Conception d'applications réactives

Prof. Jean-Marie Jacquet

Faculté d'informatique
Université de Namur, Belgique
Jean-Marie.Jacquet@unamur.be

Année académique 2024–2025

Part II: On Coordination Languages and Models

Prof. Jean-Marie Jacquet

Faculty of Computer Science
University of Namur, Belgium
Jean-Marie.Jacquet@unamur.be

Academic year 2024–2025

Content

UNIVERSITÉ DE NAMUR

Chapter I

Introduction

Table of contents

Project: default case

Subject

By using Bach/Anemone workbench, model and animate part of your project of INFO M453
"Laboratoire informatique ambiante et mobile"

Project: if default not possible

Subject

By using the coordination language Bach, model and animate a gymnastics competition

(see https://www.rookieroad.com/gymnastics/rules-and-regulations)

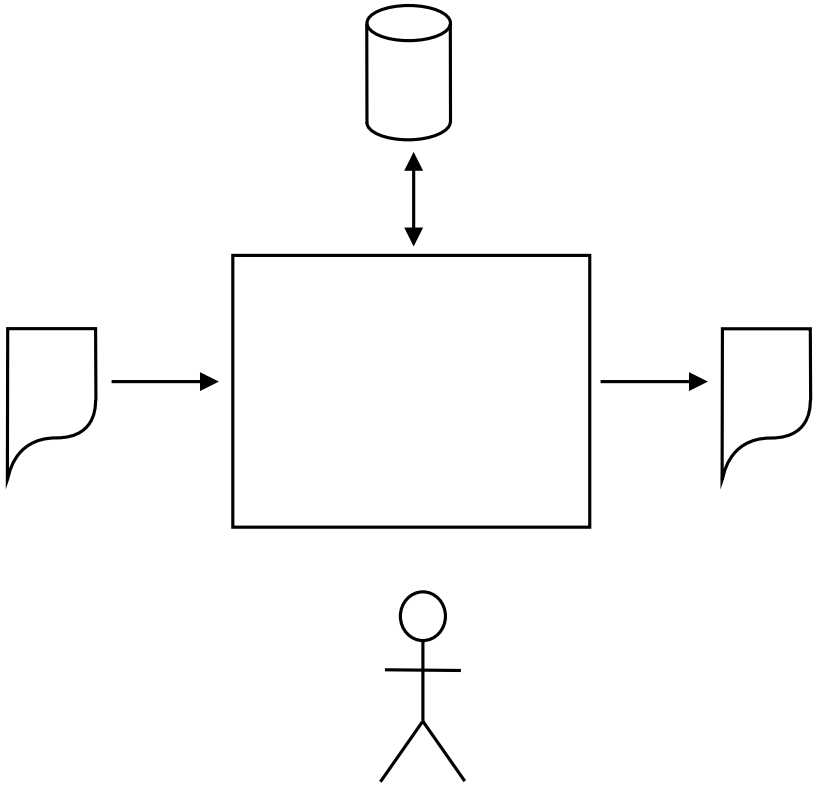
Ressources

- Bach simple interpreter: see Webcampus

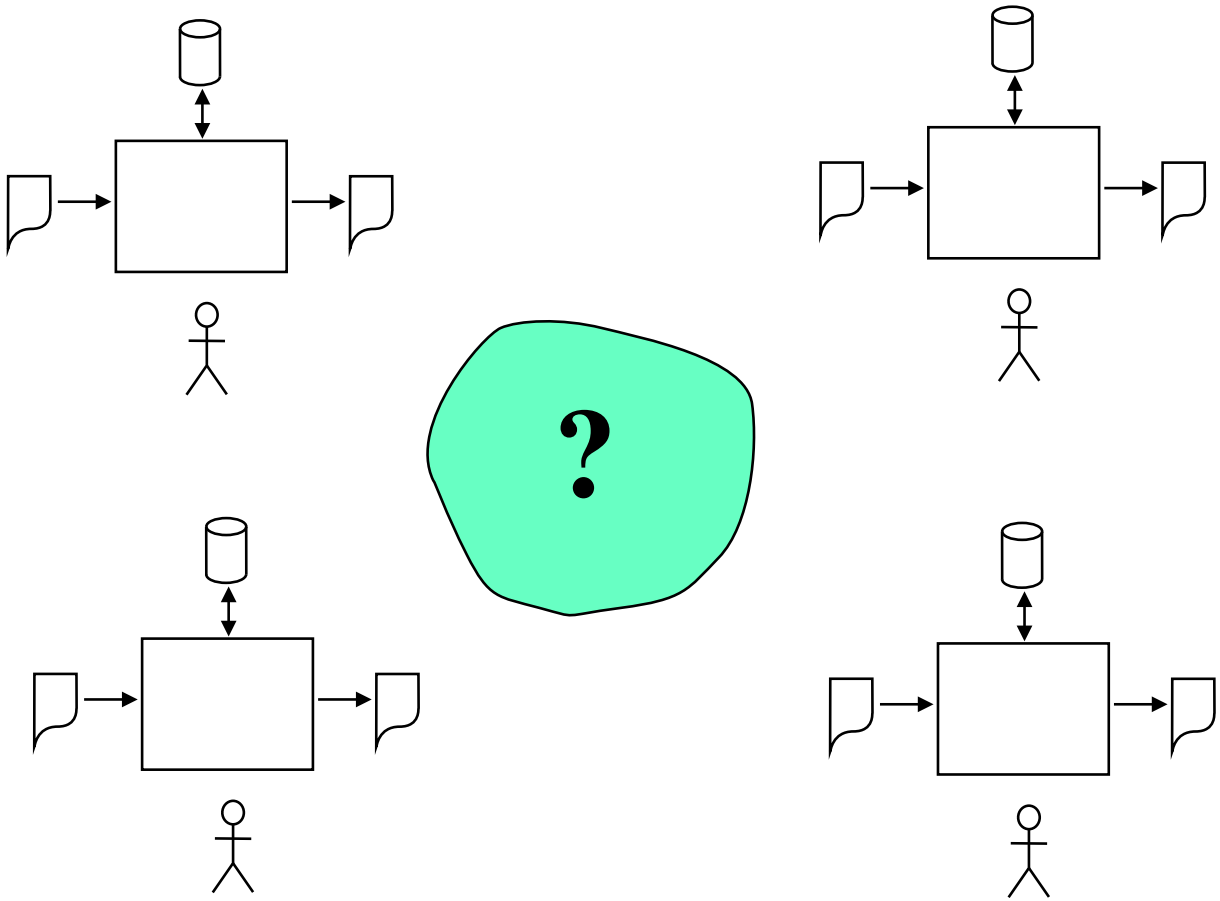
- Anemone workbench:
  - ☞ https://github.com/UNamurCSFaculty/anemone
  - ☞ see also Webcampus

Table of contents

Classical systems



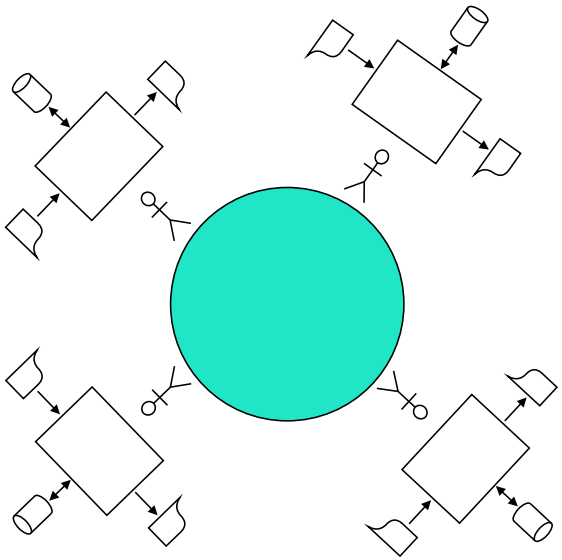
Modern systems



Different points of view

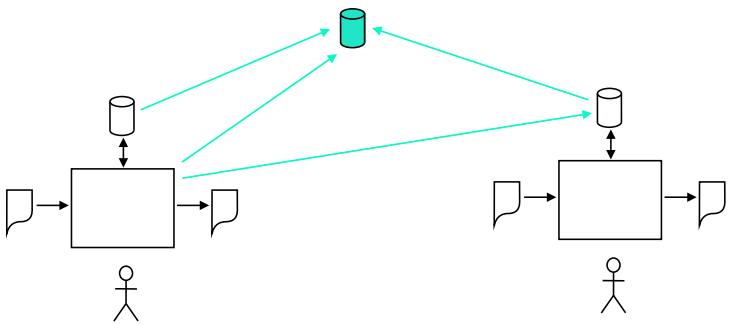
- Organizations
- Databases
- Messages
- Processes

Organizations



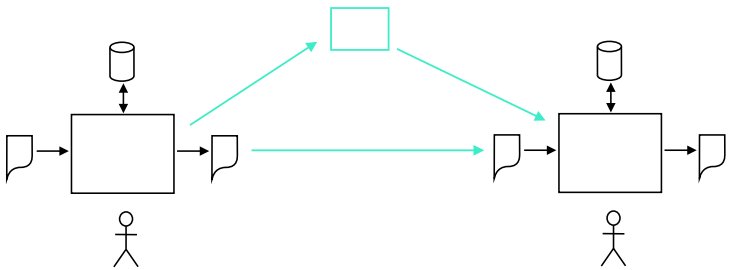
- organization structure
- agents
- roles
- desires

Databases



- DB structure
- DB integration
- code generation
- wrapper specifications and coding

Messages



- message syntax
- message semantics
- message coordination

Script langages

- Bourne shell (Unix)

```
ls -al > out.txt
mpage -P file.ps | lp -d folon
last | awk '{print $1}' | sort -u | rsh backus expand | awk '{print $1}
```

Script langages (cont'd)

- Manifold

```
process p:

   compute m1
   send m1 to q
   compute m2
   send m2 to q
   ...
   receive m
   ...
```

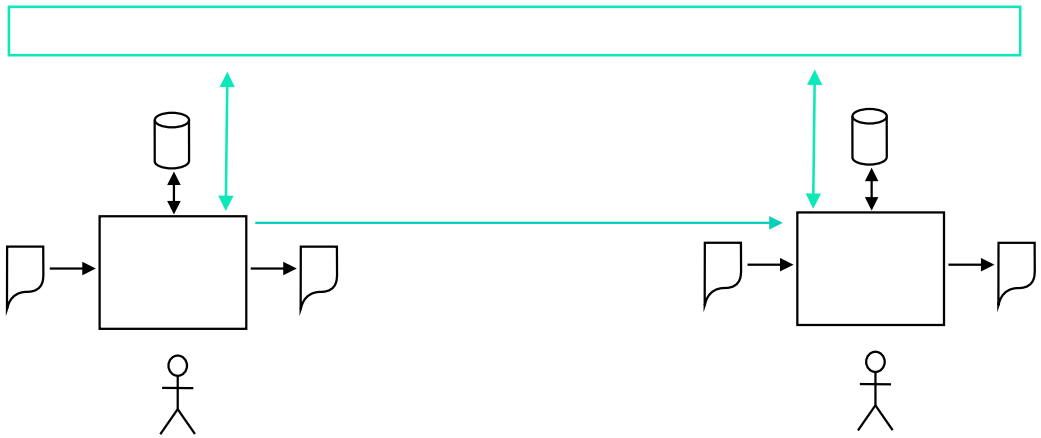
```
process q:

   ...
   receive m1
   let z be the sender of m1
   receive m2
   compute m using m1 and m2
   send m to z
   ...
```

```
process p:

   compute m1
   write m1 to
     output port o1
   compute m2
   write m2 to
     output port o2
   ...
   read m from
     input port i1
   ...
```

```
process q:

   read m1 from
     input port i1
   read m2 from
     input port i2
   compute m
     using m1 and m2
   write m to
     output port o1
```

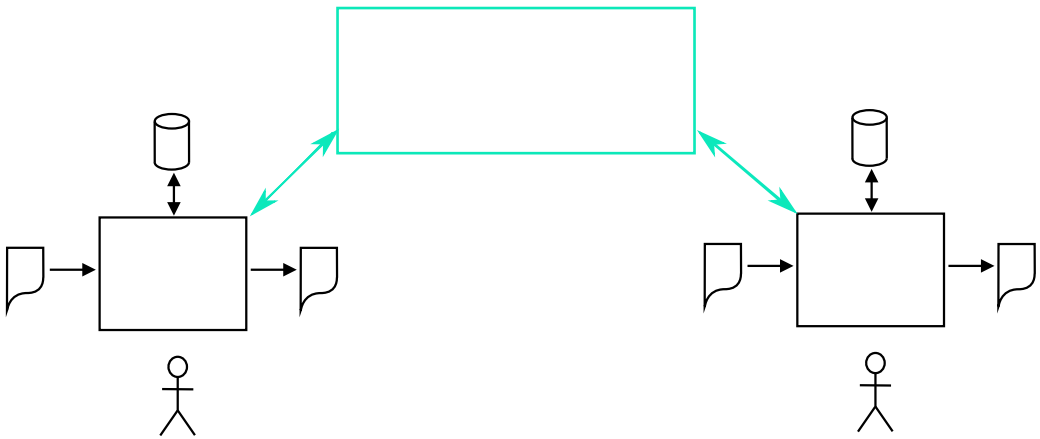
```
process c:

   create channel
      p.o1 -> q.i1
   create channel
      p.o2 -> q.i2
   create channel
      q.o1 -> p.i1
```

Processes

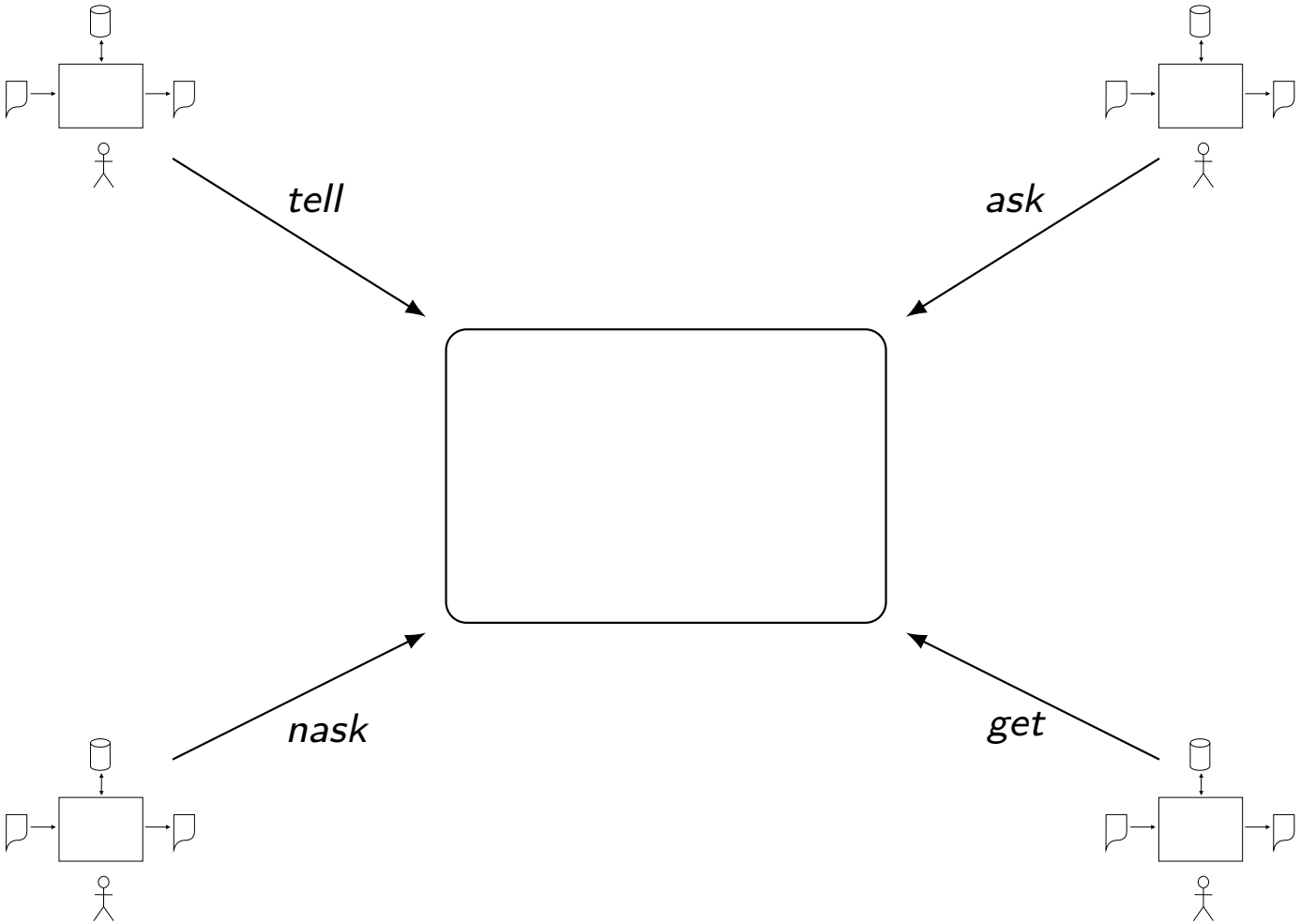


RMI, COM, DCOM, CORBA, ToolBus

Processes: coordination view



More concretely. . .



Objectives of the lecture

- expose work done in the CoordiNam laboratory
- expose basic mechanisms of coordination models and languages
- project : study through an implementation in Bach/Scala

Programming paradigms

1950s Machine langages

1960s Imperative programming

$\Rightarrow$ granularity and abstraction of actions

1970s Structured programming

$\Rightarrow$ programming methodologies

1980s Declarative programming/object-oriented programming

$\Rightarrow$ specification and interaction

Programming paradigms (cont'd)

1990s Component-based programming

$\Rightarrow$ (non sequential) patterns of interaction

2010s Reactive programming/Contextual programming

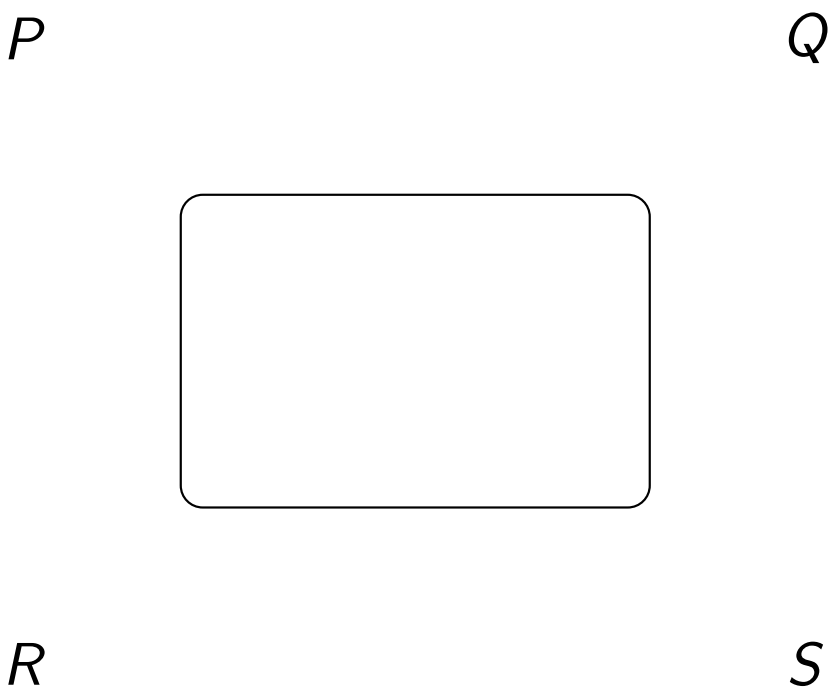
$\Rightarrow$ adaptation to new contexts

Chapter II

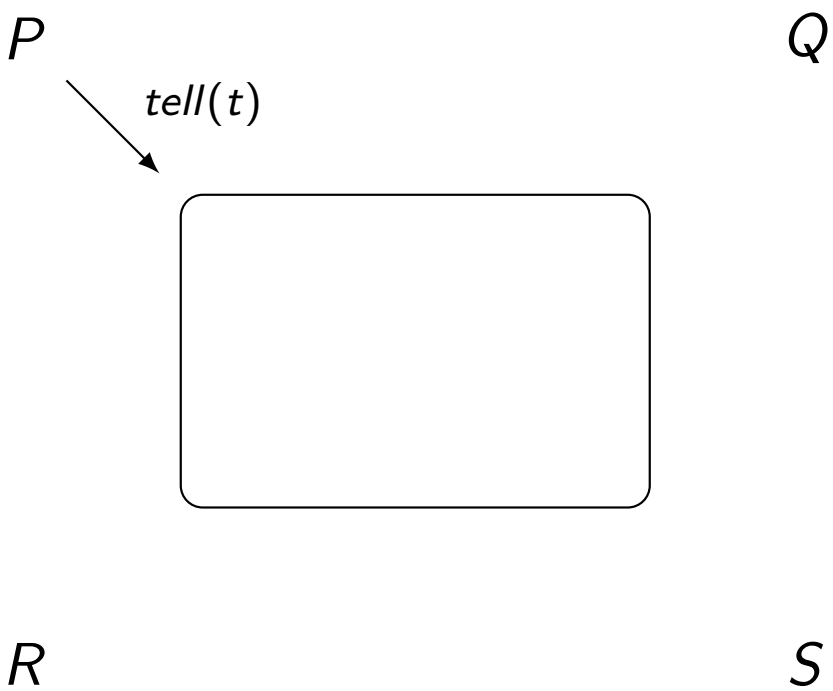
The Bach coordination language

Table of contents

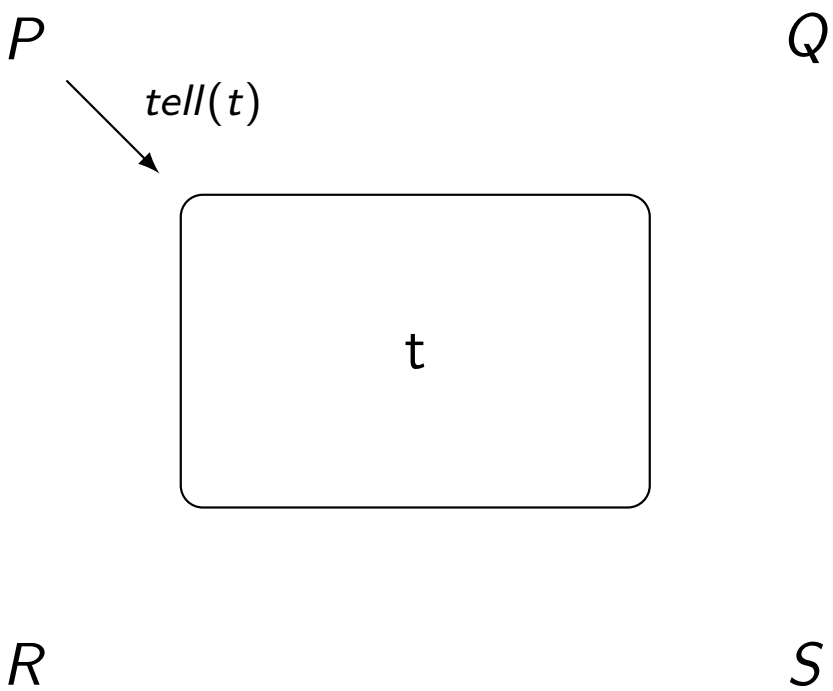
Snapshot



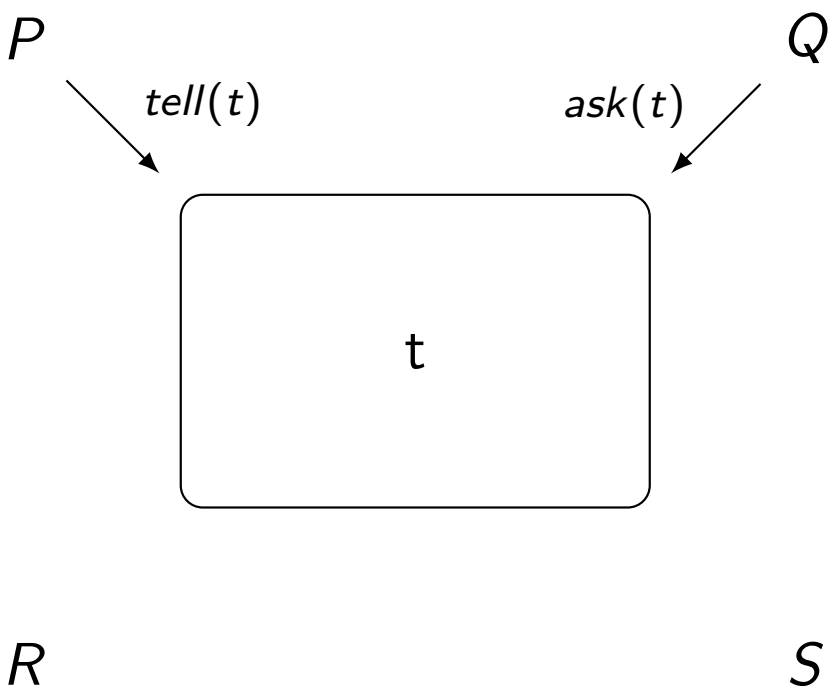
Snapshot



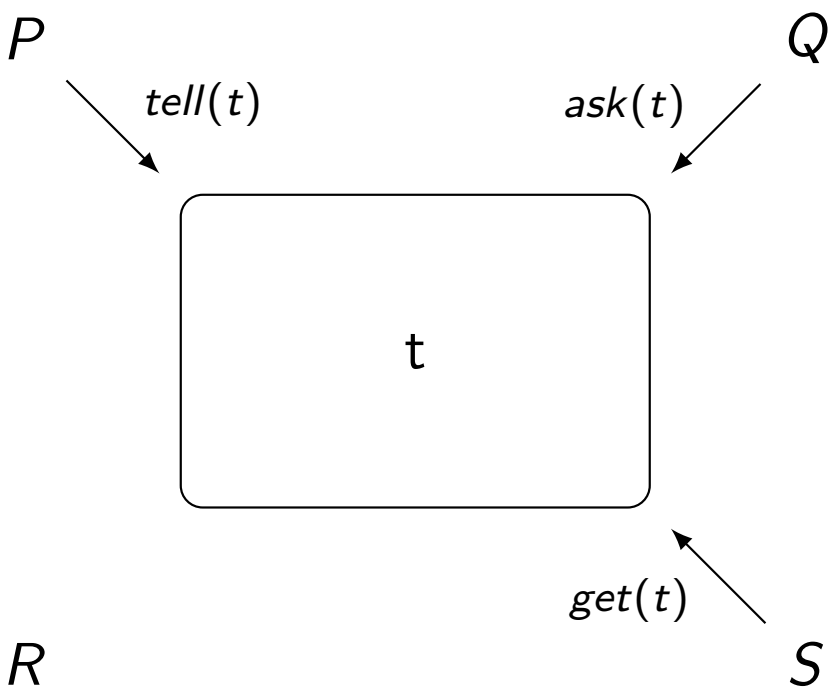
Snapshot



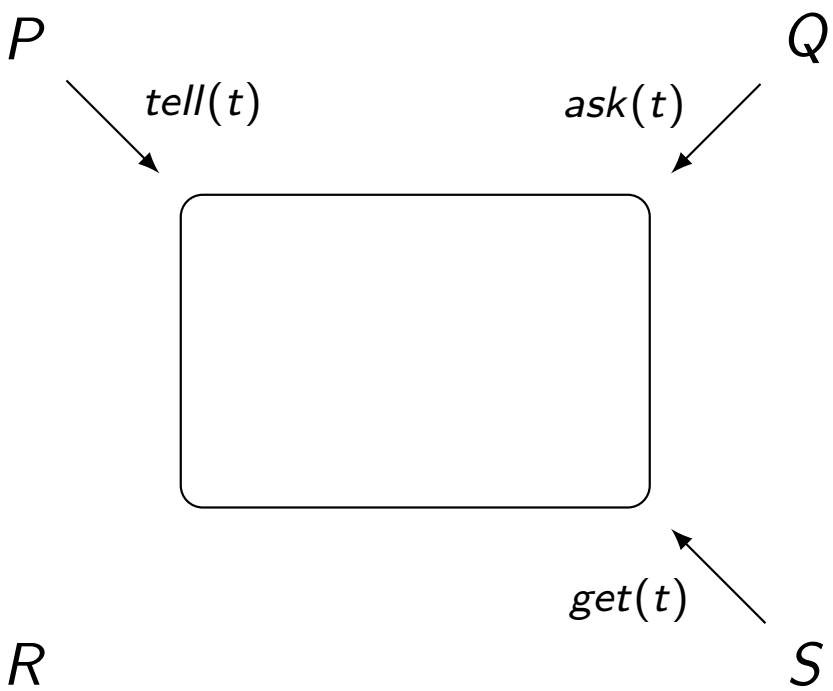
Snapshot



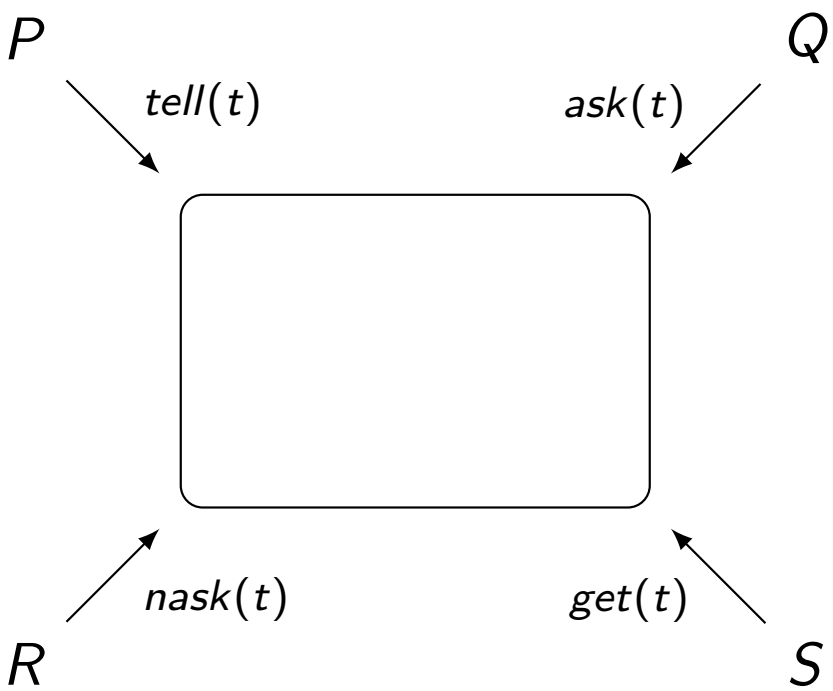
Snapshot



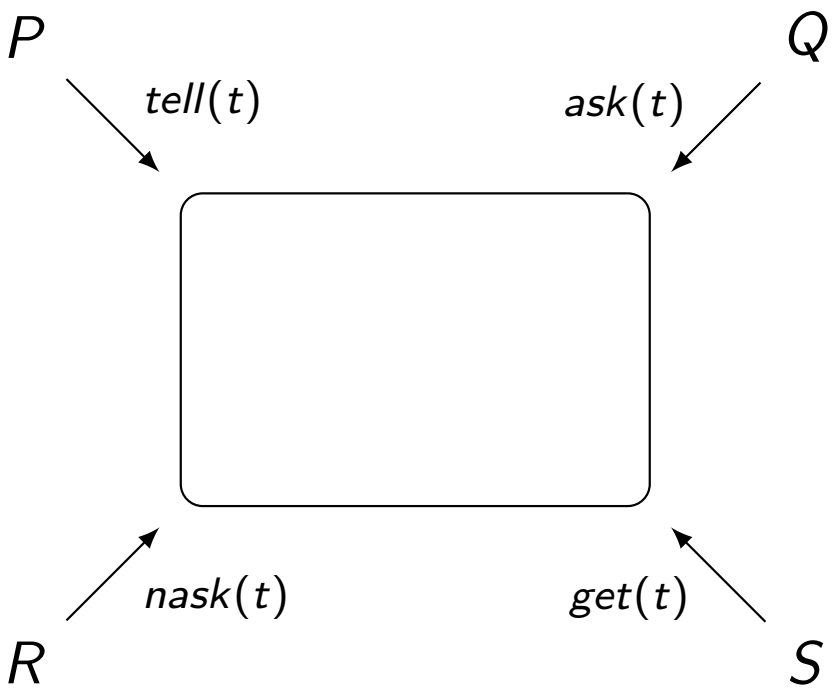
Snapshot



Snapshot



Snapshot



$$C ::= tell(t) \mid ask(t) \mid nask(t) \mid get(t)$$
$$A ::= C \mid A\,;\,A \mid A \,||\, A \mid A + A$$

Transition system (1)

$$<tell(t), \sigma> \longrightarrow <E, \sigma \cup \{t\}>$$

$$<ask(t), \sigma \cup \{t\}> \longrightarrow <E, \sigma \cup \{t\}>$$

$$\frac{t \notin \sigma}{<nask(t), \sigma> \longrightarrow <E, \sigma>}$$

$$<get(t), \sigma \cup \{t\}> \longrightarrow <E, \sigma>$$

Transition system (2)

$$\frac{<A, \sigma> \longrightarrow <A', \sigma'>}{<A \ ; \ B, \sigma> \longrightarrow <A' \ ; \ B, \sigma'>}$$

$$\frac{<A, \sigma> \longrightarrow <A', \sigma'>}{\begin{array}{c} <A \ || \ B, \sigma> \longrightarrow <A' \ || \ B, \sigma'> \\ <B \ || \ A, \sigma> \longrightarrow <B \ || \ A', \sigma'> \end{array}}$$

$$\frac{<A, \sigma> \longrightarrow <A', \sigma'>}{\begin{array}{c} <A + B, \sigma> \longrightarrow <A', \sigma'> \\ <B + A, \sigma> \longrightarrow <A', \sigma'> \end{array}}$$

Properties

- persistent broadcast communication ($><$ channel communication)
- associative memory
- clear separation between communication and computation

  $\Rightarrow$ "divide and conquer" methodology

  - code program fragments assuming that required data will eventually be available
  - compose these fragments by establishing that data is indeed provided.

  $\Rightarrow$ "coordination langage": allow for the composition of programs written in different languages

A restaurant room

$$
\begin{aligned}
room &= producer \;||\; harry \;||\; peter \\
producer &= prepare(Item); tell(Item); producer \\
harry &= get(ice); harry + get(cake); harry \\
peter &= get(water); peter + get(wine); peter
\end{aligned}
$$

Questions

1. $harry \stackrel{?}{=} harry'$

$$\begin{aligned} harry &= get(ice); harry + get(cake); harry \\ harry' &= (get(ice) + get(cake)); harry \end{aligned}$$

2. $paul \stackrel{?}{=} paul'$

$$\begin{aligned} paul &= get(water); get(bread); paul \\ &\quad + get(water); get(cheese); paul \\ paul' &= get(water); (get(bread) + get(cheese)); paul' \end{aligned}$$

Table of contents

In practice, how to construct programs?

- how to describe (real-life) problems?
- how to reason on the programs?
- how to be sure that what is described by the programs corresponds to what has to be modelled?

Objectives of Anemone

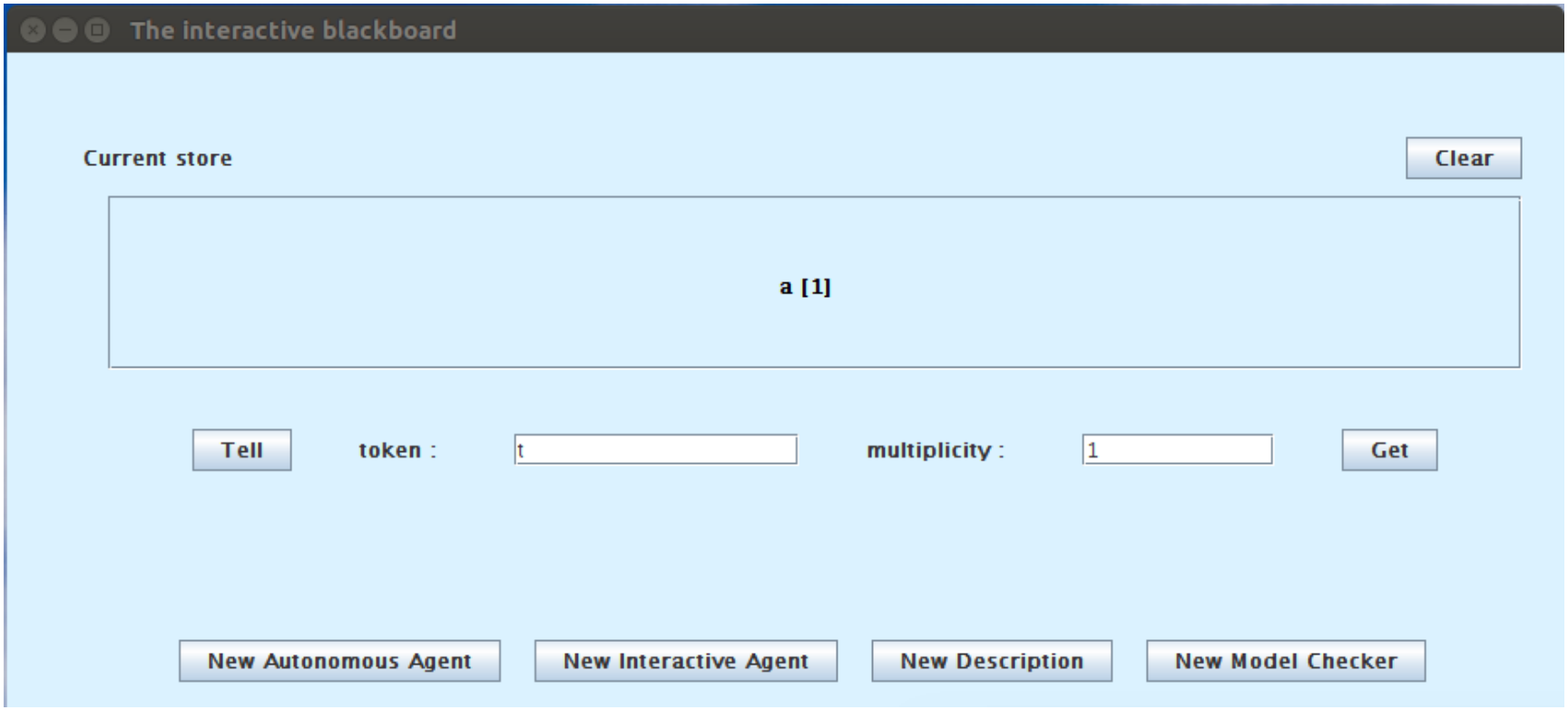
A workbench for Bach *that allows the user to*

- understand the execution of programs
  - ☞ by step by step executions
  - ☞ by automatic executions
  - ☞ by modifying the contents of the shared space
- grasp the modeling of the programs
  - ☞ by animating the execution of programs
- reason on programs
  - ☞ by model-checking properties
  - ☞ by producing traces that can be replayed

Properties of the workbench

- simple to use and to deploy
  - ☞ standalone executable file
  - ☞ process algebra that concentrates on key features
- direct relation between programs and their internal representations
  - ☞ better understanding on what is computed
  - ☞ production of meaningful traces

The Interactive Blackboard



Play with `( tell(a); get(b) ) || ( get(a); tell(b) )`

Exercice : the restaurant room in Bach

$$
\begin{aligned}
room &= producer \,||\, harry \,||\, peter \\
producer &= prepare(Item); tell(Item); producer \\
harry &= get(ice); harry + get(cake); harry \\
peter &= get(water); peter + get(wine); peter
\end{aligned}
$$

```
proc Room = Producer || Harry || Peter.
     Producer = (tell(ice) + tell(cake) +
                 tell(water) + tell(wine)); Producer.
     Harry = get(ice); Harry + get(cake); Harry.
     Peter = get(water); Peter + get(wine); Peter.
```

Exercice (cont'd)

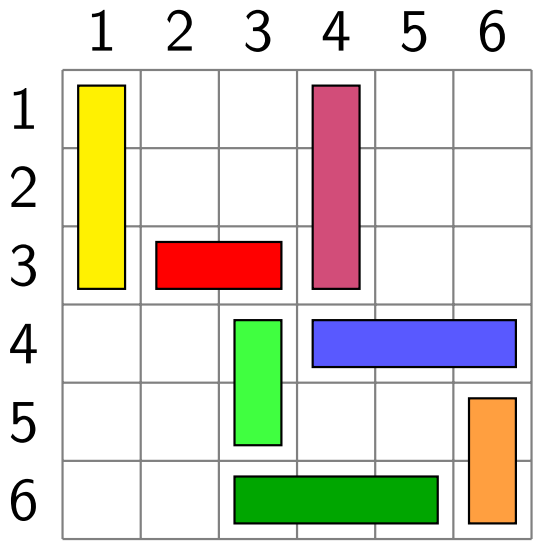
```
proc Room = Producer || Harry || Peter.
     Producer = (tell(ice) + tell(cake) +
                 tell(water) + tell(wine)); Producer.
     Harry = get(ice); Harry + get(cake); Harry.
     HarryBis = (get(ice) + get(cake)); HarryBis.
     Peter = get(water); Peter + get(wine); Peter.
     Paul = get(water); get(bread); Paul
          + get(water); get(cheese); Paul.
     PaulBis = get(water); ( get(bread) + get(cheese)) ; PaulBis.
```

- $Harry \stackrel{?}{=} HarryBis$
- $Paul \stackrel{?}{=} PaulBis$

☞ find contexts to distinguish them (if any)

Rush hour as a running example

Modeling rush-hour



- trucks and cars as concurrent agents
- competing through free places on the shared space

Language extension: data

- finite sets

```
eset RCInt = { 1, 2, 3, 4, 5, 6}.
```

- structured pieces of information
  - flat tokens: a, b, ..., t, u, ...
  - composed terms: $f(a_1, \ldots, a_n)$
    - ☞ free places as `free(i,j)`

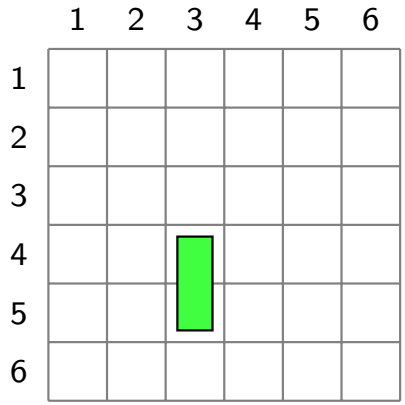
- maps and equations as rewriting rules

```
map down_car : Rows -> Rows.
eqn down_car(1) = 3.
    down_car(2) = 4.
```

Language extension: agents

$$A \quad ::= \quad Prim \mid Proc \mid A\,;A \mid A \,||\, A \mid A + A \mid C \rightarrow A \diamond A$$

```
proc VerticalCar(r: RCInt, c: RCInt) =
  ( (r>1 & r<5) -> ( get(free(pred(r),c));
                     tell(free(succ(r),c);
                     VerticalCar(pred(r),c) )
  +
  ( (r<5) -> ( get(free(down_car(r),c));
               tell(free(r,c));
               VerticalCar(succ(r),c)) ).
```



Language extension: animation

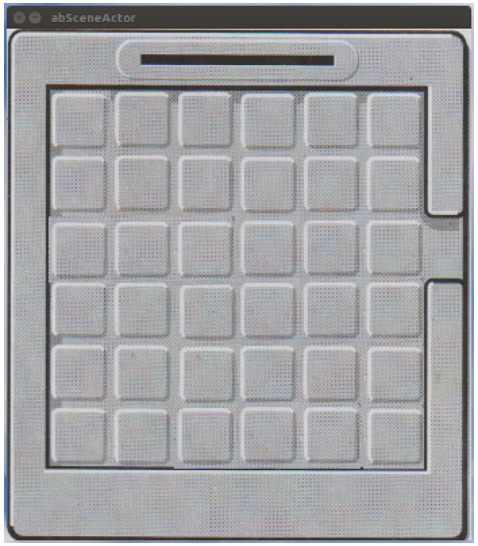
```
scene rhScene = {

    size = (562,618).
    layers = { top }.

    background = loadImage(Images/rh_empty.png).
    yellow_truck_img = loadImage(Images/yellow_truck.png).
    ...
    red_car_img_in = loadImage(Images/red_car_in.png).
    red_car_img_out = loadImage(Images/red_car_out.png).

    widget yellow_truck = {
        display = { yellow_truck_img }
        init = { wdL = top. }
    }

    ...
}
```

Language extension: animation (cont'd)

```
scene rhScene = {

    ...

    widget red_car = {
        attributes = { position in RCPlaces. }
        display = {
            position = in_grid -> red_car_img_in.
            position = out_grid -> red_car_img_out.
        }
        init = {
            wdL = top.
            position = in_grid.
        }
    }
}
```

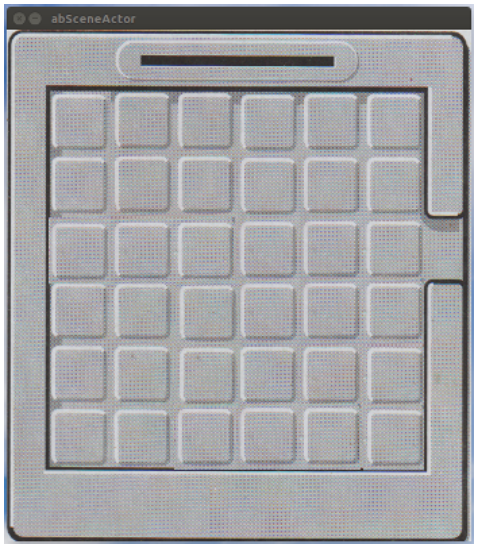
Language extension: animation (cont'd)

```
proc PlaceVehiclesOnScene =

    draw_scene;
```



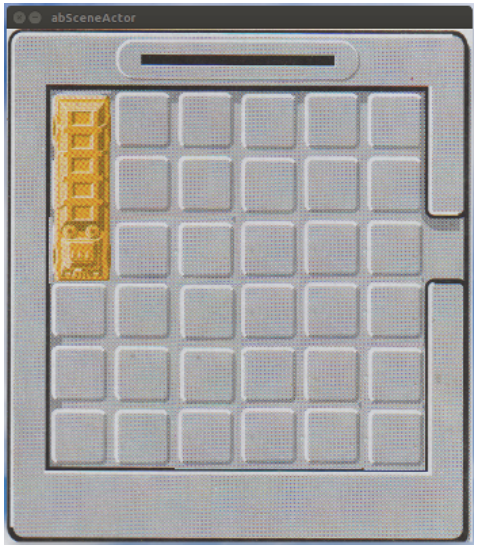
Language extension: animation (cont'd)

```
proc PlaceVehiclesOnScene =

        draw_scene;

        place_at(yellow_truck,55,80);
        ...
```

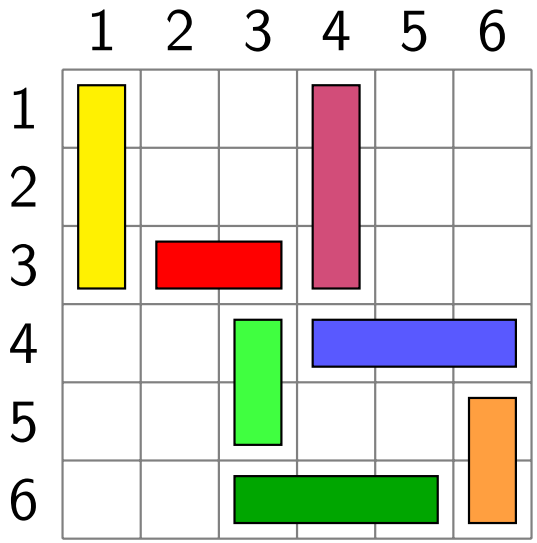


Language extension: animation (cont'd)

```
proc PlaceVehiclesOnScene =

        draw_scene;

        place_at(yellow_truck,55,80);
        ...

        show(yellow_truck);
        ...
```



Modeling rush-hour in Anim-Bach



$$\begin{aligned} \mathit{YellowTruck} &= \mathit{VerticalTruck}(1, 1, \mathit{yellow}) \\ \mathit{RedCar} &= \mathit{HorizontalCar}(3, 2, \mathit{red}) \end{aligned}$$

Rush-hour with animations

```
eset RCInt = { 1, 2, 3, 4, 5, 6 }.
     Colors = { yellow, green, blue, purple, red, orange }.

eqn x_vehicle(1) = 55.   x_vehicle(2) = 130. ...
    y_vehicle(1) = 80.   y_vehicle(2) = 155. ...

proc VerticalCar(r: RCInt, c: RCInt, p: Colors) =

       ( (r>1 & r<5) -> ( get(free(pred(r),c));
                          tell(free(succ(r),c));
                          MoveCar(p,pred(r),c);
                          VerticalCar(pred(r),c,p) ))
       +
       ( (r<5) -> ( get(free(down_car(r),c));
                    tell(free(r,c));
                    MoveCar(p,succ(r),c);
                    VerticalCar(succ(r),c,p) )).
```

Rush-hour with animations (cont'd)

```
eqn  x_vehicle(1) = 55.   x_vehicle(2) = 130. ...
     y_vehicle(1) = 80.   y_vehicle(2) = 155. ...

proc  MoveCar(p: Colors, r:RCInt, c: RCInt) =
   ( p = red -> move_to(red_car,x_vehicle(c),y_vehicle(r)) )
   +
   ( p = green -> move_to(green_car,x_vehicle(c),y_vehicle(r)) )
   +
   ( p = organge -> move_to(orange_car,x_vehicle(c),y_vehicle(r)) )
```

Solution by model checking

- Key information on the store: $\#free(1,1)$
- Basic formulae: equalities or inequalities involving integers and key information
  - ☞ $\#free(1,1) = 3$
- Propositional state formulae: combination of basic formulae by usual Boolean operators
- Scan linear temporal logic fragment:

  $TF ::= PF \mid Next\ TF \mid PF\ Until\ TF$

Code in AnimBach for rush-hour

☞ see file `rush_hour_without_GL`

Structure

- sets
- equations
- widgets
- procedures

Implementation

- Constructs as case classes
  - ☞ `AB_AST_Primitive(primitive: String, stinfo: AB_SI_ELM)`
  - ☞ `AB_AST_Agent(op: String, agi: AB_AG, agii: AB_AG)`

Implementation

- Constructs as case classes
  - ☞ `AB_AST_Primitive(primitive: String, stinfo: AB_SI_ELM)`
  - ☞ `AB_AST_Agent(op: String, agi: AB_AG, agii: AB_AG)`

Implementation (cont'd)

- Data as maps or lists
  - ☞ store, sets, procedure definitions as maps
  - ☞ map equations as lists

- Simulator as a repetition of transition steps
  - ☞ direct for primitives, sequential compositions
  - ☞ with random values for parallel and choice compositions

Implementation (cont'd)

- Data as maps or lists
  - ☞ store, sets, procedure definitions as maps
  - ☞ map equations as lists

- Simulator as a repetition of transition steps
  - ☞ direct for primitives, sequential compositions
  - ☞ with random values for parallel and choice compositions

Implementation (cont'd)

- Scene via Processing
  - ☞ declaration by means of the `setup` method
  - ☞ moving images by means of `draw` method (with linear interpolation)

- Temporal formulae by means of a home-made program
  - ☞ limited depth-first search based on simulator
  - ☞ breadth-first seach for `reach` properties

Implementation (cont'd)

- Scene via Processing
  - ☞ declaration by means of the `setup` method
  - ☞ moving images by means of `draw` method (with linear interpolation)

- Temporal formulae by means of a home-made program
  - ☞ limited depth-first search based on simulator
  - ☞ breadth-first seach for `reach` properties

Properties

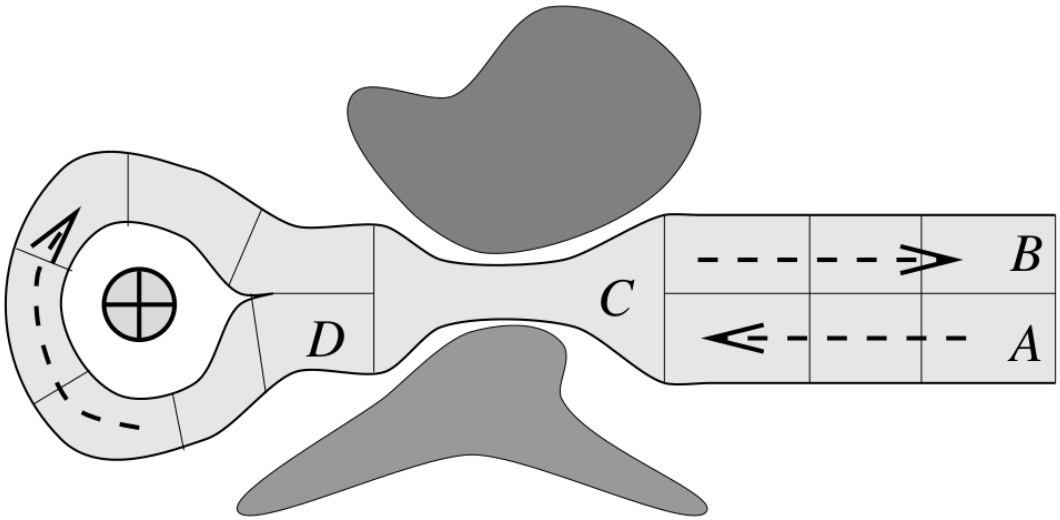
- Simple but easy to use workbench
  - low level (concurrent) executions of instructions and manipulations
  - graphical animations
  - model-checking of temporal formulae
- Process algebra Anim-Bach
  - definitions of sets, maps, scenes, processes
  - *tell*, *ask*, *get*, *nask* primitives
  - *place_at*, *move_to*, *hide*, *show* primitives
- Extensions
  - GUI refinement
  - more elaborated movements of images
  - more expressive logics and model checkers

Table of contents

Exercice 1: ping/pong

- Download the ping/pong template Ping_Pong.zip on Webcampus
- Fullfill the code of the scene `ppScene` given the images provided
- Code the `Pong` process by mimicking the `Ping` one

Exercice 2: round about



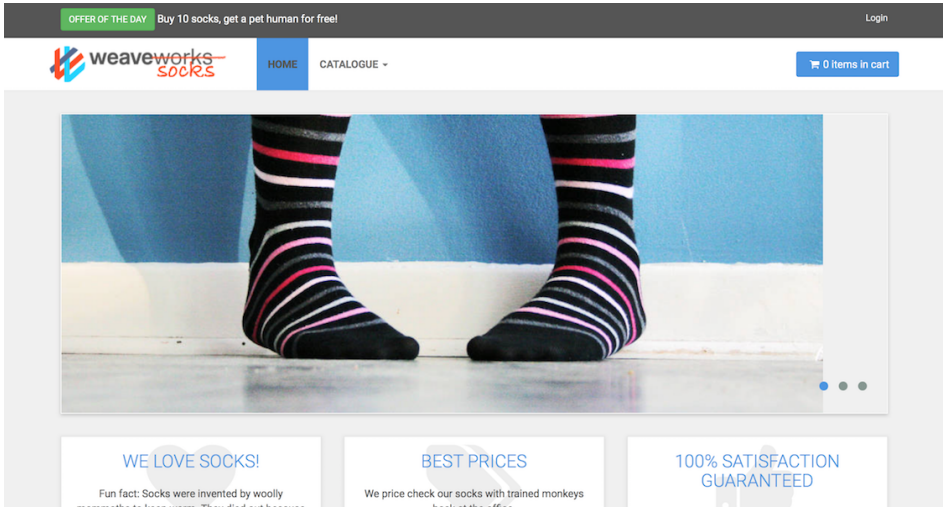
The above figure shows a road allowing visitors to drive round a monument. Cars enter the road at A and pass round the loop until they exit at B. They can only proceed forwardly (and thus never go back). A pair of rocks have reduced the road in C in such a way that only one car at a time can pass.

Exercice 2 (cont'd)

- Download the round about template Round_About.zip on Webcampus
- Fullfill the code of `round_about.mbach`
- Code each car as a process and segments as elements of the tuple space
- Give out-going (ie rightward in the picture) cars priority over in-coming ones
  - ☞ an in-coming car is not allowed to enter the single-lane segment C until one segment of D is empty.

Table of contents

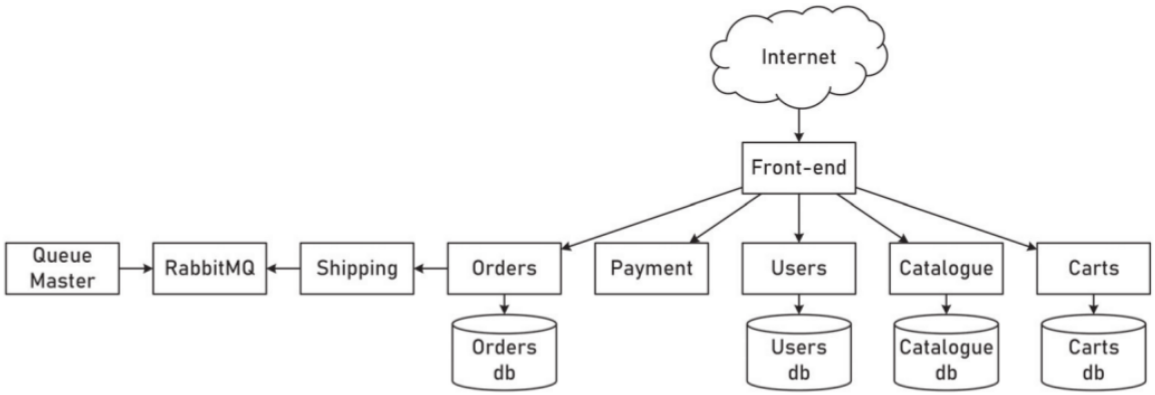
The Sock Shop Application



Sock Shop is an open source web application that simulates a user facing part of an e-commerce website that sells socks. It is a multi-component application developped for testing micro-services.

In short, the sock shop application is based on a front-end which allows a user to identify himself, to browse a catalogue, to include items in a cart, to place an order, to proceed to a payment and to have the order being shipped.

The Sock Shop Application (cont'd)



Following the document on the GitHub repository, the sock shop application obey to the architecture above. More information can be found at https://microservices-demo.github.io

Model this application

- in the form of a "sequential" specification
- in the form of micro-services running in parallel and activated on demand

Chapter III

Extensions

Table of contents

Processes as active data

Data

tell(t), get(t), ask(t), nask(t)

Processes

tellp(t), getp(t), askp(t), naskp(t)

Flight reservation system

- Problem description
  - four flights: `ba023`, `sn720`, `nw129`, `kl283`
  - questioned by three terminals (identified as 1, 2 and 3)
  - by means of `reservation` messages.

- Code

```
proc Flight_syst =
        tellp(Terminal(1)); tellp(Terminal(2));
        tellp(Terminal(3));
        tellp(Flight(ba023,80)); tellp(Flight(sn720,150));
        tellp(Flight(nw129,68)); tellp(Flight(kl283,185)).
```

Code (cont'd)

```
proc Terminal(id: sIds) =
        User_input(id, flight, seatsRequested);
        tell(reservation(flight, seatsRequested));
        get(acknowledge(flight, seatsRequested, seatsLeft, status));
        User_output(id, seatsLeft, status);
        Terminal(id).

    Flight(name: sNames, seatsAvailable: sInt) =
        get(reservation(name, seatsRequested)),
        Reserve_seats(seatsRequested, seatsAvailable, seatsLeft, status),
        tell(acknowledge(name, seatsRequested, seatsLeft, status),
        Flight(name, seatsLeft).

    User_input(...) = ...

    User_output(...) = ...

    Reserve_seats(...) = ...
```

Table of contents

Enhanced matching

Communication variables

- keep the communication through the tuple space
- use special variables to input/output data to the "host" program

Default matching

- use a left-to-right parsing
- a value matches that same value
- a communication variable matches any value

Psi-terms

- use pairs of item name – value
- perform the matching according to the item name

Formal definition (1)

ψ-term

Construct of the form $f(a_1 = v_1, \cdots, a_m = v_m)$ where

- f/n is a functor such that $m \leq n$
- the a_i's are distinct constants
- v_i denotes an integer, a string of characters, a ψ-term or a communication variable
- any communication variable appears at most once

Closed *psi*-term

A ψ-term is said to be closed if it contains no communication variable.

Formal definition (2)

Correspondance

$\psi_1 = f(a_1 = v_1, \cdots, a_l = v_l)$ corresponds to
$\psi_2 = f'(a'_1 = v'_1, \cdots, a'_m = v'_m)$ iff

1. f and f' are identical functors with same arities;
2. $\{a_i : 1 \leq i \leq l\} \subseteq \{a'_j : 1 \leq j \leq m\}$;
3. for any i such that v_i is an integer or a string of characters, if $a_i = a'_j$ then $v_i = v'_j$;
4. for any i such that v_i is a ψ-term, if $a_i = a'_j$ then v_i corresponds to v'_j.

Formal definition (3)

Binding

$$\theta : \mathit{Scvar} \rightarrow (N \cup \mathit{Sstring} \cup \mathit{Scpterm} \cup \{\bot\})$$

Matching

Let $\psi_1 = f(a_1 = v_1, \cdots, a_l = v_l)$, $\psi_2 = f'(a'_1 = v'_1, \cdots, a'_m = v'_m)$.
Assume ψ_2 is closed.
ψ_1 matches ψ_2 iff $\psi_1\theta$ corresponds to ψ_2, for some binding θ.

Table of contents

Approaches to distribution

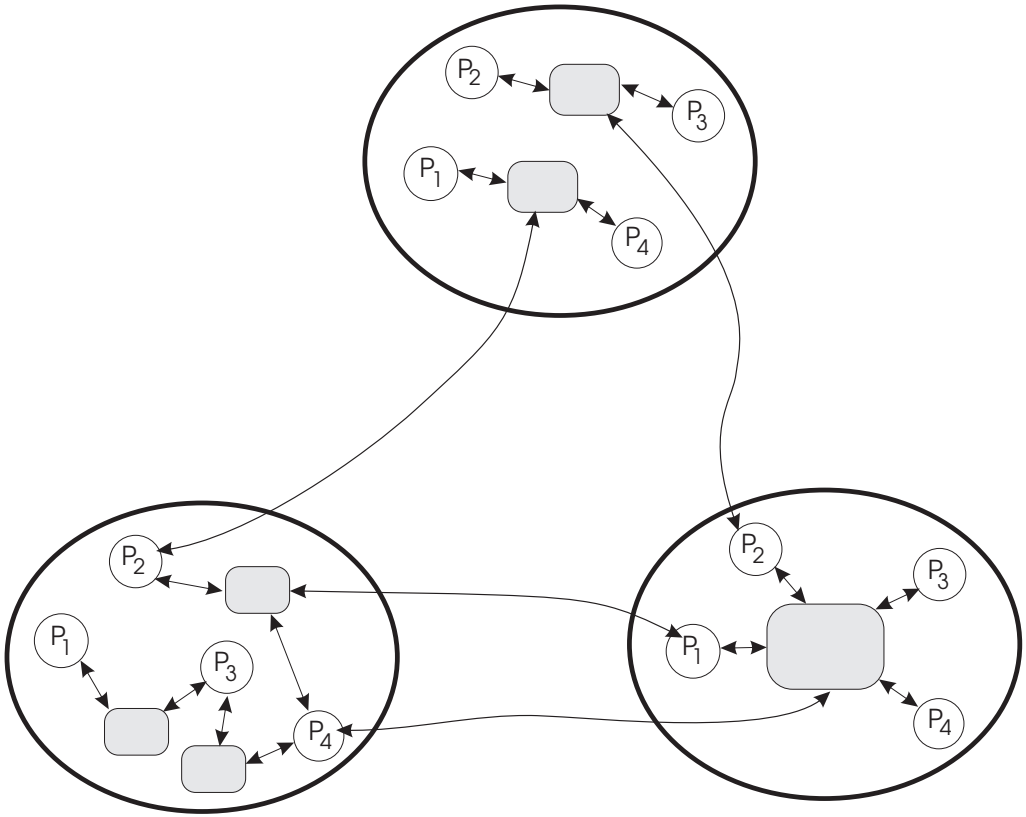
- *Classical approach:*
  - distribute a single entity (eg the dataspace)
  - use a manager to coordinate the pieces
  - think of the entity as a non-fractioned object

- *Our approach:*
  - coordinate several applications
  - avoid the use of a master manager

Four steps

1. introduce multiple blackboards,
2. distribute blackboards on computing resources,
3. introduce aliases allowing access to non-local blackboards,
4. perform load balancing by moving blackboards between locations.

Distribution in Bach



Multiple blackboards

- **Concept**

  $$Blackboard = dataspace + processes$$

- **The Bach approach**
  - blackboards manipulated by *tell*, *get*, *ask*, *nask* primitives,
  - processes can access data and processes on any blackboard.

Distribution

- **Concepts**
  - A *processor* consists of computing resources.
  - An *application* consists of the executions of several blackboards launched by a common initial instruction.
  - An *abstract machine* consists in a pair processor-application.
- **The Bach approach**
  - blackboards are created locally;
  - an execution in Bach is composed of the concurrent executions of applications;
  - the execution of a blackboard is made with respect to a program attached to the application to which the blackboard belongs;
  - access to nonlocal blackboards takes place via special links.

Aliases

- **Concepts**

  For the purposes of dynamic evolution, an extra indirection on the blackboard naming is desirable

- **The Bach approach**

  Introduce *virtual blackboards* as special blackboards containing no data and no processes but pointing to other (possibly virtual) blackboards.

Load balacing

- **Concepts**

  To dynamically balance the load of the computations, parts of executions may need to migrate from one place to another.

- **The Bach approach**

  Introduce primitives to exchange blackboards and to relink them.

Language: communication primitives

$tellbb(bbn, bbt, bbp)$	$tell(bbn, t)$	$tellp(bbn, p)$
$getbb(bbn)$	$get(bbn, t)$	$getp(bbn, p)$
$askbb(bbn)$	$ask(bbn, t)$	$askp(bbn, p)$
$naskbb(bbn)$	$nask(bbn, t)$	$naskp(bbn, p)$

$tellvb(bbn_1, bbn_2@ma_2)$
$relink(bbn_1, bbn_2@ma_2)$,
$exch(bbn_1, bbn_2@ma_2)$

Auxiliary concepts

- **Processes:** $(\Leftarrow G \diamond \theta)$
- **Contexts:**
  - ∇ is a context s.t. $\nabla[\![A]\!] = A$
  - If c is a context and if A is an agent, then

$$c \,;\, A \qquad c \parallel A \qquad A \parallel c$$

are contexts s.t.

$$\begin{aligned}(c \,;\, A)[\![A']\!] &= c[\![A']\!] \,;\, A \\ (c \parallel A)[\![A']\!] &= c[\![A']\!] \parallel A \\ (A \parallel c)[\![A']\!] &= G \parallel c[\![G']\!]\end{aligned}$$

- **Configurations:** sets of elements of the form
  $\langle bbn@ma, bbt, bbp \rangle$. (real blackboard)
  $\langle bbn_1@ma_1 \rightsquigarrow bbn_2@ma_2 \rangle$. (virtual blackboard)

Tell reductions: Real blackboard

$$(\mathsf{T}_b) \qquad \{\!|\ \langle n@ma, bt, m[\Leftarrow c[\![tellbb(bbn, bbt, bbp)]\!]\Diamond\theta]\rangle\ |\!\} \rightarrow$$

$$\{\!|\ \langle n@ma, bt, m[\Leftarrow c[\![\Box]\!]\Diamond\theta]\rangle\ |\!\} \cup \{\langle bbn@ma, bbt\theta, bbg'\rangle\}$$

$$if \left\{ \begin{array}{l} \text{no blackboard is identified by } bbn@ma \text{ in the initial} \\ \qquad \text{configuration} \\ bbt\theta \text{ is composed of closed } \psi\text{-terms} \\ bbg' \text{ is obtained from } bbg\theta \\ \qquad \text{by freshly renaming the communication variables} \end{array} \right\}$$

Tell reduction: virtual blackboard

$$(\mathsf{T}_v) \qquad \{\!|\ \langle n@ma, bt, m[\Leftarrow c[\![tellvb(bbn_1, bbn_2@ma_2)]\!]\Diamond\theta]\rangle\ |\!\} \rightarrow$$

$$\{\!|\ \langle n@ma, bt, m[\Leftarrow c[\![\Box]\!]\Diamond\theta]\rangle\ \langle bbn_1@ma \rightsquigarrow bbn_2@ma_2\rangle\ |\!\}$$

$$if \left\{ \begin{array}{l} \text{there exists a blackboard in the initial} \\ \text{configuration identified by } bbn_2@ma_2 \end{array} \right\}$$

Tell reduction: term

$$(\mathsf{T}_t) \qquad \{\!| \ \langle n@ma, bt, m[\Leftarrow c[\![tellt(bbn, t)]\!]\Diamond\theta]\rangle \ \langle bbn_0@ma_0 \rightsquigarrow bbn_1@ma_1\rangle \ \cdots \ \langle bbn_{i-1}@ma_{i-1} \rightsquigarrow bbn_i@ma_i\rangle \ \langle bbn_i@ma_i, bt', bp'\rangle \ |\!\} \rightarrow$$

$$\{\!| \ \langle n@ma, bt, m[\Leftarrow c[\![\triangle]\!]\Diamond\theta]\rangle \ \langle bbn_0@ma_0 \rightsquigarrow bbn_1@ma_1\rangle \ \cdots \ \langle bbn_{i-1}@ma_{i-1} \rightsquigarrow bbn_i@ma_i\rangle \ \langle bbn_i@ma_i, bt' + \{u\}, bp'\rangle \ |\!\}$$

$$if \left\{ \begin{array}{l} u = t\theta \text{ is closed} \\ bbn_0 = bbn \\ 0 \leq i \end{array} \right\}$$

Get reductions: virtual blackboard

$$(\mathsf{G}_v) \qquad \{\!|\ \langle n@ma, bt, m[\Leftarrow c[\![getbb(bbn)]\!]\Diamond\theta]\rangle\ \langle bbn@ma \rightsquigarrow bbn'@ma'\rangle\ |\!\} \rightarrow$$

$$\{\!|\ \langle n@ma, bt, m[\Leftarrow c[\![\Box]\!]\Diamond\theta]\rangle\ |\!\}$$

Exchange reduction

(E_1) $\quad \{\!| \ \langle n@ma, bt, m[\Leftarrow c[\![exch(bbn_1, bbn_2@ma_2)]\!]\Diamond\theta]\rangle \ \langle bbn_1@ma, bbt, bbp\rangle \ \langle bbn_2@ma_2 \rightsquigarrow bbn_1@ma\rangle \ |\!\} \rightarrow$

$\{\!| \ \langle n@ma, bt, m[\Leftarrow c[\![\Box]\!]\Diamond\theta]\rangle \ \langle bbn_1@ma \rightsquigarrow bbn_2@ma_2\rangle \ \langle bbn_2@ma_2, bbt, bbp\rangle \ |\!\}$

(E_2) $\quad \{\!| \ \langle n@ma, bt, m[\Leftarrow c[\![exch(bbn_1, bbn_2@ma_2)]\!]\Diamond\theta]\rangle \ \langle bbn_1@ma \rightsquigarrow bbn_3@ma_3\rangle \ \langle bbn_2@ma_2 \rightsquigarrow bbn_1@ma\rangle \ |\!\} \rightarrow$

$\{\!| \ \langle n@ma, bt, m[\Leftarrow c[\![\Box]\!]\Diamond\theta]\rangle \ \langle bbn_1@ma \rightsquigarrow bbn_2@ma_2\rangle \ \langle bbn_2@ma_2 \rightsquigarrow bbn_3@ma_3\rangle \ |\!\}$

Relink reduction

(Rel) $\quad \{\!|\ \langle n@ma, bt, m[\Leftarrow c[\![relink(bbn_1, bbn_3@ma_3)]\!]\Diamond\theta]\rangle \ \langle bbn_1@ma \rightsquigarrow bbn_2@ma_2\rangle\ |\!\}$

$$\rightarrow \{\!|\ \langle n@ma, bt, m[\Leftarrow c[\![\Box]\!]\Diamond\theta]\rangle \ \langle bbn_1@ma \rightsquigarrow bbn_3@ma_3\rangle\ |\!\}$$

$$if \left\{ \begin{array}{l} \text{there exists a blackboard in the initial} \\ \text{configuration identified by } bbn_3@ma_3 \end{array} \right\}$$

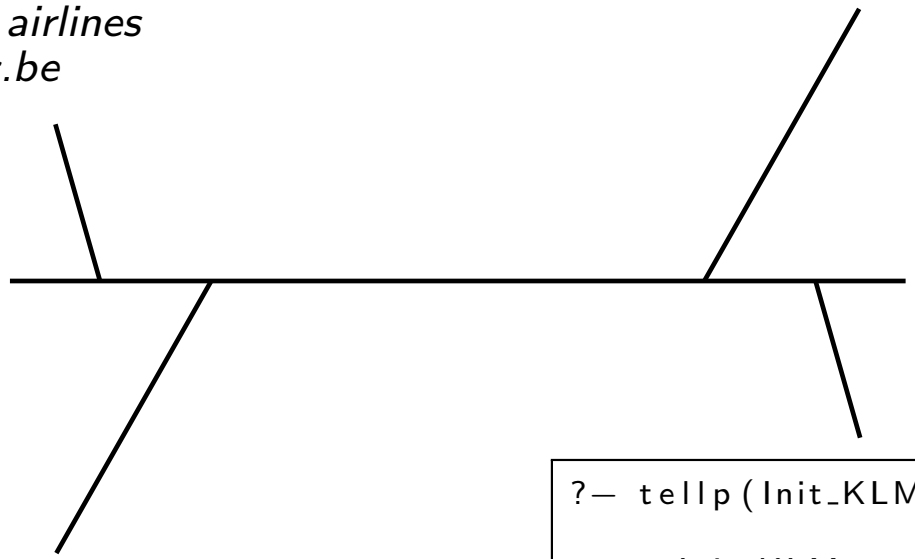
Coordinating travel agencies

```
?- tellp(Init_SN)

proc Init_SN =
   tellbb(sn_bb,[],[]);
   tellp(Flight(sn,sn023,80));
   tellp(Flight(sn,sn720,150));
   tellp(PartnerFlight(sn,lh001)).
```

Brussels airlines
batc.be

```
?- tellp(TravelAgent(2))
```

Travel agent 2
sun.sp



```
?- tellp(TravelAgent(1))
```

Travel agent 1
cuvelier.be

```
?- tellp(Init_KLM)

proc Init_KLM =
   tellbb(klm_bb,[],[]);
   tellp(Flight(klm,kl283,120));
   tellp(Flight(klm,kl034,50));
   tellp(PartnerFlight(klm,af005)).
```

KLM
schipol.nl

Travel agent

```
proc Travelagent(id: sIds) =
        User_input(flight,seatsRequested);
        tellvb(vbb,carrier(flight));
        tell(vbb,reservation(flight,seatsRequested,id));
        get(vbb,acknowledge(id,seatsLeft,status));
        getbb(vbb);
        User_output(seatsLeft,status);
        Travelagent(Id).

eqn carrier(sn023) = sn@batc.be.
    carrier(sn720) = sn@batc.be.
    carrier(kl283) = klm@schipol.nl.
    carrier(kl034) = klm@schipol.
```

Airline company

```
proc Flight(company: sCIds, flight: sFIds, seatsAvailable: sInt) =
    get(company, reservation(Flight,SeatsReq,Id));
    Reserve_seats(seatsReq,seatsAvailable,seatsLeft,status);
    tell(company, acknowledge(id,seatsLeft,status));
    Flight(company, flight, seatsLeft).

Partnerflight(company: sCIds, flight: sFIds) =
    gett(company, reservation(flight,seatsReq,id));
    tellvb(vpartner,handling(flight));
    tell(vpartner,reservation(flight,seatsReq,id));
    get(vpartner,acknowledge(id,seatsLeft,status));
    getbb(vpartner);
    tell(company, acknowledge(id,seatsLeft,status));
    Partnerflight(company, flight).
```

Table of contents

Relating blackboards: basic ideas

- *In* **primitive**

  $in(bbn, O)$: the object O is on the blackboard bbn

- **R-rules**

$$in(b_1, O_1), \cdots, in(b_j, O_j) \longrightarrow$$
$$in(b_{j+1}, O_{j+1}), \cdots, in(b_m, O_m)$$

  The presence of $O_1, \ldots, O_j$ on blackboards $b_1, \ldots, b_j$ implies the presence of $O_{j+1}, \ldots, O_m$ on blackboards $b_{j+1}, \ldots, b_m$, respectively.

Variants

- some objects need to be consumed to produce others
- this production may actually lead to the creation of new objects on the blackboards
- the selection of objects may be enhanced in two ways:
  - by using conditions to strengthen the selection of objects based on unification only in the *in* primitive,
  - by, if need be, suffixing the *in* primitive by 't' and 'p' to explicitly distinguish between data and processes

General form

$$[In(b_1, O_1) : C_1, \cdots, In(b_i, O_i) : C_i] \{In(b_{i+1}, O_{i+1}) : C_{i+1}, \cdots, In(b_j, O_j) : C_j\}$$
$$\longrightarrow [In(b_{j+1}, O_{j+1}) : C_{j+1}, \cdots, In(b_k, O_k) : C_k] \{In(b_{k+1}, O_{k+1}) : C_{k+1}, \cdots, In(b_m, O_m) : C_m\}$$

where

- the *In*'s there stay either for *in*, *int*, or *inp*
- the *C*'s represent conditions of atoms defined by Horn clauses
- the square brackets and curly brackets are respectively used to indicate modifications and no modifications.

Design decisions

- Any condition C_k $(1 \leq k \leq j)$ may involve other variables than those of O_k under the restriction that the variables used in $C_1, \ldots, C_j$ are limited to those of $O_1, \ldots, O_j$.
- Any condition C_k, $j+1 \leq k \leq l$ should include as only variables those of $O_1, \ldots, O_j$ and of O_k.
- Only those objects O_k $(1 \leq k \leq j)$ physically present on b_k are considered to produce new objects.
- Objects written are understood as being duplicated.
- The constructive telling is operated incrementally each time a new object O_k is inserted on b_k $(1 \leq k \leq j)$;
- At the activation time of an r-rule, the objects O_k that are already present are also subject to the constructive telling and therefore also induce objects.

General relations

$$Rel \equiv \{R_1, \cdots, R_m\}$$

Examples

- **The forward relation**

$$forward(b_1, b_2) \equiv \{\ [in(b_1, X)]\ \{\}\ \longrightarrow [in(b_2, X)]\ \{\}\ \}$$

- **Inheritance relation**

$$inherit(b_1, b_2) \equiv \{\ []\ \{in(b_2, X)\}\ \longrightarrow []\ \{in(b_1, X)\}\ \}$$

Examples

- **Merge relation**

$$merge(b_1, b_2, b) \equiv \left\{ \begin{array}{l} [in(b_1, X)]\ \{\} \longrightarrow [in(b, X)]\ \{\} \\ [in(b_2, X)]\ \{\} \longrightarrow [in(b, X)]\ \{\} \end{array} \right\}$$

- **The duplicate relation**

$$duplicate(b, b_1, b_2) \equiv \{\ [in(b, X)]\ \{\} \longrightarrow [in(b_1, X), in(b_2, X)]\ \{\}\ \}$$

New primitives

- *telllink*(*bbn*, *rn*)
- *getlink*(*bbn*, *rn*)
- *asklink*(*bbn*, *rn*)
- *nasklink*(*bbn*, *rn*)

Table of contents

Why time?

- **Application need**
  - Request on the Web to be satisfied in a reasonable amount of time
  - Request for an ambulance to be answered within a critical amount of time
- **The coordination community**
  - tcc, tdcc (Saraswat, Jagadeesan, Gupta, 1994, 1996)
  - tccl (De Boer, Gabbrielli, Meo, 2000)
  - Oz (Smolka, 1995)
  - JavaSpaces (Freeman, Hupfer, Arnold, 1999)

Hypothesis

Use the two-phase functioning approach

- First phase: elementary actions of statements are executed.
  - actions are supposed to take no time
  - composition operators are supposed to take no time
- Second phase: time progresses by one unit
  - when no actions can be reduced or when all the components encounter a special timed action

Example

$$
\begin{array}{rcl}
W\_Station\_lis & = & local\ w\ in \\
 & & \quad delay(10)\ ;\ compute\_sky(funchal, w)\ ; \\
 & & \quad tell_{10}(\langle funchal, w \rangle)\ ;\ W\_Station\_lis \\
 & & end \\
 & & \\
Tourist & = & local\ min, max\ in \\
 & & \quad ask_5(\langle funchal, sunny \rangle)\ ;\ fly\_to(funchal) \\
 & & \quad + \\
 & & \quad delay(5)\ ;\ drive\_to(namur) \\
 & & end
\end{array}
$$

The $\mathcal{D}$ family (1)

Syntax

$$C \quad ::= \quad tell(t) \mid ask(t) \mid get(t) \mid nask(t) \mid delay(d)$$

Semantics

$$(D1) \qquad \frac{A \neq E, A \neq A^{-}, <A, \sigma> \not\rightarrow}{<A, \sigma> \rightsquigarrow <A^{-}, \sigma>}$$

$$(D2) \qquad <delay(0), \sigma> \longrightarrow <E, \sigma>$$

The $\mathcal{D}$ family (2)

$$
\begin{aligned}
tell(t)^- &= tell(t) \\
ask(t)^- &= ask(t) \\
nask(t)^- &= nask(t) \\
get(t)^- &= get(t) \\
delay(0)^- &= delay(0) \\
delay(d)^- &= delay(d-1) \\
(B\ ;\ C)^- &= B^-\ ;\ C \\
(B \parallel C)^- &= B^- \parallel C^- \\
(B\ +\ C)^- &= B^-\ +\ C^-
\end{aligned}
$$

The $\mathcal{R}$ family (1)

Syntax

$$C \quad ::= \quad tell_d(t) \mid ask_d(t) \mid get_d(t) \mid nask_d(t)$$

Semantics

$$(T0) \qquad <tell_0(t), \sigma> \longrightarrow <E, \sigma>$$

$$(Tr) \qquad \frac{d > 0}{<tell_d(t), \sigma> \longrightarrow <E, \sigma \cup \{t_d\}>}$$

$$(Ar) \qquad \frac{d > 0}{<ask_d(t), \sigma \cup \{t_k\}> \longrightarrow <E, \sigma \cup \{t_k\}>}$$

The $\mathcal{R}$ family (2)

Semantics

$$(Nr) \quad \frac{d > 0, \nexists k : t_k \in \sigma}{<nask_d(t), \sigma> \longrightarrow <E, \sigma>}$$

$$(Gr) \quad \frac{d > 0}{<get_d(t), \sigma \cup \{t_k\}> \longrightarrow <E, \sigma>}$$

$$(Wr) \quad \frac{A \neq E, A \neq A^- \text{ or } \sigma \neq \sigma^-, <A, \sigma> \not\rightarrow}{<A, \sigma> \rightsquigarrow <A^-, \sigma^->}$$

The $\mathcal{R}$ family (2)

$$
\begin{array}{rcl}
tell_d(t)^- & = & tell_d(t) \\
ask_d(t)^- & = & ask_{max\{0,d-1\}}(t) \\
nask_d(t)^- & = & nask_{max\{0,d-1\}}(t) \\
get_d(t)^- & = & get_{max\{0,d-1\}}(t) \\
E^- & = & E \\
(B\ ;\ C)^- & = & B^-\ ;\ C \\
(B \parallel C)^- & = & B^- \parallel C^- \\
(B\ +\ C)^- & = & B^-\ +\ C^-
\end{array}
$$

$$\sigma^- = \{t_{d-1} : t_d \in \sigma, d > 1\}$$

The $\mathcal{W}$ family

Syntax

$$C \quad ::= \quad tell(t) \mid ask(t) \mid get(t) \mid nask(t) \mid wait(m)$$

Semantics

$$(W1) \qquad \frac{A \neq E, A \gg u, <A,\sigma>_u \not\rightarrow}{<A,\sigma>_u \rightsquigarrow <A,\sigma>_{u+1}}$$

$$(W2) \qquad \frac{u \geq v}{<wait(v),\sigma>_u \longrightarrow <E,\sigma>_u}$$

$A \gg u$ iff A contains a $wait(m)$ with $m > u$

The $\mathcal{I}$ family (1)

Syntax

$$C \quad ::= \quad tell_{[b:e]}(t) \mid ask_{[b:e]}(t) \mid get_{[b:e]}(t) \mid nask_{[b:e]}(t)$$

Semantics

$$(Ta) \qquad \frac{b \leq u \leq e}{<tell_{[b:e]}(t), \sigma>_u \longrightarrow <E, \sigma \cup \{t_{[u:e]}\}>_u}$$

$$(Aa) \qquad \frac{b \leq u \leq e, b' \leq u \leq e'}{<ask_{[b:e]}(t), \sigma \cup \{t_{[b':e']}\}>_u \longrightarrow <E, \sigma \cup \{t_{[b':e']}\}>_u}$$

The $\mathcal{I}$ family (2)

Semantics

$$(Na) \qquad \frac{\begin{array}{c} b \leq u \leq e, \\ \nexists b', e' : b' \leq u \leq e' \wedge t_{[b':e']} \in \sigma \end{array}}{<nask_{[b:e]}(t), \sigma>_u \longrightarrow <E, \sigma>_u}$$

$$(Ga) \qquad \frac{b \leq u \leq e, b' \leq u \leq e'}{<get_{[b:e]}(t), \sigma \cup \{t_{[b':e']}\}>_u \longrightarrow <E, \sigma>}$$

$$(Wa) \qquad \frac{A \neq E, A \gg u \text{ or } \sigma \gg u, <A, \sigma> \not\rightarrow}{<A, \sigma>_u \rightsquigarrow <A, \sigma^{+u}>_{u+1}}$$

The $\mathcal{I}$ family (3)

$$\sigma \gg u \text{ iff } \exists t_{[b:e]} \in \sigma : (e \neq \infty \wedge e > u)$$

$$\sigma^{+u} = \{t_{[max\{b,u+1\}:e]} : t_{[b:e]} \in \sigma, u+1 \leq e\}.$$

Example: delayed requests

- Abstracting the web as a tuple space, a request for information on the web may be typically programmed as follows:

$$tell_d(request)\ ;\\ (get_d(answer) + (delay(d)\ ;\ tell_\infty(no\_answer)))$$

- Similarly, exceptions after some delays may be programmed as

$$\begin{aligned} ask_d(t)\Box A &\equiv ask_d(t) + (delay(d)\ ;\ A)\\ get_d(t)\Box A &\equiv get_d(t) + (delay(d)\ ;\ A)\\ nask_d(t)\Box A &\equiv nask_d(t) + (delay(d)\ ;\ A) \end{aligned}$$

Example: Do ... watch construct

$$do\ tell_d(t)\ watch\ e\ cont\ B \equiv$$
$$(ask_\infty(e)\ ;\ B) + (nask_\infty(e)\ ;\ tell_d(t))$$

$$do\ C_d(t)\ watch\ e\ cont\ B \equiv$$
$$\begin{cases} (ask_\infty(e)\ ;\ B) \\ \quad +(nask_\infty(e)\ ;\ C_1(t)\ ;\ do\ C_{d-1}(t)\ watch\ e\ cont\ B) \\ \quad \text{if } C \in \{ask, nask, get\} \text{ and } d > 1 \\ (ask_\infty(e)\ ;\ B) + (nask_\infty(e)\ ;\ C_d(t)) \\ \quad \text{if } C \in \{ask, nask, get\} \text{ and } d \leq 1 \end{cases}$$

$$do\ delay(d)\ watch\ e\ cont\ B \equiv$$
$$\begin{cases} (ask_\infty(e)\ ;\ B) \\ \quad +(nask_\infty(e)\ ;\ delay(1)\ ;\ do\ delay(d-1)\ watch\ e\ cont\ B) \\ \quad \text{if } d \geq 1 \\ (ask_\infty(e)\ ;\ B) + nask_\infty(e) \\ \quad \text{otherwise} \end{cases}$$

Example: Do ... watch construct (2)

$$do\ X\ ;\ Y\ watch\ e\ cont\ B \equiv$$
$$do\ X\ watch\ e\ cont\ B\ ;\ do\ Y\ watch\ e\ cont\ B$$

$$do\ X\ ||\ Y\ watch\ e\ cont\ B \equiv$$
$$do\ X\ watch\ e\ cont\ B\ ||\ do\ Y\ watch\ e\ cont\ B$$

$$do\ X\ +\ Y\ watch\ e\ cont\ B \equiv$$
$$do\ X\ watch\ e\ cont\ B\ +\ do\ Y\ watch\ e\ cont\ B$$

The untimed case

- Concurrency achieved by the threads library of Solaris
- Distribution achieved by RPC/socket
- Tuple space = token-indexed list of tokens
- For each token, we keep track of
  - number of occurrences of the token
  - a list of suspended processes
- Each token is protected by its own lock
- The implementation of the basic primitives is achieved as one might guess

Timed primitives

- Implement the primitives dealing with absolute time only
- A period of validity is associated with the tokens and processes of waiting lists
- Waking-up facilities provided by the operating system are used to
  - force wait primitives to succeed when the specified waiting time has been reached
  - force timed ask, nask, get to fail when their period of validity is over
  - remove tokens whole period of validity is over

Table of contents

Applications to MANETs

- **Application view**
  - everything occurs on the local blackboard
  - synchronization based on availability of information
  - contextual information available “by magic”
- **Middelware view**
  - Blackboard relations provide this magic

    ! hosts may move out of connection
    ⇒ need for dynamic activation of relation
    ⇒ introduce events and reactions to them
  - ...

Applications to MANETs (2)

- **Middelware view (2)**
  - Timeouts
    - timed primitives allow to express timeouts
    - ! time considered with respect to local clock
    - ! get primitive handled with special protocol
  - Filter information and hosts

    ⇒ handle access rights and policy
  - Several relations may be used to satisfy a query

    ⇒ introduce priorities and make them vary in time

Events and reactions to events

- Special tuples written by dedicated processes

  $\langle connect, h \rangle$, $\langle disconnect, h \rangle$, $\langle quality, h, q \rangle$

- Reaction

$$in(\langle connected, X \rangle) \quad \Rightarrow \quad tell([in(Y)@self] \longrightarrow_b [in(Y)@X])@X$$

Access rights and policies

- Special hidden attributes of tuples, blackboard relations, primitives
- inherited from the creating processes
- accessed if capabilities are matched