Much of this class is dedicated to the discussion of the paper

C.A.Visser, G. Scollo, M.v. Sinderen, E. Brinksma
Specification Styles in Distributed System Design and Verification.
Theoretical Computer Science 89 (1991) 179-206

# SPECIFICATION STYLES

## Extensional: "pure definition"
## no implementation implications

Monolithic : Spec is tree of alternatives, i.e. *expanded*
Execution sequences are explicitly enumerated.

Constraint-oriented : Spec is parallel composition of processes.
Each process represents a constraint.
All processes (constraints) must be simultaneously satisfied.

## Intensional: suggests an implement. architecture

State-oriented : Use of state variables or of processes that describe states
Resource-oriented : Processes correspond to implem. structure

Expansion Theorem (Milner):
Every LOTOS specification can be transformed step-by-step into a (possibly infinite!) monolithic (=expanded) one.

# State-Oriented Style

**No use of parallel composition, explicit reference to system states. Two variants:**

## State-Oriented - 1st Variant
## (used in paper by Vissers et al.)

(used in paper) Use variables for states: state changes are parameter changes.

```
A(state) :=        [state = 0] -> actions ; A(1)
               []  [state = 1] -> actions ; A(0)

A(0)             to initialize
```

useful when specification requires passing state names as parameters, comparing states of different components, and similar.

## State-Oriented - 2nd Variant

Use processes for states. State changes are invocations of processes. More concise, but perhaps inadequate in some cases (if we want to pass around state values).

state0 := *actions;* state1

state1 := *actions;* state0

etc.

Specifying the QA_service in the 'other' State-Oriented style:

- define a process for each state.
- state transitions are actions,
- state changes are invocations of processes.

```
awaitQ  :=          Q?x: question;   pending Q(x)
 pendingQ(x)  :=    A!x;  awaitA
awaitA  :=          A?y:answer;  pendingA(y)
pendingA(y)  :=     Q!y;  stop
```

# Monolithic Style

Expanded view of the system.

Similar to second state-oriented style, but process names do not necessarily have meaning of system state.

# Resource-Oriented or Implementation-Oriented Style.

Describes a system architecture in terms of communicating processes. Processes corresponds to system modules.

Synchronization is usually two-way only, and models implementation-level interprocess communication.

Hiding used to hide internal synchronization events.

The most intuitively appealing, close to implementation level.

# Constraint-Oriented Style

Each process defines a set of constraints that must be collectively satisfied by the whole system.

High abstraction, no relation with implementation.

No use of hiding.
Synhcronization usually multi-way.

There are two types of constraints:

a) event sequence constraints

b) data value constraints

## a) Event sequence Constraints

**EXAMPLE.** Defining the following partial order: actions a, b, c can be executed in any order before action d.

## SPECIFYING PARTIAL ORDERING BETWEEN EVENTS

a > d   and  b > d  and  c > d                    ( > :  precedes )

(NOTE:  NO ORDER SPECIFIED BETWEEN  a, b, c )

WE CAN SPECIFY THIS BY:

```
    a;  (b;c;d;stop [] c;b;d;stop)
 [] b;  (a;c;d;stop [] c;a;d;stop)
 [] c;  (a;b;d;stop [] b;a;d;stop)
```

OR, EQUIVALENTLY, BY THREE PARALLEL PROCESSES SYNCHRONIZING ON d

```
( a; d; stop ) | [d] | ( b; d; stop ) | [d] | (c; d; stop )
```

(PROCESSES:
  INTERLEAVE     W.R.T.  a, b, c
  SYNCHRONIZE     W.R.T.  d      )

e.g.       a d b    impossible       WHY?

NOTE THE MULTI-WAY SYNCHRONIZATION

# b) Data value constraints

(by using selection predicates)

| | |
|---|---|
| P := | g ? x : int (x mod 2 = 0); P |
| | will only accept even numbers |
| Q := | g ?y:int (y mod 3 = 0); Q |
| | will only accept multiples of 3 |
| P ll Q | will only accept multiples of 6 |
| P lll Q | will accept sequences where each element is either a multiple of 2 or a multiple of 3 |
| P () Q | will have to decide at the beginning whether to act like P or like Q. |
| P ll (P lll Q) | will accept ? |

## Example of data constraints: the bounded buffer

```
process BB[input,output](n:nat) :=
	Bag[input,output] || Bound[input,output](n)
	where
		process Bag[input,output] :=
			input ?x:elem;
			(Bag[input,output]
			 ||| output !x; stop)
		endproc
		process Bound[input,output](n:nat) :=
			(output ?x:elem; Bound[input,output](n+1)
			[] [n>0] -> input ?x:elem; Bound[input,output](n-1))
		endproc
```

# Two interpretations of | |

In the constraint-oriented style, the full synchronization operator takes the logical meaning of composition of logical constraints,

while in the resource-oriented style, the same operator has the meaning of communication over shared events.

Unfortunately, however, this operator cannot quite achieve the meaning of the logical and operator: it has been noted that in general (nondeterminism!)

$$A \,||\, A \neq A$$

while instead in logic

$$A \wedge A = A$$

An input file of records (x,y)

P := g ?x:int ?y:int (x>3); P

Q := g ?x:int ?y:int (y mod 2 = 0); Q

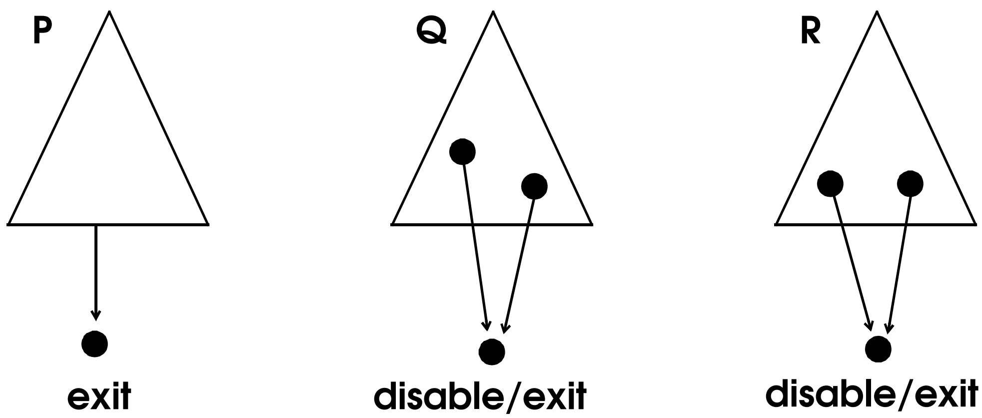
Process P||Q will accept only sequences of records such that the field x satisfies P, and field y satisfies Q

Process P|||Q will accept sequences of records such that either field x satisfies P, or field y satisfies Q.

Note: unfortunately, it is impossible to hide a field from one of the interacting processes.

# Exiting

Suppose that we have several processes, each one checking a different constraint. One process dictates the exit condition.



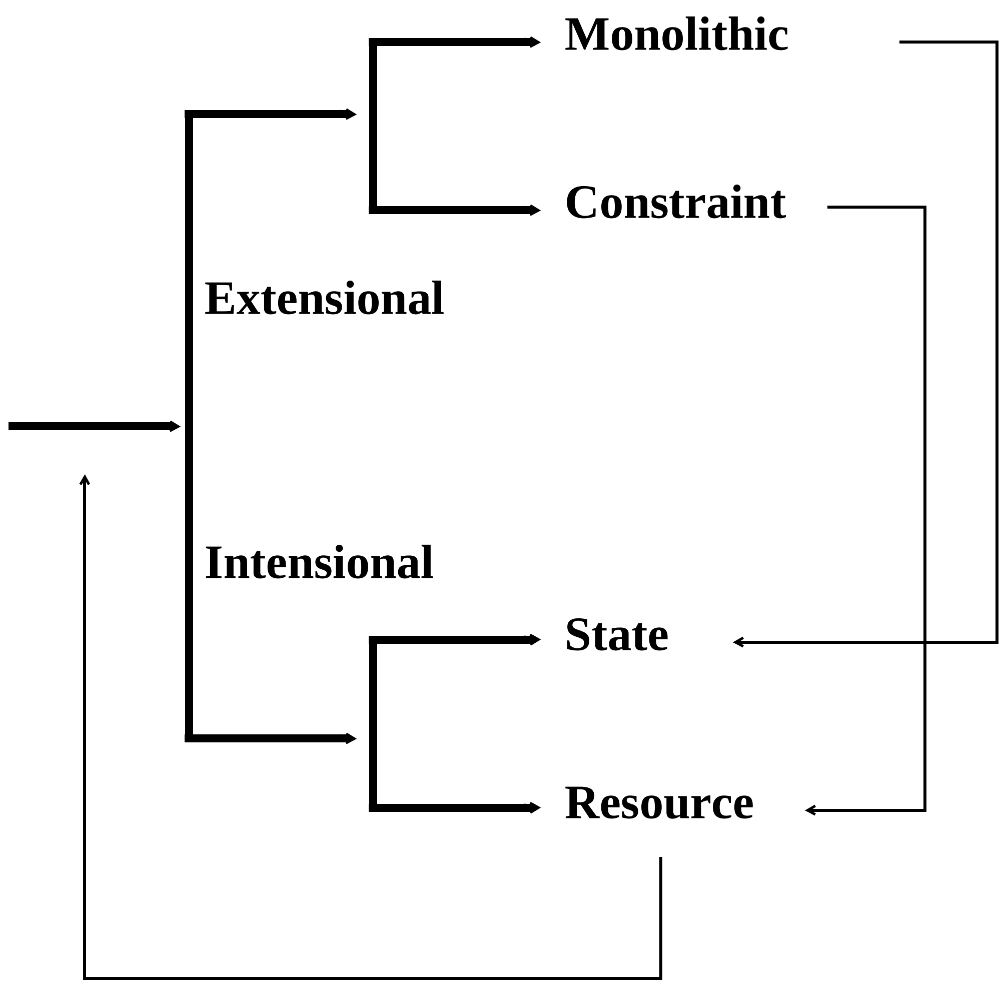

```
P :=    g?x:int (x>5 and x<9);   P

        () g!12;   exit

Q :=    ( g? x:int (x mod 3 = 0);  Q)  (>  exit

R  :=   ( g? x:int (x mod 2 = 0);   R)  (>  exit
```

Process P || Q || R will accept only 6's. When a 12 is encountered, the exit of P will force a rendez-vous of all processes on *exit* gate. All disables will then occur, and all processes will exit.

Consider the following variant of the Example 4.1 in the paper by Visser et al. It appears to be simpler, but unfortunately it contains two mistakes, corresponding to points of which the language user should be aware.

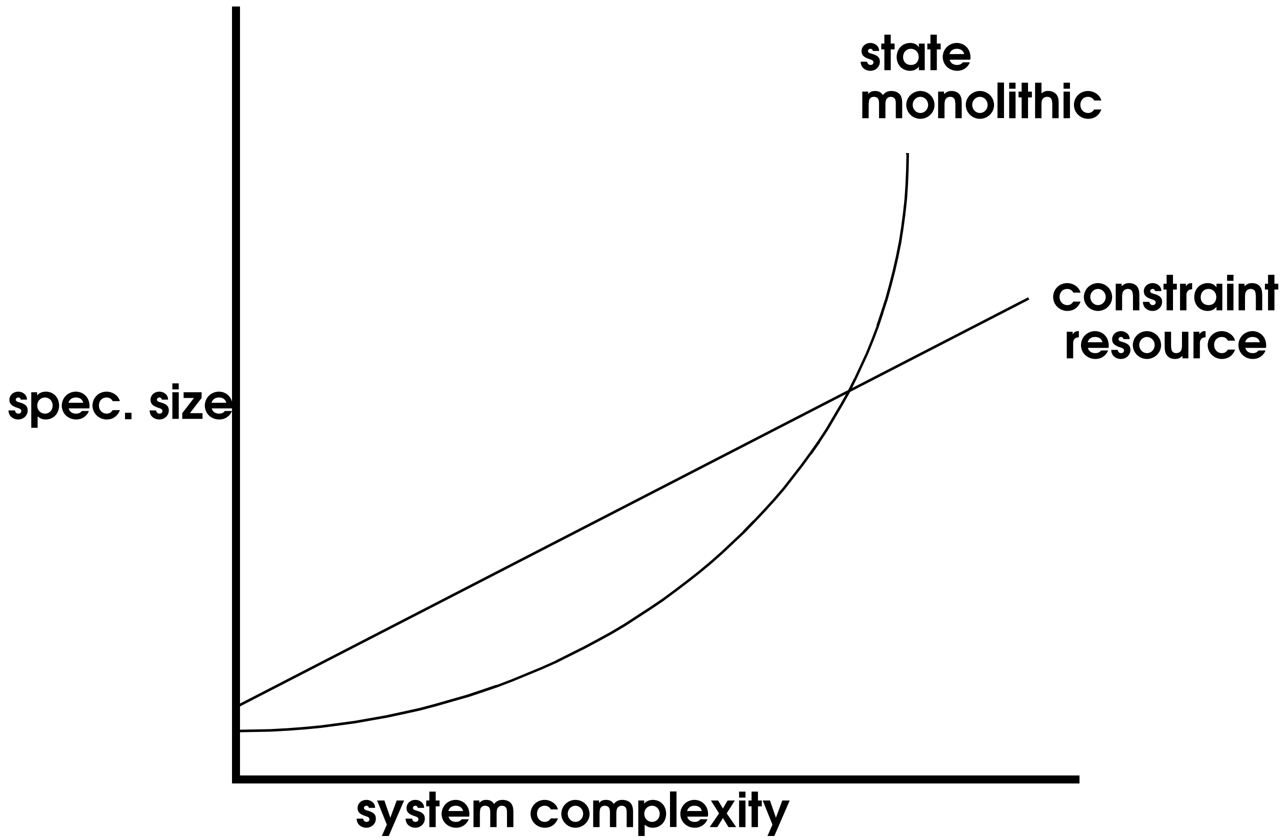
```
process QA_service[Q,A]: noexit :=
hide CLQ, CLA in
    ((Q_entity[Q,CLQ] ||| A_entity[A,CLA])
    |[CLQ,CLA]|
    CL_service[CLQ,CLA])

where

process Q_entity [X,Y]: noexit :=
    X ? x: question; Y !encode_q(x);
    Y ?d:data; X !decode_a(d); stop
endproc

process A_entity [X,Y]: noexit :=
    Y ?d:data; X !decode_q(d);
    X ?y:answer; Y !encode_a(y); stop
endproc

process CL_service [X,Y]: noexit :=
        X ?d1:data; Y !d1; Y ?d2:data; X !d2; stop
    []  X ?d1:data; Y ?d2:data; Y !d1; X !d2; stop
    []  X ?d1:data; Y ?d2:data; X !d2; Y !d1; stop
    []  Y ?d2:data; X !d2; X ?d1:data; Y !d1; stop
    []  Y ?d2:data; X ?d1:data; X !d2; Y !d1; stop
    []  Y ?d2:data; X ?d1:data; Y !d1; X !d2; stop
endproc

endproc
```

# RELATIONSHIP BETWEEN STYLES




**Arrows denote refinement relationship**

**State-oriented cannot be refined further**




More structured styles, such as constraint and resource, contain state explosion thus lead to smaller spec sizes for complex systems.

# GATES AND ADDRESSES IN LOTOS

As in CSP and CCS, only a finite, statically defined set of gates can be used in a specification. Dynamic gate creation is not possible.

How can a multiplicity of dynamically created interacting processes be defined ?

A process can be replicated by recursive call:

```
process Connections  (Usr,Ctl):  noexit  :=
      Single_Connection   (Usr,Ctl)
      |||
      i;    Connections (Usr, Ctl)
```

The gate name can be augmented by (one or more) values which represent a sort of *secondary* interaction point names.

**STATE SPLITTING!**

E.g.

A: g !5 !3

B: g !5 ?x:int

synchronization can occur on (g, 5) and if it does x in B becomes 3.

A: g !3 !3

B: g !5 ?x:int

synchronization not possible.

A: g !3 !6

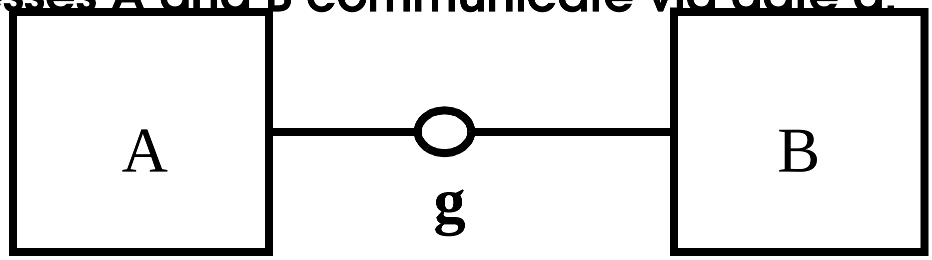
B: g !5 ?y:int . . . () g !3 !x:int . . .

synchronization can occur with the second alternative on (g,3) and if it does x becomes 6.

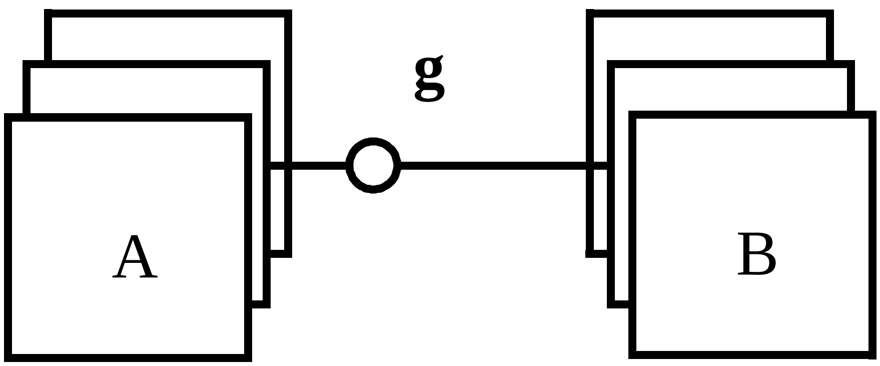
Specification of an unbounded number of interaction points becomes possible with this techniques.

## Specification level

**Processes A and B communicate via gate g.**




## Execution Level




**The various instances of A's and B's still communicate via gate g.**

**An interaction between A and B must state not only the gate g, but also additional information to distinguish the several instances of A and B.**

Example showing that the secondary interaction point name can be a parameter:

```
A(g)(1)
||
B(g)(1)

where

process A(g)(addr: CEPid): noexit :=
        . . .
        g !addr !3
        . . .
endproc

process B(g)(addr: CEPip): noexit :=
        . . .
        g !addr ?x:int
        . . .
endproc
```

It can also be determined by an interaction:

```
g  ?addr: CEPid    . . . .    A(g)(addr)
```

Or a choice between a number of possible names can be expressed:

```
choice addr: CEPid  []  A(g)(addr)
```

# Example: POTS SPECIFICATION

## WHAT ARE THE CONSTRAINTS

### LOCAL CONSTRAINTS

Appropriate sequence of events at each phone
(e.g. before dialing, one must pick up the handset)

### END_TO_END CONSTRAINTS

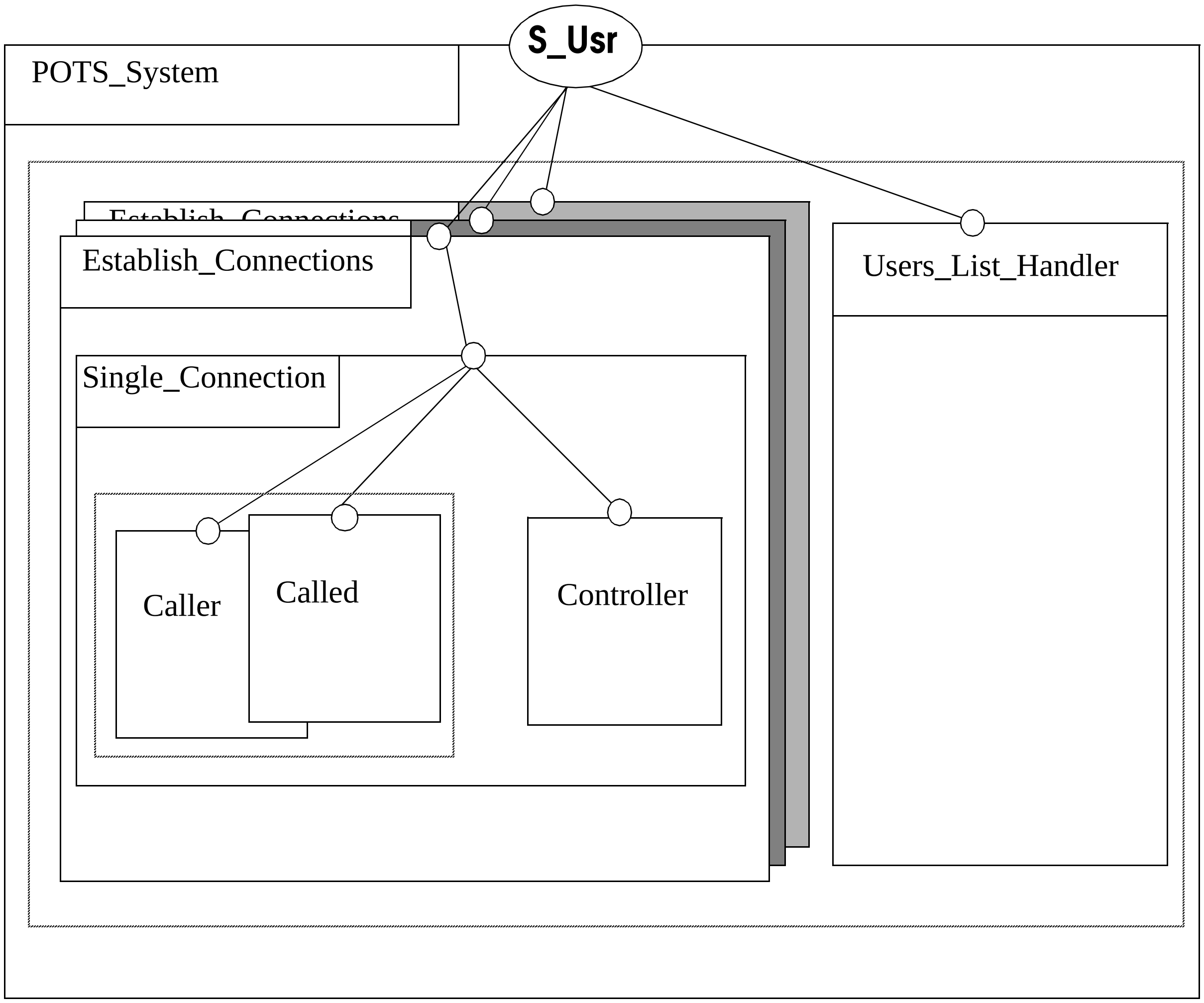
Appropriate sequence of events in each connection
(e.g. ringing at one end must follow dialing at the other)

### GLOBAL CONSTRAINTS

System-wide constraints
(e.g. no phone can be involved in two connections)

# POTS Specification in LOTOS: The Top Level

```
behavior
  (
   Establish_Connections(S_Usr)
      ||
   Users_List_Handler (S_Usr)(empty)
  )

 where
  process Establish_Connections(S_Usr):noexit:=
  (
     Single_Connection(S_Usr)
      |||
     i; Establish_Connections(S_Usr)
  )

  where
     process Single_Connection(S_Usr):noexit:=
     (
        ( Caller(S_Usr)
           |||
          Called(S_Usr)
        )
       ||
        (
          Controller(S_Usr)
          [>   Controller_Hang_Up(S_Usr)
        )
     )
```




**A graphical representation of the POTS specification's top levels**

# CONCLUSIONS

**LOTOS provides powerful concepts for the specification of distributed systems.**

**Systems can be specified in a number of ways, hence emphasizing different aspects of system behavior.**

**By using concepts of behavior equivalence, it is possible in principle to change from an aspect to another.**

**Because LOTOS is executable, LOTOS specifications can be used as abstract prototypes of systems.**

**Labelled transition systems can be used**

- as a basis for testing
- as a basis for validation activities such as model checking
- as a basis for implementation (translation LTS-> code)

**Emphasize early stages of software development cycle: design, design validation, derivation of implementation directly from design.**

**Prevent and detect software errors at the early stages of design.**

**De-emphasize late stages, when defect correction is more complex.**

# Problems....

Sharp *difference* between *data* and *behavior* formalisms.

Low expressiveness of the data formalism, e.g. in each data type equality must be defined by equations

In the behavior part it is not possible to:

> ask the question: in what state am I now? and pass the b. ex. of that state
>
> manipulate b.ex., pass them as parameters, and then execute them (not *reflective*)

From the point of view of formal verification, it is not possible to:

> reason in terms of pre- and post-conditions: e.g. given that the precondition and postcondition for A and B are known, determine the pre- and post-conditions for A |(...)| B (no *compositionality,* no Hoare logic)
>
> reason in terms of fixpoints (no monotonicity because of internal action)

expression of nondeterminism by internal action is a problem

# And more...

No explicit concept of time

(however this is only critical for the expression of *hard* real-time)

However on the other hand: too general a formalism:

designers need constrained templates, adapted to specific domains.

# Towards new generation specification languages

Define a set of combinators that can be applied to both processes and data, seen as trees

> e.g. behavior operators such as [], |[]| could also be seen as data operators
>
> to what extent can the linear logic operators be adapted for this?
>
> thus there would not be an essential difference between the representation of data and processes
>
> there would be a common underlying logic
>
> BUT... language must be executable