Formal Specification and Verification

– Formal specification

– Temporal logic

12.06.2012

Viorica Sofronie-Stokkermans

e-mail: sofronie@uni-koblenz.de

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.

Formal specification

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

  Model based specification

  transition systems, abstract state machines, timed automata — last time

  Axiom-based specification

  algebraic specification — last time

  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

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

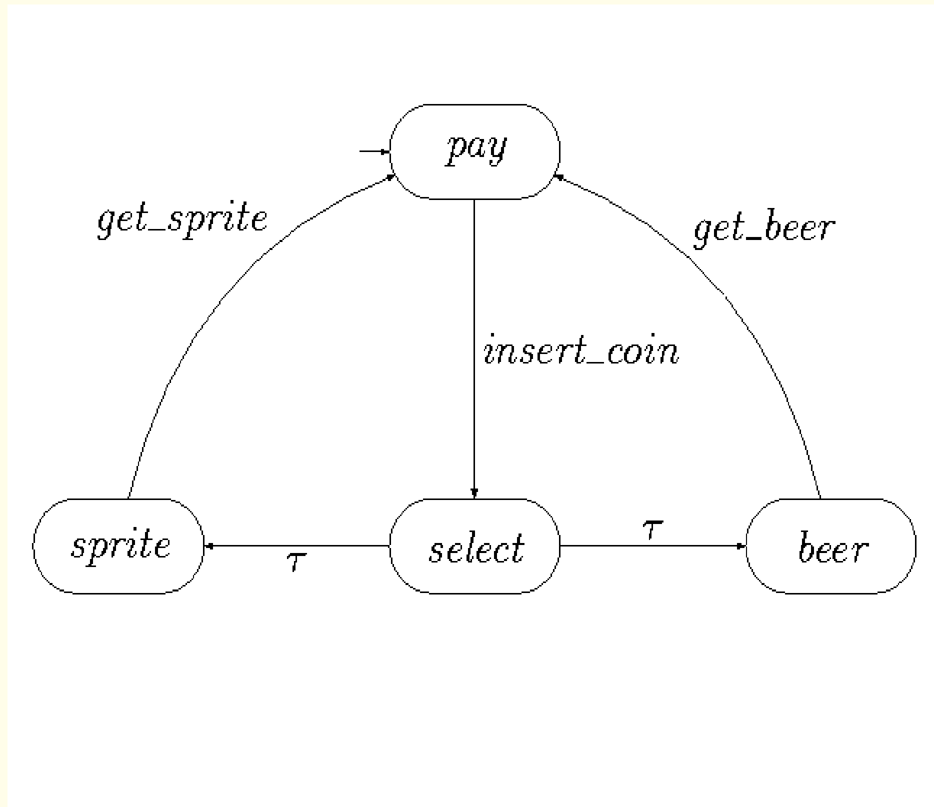
General idea:

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

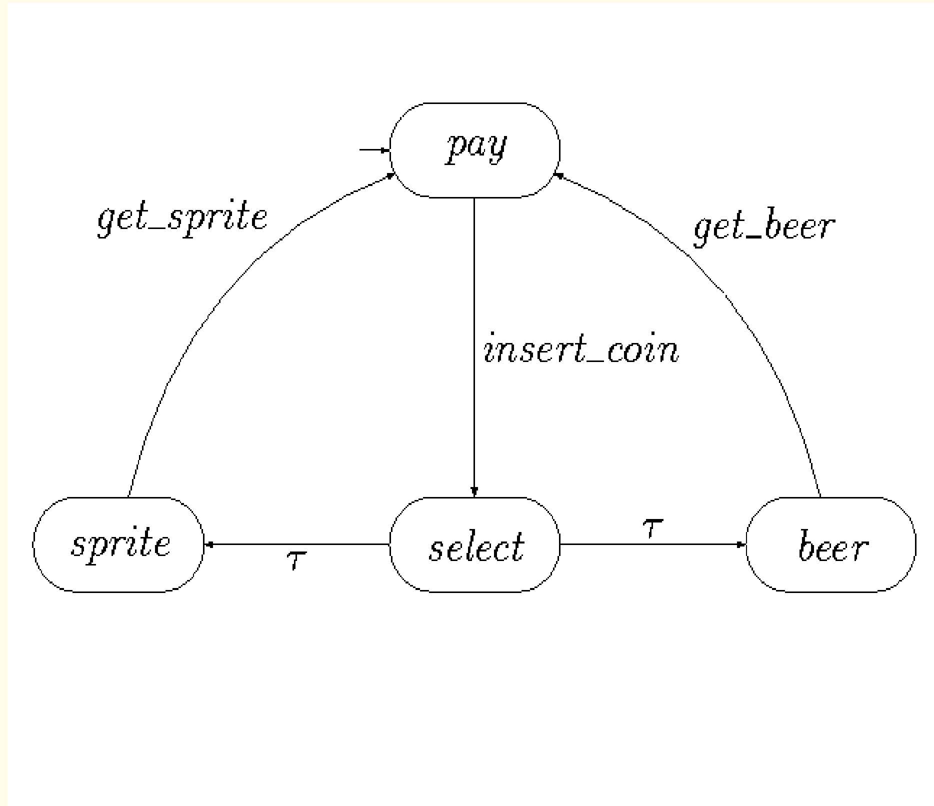
Prefix:

$P = a \rightarrow Q$ (a then Q)

where a is an event and Q a process

After event a, process P behaves like process Q

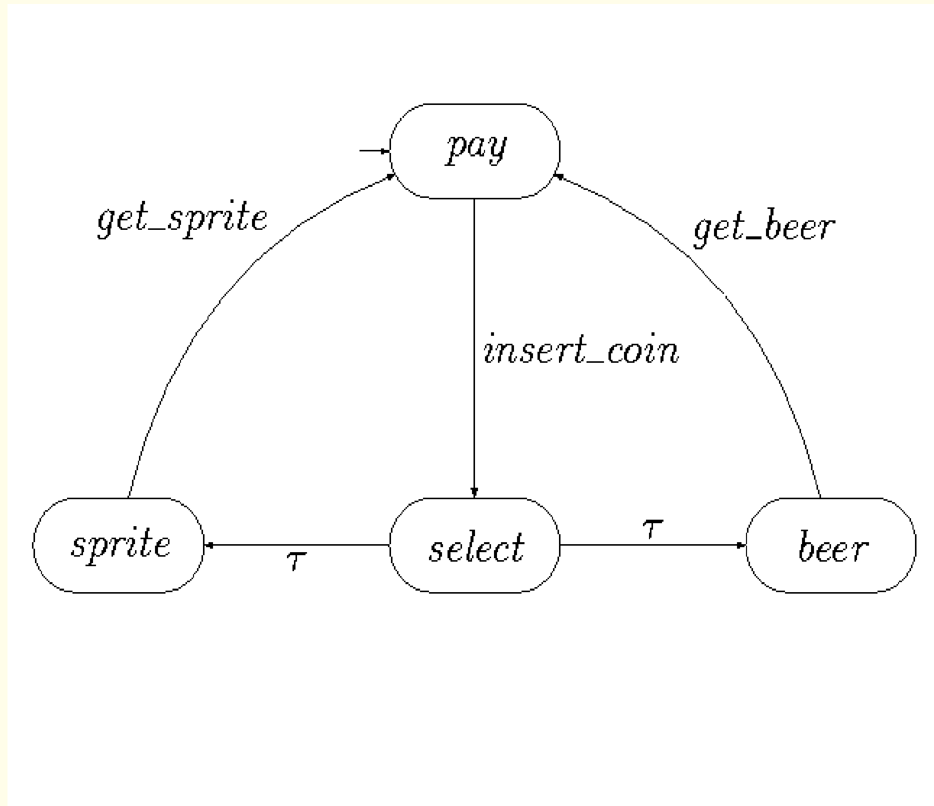
CSP: Example



A simple vending machine which consumes one coin before breaking

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

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

$$(tick \rightarrow CLOCK)$$

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

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

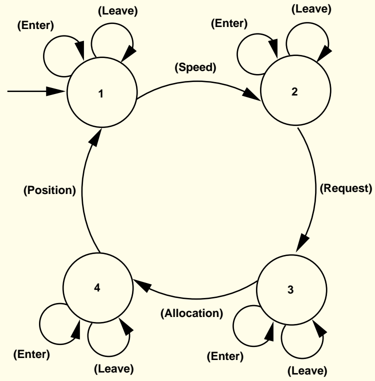
This can be regarded as an implicit definition of the behaviour of the clock.

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 $leave : [ls? : Segment;\ lt? : Train]$

local_chan $alloc, req, updPos, updSpd$

$$\mathtt{main} \stackrel{c}{=} ((enter \to \mathtt{main}) \ \square\ (leave \to \mathtt{main}) \ \square\ (updSpd \to State1))$$

$$State1 \stackrel{c}{=} ((enter \to State1) \ \square\ (leave \to State1) \ \square\ (req \to State2))$$

$$State2 \stackrel{c}{=} ((alloc \to State3) \ \square\ (enter \to State2) \ \square\ (leave \to State2))$$

$$State3 \stackrel{c}{=} ((enter \to State3) \ \square\ (leave \to State3) \ \square\ (updPos \to \mathtt{main}))$$

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

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

effect_updSpd

$\Delta(spd)$

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

⋮

CSP

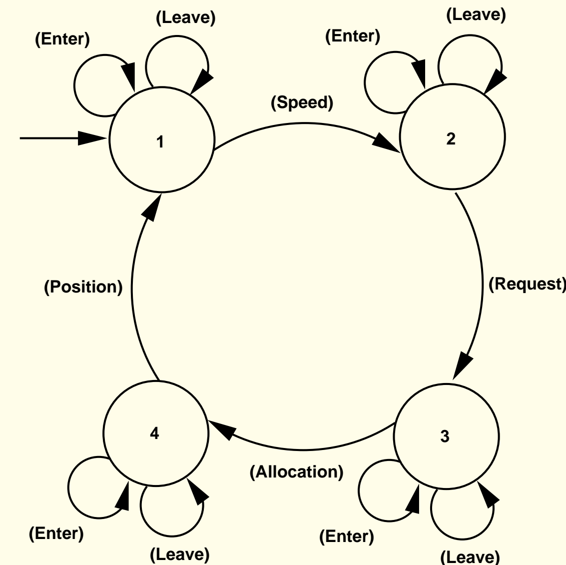
OZ

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: ————————————————————————

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

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

$\texttt{local_chan}$ $alloc, req, updPos, updSpd$

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

$State1 \stackrel{c}{=} ((req \to State2) \;\Box\; (leave \to State1) \;\Box\; (enter \to State1))$

$State2 \stackrel{c}{=} ((alloc \to State3) \;\Box\; (leave \to State2) \;\Box\; (enter \to State2))$

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

————————————————————————

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.

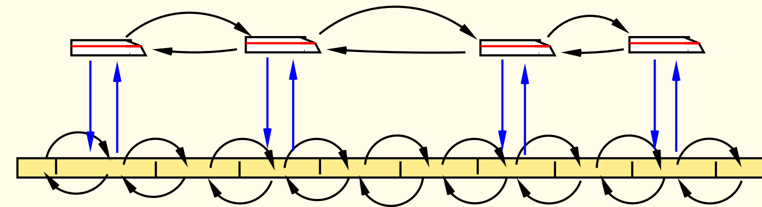
- 1. Data classes declare function symbols that can change their values during runs of the system

Data structures:

- 2-sorted pointers

train: trains

segm: segments



SegmentData

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

TrainData

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