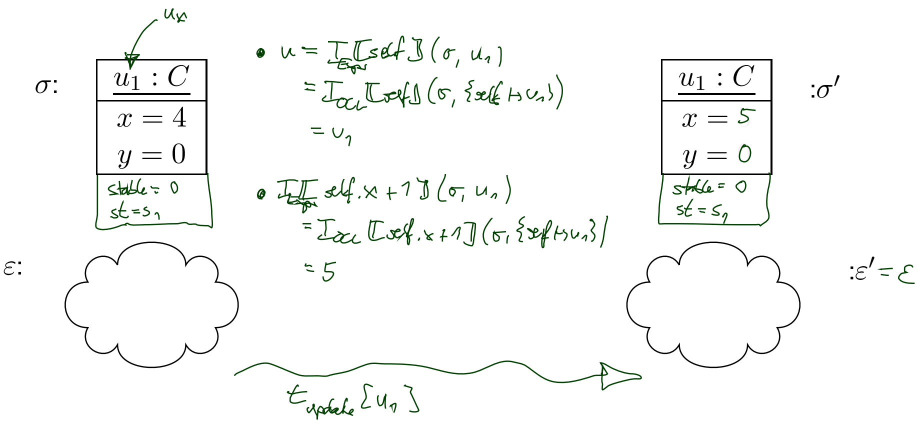
Not defined if $I[\![expr_1]\!](\sigma, u_x)$ or $I[\![expr_2]\!](\sigma, u_x)$ not defined.

Update Transformer Example

$\mathcal{SM}_C$:



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



Transformer: Send

abstract syntax	**concrete syntax**
$\mathtt{send}(E(expr_1, ..., expr_n), expr_{dst})$	

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

$E \in \mathscr{E}$; $atr(E) = \{v_1 : T_1, \dots, v_n : T_n\}$; $expr_i : T_i, 1 \le i \le n$;
$expr_{dst} : T_D, C, D \in \mathscr{C} \setminus \mathscr{E}$;
all expressions obey visibility and navigability in C

semantics

$$(\sigma', \varepsilon') \in t_{\mathtt{send}(E(expr_1, ..., expr_n), expr_{dst})}[u_x](\sigma, \varepsilon)$$

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

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

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

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

observables

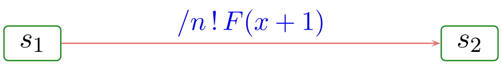
$$Obs_{\mathtt{send}}[u_x] = \{(u_e, u_{dst})\}$$

(error) conditions

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

Send Transformer Example

$\mathcal{SM}_C$:



$$t_{\mathtt{send}(expr_{src}, E(expr_1,\dots,expr_n), expr_{dst})}[u_x](\sigma,\varepsilon) \ni (\sigma',\varepsilon') \text{ iff } \varepsilon' = \varepsilon \oplus (u_{dst}, u);$$
$$\sigma' = \sigma \,\dot\cup\, \{u \mapsto \{v_i \mapsto d_i \mid 1 \le i \le n\}\};\ u_{dst} = I[\![expr_{dst}]\!](\sigma, u_x) \in \mathrm{dom}(\sigma);$$
$$d_i = I[\![expr_i]\!](\sigma,u_x),\ 1 \le i \le n;\ u \in \mathscr{D}(E) \text{ a fresh identity;}$$

σ:

$\underline{u_1 : C}$
$x = 5$

$:\sigma'$

ε:

$:\varepsilon'$

Sequential Composition of Transformers

- **Sequential composition** $t_1 \circ t_2$ of transformers t_1 and t_2 is canonically defined as

$$(t_2 \circ t_1)[u_x](\sigma,\varepsilon) = t_2[u_x](t_1[u_x](\sigma,\varepsilon))$$

  with observation

$$Obs_{(t_2 \circ t_1)}[u_x](\sigma,\varepsilon) = Obs_{t_1}[u_x](\sigma,\varepsilon) \cup Obs_{t_2}[u_x](t_1(\sigma,\varepsilon)).$$

- **Clear**: not defined if one the two intermediate "micro steps" is not defined.

Transformers And Denotational Semantics

Observation: our transformers are in principle the **denotational semantics** of the actions/action sequences. The trivial case, to be precise.

Note: with the previous examples, we can capture

- empty statements, skips,
- assignments,
- conditionals (by normalisation and auxiliary variables),
- create/destroy (later),

but not **possibly diverging loops**.

Our (Simple) Approach: if the action language is, e.g. Java, then (**syntactically**) forbid loops and calls of recursive functions.

Other Approach: use full blown denotational semantics.

No show-stopper, because loops in the action annotation can be converted into transition cycles in the state machine.

Tell Them What You've Told Them...

- A **ether** is an abstract representation of different possible "event pools" like
  - FIFO queues (shared, or per sender),
  - One-place buffers,
  - ...
- A **system configuration** consists of
  - an **event pool** (pending messages),
  - a **system state** over a signature with **implicit attributes** for
    - current state,
    - stability,
    - etc.
- Transitions are labelled with **actions**, the effect of actions is explained by **transformers**, transformers may modify **system state** and **ether**.

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.