

Formal Specification and Verification

Formal specification (2)

3.06.2014

Viorica Sofronie-Stokkermans

e-mail: sofronie@uni-koblenz.de

Until now

- Logic
- Formal specification (generalities)

Algebraic specification

Transition systems

Formal specification

- **Specification languages for describing programs/processes/systems**

Model based specification

transition systems, abstract state machines, specifications based on set theory

Axiom-based specification

algebraic specification

Declarative specifications

logic based languages (Prolog)

functional languages, λ -calculus (Scheme, Haskell, OCaml, ...)

rewriting systems (very close to algebraic specification): ELAN, SPIKE, ...

- **Specification languages for properties of programs/processes/systems**

Temporal logic

Formal specification

- **Specification languages for describing programs/processes/systems**

Model based specification

transition systems, abstract state machines, specifications based on set theory

Axiom-based specification

algebraic specification

Declarative specifications

logic based languages (Prolog)

functional languages, λ -calculus (Scheme, Haskell, OCaml)

rewriting systems (very close to algebraic specification): ELAN, SPIKE

- **Specification languages for properties of programs/processes/systems**

Temporal logic

Transition systems

- Model to describe the behaviour of systems
- Digraphs where nodes represent states, and edges model transitions
- **State:** Examples
 - the current colour of a traffic light
 - the current values of all program variables + the program counter
 - the current value of the registers together with the values of the input bits
- **Transition** (“state change”): Examples
 - a switch from one colour to another
 - the execution of a program statement
 - the change of the registers and output bits for a new input

Transition systems

Definition.

A transition system TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where:

- S is a set of states
- Act is a set of actions
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation
- $I \subseteq S$ is a set of initial states
- AP is a set of atomic propositions
- $L : S \rightarrow 2^{AP}$ is a labeling function

S and Act are either finite or countably infinite

Notation: $s \xrightarrow{\alpha} s'$ instead of $(s, \alpha, s') \in \rightarrow$.

Direct successors and predecessors

$$Post(s, \alpha) = \{s' \in S \mid s \xrightarrow{\alpha} s'\},$$

$$Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$$

$$Pre(s, \alpha) = \{s' \in S \mid s' \xrightarrow{\alpha} s\},$$

$$Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha)$$

$$Post(C, \alpha) = \bigcup_{s \in C} Post(s, \alpha),$$

$$Post(C) = \bigcup_{\alpha \in Act} Post(C, \alpha) \quad \text{for } C \subseteq S$$

$$Pre(C, \alpha) = \bigcup_{s \in C} Pre(s, \alpha),$$

$$Pre(C) = \bigcup_{\alpha \in Act} Pre(C, \alpha) \quad \text{for } C \subseteq S$$

State s is called **terminal** if and only if $Post(s) = \emptyset$

Non-determinism

Nondeterminism is a feature!

- to model **concurrency by interleaving**
 - no assumption about the relative speed of processes
- to model **implementation freedom**
 - only describes what a system should do, not how
- to model **under-specified** systems, or **abstractions of real systems**
 - use incomplete information

In automata theory, nondeterminism may be exponentially more succinct but that's not the issue here!

Reachable states

Definition. State $s \in S$ is called **reachable** in TS if there exists an initial, finite execution fragment

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} s_n = s$$

$\text{Reach}(TS)$ denotes the set of all reachable states in TS .

Detailed description of states

Variables; Predicates

Program graph representation

Program graphs

A **program graph** PG over set Var of typed variables is a tuple

$$(Loc, Act, Effect, \rightarrow, Loc_0, g_0)$$

where

- Loc is a set of locations with initial locations $Loc_0 \subseteq Loc$
- Act is a set of actions
- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$ is the effect function
- $\rightarrow \subseteq Loc \times (\underbrace{Cond(Var)}_{\text{Boolean conditions on } Var} \times Act) \times Loc$, transition relation
- $g_0 \in Cond(Var)$ is the initial condition.

Notation: $l \xrightarrow{g:\alpha} l'$ denotes $(l, g, \alpha, l') \in \rightarrow$.

From program graphs to transition systems

- Basic strategy: **unfolding**
 - state = location (current control) l + data valuation β (l, β)
 - initial state = initial location + data valuation satisfying the initial condition g_0
- Propositions and labeling
 - propositions: “at l ” and “ $x \in D$ ” for $D \subseteq \text{dom}(x)$
 - $\langle l, \beta \rangle$ is labeled with “at l ” and all conditions that hold in β .
- $l \xrightarrow{g:\alpha} l'$ and g holds in β then $\langle l, \beta \rangle \xrightarrow{\alpha} \langle l', \text{Effect}(\langle l, \beta \rangle) \rangle$

Transition systems for program graphs

The transition system $TS(PG)$ of program graph

$$PG = (Loc, Act, Effect, \rightarrow, Loc_0, g_0)$$

over set Var of variables is the tuple $(S, Act, \rightarrow, I, AP, L)$ where:

- $S = Loc \times Eval(Var)$
- $\rightarrow S \times Act \times S$ is defined by the rule:
If $I \xrightarrow{g:\alpha} I'$ and $\beta \models g$ then $\langle I, \beta \rangle \xrightarrow{\alpha} \langle I', Effect(\langle I, \beta \rangle) \rangle$
- $I = \{ \langle I, \beta \rangle \mid I \in Loc_0, \beta \models g_0 \}$
- $AP = Loc \cup Cond(Var)$ and
- $L(\langle I, \beta \rangle) = \{I\} \cup \{g \in Cond(Var) \mid \beta \models g\}$.

Generalizations of transition systems

- More detailed description of states: Abstract state machines
- Emphasis on processes and their interdependency: CSP
- Durations: Timed automata
- Continuous evolution + discrete control: Hybrid automata

Timed automata

- transition systems + timing constraints

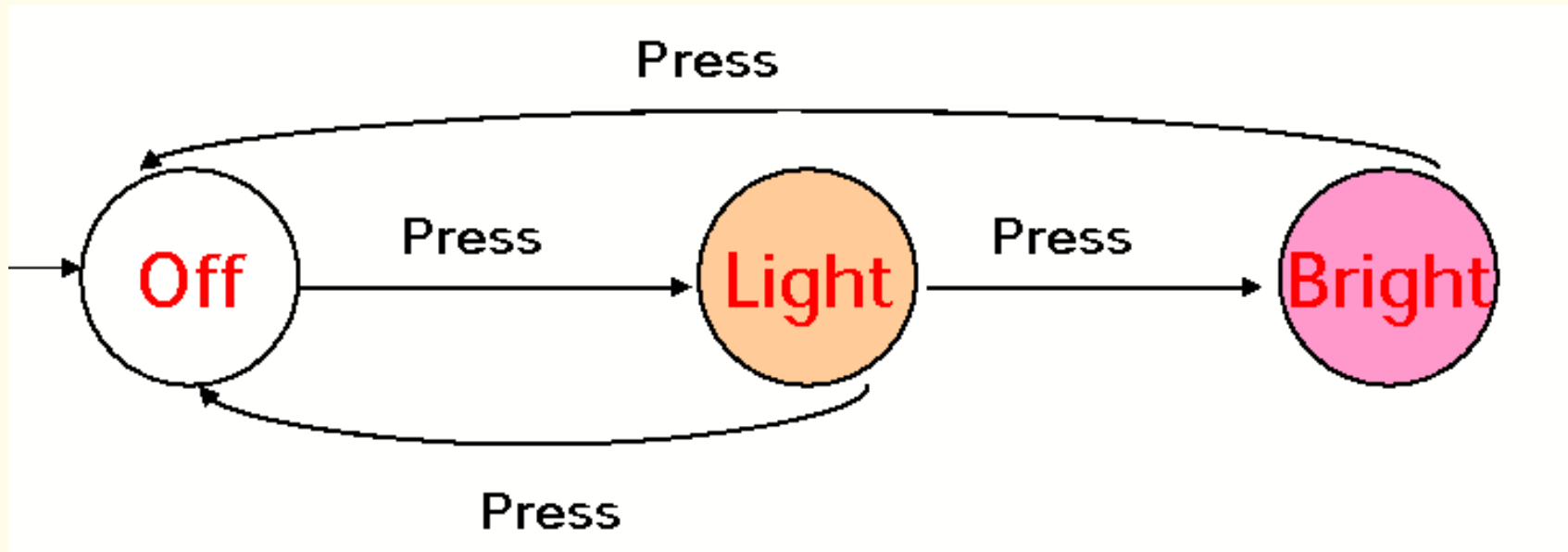
Timed automata

A timed automaton is a finite automaton extended with a finite set of real-valued clocks. During a run of a timed automaton, clock values increase all with the same speed. Along the transitions of the automaton, clock values can be compared to integers. These comparisons form guards that may enable or disable transitions and by doing so constrain the possible behaviors of the automaton. Further, clocks can be reset.

Timed automata can be used to model and analyse the timing behavior of computer systems, e.g., real-time systems or networks.

Timed automata

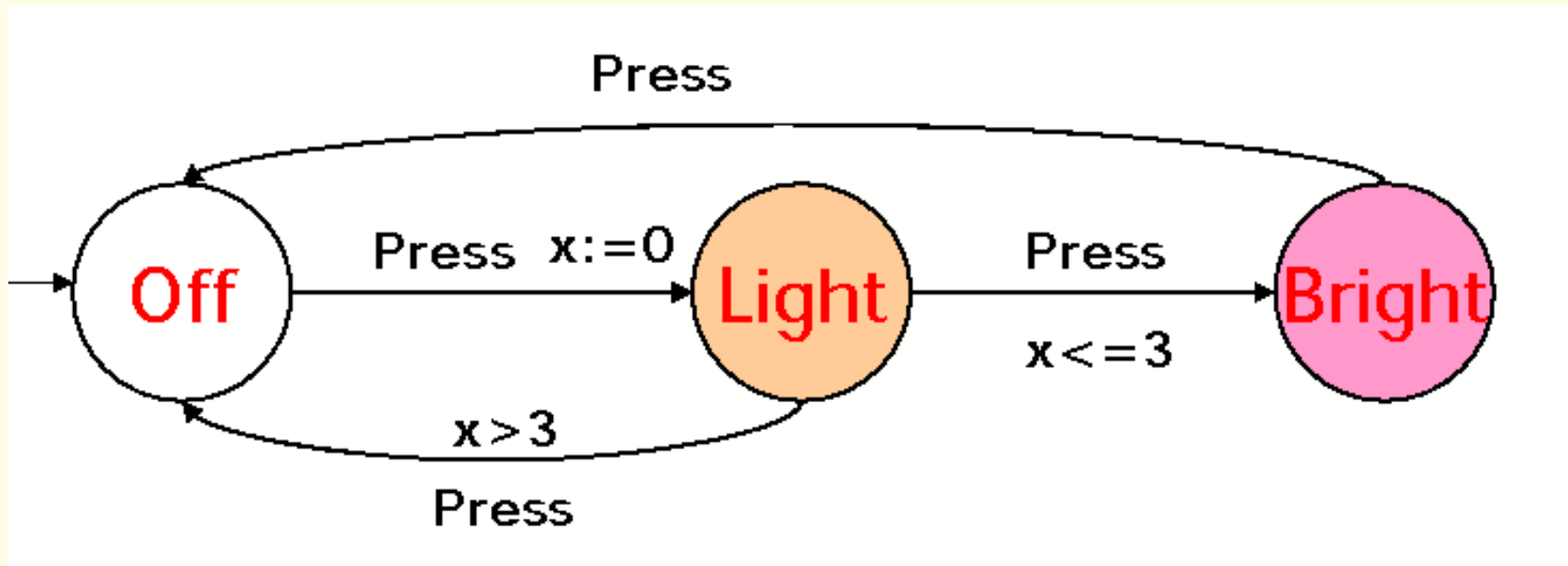
Example: Simple Light Control



WANT: if press is issued twice quickly then the light will get brighter; otherwise the light is turned off.

Timed automata

Example: Simple Light Control



Solution: Add a real-valued clock x

Adding continuous variables to transition systems

Timed automata: Syntax

- A finite set Loc of locations
- A subset $Loc_0 \subseteq Loc$ of initial locations
- A finite set Act of labels (alphabet, actions)
- A finite set X of clocks
- Invariant $Inv(l)$ for each location $l \in Loc$: (clock constraint over X)
- A finite set E of edges. Each edge has:
 - source location l , target location l'
 - label $a \in Act$ (empty labels also allowed)
 - guard g (a clock constraint over X)
 - a subset X' of clocks to be reset

Timed automata: Semantics

For a timed automaton

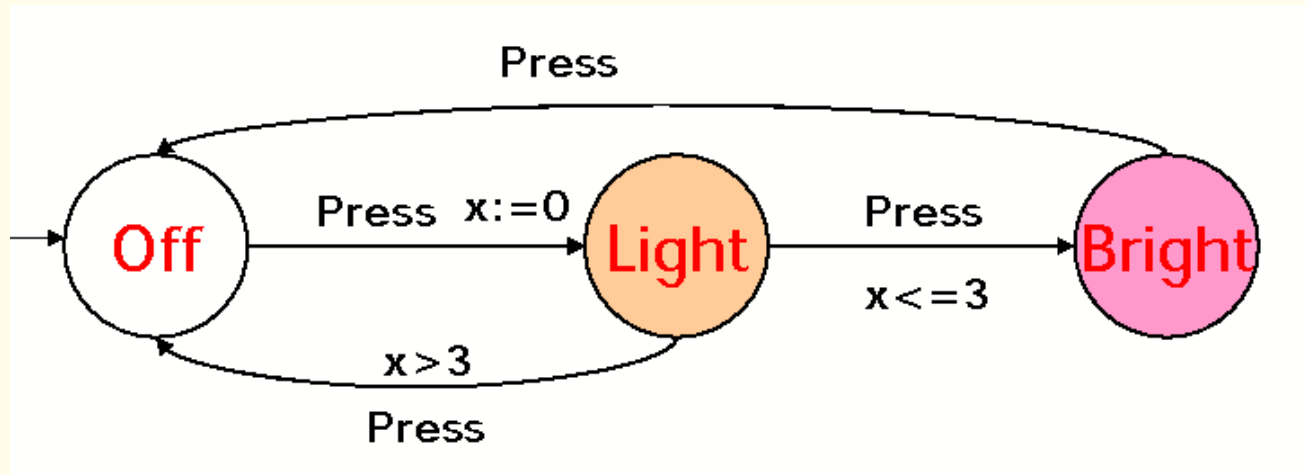
$$A = (Loc, Loc_0, Act, X, \{Inv_l\}_{l \in Loc}, E)$$

define an infinite state transition system $S(A)$:

- **States** S : a state s is a pair (l, v) , where l is a location, and v is a clock vector, mapping clocks in X to \mathbb{R} , satisfying $Inv(l)$
- **Initial States**: (l, v) is initial state if l is in Loc_0 and $v(x) = 0$
- **Elapse of time transitions**: for each nonnegative real number d , $(l, v) \xrightarrow{d} (l, v + d)$ if both v and $v + d$ satisfy $Inv(l)$
- **Location switch transitions**: $(l, v) \xrightarrow{a} (l', v')$ if there is an edge (l, a, g, X', l') such that v satisfies g and $v' = v[\{x \mapsto 0 \mid x \in X'\}]$.

Timed automata

Example: Simple Light Control



Timed automaton:

$Loc = \{Off, Light, Bright\}$, $Loc_0 = \{Off\}$, $Act = \{Press\}$

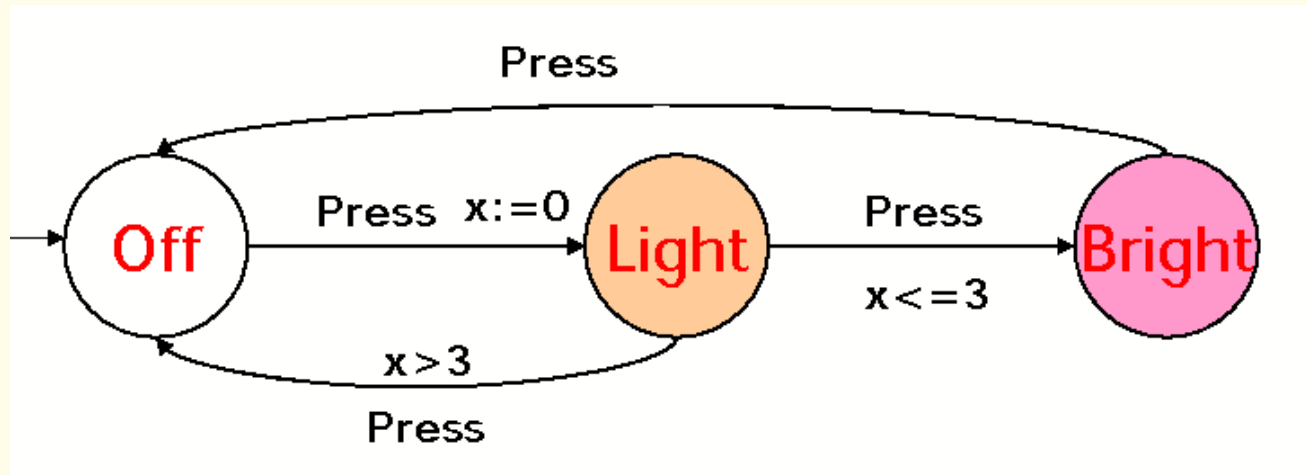
$X = \{x\}$; $Inv(Off) = Inv(Light) = Inv(Bright) = (x \geq 0)$

Edges: $(Off, Press, \top, \{x\}, Light)$, $(Light, Press, x > 3, \emptyset, Off)$

$(Light, Press, x \leq 3, \emptyset, Bright)$, $(Bright, Press, \top, \emptyset, Off)$

Timed automata

Example: Simple Light Control



States: (Off, v), (Light, v), (Bright, v) (v value of clock x).

Initial state: (Off, 0).

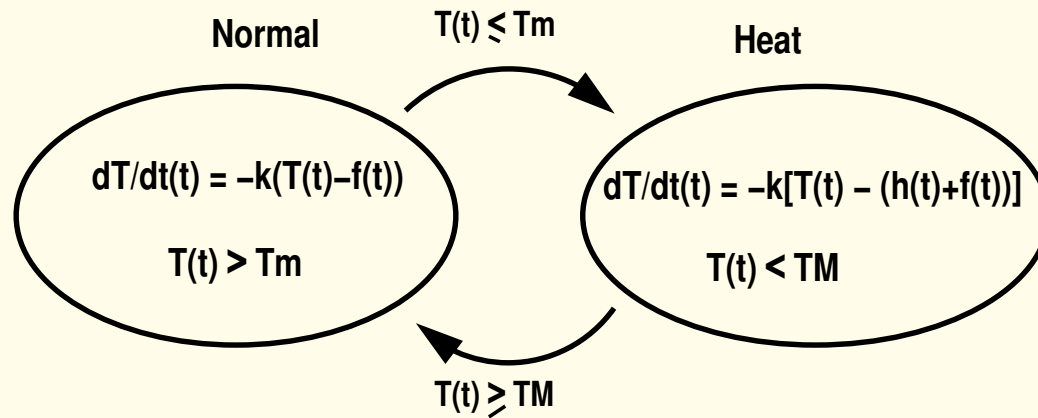
Transitions (Examples)

Elapse of time: (Off, 10) $\xrightarrow{5}$ (Off, 15)

Location switch: (Off, 10) $\xrightarrow{\text{Press}}$ (Light, 0)

Hybrid Automata

Hybrid Automata



$f : \mathbb{R} \rightarrow \mathbb{R}$ evolution of external temperature

$h : \mathbb{R} \rightarrow \mathbb{R}$ evolution of heater temperature

Hybrid Automata

Hybrid automaton (HA) $S = (X, Q, \text{flow}, \text{Inv}, \text{Init}, E, \text{jump})$ where:

- (1) $X = \{x_1, \dots, x_n\}$ finite set of real valued variables
 Q finite set of control modes
- (2) $\{\text{flow}_q \mid q \in Q\}$ specify the continuous dynamics in each control mode
(flow_q predicate over $\{x_1, \dots, x_n\} \cup \{\dot{x}_1, \dots, \dot{x}_n\}$).
- (3) $\{\text{Inv}_q \mid q \in Q\}$ mode invariants (predicates over X).
- (4) $\{\text{Init}_q \mid q \in Q\}$ initial states for control modes (predicates over X).
- (5) E : control switches (finite multiset with elements in $Q \times Q$).
- (6) $\{\text{guard}_e \mid e \in E\}$ guards for control switches (predicates over X).
- (7) Jump conditions $\{\text{jump}_e \mid e \in E\}$, (predicates over $X \cup X'$), where $X' = \{x'_1, \dots, x'_n\}$ is a copy of X consisting of “primed” variables.

Linear Hybrid Automata

Atomic linear predicate: linear inequality (e.g. $3x_1 - x_2 + 7x_5 \leq 4$).

Convex linear predicate: finite conjunction of linear inequalities.

A state assertion s for S : family $\{s(q) \mid q \in Q\}$, where $s(q)$ is a predicate over X (expressing constraints which hold in state s for mode q).

Definition [Henzinger 1997] A linear hybrid automaton (LHA) is a hybrid automaton which satisfies the following requirements:

(1) **Linearity:**

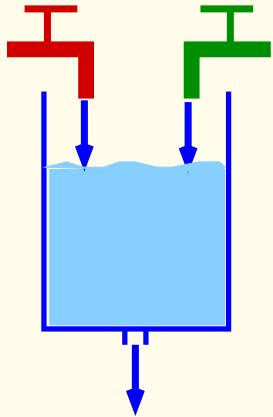
- For every $q \in Q$, flow_q , Inv_q , and Init_q are convex linear predicates.
- For every $e = (q, q') \in E$, jump_e and guard_e are convex linear predicates.

We assume that flow_q are conjunctions of *non-strict* inequalities.

(2) **Flow independence:**

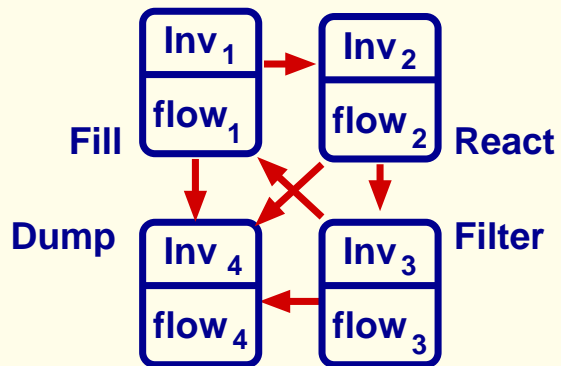
For every $q \in Q$, flow_q is a predicate over \dot{X} only.

Example

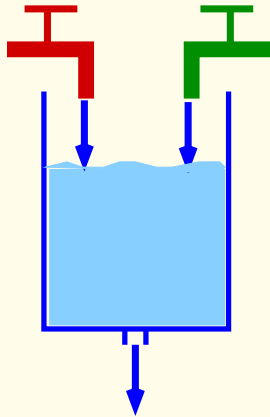


Chemical plant

Two substances are mixed; they react; the resulting product is filtered out; then the procedure is repeated.



Example

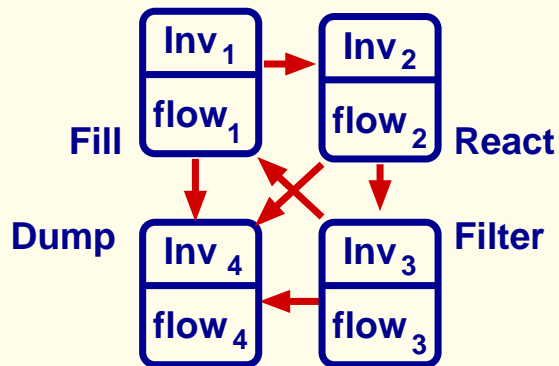


Chemical plant

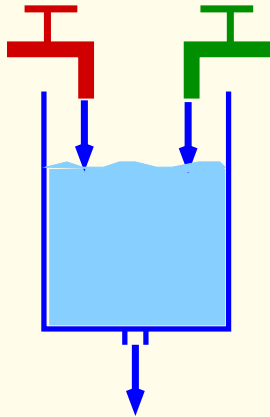
Two substances are mixed; they react; the resulting product is filtered out; then the procedure is repeated.

Check:

- No overflow
- Substances in the right proportion
- If substances in wrong proportion, tank can be drained in ≤ 200 s.



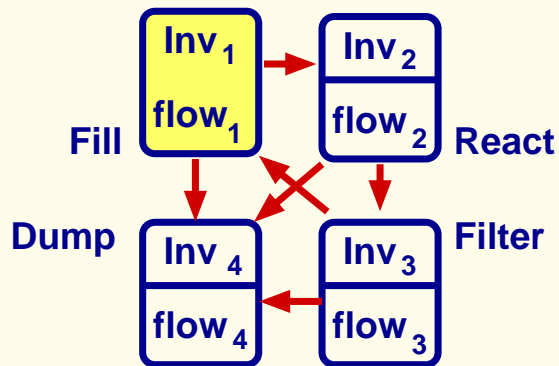
Example



Mode 1: Fill Temperature is low, 1 and 2 do not react. Substances 1 and 2 (possibly mixed with a small quantity of 3) are filled in the tank in equal quantities up to a margin of error.

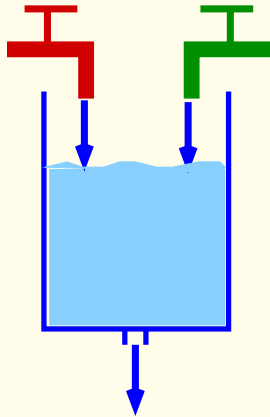
$$\text{Inv}_1 \quad x_1 + x_2 + x_3 \leq L_f \wedge \bigwedge_{i=1}^3 x_i \geq 0 \wedge \\ -\epsilon_a \leq x_1 - x_2 \leq \epsilon_a \wedge 0 \leq x_3 \leq \min$$

$$\text{flow}_1 \quad \dot{x}_1 \geq \text{dmin} \wedge \dot{x}_2 \geq \text{dmin} \wedge \dot{x}_3 = 0 \wedge -\delta_a \leq \dot{x}_1 - \dot{x}_2 \leq \delta_a$$



If proportion not kept: system jumps into mode 4 (**Dump**);
If the total quantity of substances exceeds level L_f (tank filled) the system jumps into mode 2 (**React**).

Example



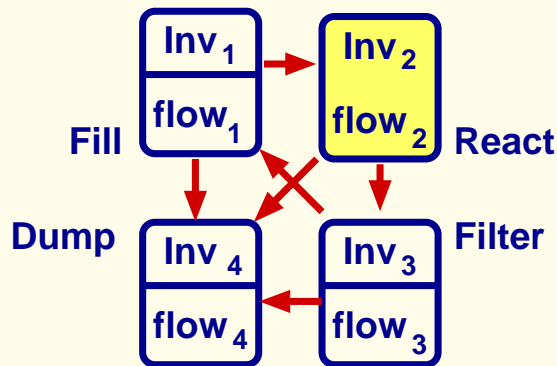
Mode 2: React Temperature is high. Substances 1 and 2 react. The reaction consumes equal quantities of substances 1 and 2 and produces substance 3.

$$\text{Inv}_2 \quad L_f \leq x_1 + x_2 + x_3 \leq L_{\text{overflow}} \wedge \bigwedge_{i=1}^3 x_i \geq 0 \wedge$$

$$-\epsilon_a \leq x_1 - x_2 \leq \epsilon_a \wedge 0 \leq x_3 \leq \max$$

$$\text{flow}_2 \quad \dot{x}_1 \leq -d_{\min} \wedge \dot{x}_2 \leq -d_{\min} \wedge \dot{x}_3 \geq d_{\min}$$

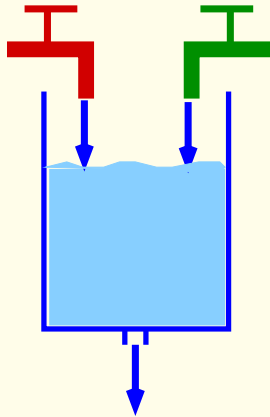
$$\wedge \dot{x}_1 = \dot{x}_2 \wedge \dot{x}_3 + \dot{x}_1 + \dot{x}_2 = 0$$



If the proportion between substances 1 and 2 is not kept the system jumps into mode 4 (**Dump**);

If the total quantity of substances 1 and 2 is below some minimal level min the system jumps into mode 3 (**Filter**).

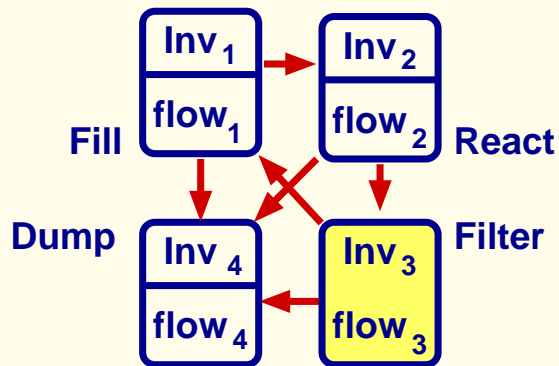
Example



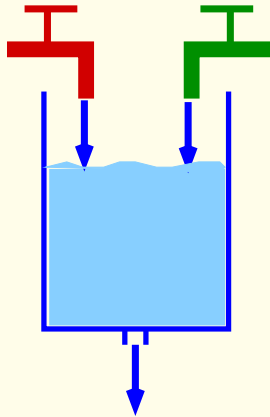
Mode 3: Filter Temperature is low. Substance 3 is filtered out.

$$\begin{aligned} \text{Inv}_3 \quad & x_1 + x_2 + x_3 \leq L_{\text{overflow}} \quad \wedge \quad \bigwedge_{i=1}^3 x_i \geq 0 \quad \wedge \\ & -\epsilon_a \leq x_1 - x_2 \leq \epsilon_a \quad \wedge \quad x_3 \geq \text{min} \\ \text{flow}_3 \quad & \dot{x}_1 = 0 \wedge \dot{x}_2 = 0 \quad \wedge \quad \dot{x}_3 \leq -d_{\text{min}} \end{aligned}$$

If proportion not kept: system jumps into mode 4 (**Dump**);
 Otherwise, if the concentration of substance 3 is below some minimal level min the system jumps into mode 1 (**Fill**).



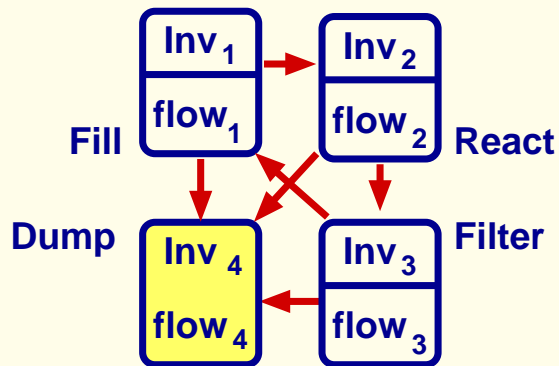
Example



Mode 4: Dump The content of the tank is emptied.

For simplicity we assume that this happens instantaneously:

$$\text{Inv}_4 : \bigwedge_{i=1}^3 x_i = 0 \text{ and } \text{flow}_4 : \bigwedge_{i=1}^3 \dot{x}_i = 0.$$



Remark

The material on ASMs is not required for the exam (only the general idea)

The definitions of timed automata and hybrid automata are required for the exam.

More complex specifications and specification languages

- Languages for describing various processes
- Languages based on Set theory (OZ, B)
- Languages for describing durations
- Complex languages

More complex specifications and specification languages

- Languages for describing various processes
- Languages based on Set theory (OZ, B)
- Languages for describing durations
- Complex languages

CSP

Communicating Sequential Processes, or CSP, is a language for describing processes and patterns of interaction between them.

It is supported by an elegant, mathematical theory, a set of proof tools, and an extensive literature.

CSP

Communicating Sequential Processes, or CSP, is a language for describing processes and patterns of interaction between them.

It is supported by an elegant, mathematical theory, a set of proof tools, and an extensive literature.

- Each process: transition system
- Operations on processes: sequential, parallel composition
effects on states

CSP

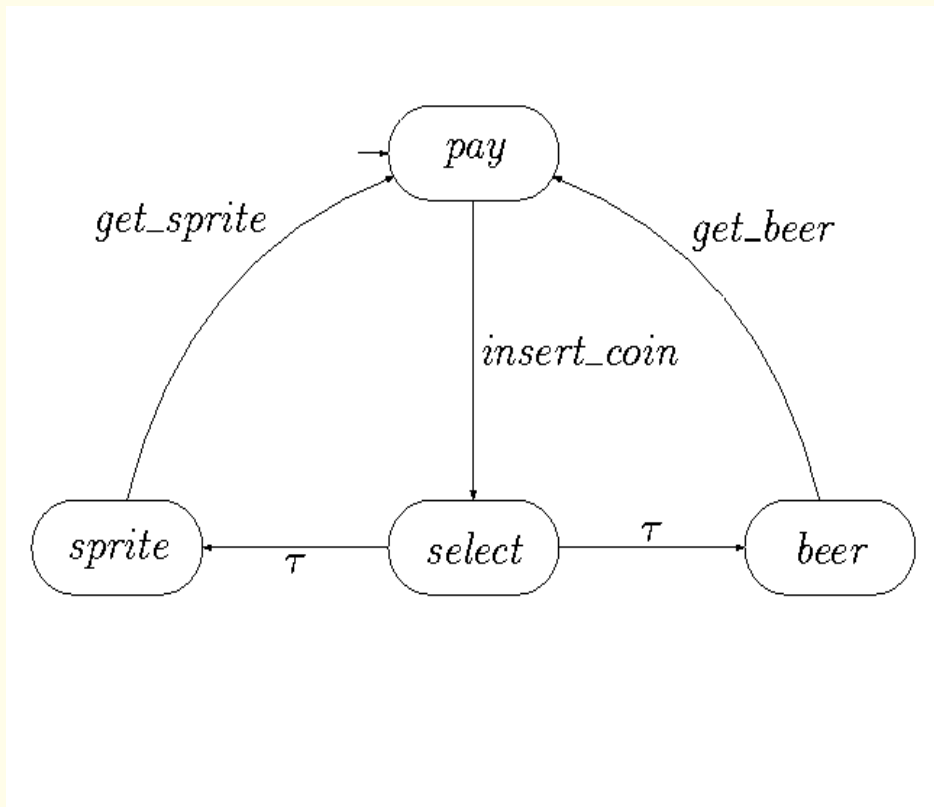
General idea:

Given:

- Set of event names
- Process: behaviour pattern of an object (insofar as it can be described in terms of the limited set of events selected as its alphabet)

CSP

Example:



Events: insert-coin, get-sprite, get-beer

CSP

Prefix:

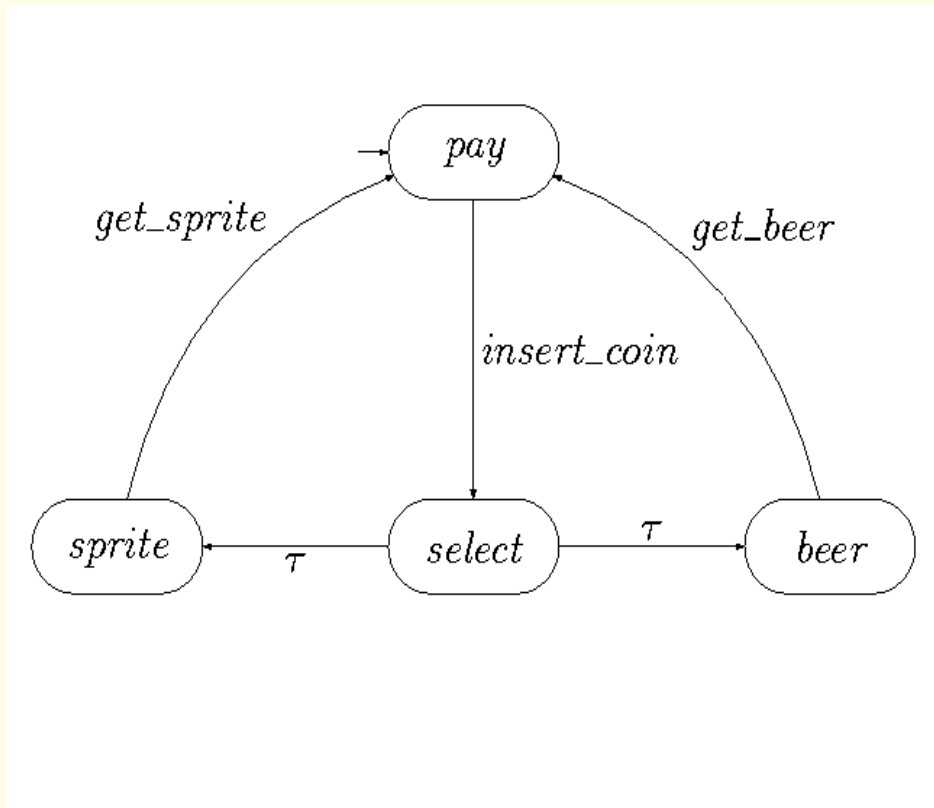
$$P = a \rightarrow Q$$

(*a* then *Q*)

where *a* is an event and *Q* a process

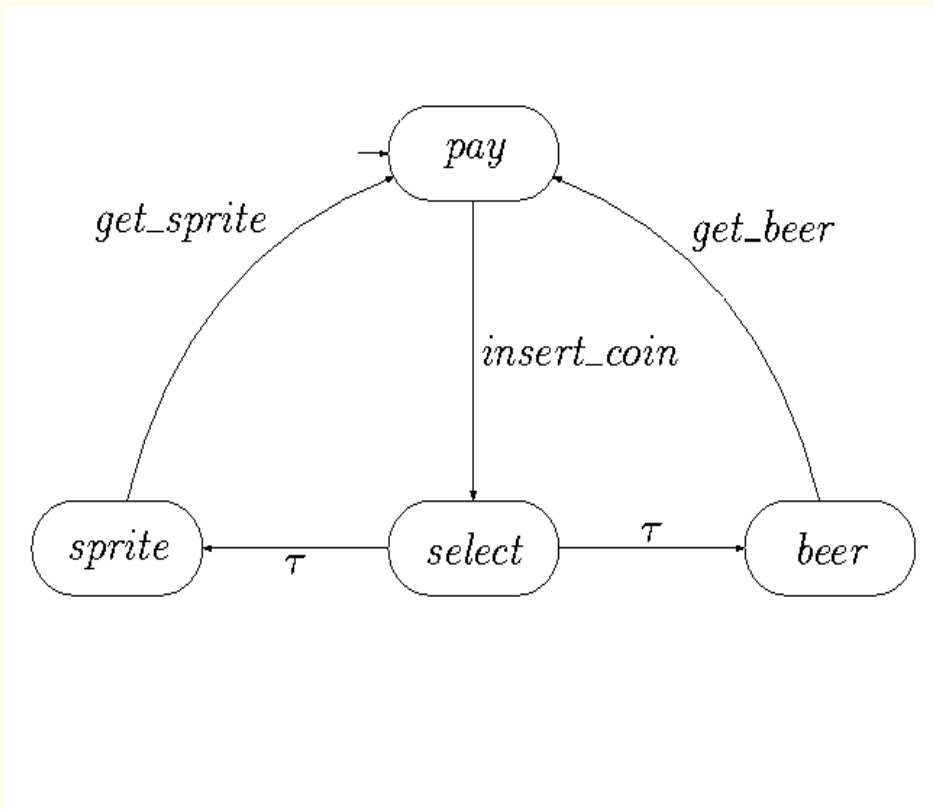
After event *a*, process *P* behaves like process *Q*

CSP: Example



A simple vending machine which consumes one coin before breaking
(*insert-coin* \rightarrow *STOP*)

CSP: Example



A simple vending machine that successfully serves two customers before breaking

$(insert_coin \rightarrow (get_sprite \rightarrow (insert_coin \rightarrow (get_beer \rightarrow STOP))))$

CSP

Example: (recursive definitions)

Consider the simplest possible everlasting object, a clock which never does anything but tick (the act of winding is deliberately ignored)

$$Events(CLOCK) = \{tick\}$$

Consider next an object that behaves exactly like the clock, except that it first emits a single tick

$$(tick \rightarrow CLOCK)$$

The behaviour of this object is indistinguishable from that of the original clock. This reasoning leads to formulation of the equation

$$CLOCK = (tick \rightarrow CLOCK)$$

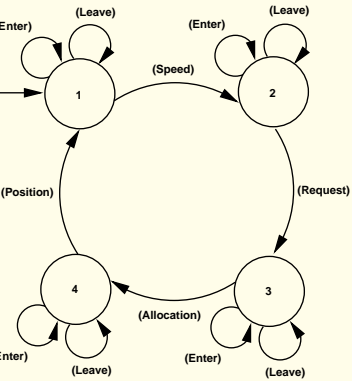
This can be regarded as an implicit definition of the behaviour of the clock.

Modular Specifications: CSP-OZ-DC (COD)

COD [Hoenicke,Olderog'02] allows us to specify in a modular way:

- the control flow of a system
using Communicating Sequential Processes (CSP)
- the state space and its change
using Object-Z (OZ)
- (dense) real-time constraints over durations of events
using the Duration Calculus (DC)

Example: Controller for line track (RBC)



RBC

```
method enter : [s1? : Segment; t0? : Train; t1? : Train; t2? : Train]
method leave : [l? : Segment; lt? : Train]
local_chan alloc, req, updPos, updSpd
```

```
main ≜ ((enter → main)
□ (leave → main)
□ (updSpd → State1))
State1 ≜ ((enter → State1)
□ (leave → State1)
□ (req → State2))
```

SegmentData

```
train : Segment → Train [Train on segment]
req : Segment → ℤ [Requested by train]
alloc : Segment → ℤ [Allocated by train]
```

```
State2 ≜ ((alloc → State3)
□ (enter → State2)
□ (leave → State2))
State3 ≜ ((enter → State3)
□ (leave → State3)
□ (updPos → main))
```

TrainData

```
segm : Train → Segment [Train segment]
next : Train → Train [Next train]
spd : Train → ℝ [Speed]
pos : Train → ℝ [Current position]
prev : Train → Train [Prev. train]
```

sd : SegmentData
td : TrainData

```
∀t : TrainΓ tid(t) > 0
∀t1, t2 : Train | t1 ≠ t2Γ tid(t1) ≠ tid(t2)
∀s : SegmentΓ prevs(nexts(s)) = s
∀s : SegmentΓ nexts(prevs(s)) = s
∀s : SegmentΓ sid(s) > 0
∀s : SegmentΓ sid(nexts(s)) > sid(s)
∀s1, s2 : Segment | s1 ≠ s2Γ sid(s1) ≠ sid(s2)
∀s : Segment | s ≠ snilΓ length(s) > d + gmax · Δt
∀s : Segment | s ≠ snilΓ 0 < lmax(s) ∧ lmax(s) ≤ gmax
∀s : SegmentΓ lmax(s) ≥ lmax(prevs(s)) - decmax · Δt
∀s1, s2 : SegmentΓ tid(incoming(s1)) ≠ tid(train(s2))
```

effect_updSpd

Δ(spdx)

```
∀t : Train | pos(t) < length(segmx(t)) - d ∧ spd(t) - decmax · Δt > 0
Γ max{0, spd(t) - decmax · Δt} ≤ spd'(t) ≤ lmax(segmx(t))
∀t : Train | pos(t) ≥ length(segmx(t)) - d ∧ alloc(nexts(segmx(t))) = tid(t)
Γ max{0, spd(t) - decmax · Δt} ≤ spd'(t) ≤ min{lmax(segmx(t)), lmax(nexts(segmx(t)))}
∀t : Train | pos(t) ≥ length(segmx(t)) - d ∧ ¬ alloc(nexts(segmx(t))) = tid(t)
Γ spd'(t) = max{0, spd(t) - decmax · Δt}
```

CSP

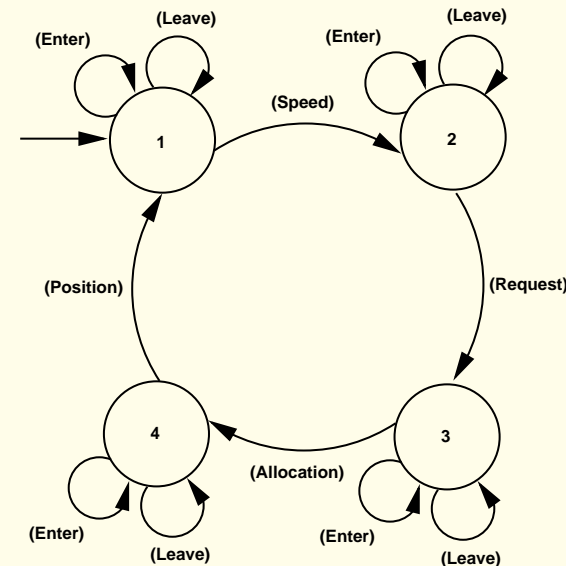
OZ

Example: Controller for line track (RBC)

CSP part: specifies the processes and their interdependency.

The RBC system passes repeatedly through four phases, modeled by events:

- **updSpd** (speed update)
- **req** (request update)
- **alloc** (allocation update)
- **updPos** (position update)



Between these events, trains may leave or enter the track (at specific segments), modeled by the events **leave** and **enter**.

Example: Controller for line track (RBC)

CSP part: specifies the processes and their interdependency.

The RBC system passes repeatedly through four phases, modeled by events with corresponding COD schemata:

CSP: _____

method *enter* : [s1? : Segment; t0? : Train; t1? : Train; t2? : Train]

method *leave* : [ls? : Segment; lt? : Train]

local_chan *alloc*, *req*, *updPos*, *updSpd*

$\text{main} \stackrel{c}{=} ((\text{updSpd} \rightarrow \text{State1}) \quad \text{State1} \stackrel{c}{=} ((\text{req} \rightarrow \text{State2}) \quad \text{State2} \stackrel{c}{=} ((\text{alloc} \rightarrow \text{State3}) \quad \text{State3} \stackrel{c}{=} ((\text{updPos} \rightarrow \text{main})$
 $\square(\text{leave} \rightarrow \text{main}) \quad \square(\text{leave} \rightarrow \text{State1}) \quad \square(\text{leave} \rightarrow \text{State2}) \quad \square(\text{leave} \rightarrow \text{State3})$
 $\square(\text{enter} \rightarrow \text{main})) \quad \square(\text{enter} \rightarrow \text{State1})) \quad \square(\text{enter} \rightarrow \text{State2})) \quad \square(\text{enter} \rightarrow \text{State3}))$

Example: Controller for line track (RBC)

OZ part. Consists of data classes, axioms, the `Init` schema, update rules.

Example: Controller for line track (RBC)

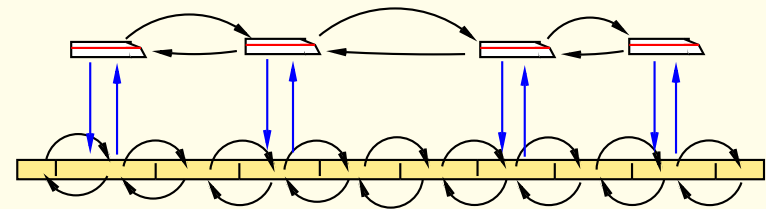
OZ part. Consists of data classes, axioms, the `Init` schema, update rules.

- 1. **Data classes** declare function symbols that can change their values during runs of the system

Data structures:

- 2-sorted pointers

train: trains
 segm: segments



<i>SegmentData</i>	
$train : Segment \rightarrow Train$	[Train on segment]
$req : Segment \rightarrow \mathbb{Z}$	[Requested by train]
$alloc : Segment \rightarrow \mathbb{Z}$	[Allocated by train]

<i>TrainData</i>	
$segm : Train \rightarrow Segment$	[Train segment]
$next : Train \rightarrow Train$	[Next train]
$spd : Train \rightarrow \mathbb{R}$	[Speed]
$pos : Train \rightarrow \mathbb{R}$	[Current position]
$prev : Train \rightarrow Train$	[Prev. train]

Example: Controller for line track (RBC)

OZ part. Consists of data classes, axioms, the `Init` schema, update rules.

- **1. Data classes** declare function symbols that can change their values during runs of the system, and are used in the OZ part of the specification.
- **2. Axioms:** define properties of the data structures and system parameters which do not change
 - $gmax : \mathbb{R}$ (the global maximum speed),
 - $decmax : \mathbb{R}$ (the maximum deceleration of trains),
 - $d : \mathbb{R}$ (a safety distance between trains),
 - Properties of the data structures used to model trains/segments

Example: Controller for line track (RBC)

OZ part. Consists of data classes, axioms, the `Init` schema, update rules.

- **3. Init schema.** describes the initial state of the system.
 - trains - doubly-linked list; placed correctly on the track segments
 - all trains respect their speed limits.
- **4. Update rules** specify updates of the state space executed when the corresponding event from the CSP part is performed.

Example: Speed update

effect_updSpd

$\Delta(\text{spd})$

$\forall t : \text{Train} \mid \text{pos}(t) < \text{length}(\text{segm}(t)) - d \wedge \text{spd}(t) - \text{decmax} \cdot \Delta t > 0$

$\Gamma \max\{0, \text{spd}(t) - \text{decmax} \cdot \Delta t\} \leq \text{spd}'(t) \leq \text{lmax}(\text{segm}(t))$

$\forall t : \text{Train} \mid \text{pos}(t) \geq \text{length}(\text{segm}(t)) - d \wedge \text{alloc}(\text{nexts}(\text{segm}(t))) = \text{tid}(t)$

$\Gamma \max\{0, \text{spd}(t) - \text{decmax} \cdot \Delta t\} \leq \text{spd}'(t) \leq \min\{\text{lmax}(\text{segm}(t)), \text{lmax}(\text{nexts}(\text{segm}(t)))\}$

$\forall t : \text{Train} \mid \text{pos}(t) \geq \text{length}(\text{segm}(t)) - d \wedge \neg \text{alloc}(\text{nexts}(\text{segm}(t))) = \text{tid}(t)$

$\Gamma \text{spd}'(t) = \max\{0, \text{spd}(t) - \text{decmax} \cdot \Delta t\}$

Formal specification

- Specification for program/system
- Specification for properties of program/system

Verification tasks:

Check that the specification of the program/system has the required properties.