Software Design, Modelling and Analysis in UML

Lecture 13: Core State Machines IV

2012-12-12

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

- System configuration
- Transformer

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 cont'd
  - Examples for transformer
  - Run-to-completion Step
  - Putting It All Together

System Configuration, Ether, Transformer

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

  $V_0 \,\dot\cup\, \{\langle stable : Bool, -, true, \emptyset\rangle\}$

  $\dot\cup\, \{\langle st_C : S_{M_C}, +, s_0, \emptyset\rangle \mid C \in \mathscr{C}\}$

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

  $\dot\cup\, \{\langle params_E : E_{0,1}, +, \emptyset, \emptyset\rangle \mid E \in \mathscr{E}_0\},$

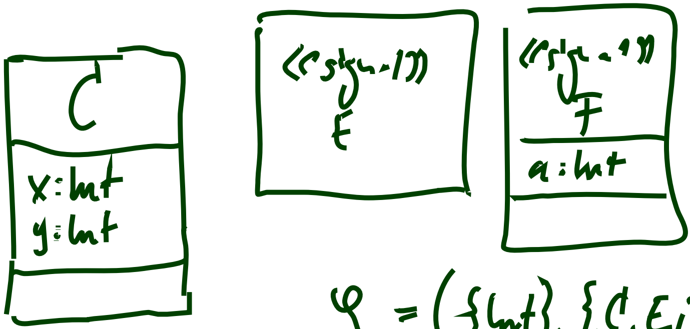
  $\{C \mapsto atr_0(C)$

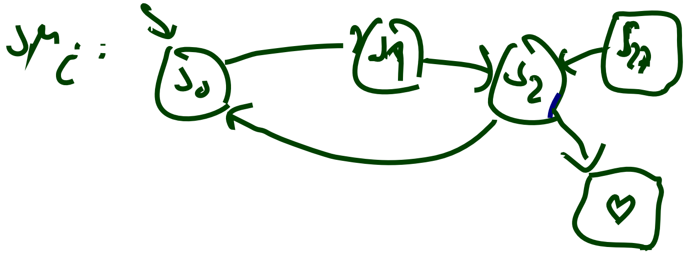
  $\cup\, \{stable, st_C\} \cup \{params_E \mid E \in \mathscr{E}_0\} \mid C \in \mathscr{C}\}, \quad \mathscr{E}_0)$

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

  states of state machine M_C of C

- $\underbrace{\sigma(u)(r)}_{\in 2^{\mathscr{D}(C)}, \text{ if } r : C_{0,1} \text{ or } r : C_*} \cap \mathscr{D}(\mathscr{E}_0) = \emptyset$ for each $u \in \mathrm{dom}(\sigma)$ and $r \in V_{0_{0,1,*}}$ (e.g. $r : C_{0,1}$)





$\mathcal{D}_0(Int) = \mathbb{Z}$

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

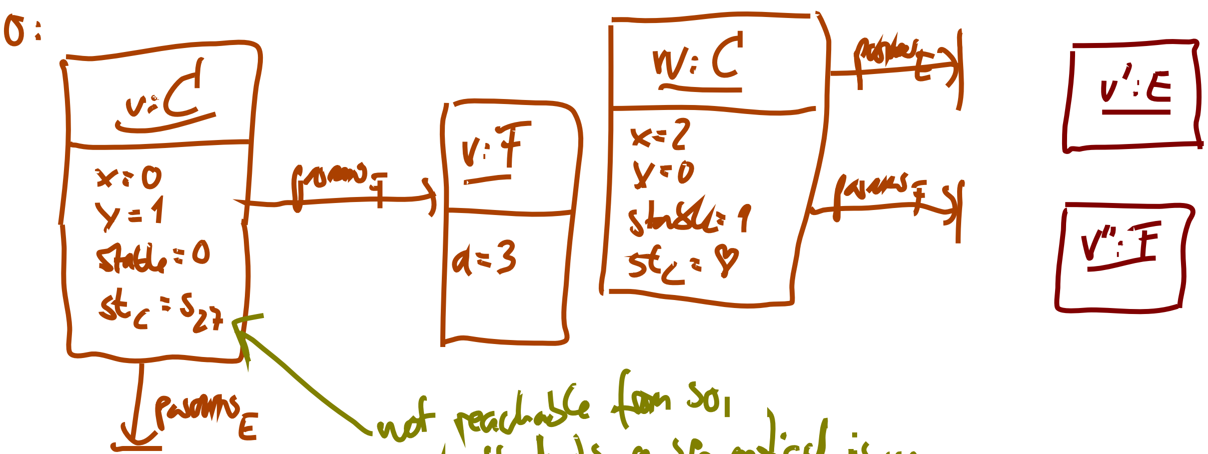
$\mathcal{S} = (\{Int, S_{M_C}\}, \{C, E, F\}, \{x, y, a : Int, stable : \overset{Bool}{Bool}, st_C : S_{M_C}, params_E : E_{0,1}, params_F : F_{0,1}\},$
$\{C \mapsto \{x, y, stable, st_C, params_E, params_F\}, E \mapsto \emptyset, F \mapsto \{a\}\}, \{E, F\})$

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

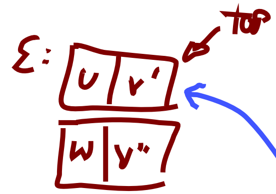
$\mathcal{D}(S_{M_C}) = \{s_0, s_1, s_2, ♡, s_{27}\}$

if Eth is shared FIFO queue

σ :

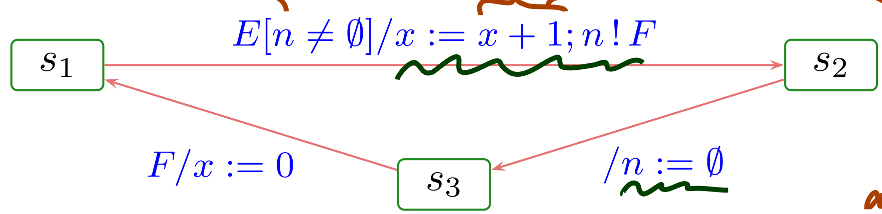


not reachable from s_0, but that is a semantical issue



"v'' (an instance of F) is ready for w"

Where are we?



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

abbrev. this.x + 1

here

- **Wanted**: a labelled transition relation

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

  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"**.

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

$\dot\cup\{\bot\})$

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.

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 $\mathscr{S}$

Transformer: Skip

abstract syntax **concrete syntax**

`skip` *skip*

intuitive semantics

do nothing

well-typedness

./.

semantics

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

observables

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

(error) conditions

Transformer: Update

abstract syntax

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

concrete syntax

$expr_1.v := expr_2$

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 $v : \tau \in atr(C)$; $\quad expr_2 : \tau$;

$expr_1, expr_2$ obey visibility and navigability

semantics

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

where $\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[\![expr_2]\!](\sigma, \beta)]]$ with

$u = I[\![expr_1]\!](\sigma, \beta)$, $\beta = \{\text{this} \mapsto u_x\}$.

observables

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

(error) conditions

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

Update Transformer Example

other example: $\texttt{this.n.x := x+1;}$ with $\underbrace{\texttt{this}}_{u_x}$ and $\texttt{this.n}$ = $expr_1$

$\mathcal{SM}_C$:

$$s_1 \xrightarrow{\;/\underbrace{x := x + 1}\;} s_2$$

$\underbrace{\texttt{this.x}}_{expr_1} := \underbrace{x+1}_{expr_2}$

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

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

σ:

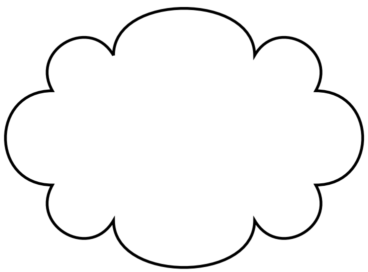
$\underline{u_1 : C}$
$x = 4$
$y = 0$

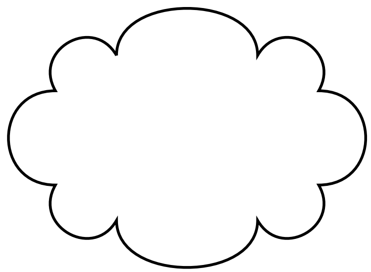
$\longrightarrow$

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

$:\sigma'$

ε:





$:\varepsilon'$

Transformer: Send

abstract syntax

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

concrete syntax

$expr_{dst}\ !\ E(expr_1, \dots, expr_n)$

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

don't send to signal instances

$$expr_{dst} : \tau_D,\ C, D \in \mathscr{C} \setminus \mathscr{E};\ E \in \mathscr{E};$$

$$atr(E) = \{v_1 : \tau_1, \dots, v_n : \tau_n\};\ expr_i : \tau_i,\ 1 \leq i \leq n;$$

all expressions obey visibility and navigability in C

semantics

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

the new signal instance

where $\sigma' = \sigma \,\dot\cup\, \{u \mapsto \{v_i \mapsto d_i \mid 1 \leq i \leq n\}\}$; $\varepsilon' = \varepsilon \oplus (u_{dst}, u)$;
if $u_{dst} = I[\![expr_{dst}]\!](\sigma, \beta) \in \mathrm{dom}(\sigma)$; $d_i = I[\![expr_i]\!](\sigma, \beta)$ for $1 \leq i \leq n$;
$u \in \mathscr{D}(E)$ 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 $u_{dst} \notin \mathrm{dom}(\sigma)$; $\beta = \{\mathsf{this} \mapsto u_x\}$.

observables

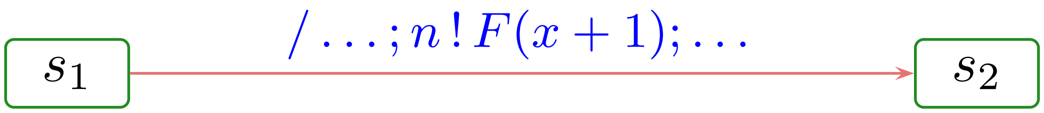
$$Obs_{\mathtt{send}}[u_x] = \{(u_x, u, \underbrace{(E, d_1, \dots, d_n)}_{\text{an event}}, u_{dst})\}$$

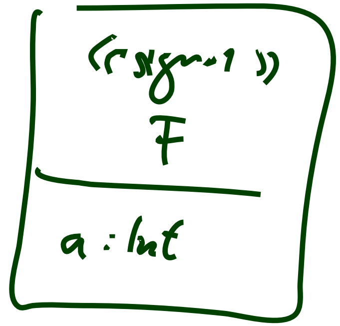
(error) conditions

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

Send Transformer Example

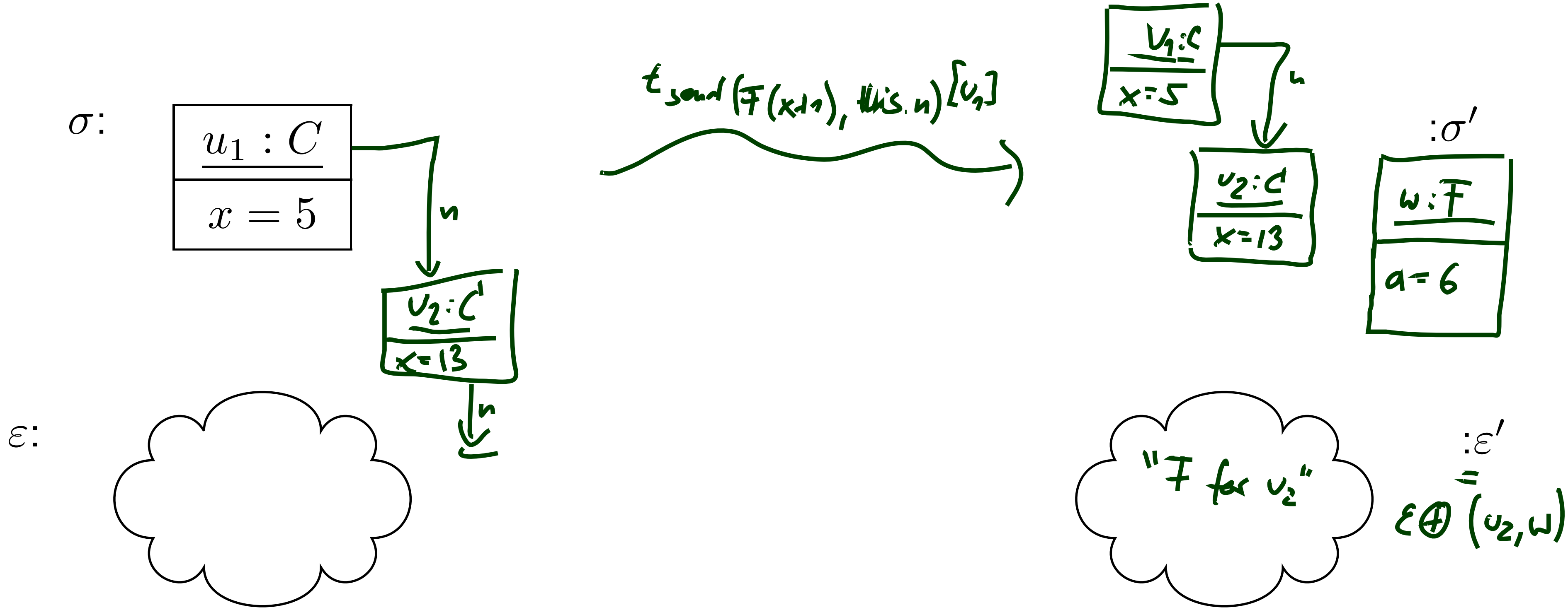
$\mathcal{SM}_C$:





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

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



Transformer: Create

(*) so not: $x := (\text{new } C).x + (\text{new } C).y$

if needed:

$tmp_1 := \text{new } C;$
$tmp_2 := \text{new } C;$
$x := tmp_1.x + tmp_2.y$
$tmp_1 := \text{NULL};$
$tmp_2 := \text{NULL};$

abstract syntax | **concrete syntax**

$\texttt{create}(C, expr, v)$ | $expr.v := \text{new } C$

intuitive semantics

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

well-typedness

$$expr : \tau_D,\ v \in atr(D),\ atr(C) = \{\langle v_1 : \tau_1, expr_i^0 \rangle \mid 1 \le i \le n\}$$

semantics

$$\ldots$$

observables

$$\ldots$$

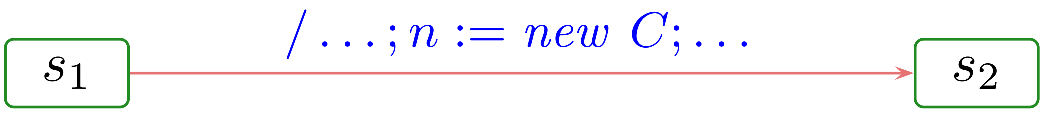
(error) conditions

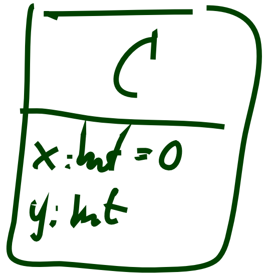
$I[\![expr]\!](\sigma, \beta)$ not defined.

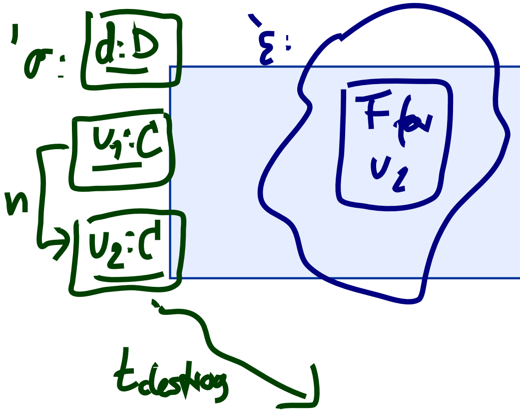
- We use an "and assign"-action for simplicity — it doesn't add or remove expressive power, but moving creation to the expression language raises all kinds of other problems such as order of evaluation (and thus creation). (*)
- Also for simplicity: no parameters to construction ($\sim$ parameters of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

Create Transformer Example

$\mathcal{SM}_C$:



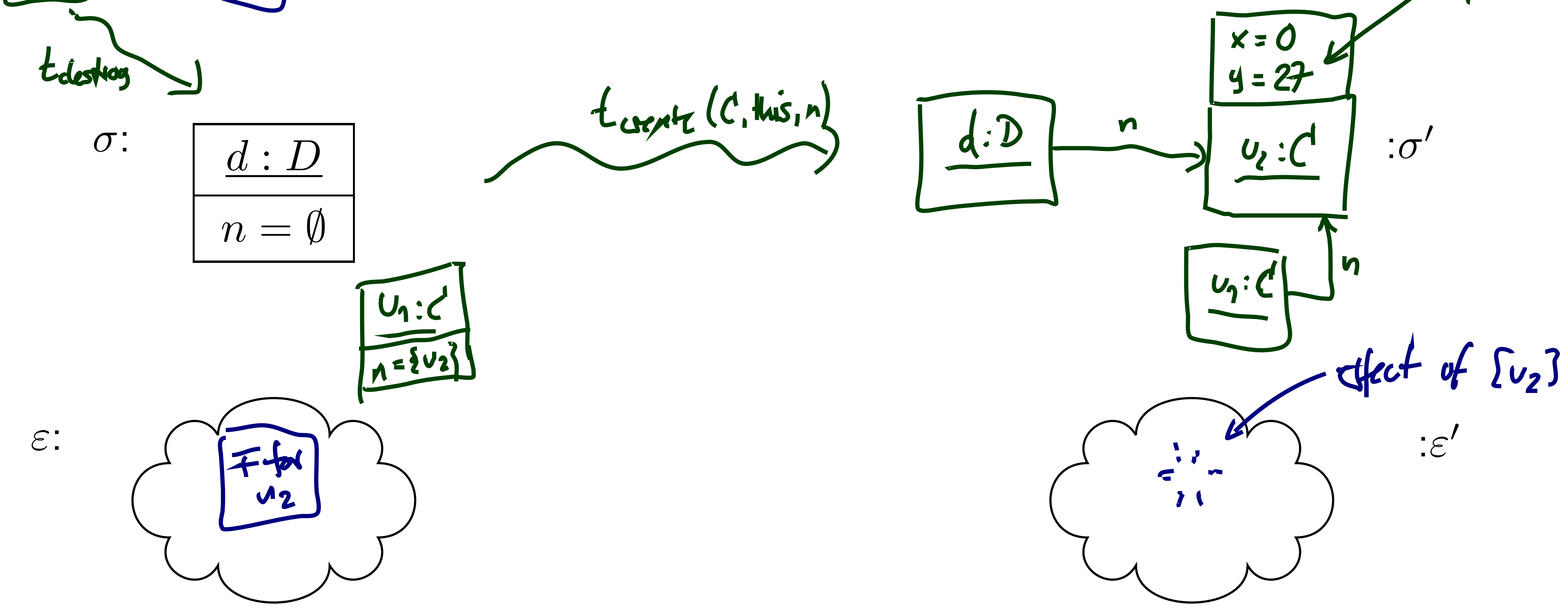




create$(C, expr, v)$

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

non-det. pick



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

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

concrete syntax

intuitive semantics

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

well-typedness

$$expr : \tau_D,\ v \in atr(D),\ atr(C) = \{\langle v_1 : \tau_1, expr_i^0\rangle \mid 1 \le i \le n\},\ C \notin \mathscr{E}$$

semantics

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

iff $\sigma' = \underbrace{\sigma[u_0 \mapsto \sigma(u_0)[v \mapsto u]]}_{\text{updating } v \text{ in } u_0 \text{ in } \sigma} \cup \underbrace{\{u \mapsto \{v_i \mapsto d_i \mid 1 \le i \le n\}\}}_{\text{add new object to } \sigma}$,

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

cleanup operation

$u_0 = I[\![expr]\!](\sigma, \beta)$; $d_i = I[\![expr_i^0]\!](\sigma, \beta)$ if $expr_i^0 \neq$ '' and arbitrary value from $\mathscr{D}(\tau_i)$ otherwise; $\beta = \{\text{this} \mapsto u_x\}$.

observables

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

(error) conditions

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

Transformer: Destroy

abstract syntax

$\texttt{destroy}(expr)$

concrete syntax

delete expr

intuitive semantics

Destroy the object denoted by expression $expr$.

well-typedness

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

semantics

$$\ldots$$

observables

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

(error) conditions

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

Destroy Transformer Example

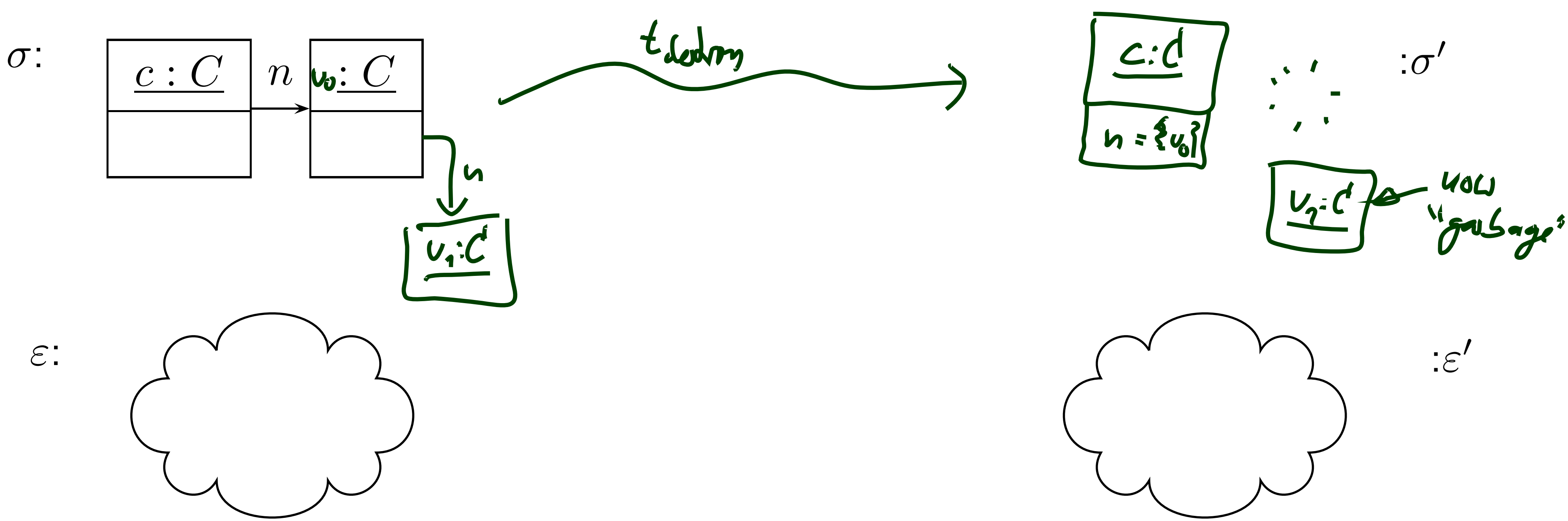
$\mathcal{SM}_C$:

$(\sigma'', \varepsilon'') = (\sigma', \varepsilon')$

$t_{destroy(n)}$

destroy($expr$)

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



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?
- 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

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

semantics

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

function restriction

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

observables

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

(error) conditions

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

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.

$x := x+1;\ n.y := 27;\ n!F$

$t_{update}(t_{update}(t_{send}[\sim](\sigma, \varepsilon)))$

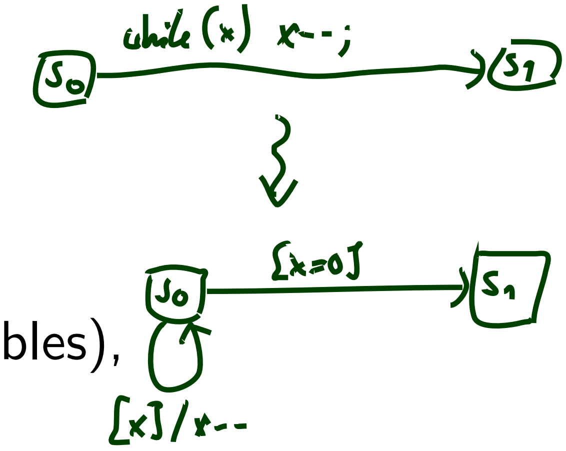
Transformers And Denotational Semantics

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,

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.

Run-to-completion Step

Transition Relation, Computation

Definition. Let A be a set of **actions** and S a (not necessarily finite) set of of **states**.

We call

$$\rightarrow \; \subseteq S \times A \times S$$

a (labelled) **transition relation**.

Let $S_0 \subseteq S$ be a set of **initial states**. A sequence

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots$$

with $s_i \in S$, $a_i \in A$ is called **computation** of the **labelled transition system** $(S, \rightarrow, S_0)$ if and only if

- **initiation**: $s_0 \in S_0$
- **consecution**: $(s_i, a_i, s_{i+1}) \in \rightarrow$ for $i \in \mathbb{N}_0$.

Note: for simplicity, we only consider infinite runs.

Active vs. Passive Classes/Objects

- **Note**: From now on, assume that all classes are **active** for simplicity.

  We'll later briefly discuss the Rhapsody framework which proposes a way how to integrate non-active objects.

- **Note**: The following RTC "algorithm" follows [Harel and Gery, 1997] (i.e. the one realised by the Rhapsody code generation) where the standard is ambiguous or leaves choices.

From Core State Machines to LTS

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

We say, the state machines induce the following labelled transition relation on states $S := (\Sigma_{\mathscr{S}}^{\mathscr{D}} \,\dot\cup\, \{\#\} \times Eth)$ with actions $A := \left(2^{\mathscr{D}(\mathscr{C}) \times (\mathscr{D}(\mathscr{E}) \,\dot\cup\, \{\perp\}) \times Evs(\mathscr{E},\mathscr{D}) \times \mathscr{D}(\mathscr{C})}\right)^2$:

$\mathscr{D}(\mathscr{C}) \times$

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

  if and only if

  (i) an event with destination u is discarded,

  (ii) an event is dispatched to u, i.e. stable object processes an event, or

  (iii) run-to-completion processing by u commences, i.e. object u is not stable and continues to process an event,

  (iv) the environment interacts with object u,

- $s \xrightarrow{(cons, \emptyset)} \#$ ← "error"

  if and only if

  (v) $s = \#$ and $cons = \emptyset$, or an error condition occurs during consumption of $cons$.

(i) Discarding An Event

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

if

- an E-event (instance of signal E) is ready in ε for object u of a class $\mathscr{C}$, i.e. if

$$u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C) \wedge \exists\, u_E \in \mathscr{D}(\mathscr{E}) : u_E \in ready(\varepsilon, u)$$

- u is stable and in state machine state s, i.e. $\sigma(u)(stable) = 1$ and $\sigma(u)(st) = s$,
- but there is no corresponding transition enabled (all transitions incident with current state of u either have other triggers or the guard is not satisfied)

$$\forall\, (s, F, expr, act, s') \in \rightarrow (\mathcal{SM}_C) : F \neq E \vee I[\![expr]\!](\tilde{\sigma}) = 0$$

! with $\tilde{\sigma}$: see slide 30

and

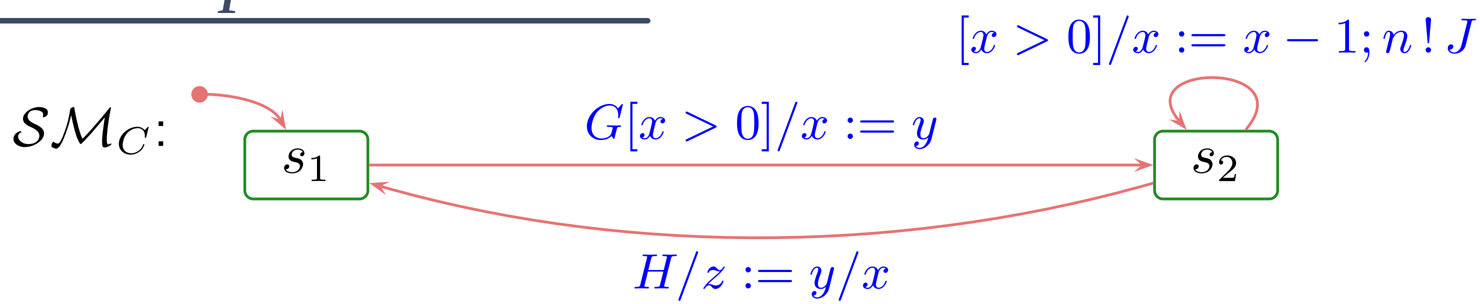
- the system configuration doesn't change, i.e. $\sigma' = \sigma$
- the event u_E is removed from the ether, i.e.

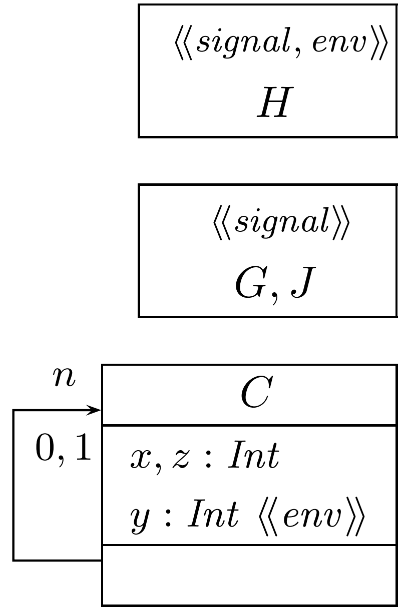
$$\varepsilon' = \varepsilon \ominus u_E,$$

- consumption of u_E is observed, i.e.

$$cons = \{(u, \overbrace{(E, \sigma(u_E))}^{u_E})\}, Snd = \emptyset.$$

Example: Discard





σ:

$\underline{c : C}$
$x = 1, z = 0, y = 2$
$st = s_1$
$stable = 1$

$\xrightarrow{c}$ $\sigma' = \sigma$

ε: J for c, G for c

$\varepsilon' = \varepsilon \ominus u_J$

G for c

- $\exists\, u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C)$
  $\exists\, u_E \in \mathscr{D}(\mathscr{E}) : u_E \in ready(\varepsilon, u)$
- $\forall\,(s, F, expr, act, s') \in \rightarrow (\mathcal{SM}_C)$:
  $F \neq E \vee I[\![expr]\!](\sigma) = 0$
- $\sigma(u)(stable) = 1,\ \sigma(u)(st) = s,$
- $\sigma' = \sigma,\ \varepsilon' = \varepsilon \ominus u_E$
- $cons = \{(u, (E, \sigma(u_E)))\},\ Snd = \emptyset$

(ii) Dispatch

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

- $u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C) \wedge \exists\, u_E \in \mathscr{D}(\mathscr{E}) : u_E \in \mathit{ready}(\varepsilon, u)$
- u is stable and in state machine state s, i.e. $\sigma(u)(\mathit{stable}) = 1$, and $\sigma(u)(st) = s$,
- a transition is enabled, i.e.

  $$\exists\, (s, F, \mathit{expr}, \mathit{act}, s') \in \to (\mathcal{SM}_C) : F = E \wedge I[\![\mathit{expr}]\!](\tilde{\sigma}) = 1$$

  where $\tilde{\sigma} = \sigma[u.\mathit{params}_E \mapsto u_E]$.

and

- (σ', ε') results from applying t_{act} to (σ, ε) and removing u_E from the ether, i.e.

  $$(\sigma'', \varepsilon') = t_{act}(\tilde{\sigma}, \varepsilon \ominus u_E),$$

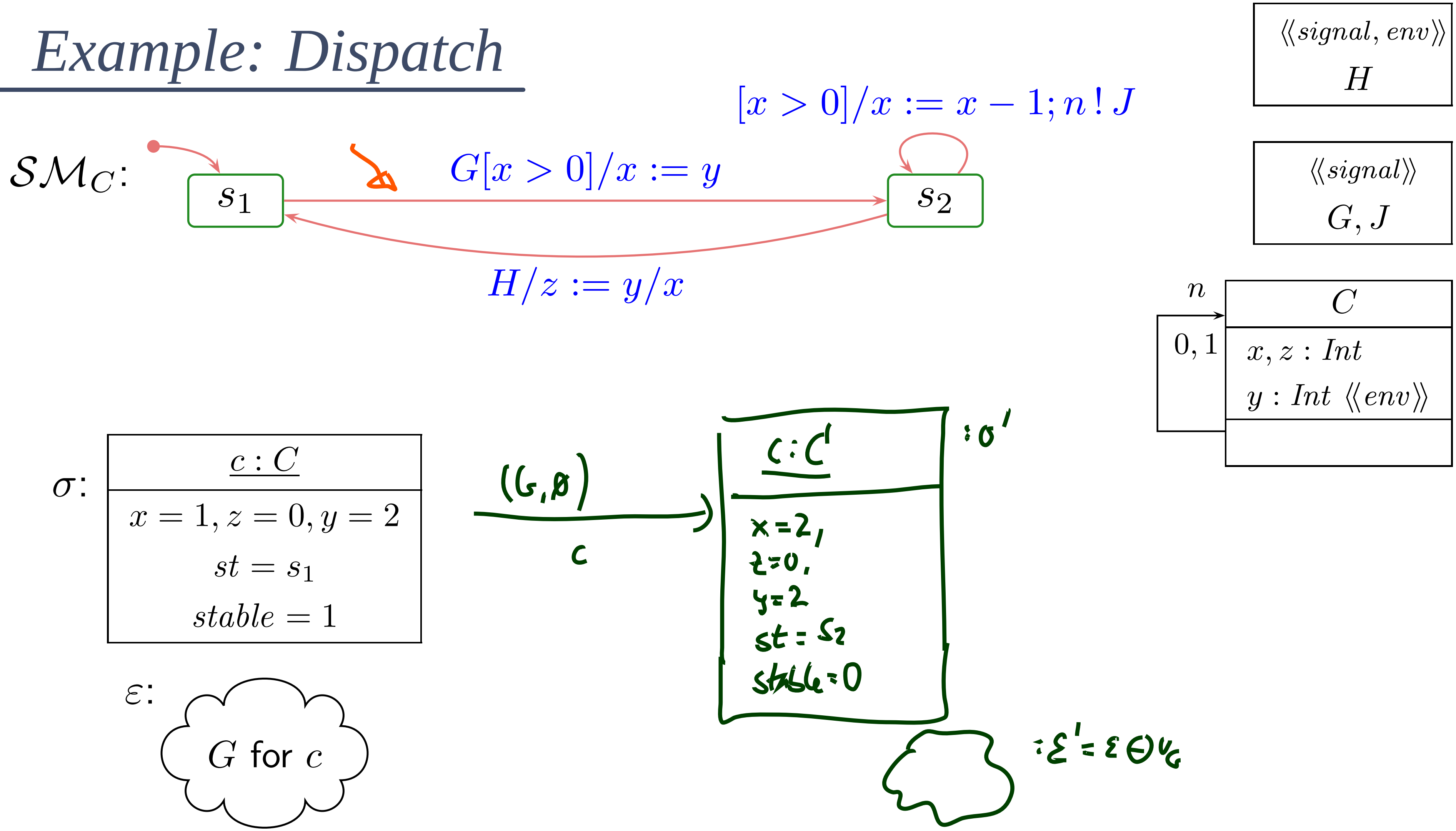
  $$\sigma' = (\sigma''[u.st \mapsto s', u.\mathit{stable} \mapsto b, u.\mathit{params}_E \mapsto \emptyset])|_{\mathscr{D}(\mathscr{C}) \setminus \{u_E\}}$$

  remove signal instance

  where b **depends**:
  - If u becomes stable in s', then $b = 1$. It **does** become stable if and only if there is no transition **without trigger** enabled for u in (σ', ε').
  - Otherwise $b = 0$.
- Consumption of u_E and the side effects of the action are observed, i.e.

  $$\mathit{cons} = \{(u, (E, \sigma(u_E)))\}, \mathit{Snd} = \mathit{Obs}_{t_{act}}(\tilde{\sigma}, \varepsilon \ominus u_E).$$

Example: Dispatch



- $\exists\, u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C)$
  $\exists\, u_E \in \mathscr{D}(\mathscr{E}) : u_E \in ready(\varepsilon, u)$
- $\exists\, (s, F, expr, act, s') \in \to (\mathcal{SM}_C)$:
  $F = E \wedge I[\![expr]\!](\tilde{\sigma}) = 1$
- $\tilde{\sigma} = \sigma[u.params_E \mapsto u_E]$.
- $\sigma(u)(stable) = 1,\ \sigma(u)(st) = s,$
- $(\sigma'', \varepsilon') = t_{act}(\tilde{\sigma}, \varepsilon \ominus u_E)$
- $\sigma' = (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset])|_{\mathscr{D}(\mathscr{C}) \setminus \{u_E\}}$
- $cons = \{(u, (E, \sigma(u_E)))\},\ Snd = Obs_{t_{act}}(\tilde{\sigma}, \varepsilon \ominus u_E)$

(iii) Commence Run-to-Completion

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

if

- there is an unstable object u of a class $\mathscr{C}$, i.e.

$$u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C) \wedge \sigma(u)(stable) = 0$$

- there is a transition without trigger enabled from the current state $s = \sigma(u)(st)$, i.e.

$$\exists\, (s, \_, expr, act, s') \in \longrightarrow (\mathcal{SM}_C) : I[\![expr]\!](\sigma) = 1$$

and

- (σ', ε') results from applying t_{act} to (σ, ε), i.e.

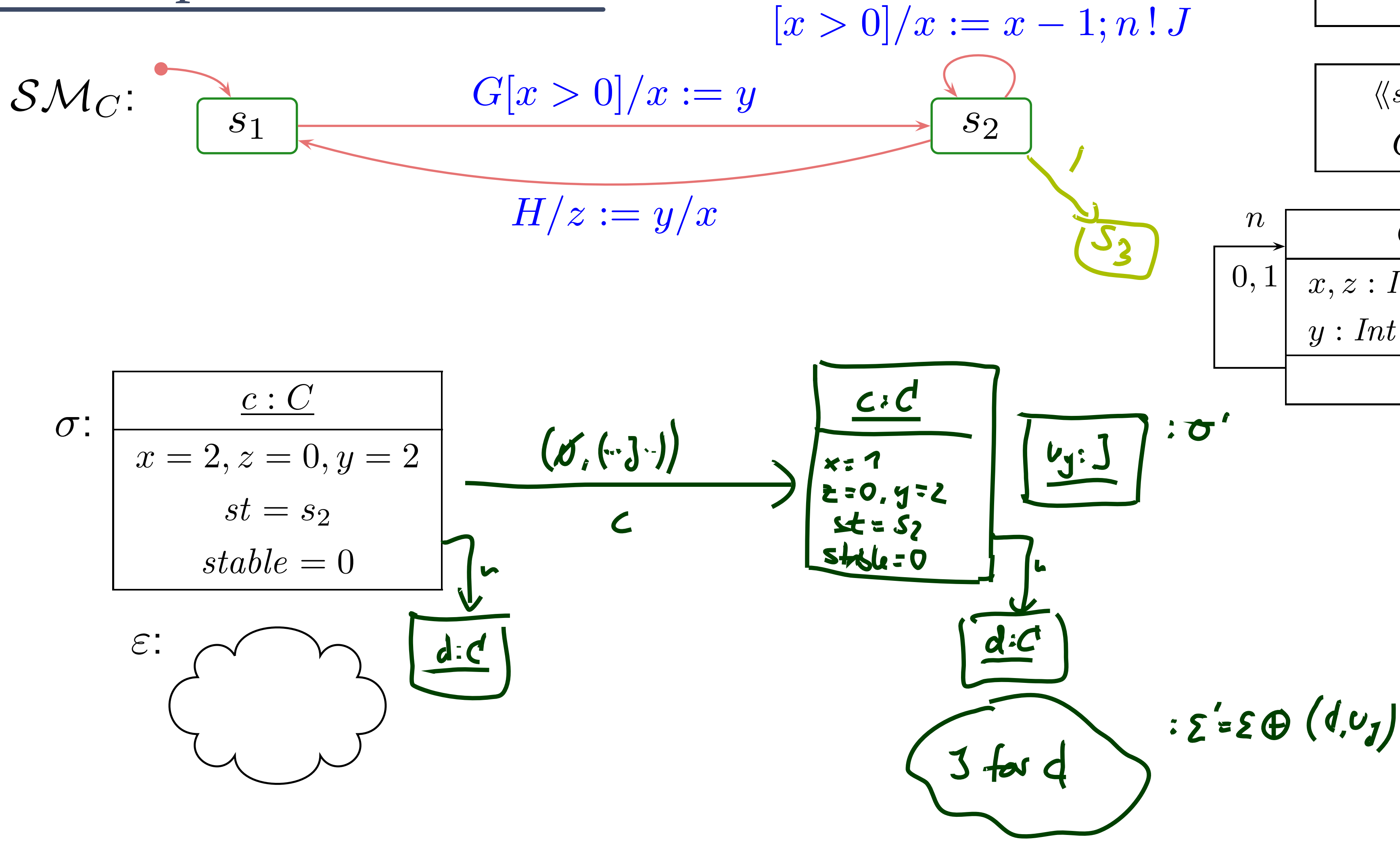
$$(\sigma'', \varepsilon') \in t_{act}[u](\sigma, \varepsilon), \quad \sigma' = \sigma''[u.st \mapsto s', u.stable \mapsto b]$$

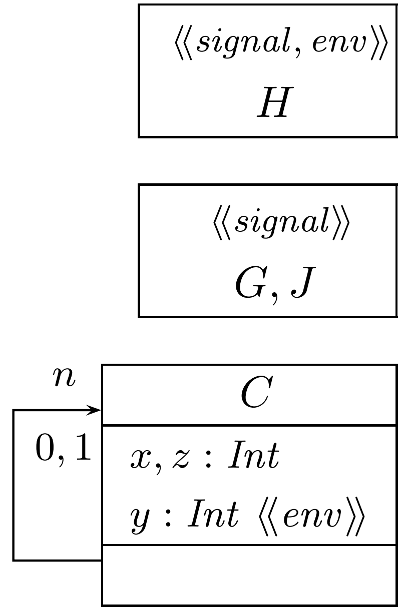
  where b **depends** as before.

- Only the side effects of the action are observed, i.e.

$$cons = \emptyset, Snd = Obs_{t_{act}}(\sigma, \varepsilon).$$

Example: Commence





- $\exists\, u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C) : \sigma(u)(stable) = 0$
- $\exists\, (s, \_, expr, act, s') \in \longrightarrow (\mathcal{SM}_C) : I[\![expr]\!](\sigma) = 1$
- $\sigma(u)(stable) = 1,\ \sigma(u)(st) = s,$
- $(\sigma'', \varepsilon') = t_{act}(\sigma, \varepsilon),$
  $\sigma' = \sigma''[u.st \mapsto s', u.stable \mapsto b]$
- $cons = \emptyset,\ Snd = Obs_{t_{act}}(\sigma, \varepsilon)$

(iv) Environment Interaction

Assume that a set $\mathscr{E}_{env} \subseteq \mathscr{E}$ is designated as **environment** signal/ ~~**events**~~ and a set of attributes $v_{env} \subseteq V$ is designated as **input attributes**.

Then

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

if

- an environment event $E \in \mathscr{E}_{env}$ is spontaneously sent to an alive object $u \in \mathscr{D}(\sigma)$, i.e.

  $$\sigma' = \sigma \mathbin{\dot\cup} \overbrace{\{u_E \mapsto \{v_i \mapsto d_i \mid 1 \leq i \leq n\}}^{\text{one new instance of } E}, \quad \varepsilon' = \varepsilon \oplus u_E$$

  where $u_E \notin \mathrm{dom}(\sigma)$ and $atr(E) = \{v_1, \ldots, v_n\}$.
- Sending of the event is observed, i.e. $cons = \emptyset$, $Snd = \{(env, E(\vec{d}))\}$.

or

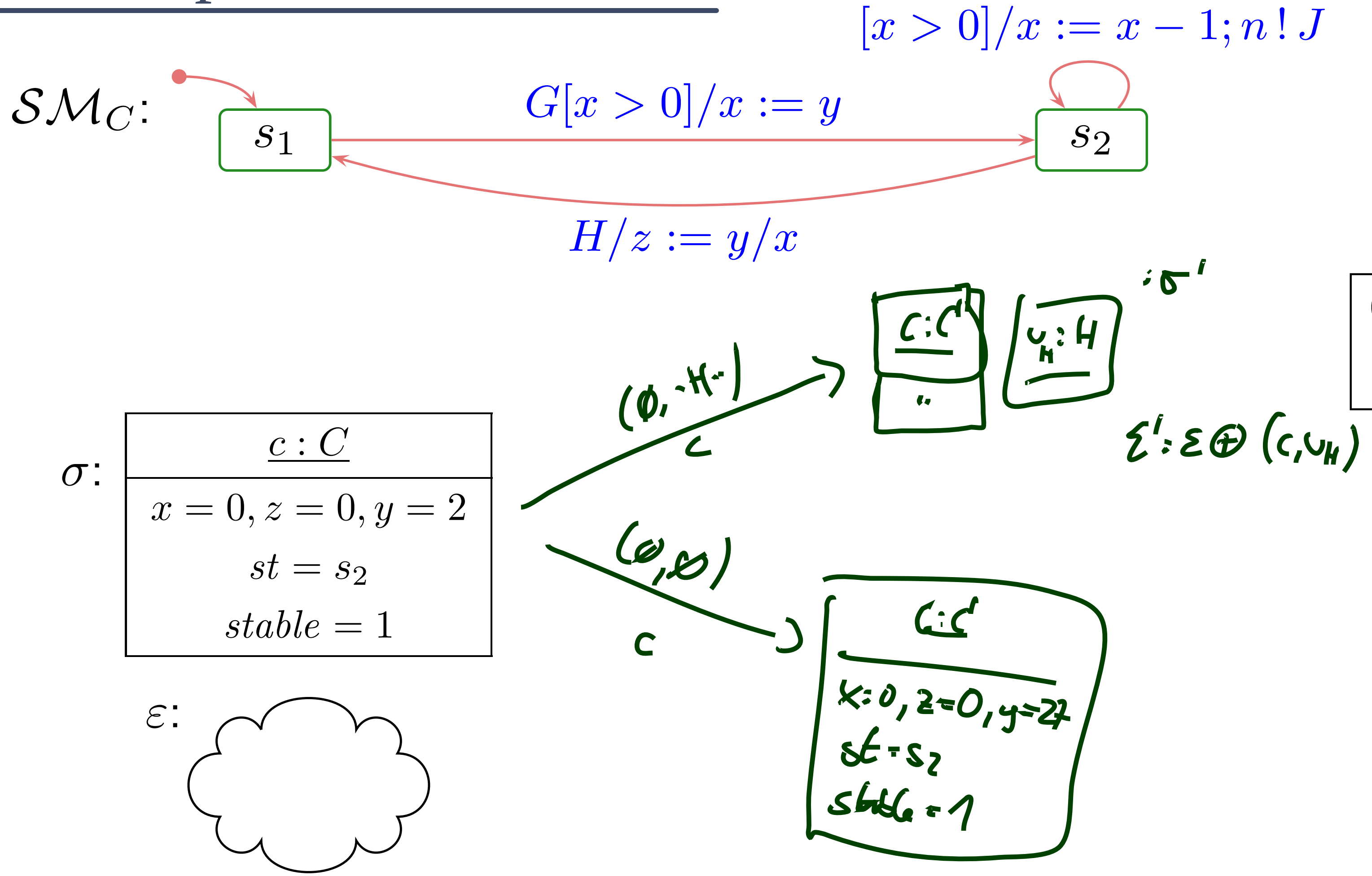
non-det.

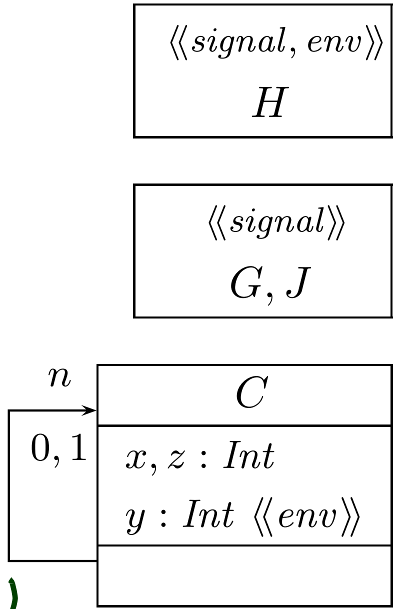
- Values of input attributes change freely in alive objects, i.e.

  $$\forall\, v \in V\ \forall\, u \in \mathrm{dom}(\sigma) : \sigma'(u)(v) \neq \sigma(u)(v) \implies v \in V_{env}.$$

  and no objects appear or disappear, i.e. $\mathrm{dom}(\sigma') = \mathrm{dom}(\sigma)$.
- $\varepsilon' = \varepsilon$.

Example: Environment





- $\sigma' = \sigma \mathbin{\dot\cup} \{u_E \mapsto \{v_i \mapsto d_i \mid 1 \leq i \leq n\}$
- $\varepsilon' = \varepsilon \oplus u_E$ where $u_E \notin \operatorname{dom}(\sigma)$ and $atr(E) = \{v_1, \ldots, v_n\}$.
- $u \in \operatorname{dom}(\sigma)$
- $cons = \emptyset$, $Snd = \{(env, E(\vec{d}))\}$.

(v) Error Conditions

$$s \xrightarrow[u]{(cons,Snd)} \#$$

if, in (i), (ii) or (iii), (*)

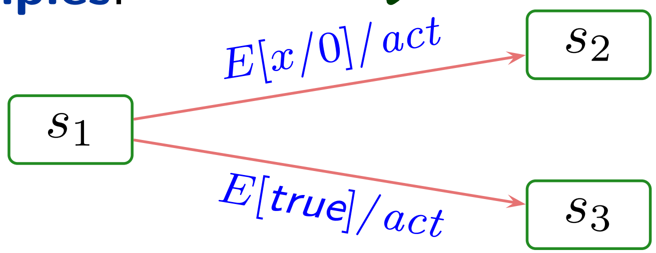
- $I[\![expr]\!]$ is not defined for σ, or
- t_{act} is not defined for (σ, ε), i.e. $t_{act}[u](\sigma,\varepsilon) = \emptyset$

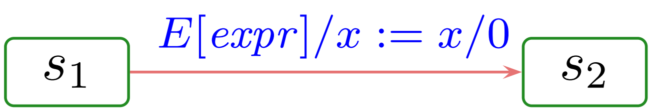
and

- consumption **is observed** according to (i), (ii) or (iii), (*) but $Snd = \emptyset$.

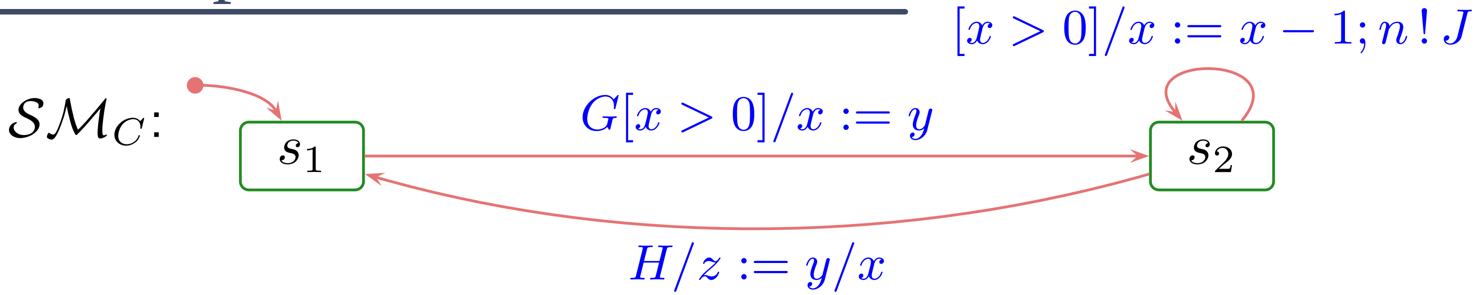
if we add . (*), this is leading to # with F ready

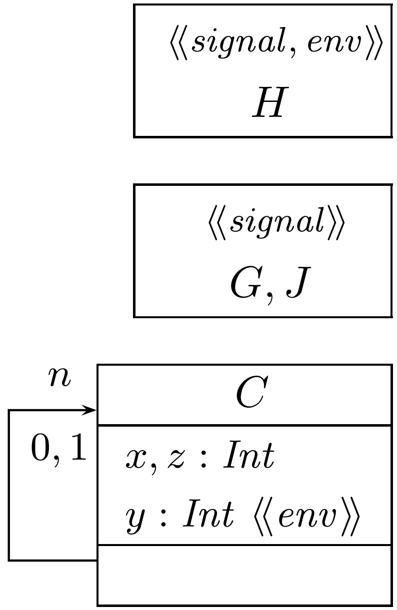
Examples:

- 

- 

Example: Error Condition



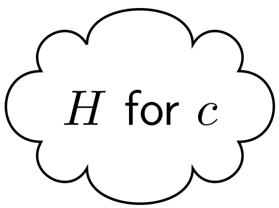


σ:

$\underline{c : C}$
$x = 0, z = 0, y = 27$ $st = s_2$ $stable = 1$

$\xrightarrow[c]{(H,\emptyset)}$ #

ε:



- $I[\![expr]\!]$ not defined for σ, or
- t_{act} is not defined for (σ, ε)
- consumption according to (ii) or (iii)
- $Snd = \emptyset$

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.

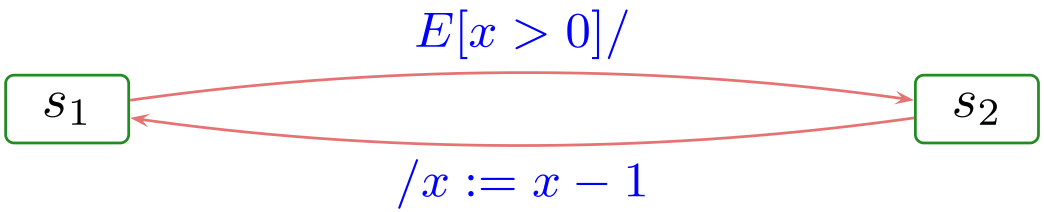
Notions of Steps: The Run-to-Completion Step

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

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.