Formal Specification and Verification

Formal specification (2)

3.12.2018

Viorica Sofronie-Stokkermans

e-mail: sofronie@uni-koblenz.de

Until now

- Logic
- Formal specification (generalities)

  Algebraic specification

  Transition systems

Transition systems

Transition systems

- Executions
- Modeling data-dependent systems

Last time: 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

Last time: Transition systems

Definition.

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

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

S and Act are either finite or countably infinite

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

Last time: Direct successors and predecessors

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

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

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

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

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

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

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

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

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

Reach(TS) denotes the set of all reachable states in TS.

Detailed description of states

Variables; Predicates

$\mapsto$ Program graph representation

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

From program graphs to transition systems

- Basic strategy: unfolding
  - state = location (current control) l + data valuation β $\qquad (l, \beta)$
  - initial state = initial location + data valuation satisfying the initial condition g_0

- Propositions and labeling
  - propositions: "at l" and "$x \in D$" for $D \subseteq dom(x)$
  - $< l, \beta >$ is labeled with "at l" and all conditions that hold in β.

- $l \overset{g:\alpha}{\rightarrow} l'$ and g holds in β then $< l, \beta > \overset{\alpha}{\rightarrow} < l', Effect(< l, \beta >) >$

Transition systems for program graphs

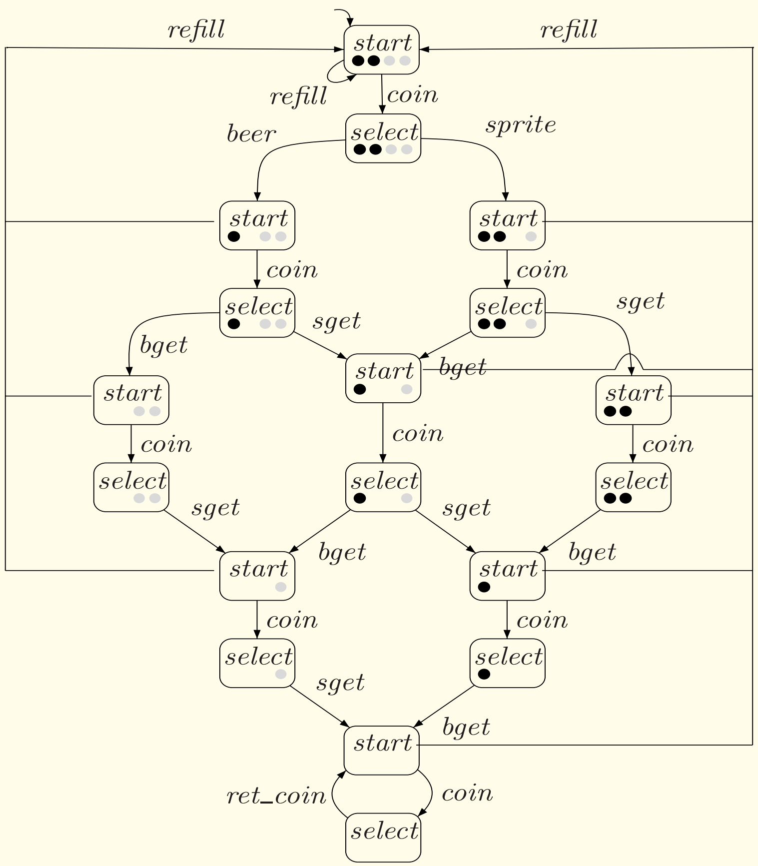
The transition system $TS(PG)$ of program graph

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

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 $l \overset{g:\alpha}{\rightarrow} l'$ and $\beta \models g$ then $< l, \beta > \overset{\alpha}{\rightarrow} < l', Effect(< l, \beta >) >$
- $I = \{< l, \beta >|\ l \in \mathrm{Loc}_0, \beta \models g_0\}$
- $AP = Loc \cup Cond(Var)$ and
- $L(< l, \beta >) = \{l\} \cup \{g \in Cond(Var) \mid \beta \models g\}$.

Transition systems for program graphs



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

- Probabilistic systems: Markov chains, Probabilistic hybrid automata, ...

Abstract state machines (ASM)

Purpose

Formalism for modelling/formalising (sequential) algorithms

Not: Computability / complexity analysis

Invented/developed by

Yuri Gurevich, 1988

Old name

Evolving algebras

ASMs

Three Postulates

Sequential Time Postulate:

An algorithm can be described by defining a set of states, a subset of initial states, and a state transformation function

Abstract State Postulate:

States can be described as first-order structures

Bounded Exploration Postulate:

An algorithm explores only finitely many elements in a state to decide what the next state is. There is a finite number of names (terms) for all these "interesting" elements in all states.

Example: Computing Squares

Initial State

$square = 0$

$count = 0$

ASM for computing the square of input

if $input < 0$ then

    $input :=$ - $input$

else if $input > 0 \wedge count < input$ then

  par

    $square := square + input$

    $count := count + 1$

  endpar

The Sequential Time Postulate

Sequential algorithm

An algorithm is associated with

- a set S of states
- a set $I \subseteq S$ of initial states
- A function $\tau : S \to S$
  (the one-step transformation of the algorithm)

Run (computation)

A run (computation) is a sequence $X_0, X_1, X_2 \ldots$ of states such that

- $X_0 \in I$
- $\tau(X_i) = X_{i+1}$ for all $i \geq 0$

Remark

Remark: In this formalism, algorithms are deterministic

$\tau : S \to S$ can be also viewed as a relation $R \subseteq S \times \{\tau\} \times S$ with

$$(s, \tau, s') \in R \text{ iff } \tau(s) = s'.$$

The Abstract State Postulate

States are first-order structures where

- all states have the same vocabulary (signature)
- the transformation τ does not change the base set (universe)
- S and I are closed under isomorphism
- if f is an isomorphism from a state X onto a state Y, then f is also an isomorphism from $\tau(X)$ onto $\tau(Y)$.

Example: Trees

Vocabulary

nodes:	unary, boolean:	the class of nodes (type/universe)
strings:	unary, boolean:	the class of strings
parent:	unary:	the parent node
firstChild:	unary:	the first child node
nextSibling:	unary:	the first sibling
label:	unary:	node label
c:	constant:	the current node

Vocabulary (Signature)

Signatures: A signature is a finite set of function symbols, where

- each symbol is assigned an arity $n \geq 0$
- symbols can be marked relational (predicates)
- symbols can be marked static (default: dynamic)

Vocabulary (Signature)

Signatures: A signature is a finite set of function symbols, where

- each symbol is assigned an arity $n \geq 0$
- symbols can be marked relational (predicates)
- symbols can be marked static (default: dynamic)

Remark: This is not a restriction

• predicates with arity n can be regarded as functions with arity

$s \dots s \to \text{bool}$

where s is the usual sort (for terms) and bool is a different sort

• The sort bool is described using a unary predicate Bool

• The sort Bool contains all formulae, in particular also $\top, \bot$ ("relational constants")

Vocabulary (Signature)

Signatures: A signature is a finite set of function symbols, where

- each symbol is assigned an arity $n \geq 0$
- symbols can be marked relational (predicates)
- symbols can be marked static (default: dynamic)

Each signature contains

- the constant *undef* ("undefined")
- the relational constants $\top$ (true), $\bot$ (false)
- the unary relational symbols *Boole*, $\neg$
- the binary relational symbols $\wedge, \vee, \rightarrow, \leftrightarrow, \approx$

These special symbols are all static

Vocabulary (Signature)

Signatures: A signature is a finite set of function/predicate symbols, where

- each symbol is assigned an arity $n \geq 0$
- symbols can be marked static (default: dynamic)

Each signature contains

- the constant *undef* ("undefined")
- the relational constants *true*, *false*
- the unary relational symbols *Boole*, $\neg$
- the binary relational symbols $\wedge, \vee, \rightarrow, \leftrightarrow, \approx$

These special symbols are all static

There is an infinite set of variables
Terms are built as usual from variables and function symbols
Formulae are built as usual

First-order Structures (States)

First-order structures (states) consist of

- a non-empty universe (called BaseSet)
- an interpretation of the symbols in the signature

Restrictions on states

- $0, 1, \mathit{undef} \in \text{BaseSet}$ (different)
- $\bot_{\mathcal{A}} = 0$, $\top_{\mathcal{A}} = 1$
- $\mathit{undef}_{\mathcal{A}} = \mathit{undef}$
- If f relational then $f_{\mathcal{A}} : \text{BaseSet} \to \{0, 1\}$
- $\mathit{Boole}_{\mathcal{A}} = \{0, 1\}$
- $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ are interpreted as usual

The reserve of a state

Reserve: Consists of the elements that are "unknown" in a state

The reserve of a state must be infinite

Extended States

Variable assignment

A function $\beta : \mathit{Var} \to \mathrm{BaseSet}$

(boolean variables are assigned 0 or 1)

Extended state

A pair $(\mathcal{A}, \beta)$ consisting of a state $\mathcal{A}$ and a variable assignment β.

Extended States

Variable assignment

A function $\beta : \mathit{Var} \rightarrow \mathrm{BaseSet}$

(boolean variables are assigned 0 or 1)

Extended state

A pair $(\mathcal{A}, \beta)$ consisting of a state $\mathcal{A}$ and a variable assignment β.

Evaluation of terms and formulae: as usual

Example: Trees

Vocabulary

nodes:	unary, boolean:	the class of nodes (type/universe)
strings:	unary, boolean:	the class of strings
parent:	unary:	the parent node
firstChild:	unary:	the first child node
nextSibling:	unary:	the first sibling
label:	unary:	node label
c:	constant:	the current node

Example: Trees

Terms

$parent(parent(c))$

$label(firstChild(c))$

$parent(firstChild(c)) = c$ (Boolean, formula)

$nodes(x) \rightarrow parent(x) = parent(nextSibling(x))$

(x is a variable)

Isomorphism

Lemma (Isomorphism)

Isomorphic states (structures) are indistinguishable by ground terms:

Justification for postulate

Algorithm must have the same behaviour for indistinguishable states

Isomorphic states are different representations of the same abstract state!

State updates

Locations. A location is a pair $(f, \overline{a})$ with

- f an n-ary function symbol
- $\overline{a} \in \text{BaseSet}^n$ an n-tuple

Examples

$(parent, a)$, $(firstChild, a)$, $(nextSibling, a)$, $(c,)$

State updates

Locations. A location is a pair $(f, \overline{a})$ with

- f an n-ary function symbol
- $\overline{a} \in \text{BaseSet}^n$ an n-tuple

Examples

$(parent, a)$, $(firstChild, a)$, $(nextSibling, a)$, $(c,)$

An update is a triple $(f, \overline{a}, b)$ with

- $(f, \overline{a})$ a location
- f not static
- $b \in \text{BaseSet}$
- if f is relational, then $b \in \{0, 1\}$

State updates

Locations. A location is a pair $(f, \overline{a})$ with

- f an n-ary function symbol
- $\overline{a} \in \text{BaseSet}^n$ an n-tuple

Examples

(parent, a), $(\mathit{firstChild}, a)$, $(\mathit{nextSibling}, a)$, $(c,)$

An update is a triple $(f, \overline{a}, b)$ with

- $(f, \overline{a})$ a location
- f not static
- $b \in \text{BaseSet}$
- if f is relational, then $b \in \{0, 1\}$

Intended meaning:

f is changed by changing $f(\overline{a})$ to b.

State updates

Locations. A location is a pair $(f, \overline{a})$ with

- f an n-ary function symbol
- $\overline{a} \in \mathrm{BaseSet}^n$ an n-tuple

Examples

(parent, a), $(\mathit{firstChild}, a)$, $(\mathit{nextSibling}, a)$, $(c,)$

An update is a triple $(f, \overline{a}, b)$ with

- $(f, \overline{a})$ a location
- f not static
- $b \in \mathrm{BaseSet}$
- if f is relational, then $b \in \{\mathit{tt}, \mathit{ff}\}$

Intended meaning:

f is changed by changing $f(\overline{a})$ to b.

An update is trivial if $f_{\mathcal{A}}(\overline{a}) = b$

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.

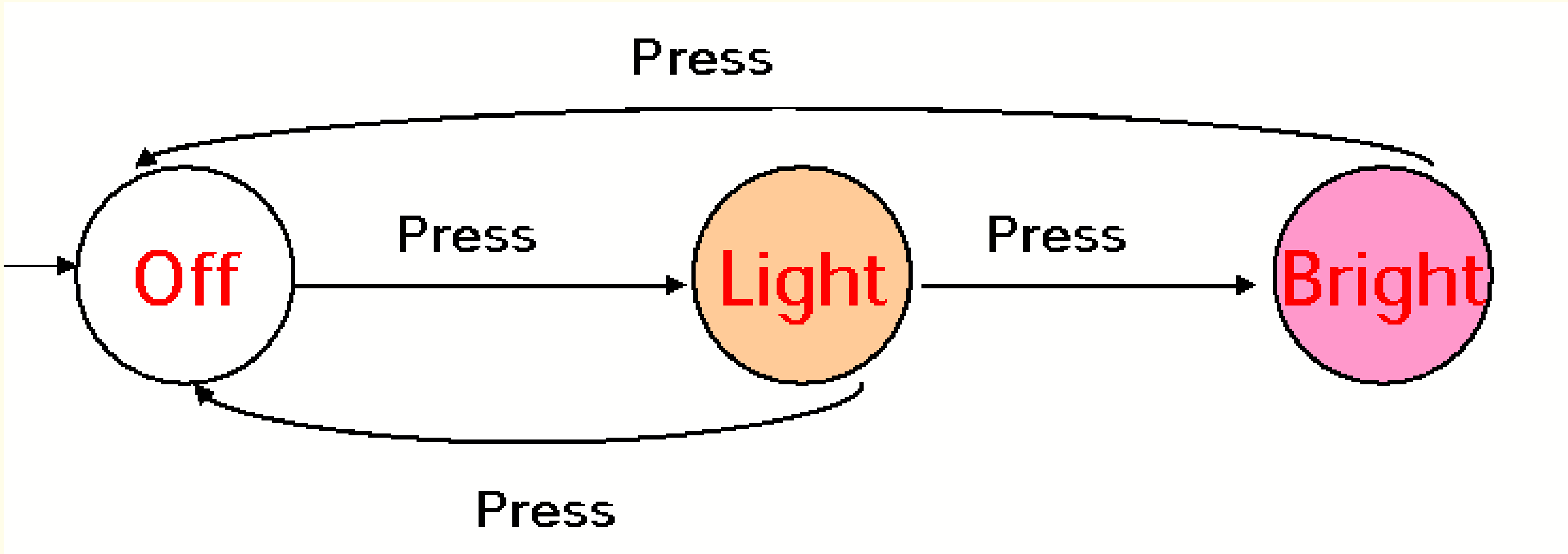
Timed automata

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

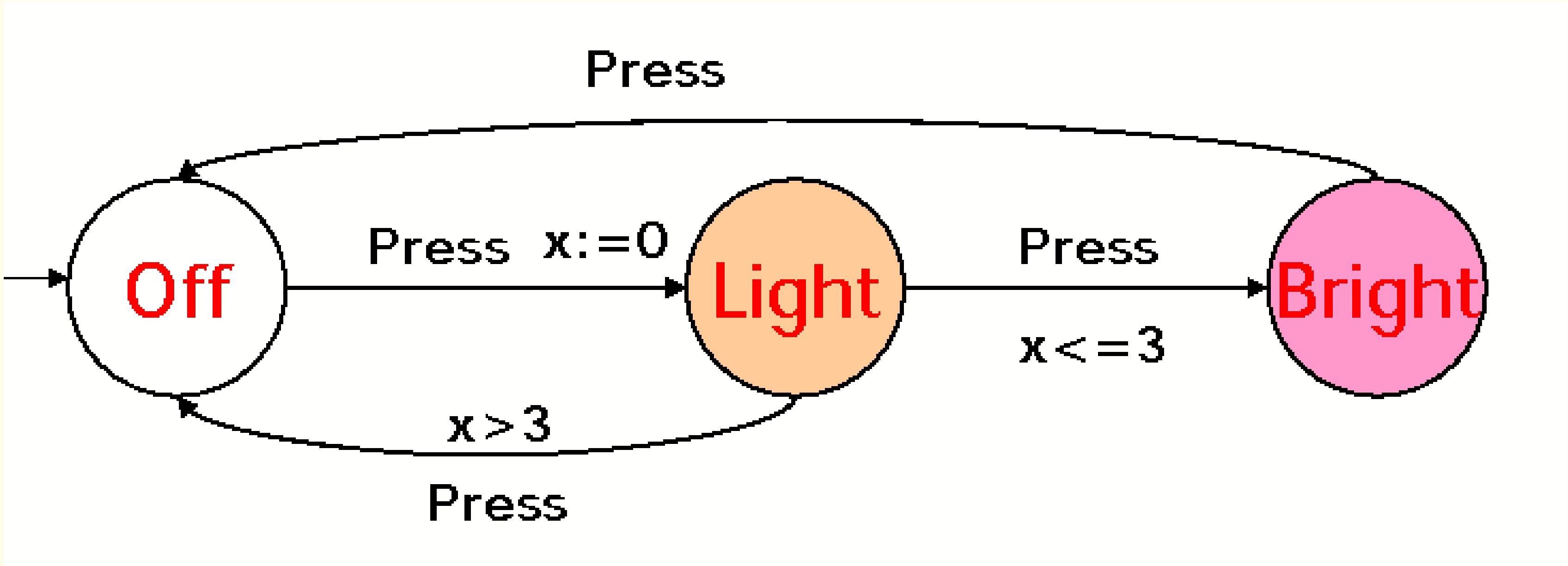
Example: Simple Light Control



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

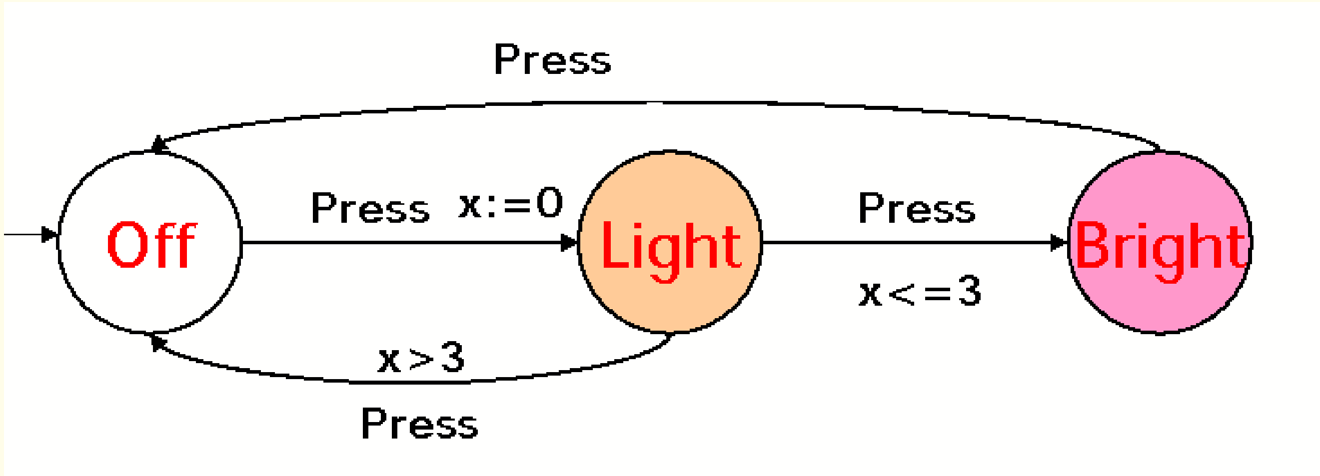
$$A = (Loc, Loc_0, Act, X, \{Inv_l\}_{l \in Loc}, E)$$

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:

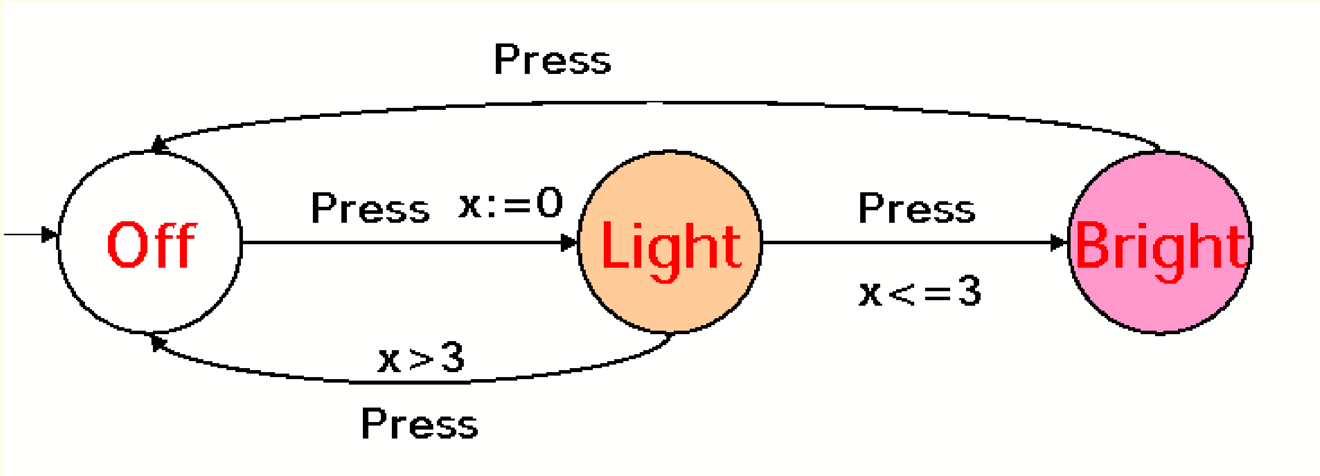
$Loc = \{\text{Off}, \text{Light}, \text{Bright}\},\ Loc_0 = \{\text{Off}\}, \quad Act = \{\text{Press}\}$

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

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

Timed automata

Example: Simple Light Control



States: (Off, v), (Light, v), (Bright, v) (v value of clock x).
Initial state: $(\mathrm{Off}, 0)$.

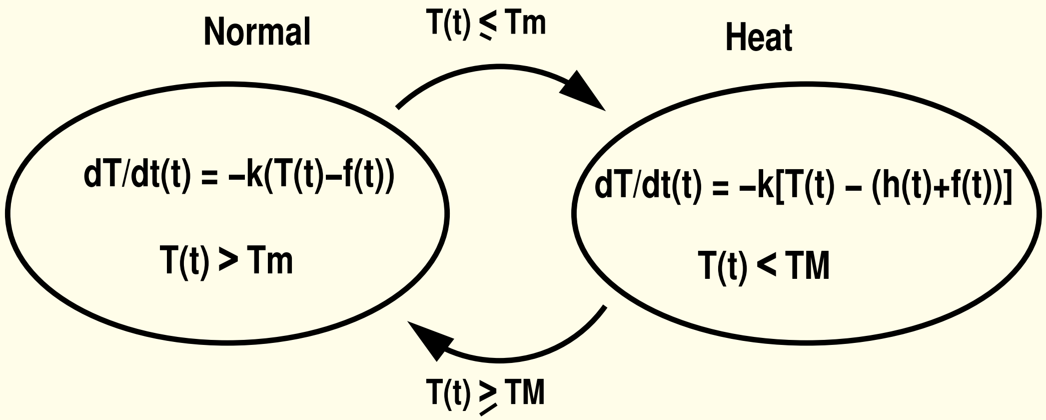
Transitions (Examples)

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

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

Hybrid Automata

Hybrid Automata



f : R −> R evolution of external temperature

h : R −> R evolution of heater temperature

Hybrid Automata

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

(1) $X = \{x_1, \dots, x_n\}$ finite set of real valued variables
Q finite set of control modes

(2) $\{\mathrm{flow}_q \mid q \in Q\}$ 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) $\{\mathrm{Inv}_q \mid q \in Q\}$ mode invariants (predicates over X).

(4) $\{\mathrm{Init}_q \mid q \in Q\}$ initial states for control modes (predicates over X).

(5) E: control switches (finite multiset with elements in $Q \times Q$).

(6) $\{\mathrm{guard}_e \mid e \in E\}$ guards for control switches (predicates over X).

(7) Jump conditions $\{\mathrm{jump}_e \mid e \in E\}$, (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:

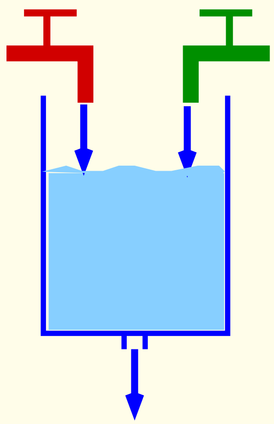
- For every $q \in Q$, flow_q, Inv_q, and Init_q are convex linear predicates.
- For every $e = (q, q') \in E$, jump_e and guard_e are convex linear predicates.

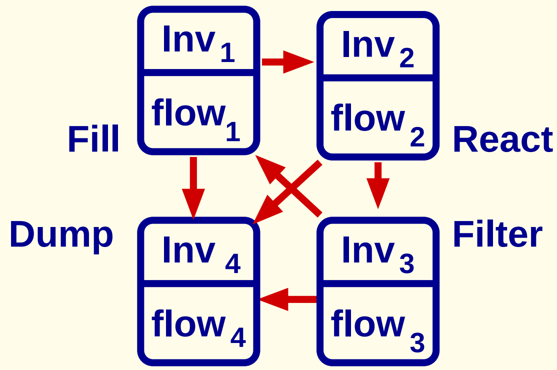
We assume that flow_q are conjunctions of *non-strict* inequalities.

(2) Flow independence:

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

Example

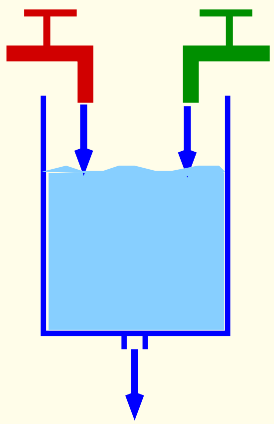


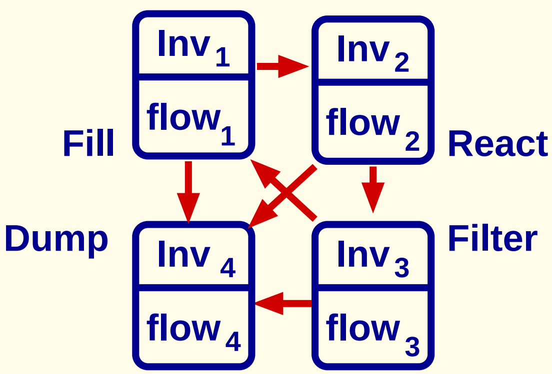


Chemical plant

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

Example





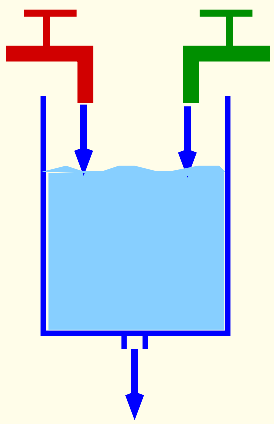
Chemical plant

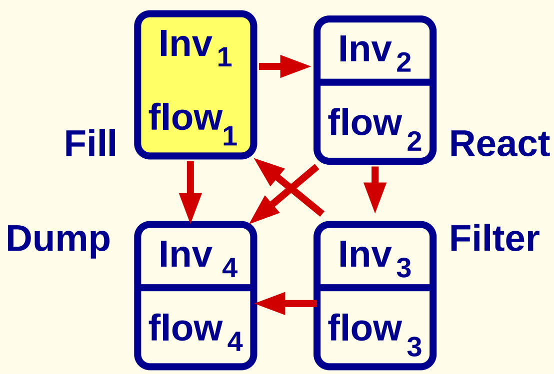
Two substances are mixed; they react;
the resulting product is filtered out;
then the procedure is repeated.

Check:

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

Example



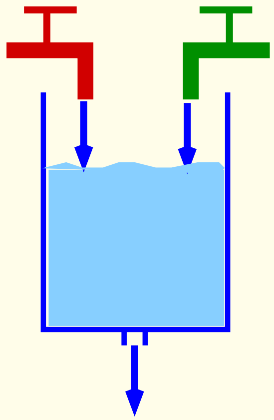


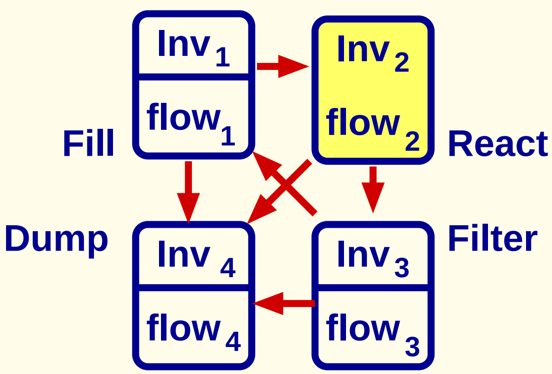
Mode 1: Fill Temperature is low, 1 and 2 do not react. Substances 1 and 2 (possibly mixed with a small quantity of 3) are filled in the tank in equal quantities up to a margin of error.

$$
\begin{array}{ll}
\mathrm{Inv}_1 & x_1 + x_2 + x_3 \leq L_f \;\wedge\; \bigwedge_{i=1}^{3} x_i \geq 0 \;\wedge \\
 & -\epsilon_a \leq x_1 - x_2 \leq \epsilon_a \;\wedge\; 0 \leq x_3 \leq \min \\
\mathrm{flow}_1 & \dot{x_1} \geq \mathrm{dmin} \wedge \dot{x_2} \geq \mathrm{dmin} \wedge \dot{x_3} = 0 \wedge -\delta_a \leq \dot{x_1} - \dot{x_2} \leq \delta_a
\end{array}
$$

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



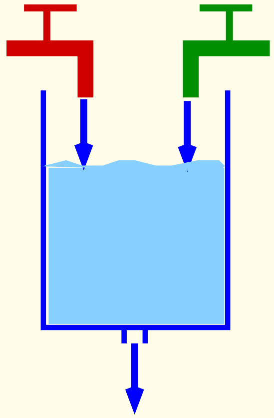


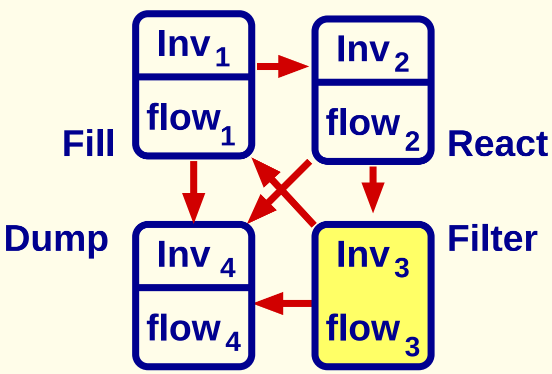
Mode 2: React Temparature is high. Substances 1 and 2 react. The reaction consumes equal quantities of substances 1 and 2 and produces substance 3.

$$
\begin{array}{ll}
\mathrm{Inv}_2 & L_f \leq x_1 + x_2 + x_3 \leq L_{\mathrm{overflow}} \ \wedge \ \bigwedge_{i=1}^{3} x_i \geq 0 \ \wedge \\
 & -\epsilon_a \leq x_1 - x_2 \leq \epsilon_a \ \wedge \ 0 \leq x_3 \leq \mathrm{max} \\
\mathrm{flow}_2 & \dot{x}_1 \leq -\mathrm{dmin} \wedge \dot{x}_2 \leq -\mathrm{dmin} \wedge .x_3 \geq \mathrm{dmin} \\
 & \wedge\ \dot{x}_1 = \dot{x}_2 \ \wedge \ \dot{x}_3 + \dot{x}_1 + \dot{x}_2 = 0
\end{array}
$$

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





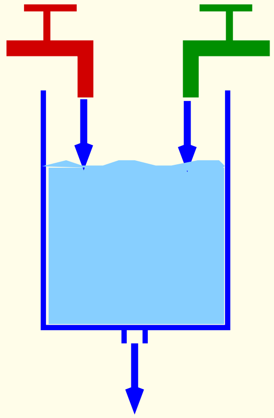
Mode 3: Filter Temperature is low. Substance 3 is filtered out.

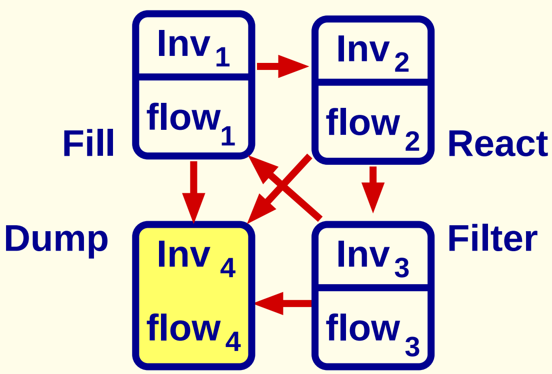
$$
\begin{array}{ll}
\mathrm{Inv}_3 & x_1 + x_2 + x_3 \leq L_{\mathrm{overflow}} \quad \wedge \quad \bigwedge_{i=1}^{3} x_i \geq 0 \quad \wedge \\
 & -\epsilon_a \leq x_1 - x_2 \leq \epsilon_a \quad \wedge \quad x_3 \geq \mathrm{min} \\
\mathrm{flow}_3 & \dot{x}_1 = 0 \wedge \dot{x}_2 = 0 \quad \wedge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

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:

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

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

  efects on states

CSP

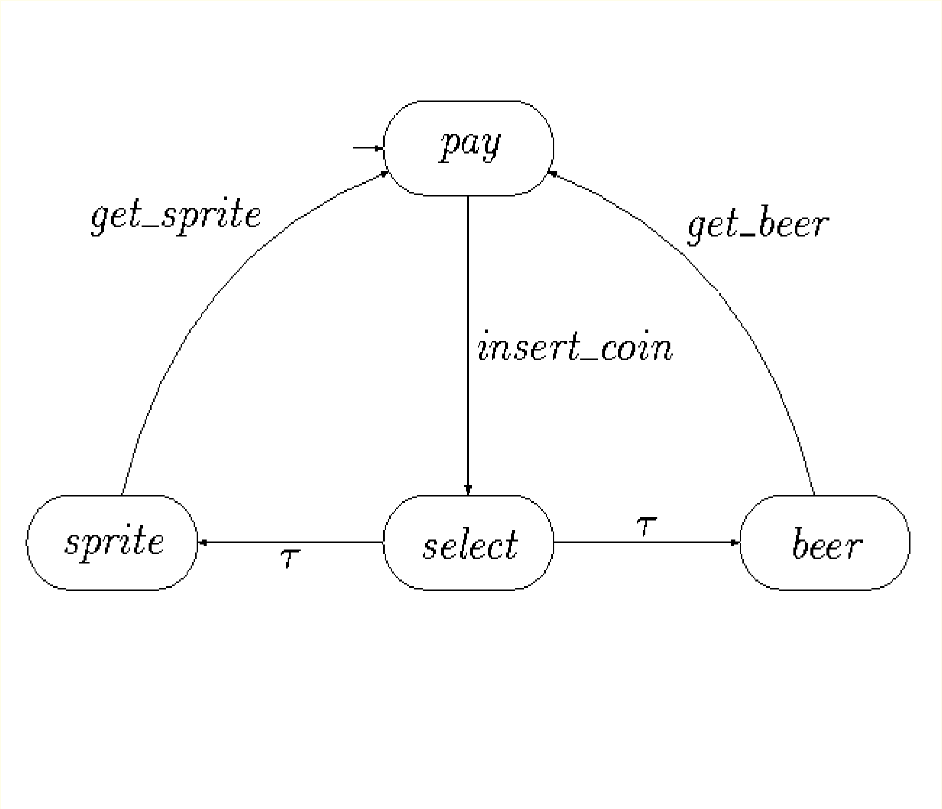
General idea:

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

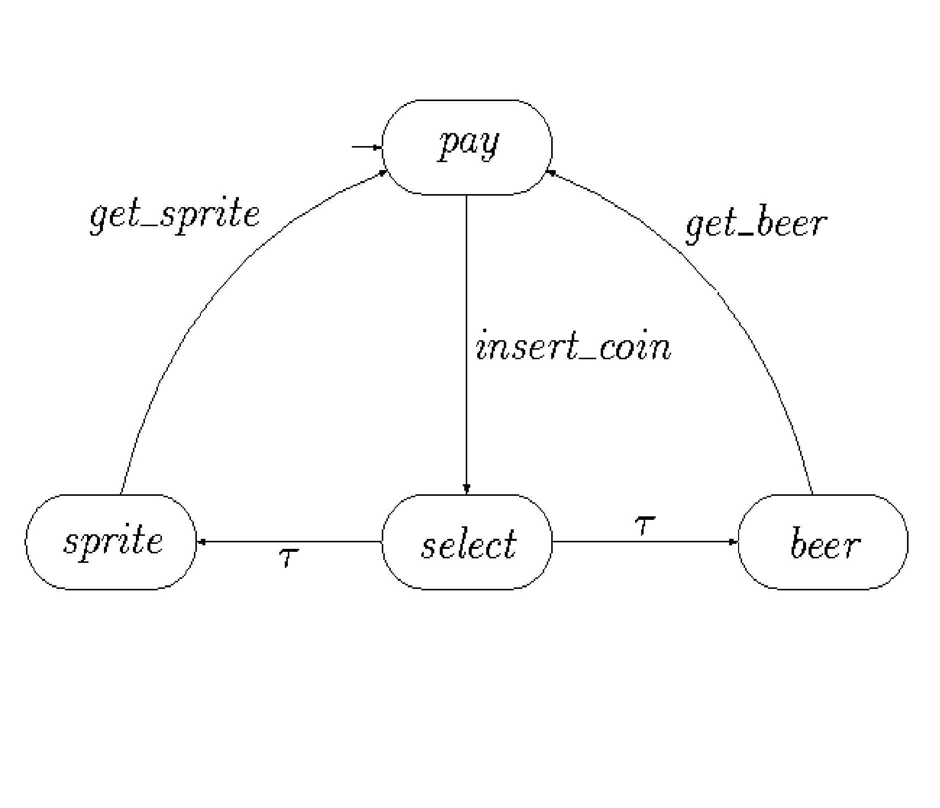
Prefix:

$P = a \to Q$ (*a* then Q)

where a is an event and Q a process

After event a, process P behaves like process Q

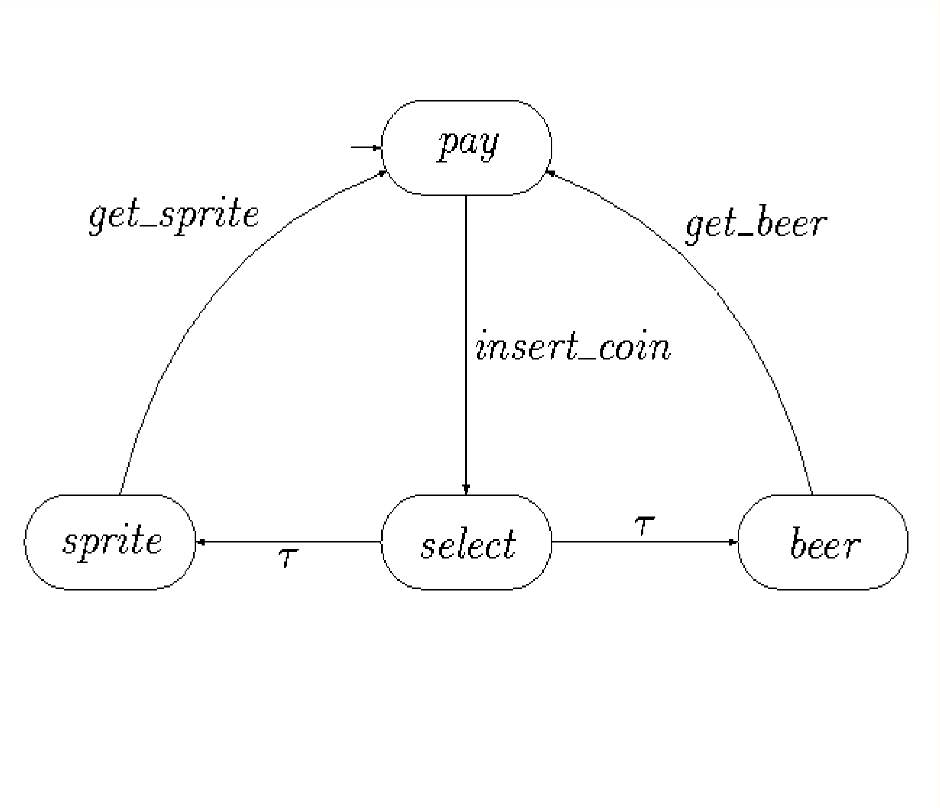
CSP: Example



A simple vending machine which consumes one coin before breaking

$$(\textit{insert-coin} \rightarrow \textit{STOP})$$

CSP: Example



A simple vending machine that successfully serves two customers before breaking

$$(\textit{insert-coint} \rightarrow (\textit{get-sprite} \rightarrow (\textit{insert-coin} \rightarrow (\textit{get-beer} \rightarrow \textit{STOP}))))$$

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

$$(tick \to CLOCK)$$

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

$$CLOCK = (tick \to CLOCK)$$

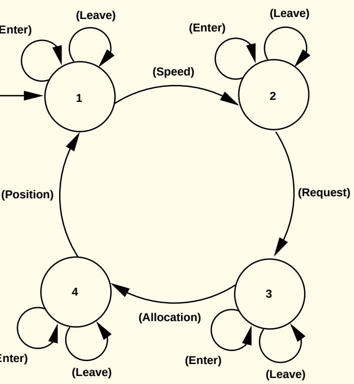
This can be regarded as an implicit definition of the behaviour of the clock.

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

$\texttt{method}\ enter : [s1? : Segment;\ t0? : Train;\ t1? : Train;\ t2? : Train]$

$\texttt{method}\ leave : [ls? : Segment;\ lt? : Train]$

$\texttt{local\_chan}\ alloc, req, updPos, updSpd$

$\texttt{main} \mathrel{\hat{=}} ((enter \to \texttt{main}) \;\Box\; (leave \to \texttt{main}) \;\Box\; (updSpd \to State1))$

$State1 \mathrel{\hat{=}} ((enter \to State1) \;\Box\; (leave \to State1) \;\Box\; (req \to State2))$

$State2 \mathrel{\hat{=}} ((alloc \to State3) \;\Box\; (enter \to State2) \;\Box\; (leave \to State2))$

$State3 \mathrel{\hat{=}} ((enter \to State3) \;\Box\; (leave \to State3) \;\Box\; (updPos \to \texttt{main}))$

SegmentData

$train : Segment \to Train$	[Train on segment]
$req : Segment \to \mathbb{Z}$	[Requested by train]
$alloc : Segment \to \mathbb{Z}$	[Allocated by train]

TrainData

$segm : Train \to Segment$	[Train segment]
$next : Train \to Train$	[Next train]
$spd : Train \to \mathbb{R}$	[Speed]
$pos : Train \to \mathbb{R}$	[Current position]
$prev : Train \to Train$	[Prev. train]

$sd : SegmentData$
$td : TrainData$

$\forall t : Train \Gamma tid(t) > 0$
$\forall t1, t2 : Train \mid t1 \neq t2 \Gamma tid(t1) \neq tid(t2)$
$\forall s : Segment \Gamma prevs(nexts(s)) = s$
$\forall s : Segment \Gamma nexts(prevs(s)) = s$
$\forall s : Segment \Gamma sid(s) > 0$
$\forall s : Segment \Gamma sid(nexts(s)) > sid(s)$
$\forall s1, s2 : Segment \mid s1 \neq s2 \Gamma sid(s1) \neq sid(s2)$
$\forall s : Segment \mid s \neq snil \Gamma length(s) > d + gmax \cdot \Delta t$
$\forall s : Segment \mid s \neq snil \Gamma 0 < lmax(s) \wedge lmax(s) \leq gmax$
$\forall s : Segment \Gamma lmax(s) \geq lmax(prevs(s)) - decmax \cdot \Delta t$
$\forall s1, s2 : Segment \Gamma tid(incoming(s1)) \neq tid(train(s2))$

Init

$\forall t : Train \Gamma train(segm(t)) = t$
$\forall t : Train \Gamma next(prev(t)) = t$
$\forall t : Train \Gamma prev(next(t)) = t$
$\forall t : Train \Gamma 0 \leq pos(t) \leq length(segm(t))$
$\forall t : Train \Gamma 0 \leq spd(t) \leq lmax(segm(t))$
$\forall t : Train \Gamma alloc(segm(t)) = tid(t)$
$\forall t : Train \Gamma alloc(nexts(segm(t))) = tid(t)$
$\quad \vee\ length(segm(t)) - bd(spd(t)) > pos(t)$
$\forall s : Segment \Gamma segm(train(s)) = s$

$\texttt{effect\_}updSpd$

$\Delta(spd)$

$\forall t : Train \mid pos(t) < length(segm(t)) - d \wedge spd(t) - decmax \cdot \Delta t > 0$
$\quad \Gamma \max\{0, spd(t) - decmax \cdot \Delta t\} \leq spd'(t) \leq lmax(segm(t))$
$\forall t : Train \mid pos(t) \geq length(segm(t)) - d \wedge alloc(nexts(segm(t))) = tid(t)$
$\quad \Gamma \max\{0, spd(t) - decmax \cdot \Delta t\} \leq spd'(t) \leq \min\{lmax(segm(t)), lmax(nexts(segm(t)))\}$
$\forall t : Train \mid pos(t) \geq length(segm(t)) - d \wedge \neg\ alloc(nexts(segm(t))) = tid(t)$
$\quad \Gamma spd'(t) = \max\{0, spd(t) - decmax \cdot \Delta t\}$

$\vdots$

CSP

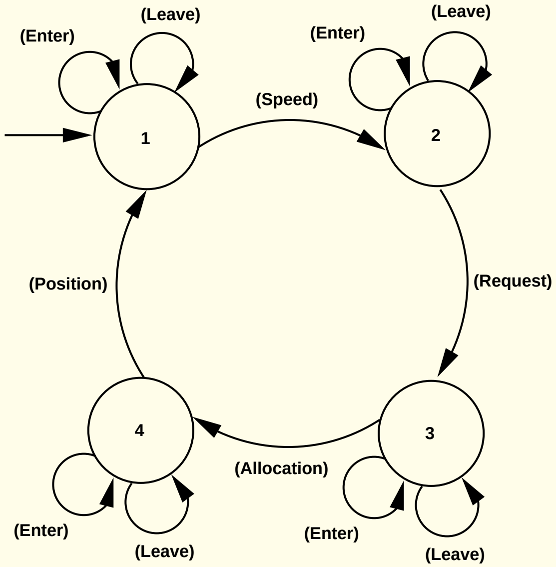
OZ

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:

$\texttt{method}\ \mathit{enter} : [s1? : \mathit{Segment};\ t0? : \mathit{Train};\ t1? : \mathit{Train};\ t2? : \mathit{Train}]$

$\texttt{method}\ \mathit{leave} : [ls? : \mathit{Segment};\ lt? : \mathit{Train}]$

$\texttt{local\_chan}\ \mathit{alloc}, \mathit{req}, \mathit{updPos}, \mathit{updSpd}$

$$\begin{array}{llll}
\texttt{main} \overset{c}{=} ((\mathit{updSpd} \to \mathit{State}1) & \mathit{State}1 \overset{c}{=} ((\mathit{req} \to \mathit{State}2) & \mathit{State}2 \overset{c}{=} ((\mathit{alloc} \to \mathit{State}3) & \mathit{State}3 \overset{c}{=} ((\mathit{updPos} \to \texttt{main}) \\
\quad \Box (\mathit{leave} \to \texttt{main}) & \quad \Box (\mathit{leave} \to \mathit{State}1) & \quad \Box (\mathit{leave} \to \mathit{State}2) & \quad \Box (\mathit{leave} \to \mathit{State}3) \\
\quad \Box (\mathit{enter} \to \texttt{main})) & \quad \Box (\mathit{enter} \to \mathit{State}1)) & \quad \Box (\mathit{enter} \to \mathit{State}2)) & \quad \Box (\mathit{enter} \to \mathit{State}3))
\end{array}$$

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.

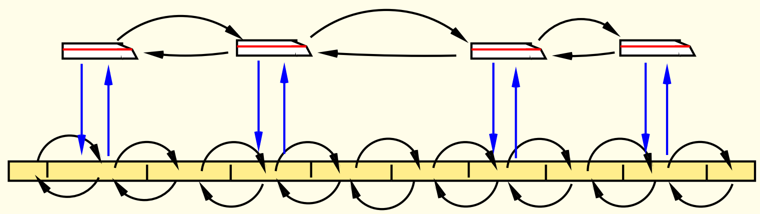
- 1. Data classes declare function symbols that can change their values during runs of the system

  Data structures:

  - 2-sorted pointers

    train: trains

    segm: segments



SegmentData

$train : Segment \to Train$	[Train on segment]
$req : Segment \to \mathbb{Z}$	[Requested by train]
$alloc : Segment \to \mathbb{Z}$	[Allocated by train]

TrainData

$segm : Train \to Segment$	[Train segment]
$next : Train \to Train$	[Next train]
$spd : Train \to \mathbb{R}$	[Speed]
$pos : Train \to \mathbb{R}$	[Current position]
$prev : Train \to Train$	[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*

  $\Delta(spd)$

  $$\forall t : Train \mid pos(t) < length(segm(t)) - d \wedge spd(t) - decmax \cdot \Delta t > 0$$
  $$\quad \Gamma \max\{0, spd(t) - decmax \cdot \Delta t\} \leq spd'(t) \leq lmax(segm(t))$$
  $$\forall t : Train \mid pos(t) \geq length(segm(t)) - d \wedge alloc(nexts(segm(t))) = tid(t)$$
  $$\quad \Gamma \max\{0, spd(t) - decmax \cdot \Delta t\} \leq spd'(t) \leq \min\{lmax(segm(t)), lmax(nexts(segm(t)))\}$$
  $$\forall t : Train \mid pos(t) \geq length(segm(t)) - d \wedge \neg\, alloc(nexts(segm(t))) = tid(t)$$
  $$\quad \Gamma spd'(t) = \max\{0, spd(t) - decmax \cdot \Delta t\}$$

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.