Embedded system design with the Polychronous paradigm

Albert Benveniste

Inria/Irisa

With contributions by lots of people:

Paul Le Guernic, Benoît Caillaud, Thierry Gautier, Jean-Pierre Talpin, Loic Besnard

---

SwSTE07 Conference October 30-31 2007 Israel

Updated April 2009, Chess seminar, December 2009, February 2011

Contents

- Motivation
- Signal Model of Computation and Communication
- Key data structure for Signal compilation: clock-and-causality calculus
- Constructive Semantics
- Separate compilation and a notion of interface
- Deploying over distributed (possibly asynchronous) architectures
- Use in architecture modeling and analysis

The key challenge of embedded system design

- Supporting overall design flow
- Addressing heterogeneity in:
  - System
  - Skills/teams/tools/methods
- Supporting complex OEM/supplier chains

Catia

Nastran

XX_tool

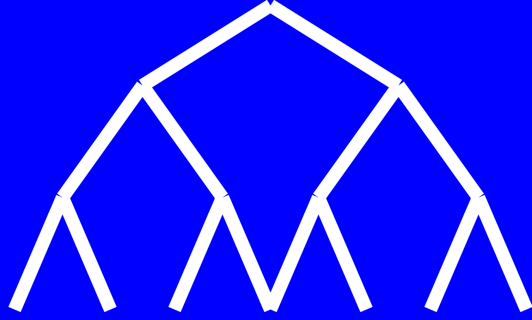
Simulink

Scade

Rhapsody

Reliability tool

...



PLM

CAN

Flexray

ARINC 653

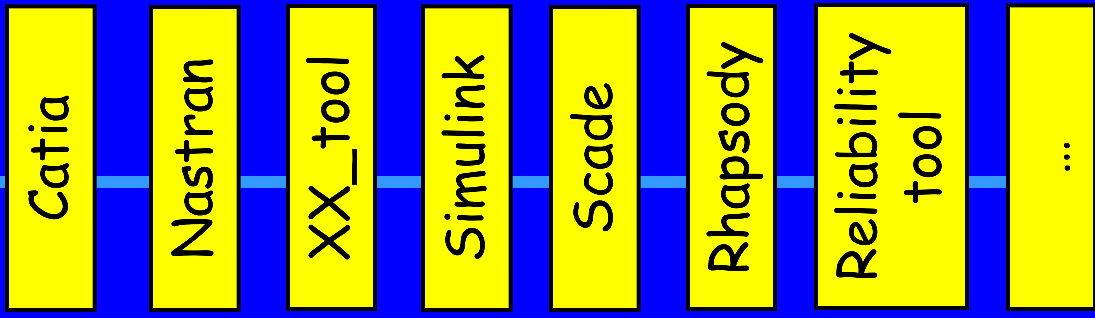
Topology

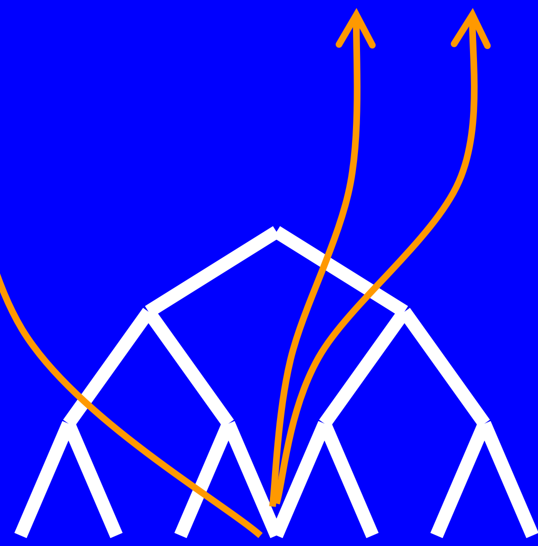
ECU/LRU

Profiling

Energy

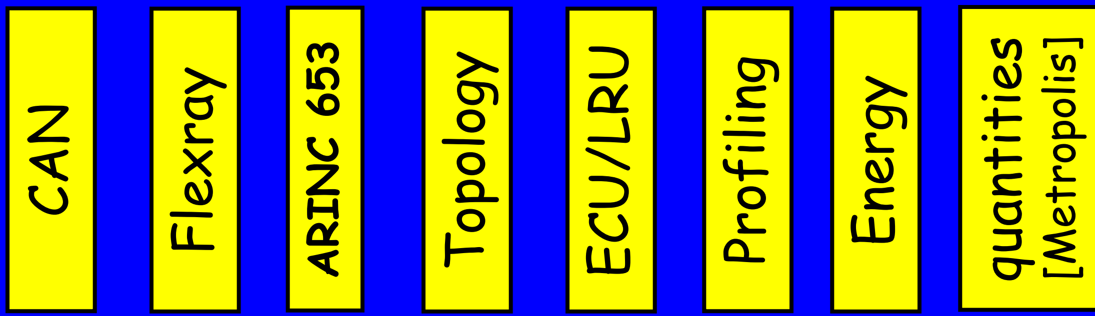
quantities [Metropolis]

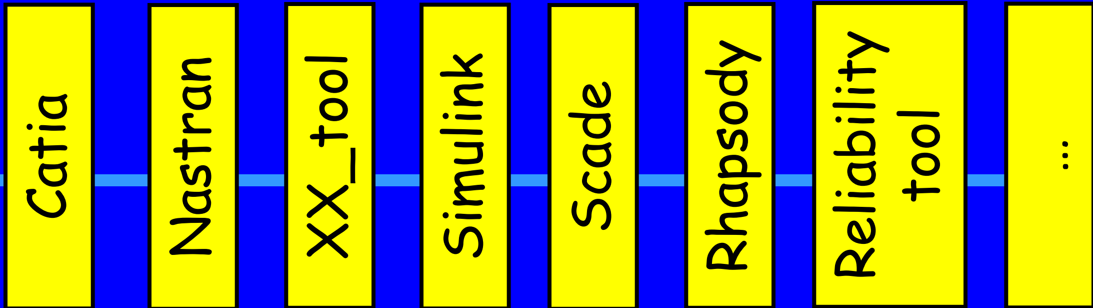


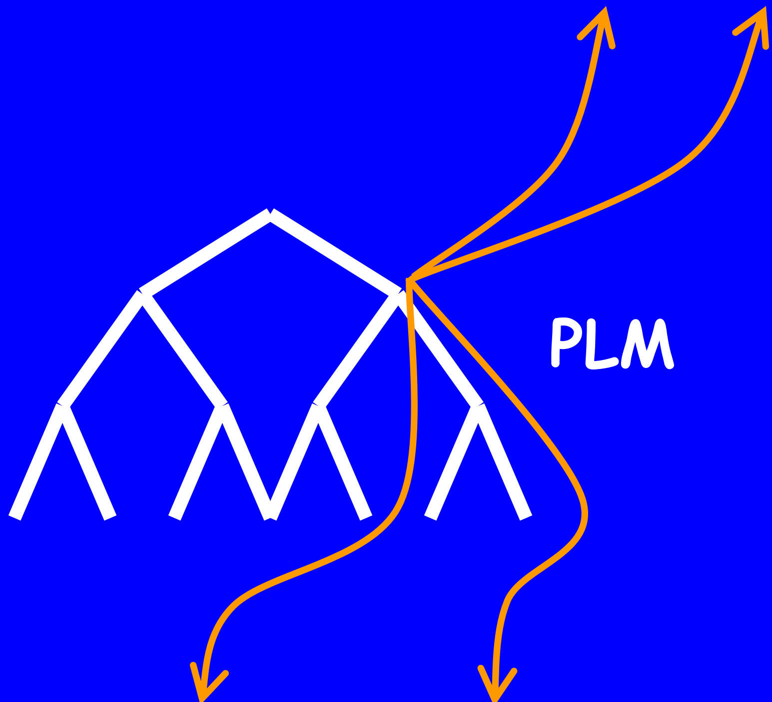


PLM

This leaf is a valve
it has a Catia model
a Simulink physical model
and a SCADE software





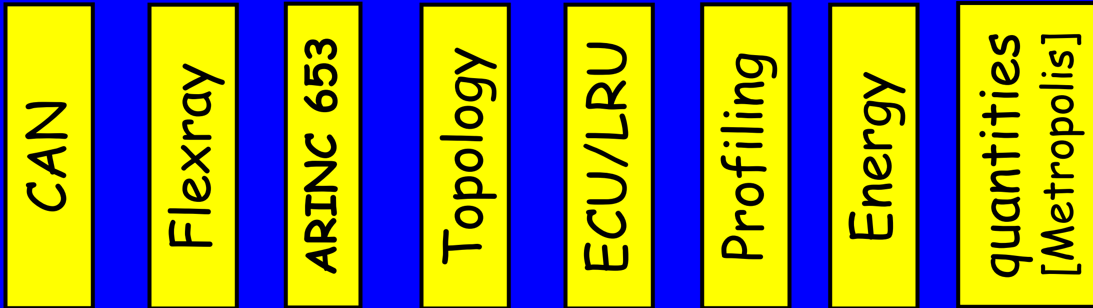


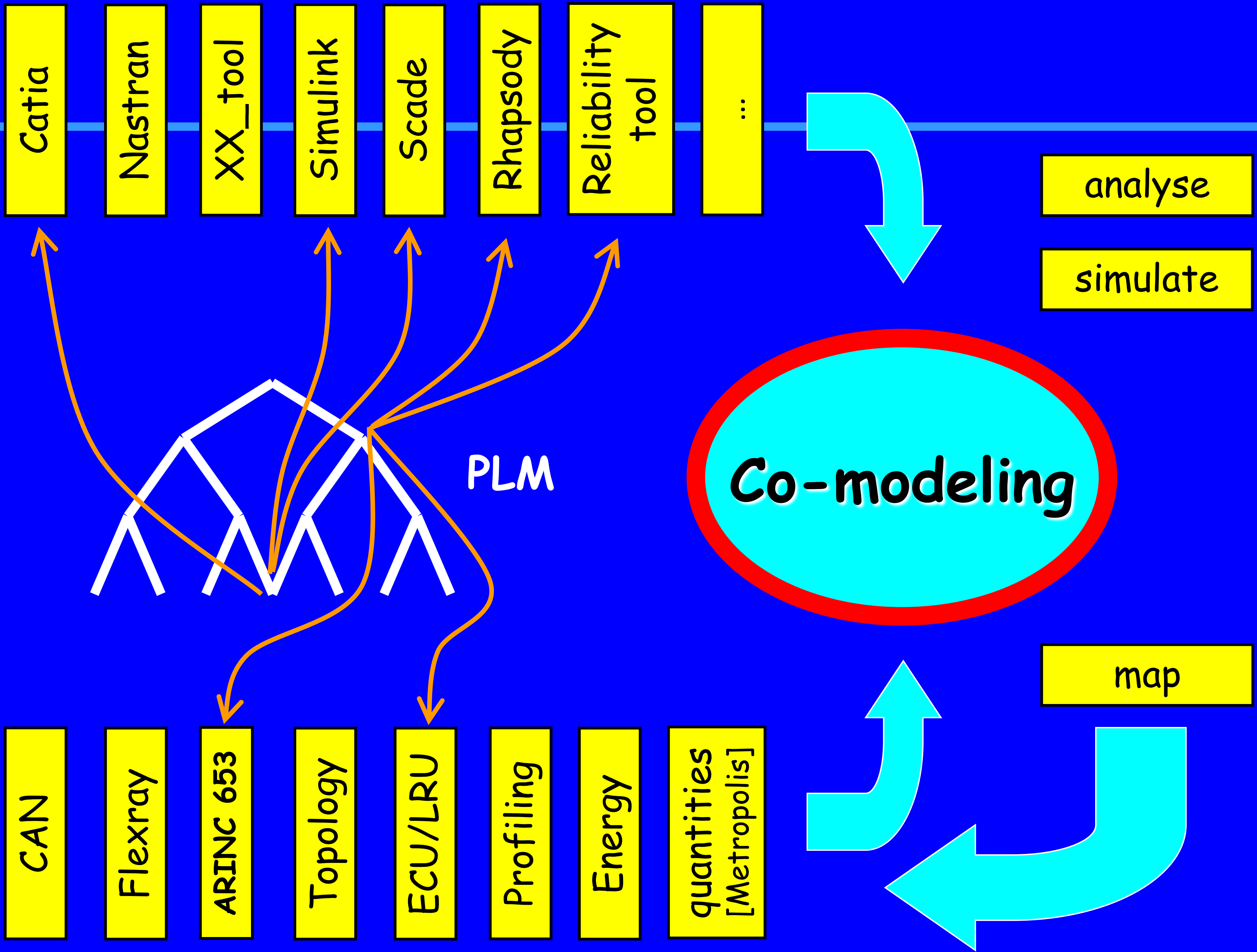
This subsystem

has a Rhapsody model

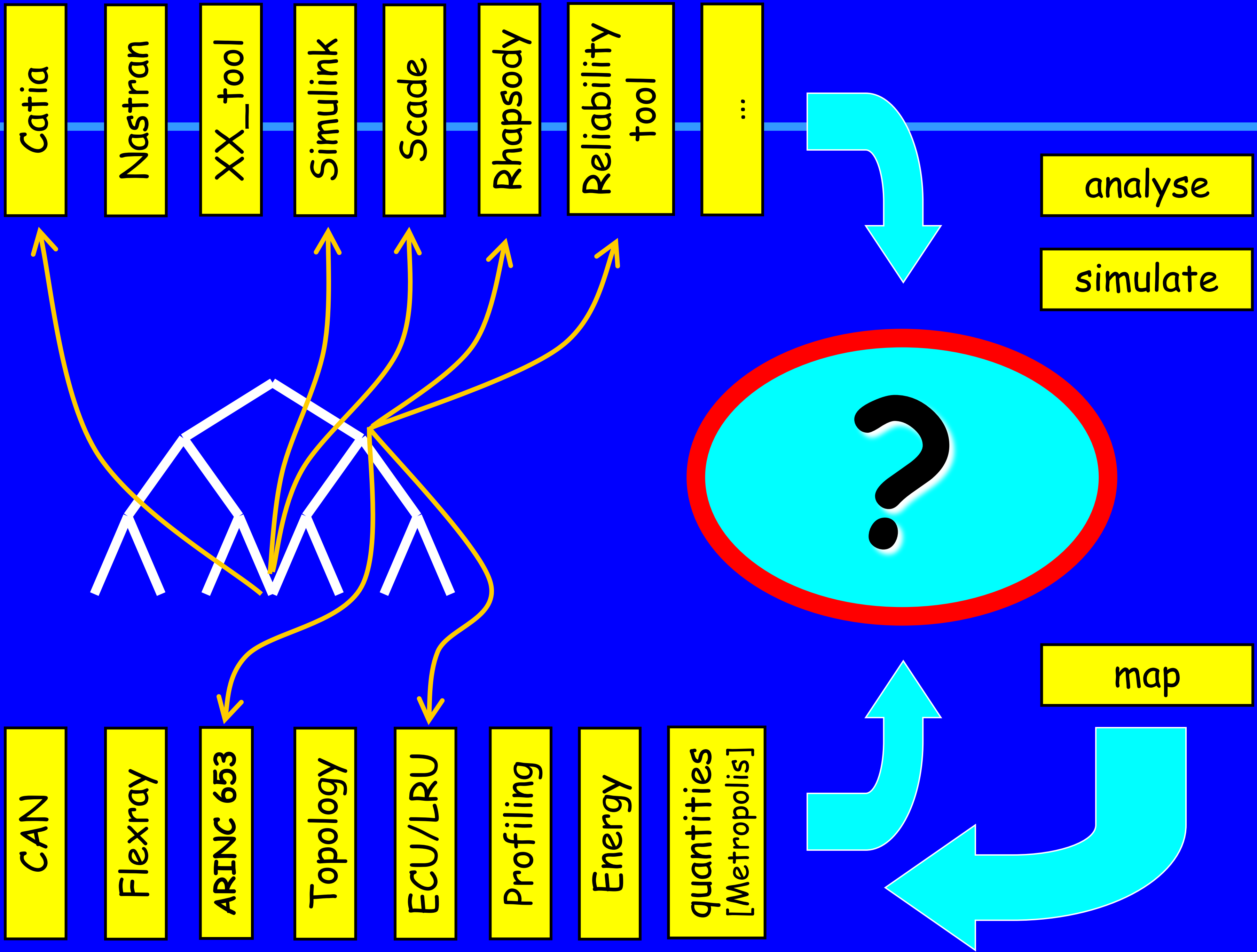
and a reliability model

it is deployed on a set of LRU's communicating over an ARINC 643 bus





Catia
Nastran
XX_tool
Simulink
Scade
Rhapsody
Reliability tool
...
analyse
simulate
PLM
Co-modeling
map
CAN
Flexray
ARINC 653
Topology
ECU/LRU
Profiling
Energy
quantities [Metropolis]



Catia
Nastran
XX_tool
Simulink
Scade
Rhapsody
Reliability tool
...
analyse
simulate
?
map
CAN
Flexray
ARINC 653
Topology
ECU/LRU
Profiling
Energy
quantities [Metropolis]

Aim of this course is
to substantiate why
the Polychronous paradigm
is a good solution
for system level co-modeling

I am kindly asking you for a while to forget everything you know about embedded software…

- RTOS, RME, tasking, scheduling
- Bus, networking
- Components, reuse, COTS
- Architectures, dependability
- Simulink, ladder diagrams
- Whateveyou…

… and to remember instead some basic maths you learnt while being undergraduate …

- Specifying sets or surfaces via equations
- Intersecting them by means of systems of equations
- Systems of equations do combine easily: they are just systems of systems of equations = systems of equations
- Systems of equations possess 0 / 1 / many solutions
- Equations can be dynamical (differential equations, difference equations)
- Solving equations may be difficult

… and to compare it with your actual engineering knowledge

- Specifying sets or surfaces via equations
- You want to (re)use components

… and to compare it with your actual engineering knowledge

- Specifying sets or surfaces via equations
- Intersecting them by means of systems of equations

- You want to (re)use components
- Designing systems by assembling components

… and to compare it with your actual engineering knowledge

- Specifying sets or surfaces via equations
- Intersecting them by means of systems of equations
- Systems of equations do combine easily: they are just systems of systems of equations = systems of equations
- You want to (re)use components
- Designing systems by assembling components
- Computer scientists have developed huge bodies of theory about how to compose things; standardisation bodies have debated this even more

… and to compare it with your actual engineering knowledge

- Specifying sets or surfaces via equations
- Intersecting them by means of systems of equations
- Systems of equations do combine easily: they are just systems of systems of equations = systems of equations
- Systems of equations possess no / one / many solutions

- You want to (re)use components
- Designing systems by assembling components
- Computer scientists have developed huge bodies of theory about how to compose things; standardisation bodies have debated this even more
- Composition may be blocking or may not be executable due to nondeterminism; otherwise we are happy

… and to compare it with your actual engineering knowledge

- Specifying sets or surfaces via equations
- Intersecting them by means of systems of equations
- Systems of equations do combine easily: they are just systems of systems of equations = systems of equations
- Systems of equations possess no / one / many solutions
- Equations can be dynamical (differential equations, difference equations)

- You want to (re)use components
- Designing systems by assembling components
- Computer scientists have developed huge bodies of theory about how to compose things; standardisation bodies have debated this even more
- Composition may be blocking or may not be executable due to nondeterminism; otherwise we are happy
- Embedded systems involve reactive and real-time components

… and to compare it with your actual engineering knowledge

- Specifying sets or surfaces via equations
- Intersecting them by means of systems of equations
- Systems of equations do combine easily: they are just systems of systems of equations = systems of equations
- Systems of equations possess no / one / many solutions
- Equations can be dynamical (differential equations, difference equations)
- Solving equations may be difficult

- You want to (re)use components
- Designing systems by assembling components
- Computer scientists have developed huge bodies of theory about how to compose things; standardisation bodies have debated this even more
- Composition may be blocking or may not be executable due to nondeterminism; otherwise we are happy
- Embedded systems involve reactive and real-time components
- Get executable code for the component and run it

Remarks: some things are easier in math, other are easier in software engineering

- Specifying sets or surfaces via equations
- Intersecting them by means of systems of equations
- Systems of equations do combine easily: they are just systems of systems of equations = systems of equations
- Systems of equations possess no / one / many solutions
- Equations can be dynamical (differential equations, difference equations)
- Solving equations may be difficult

easy: composing
difficult: executing

- You want to (re)use components
- Component based system design
- Computer scientists have developed huge bodies of theory about how to compose things; standardisation bodies have debated this even more
- Composition may be blocking or may not be executable due to nondeterminism; otherwise we are happy
- Embedded systems involve reactive and real-time components
- Get executable code for the component and run it

composing: difficult
executing: « easy »

Guess which is the best paradigm for component based system design?

Signal/Polychrony has chosen to rely on the math paradigm

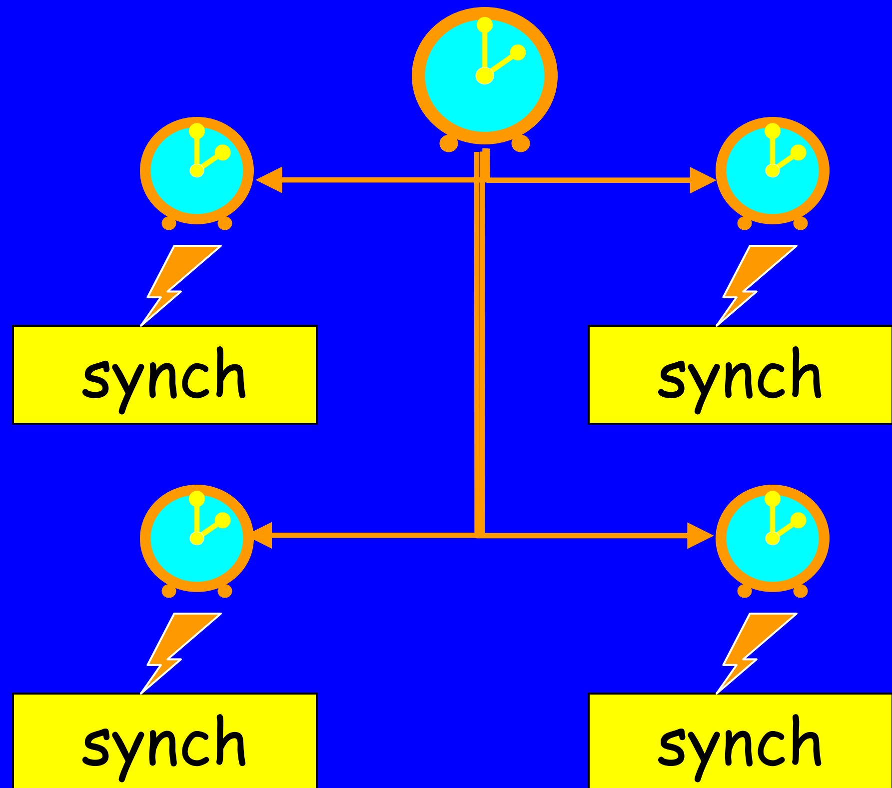
- This has made much more easy all issues related to composition of components
- Some more traditional issues like compilation and code generation become (very) difficult – the theory for it amounts to solving equations in particular domains

Contents

- Motivation
- Signal Model of Computation and Communication
- Key data structure for Signal compilation: clock-and-causality calculus
- Constructive Semantics
- Separate compilation and a notion of interface
- Deploying over distributed (possibly asynchronous) architectures
- Use in architecture modeling and analysis

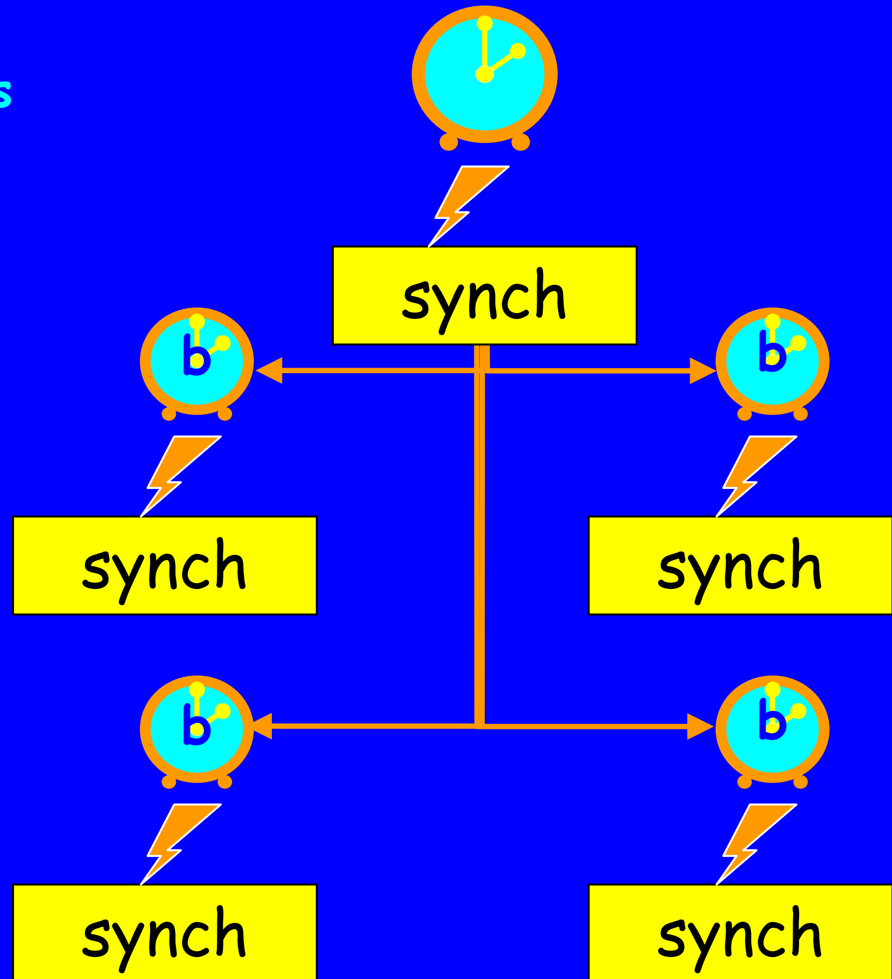
The Essence of Polychrony (1)

- Synchronous hardware
- Ed Lee's Synchronous dataflow
- Simple Simulink diagrams
  - Each module has a single clock that triggers all signals
  - The different clocks are derived from a global master clock via a frequency/phase mechanism
  - Systems of difference equations in maths



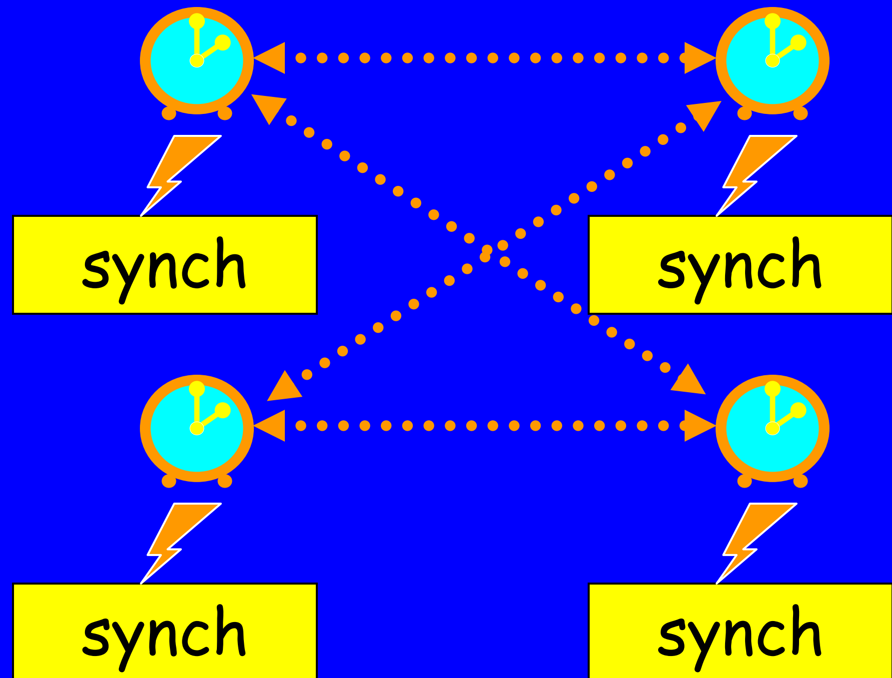
The Essence of Polychrony (2)

- Synchronous software:
  - "if-then-else" statements activate their children statements
  - Each module has a single clock that triggers all signals
  - The different clocks are derived from a global master clock via a boolean clock mechanism, recursively



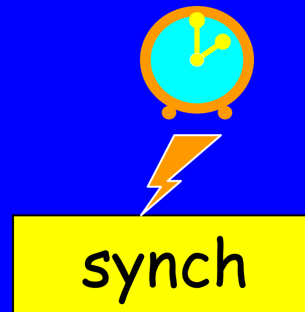
The Essence of Polychrony (3)

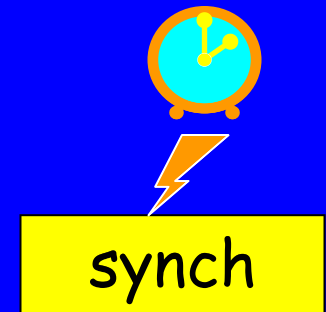
- <u>Open systems</u> with synchronous software components: Polychronous
  - Prepared to accept more components
  - Each module is synchronous software
  - There is no global master clock
  - The different clocks can be <u>related</u> by synchronization constraints



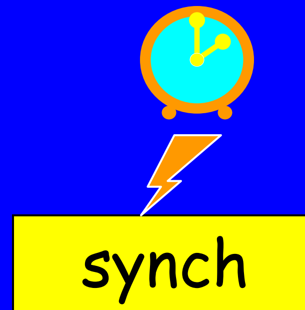
The Essence of Polychrony (4)

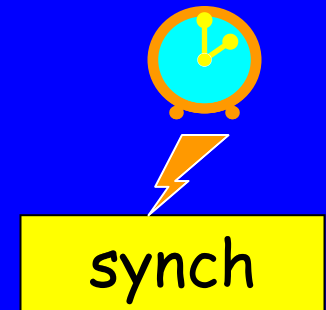
- Open systems with synchronous software components: Polychronous





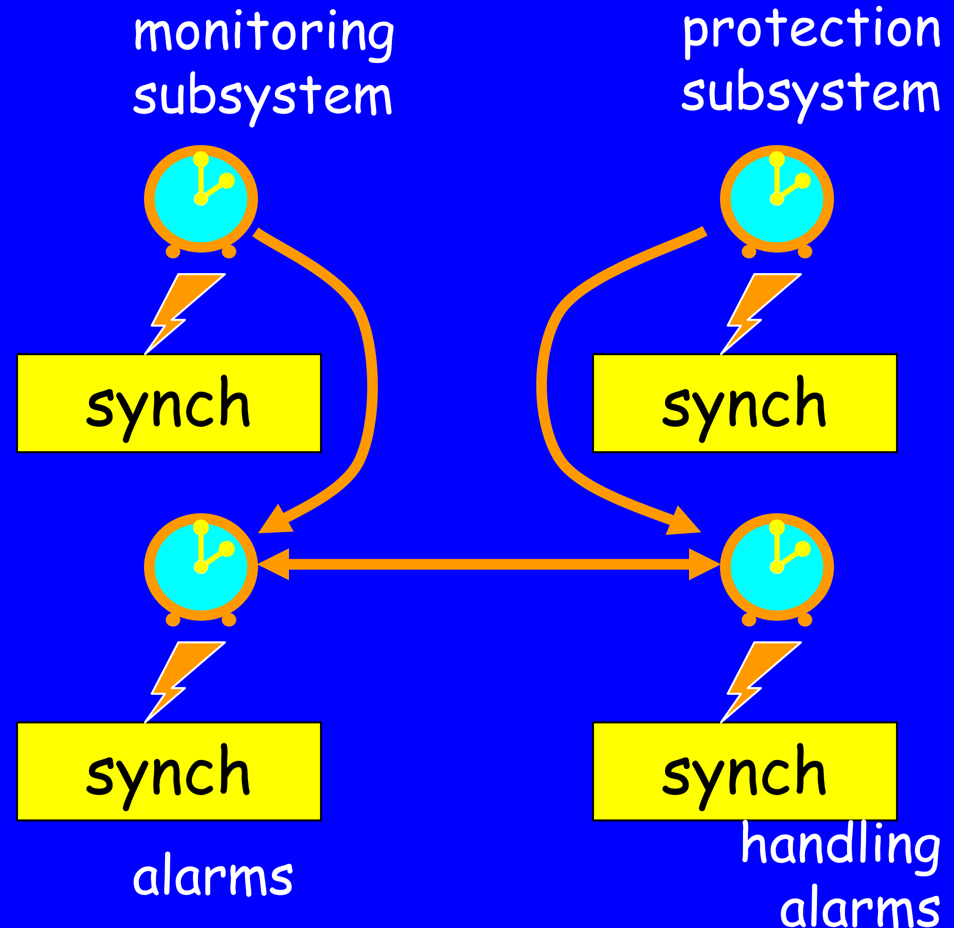
  - If no synchronization constraints: asynchrony!!





The Essence of Polychrony (5)

- Open systems with synchronous software components: Polychronous
  - The different clocks can be related by synchronization constraints



Pushing Polychrony to its limits: Signal

- Open systems with synchronous software components: Polychronous
  - Prepared to accept more components
  - Each module is synchronous software
  - There is no global master clock
  - The different clocks can be related by synchronization constraints

- Each individual signal has its own private clock
- There is no global master clock
- The different clocks can be related by synchronization constraints

Pushing Polychrony to its limits: Signal

- Clocks act as a typing system
- Clock synchronization is difficult by hand
- Hence clock synchronization is synthesized, not verified
- This is unlike Lustre, where clocks are only verified
  ⇒ replaced in practice by *activation conditions*, which do not act as types

These design choices are debatable

- Each individual signal has its own private clock
- There is no global master clock
- The different clocks can be related by synchronization constraints

Signal/Polychrony: zooming in

Below is a pseudo code that we shall discuss

```
loop
     [ when X>0 then
          [get Y ; emit Z=Y+pre(Y)] ]
 ||
     [ present X then
       [ present Z then emit U=X+Z
                    else emit V=2X ] ]
end
```

Signal/Polychrony: zooming in

```
loop  % unique non terminating while loop
      [ when X>0 then
           [get Y ; emit Z=Y+pre(Y)] ]
 ||
      [ present X then
        [ present Z then emit U=X+Z
                       else emit V=2X ] ]
end
```

Signal/Polychrony: zooming in

```
loop
     [ when X>0 then
          [get Y ; emit Z=Y+pre(Y)] ]
  ||   % perfectly synchronous parallel composition
     [ present X then
        [ present Z then emit U=X+Z
                         else emit V=2X ] ]
end
```

Signal/Polychrony: zooming in

```
loop
    [ when X>0 then
        [get Y ; emit Z=Y+pre(Y)] ]
 ||
    [ present X then
      [ present Z then emit U=X+Z
                  else emit V=2X ] ]
end
```

Signal/Polychrony: zooming in

```
loop
     [ when X>0 then
          [get Y ; emit Z=Y+pre(Y)] ]
end
```

?X	3	-2	5	0	-1	-3	2	1
?Y	2		1				5	3
!Z	*		3				6	8

Signal/Polychrony: zooming in

```
loop
      [ when X>0 then
            [get Y ; emit Z=Y+pre(Y)] ]
 ||
       [ present X then
         [ present Z then emit U=X+Z
                     else emit V=2X ] ]
end
```

Signal/Polychrony: zooming in

loop

?X	3	-2	5	0	-1	-3	2	1
?Z	*		3				6	8
!V		-4		0	-2	-6		

```
[ present X then
  [ present Z then emit U=X+Z
                  else emit V=2X ] ]
end
```

Signal/Polychrony: zooming in

```
loop
     [ when X>0 then
          [get Y ; emit Z=Y+pre(Y)] ]
  ||  % synchronizes the two blocks by unification
     [ present X then
       [ present Z then emit U=X+Z
                   else emit V=2X ] ]
end
```

Signal/Polychrony: zooming in

loop

?X	3	-2	5	0	-1	-3	2	1
?Y	2		1				5	3
!Z	*		3				6	8

|| % synchronizes the two blocks by unification

?X	3	-2	5	0	-1	-3	2	1
?Z	*		3				6	8
!V		-4		0	-2	-6		

end

Polychrony: the Signal syntax

```
loop
      [ when X>0 then
            [get Y ; emit Z=Y+pre(Y)] ]
 ||
      [ present X then
        [ present Z then emit U=X+Z
                    else emit V=2X ] ]
end
```

pseudo-code

```
 (
   ( Y ^= when X>0
   | Z := Y + Y$1 )
 |
   ( U := (X when event Z) + Z
   | V := 2X when not event Z )
 )
```

Signal syntax

(systems of dataflow equations)

Polychrony: the Signal syntax & synchro

```
(
  ( Y ^= when X>0
  | Z ^= Y ^= Y$1 )
|
  ( U ^= Z
  | V ^= X ^- Z )
)
|
(
  ( Y ^= when X>0
  | Z := Y + Y$1 )
|
  ( U := (X when event Z) + Z
  | V := 2X when not event Z )
)
```

^= , ^-
equality and
difference
of clocks

clock calculus

Signal syntax

Contents

- Motivation
- Signal Model of Computation and Communication
- Key data structure for Signal compilation: clock-and-causality calculus
- Constructive Semantics
- Separate compilation and a notion of interface
- Deploying over distributed (possibly asynchronous) architectures
- Use in architecture modeling and analysis

What is the problem?

- As we have seen, Signal programs are systems of equations with dynamics
- This makes parallel composition straightforward defining and studying: associative and commutative (allows for flexible re-architecturing)
- However, there is no free lunch: systems of equations are declarative, not operational. Producing executable code in fact amounts to "solving" these systems of equations for their outputs, regarding inputs as free parameters. Think of

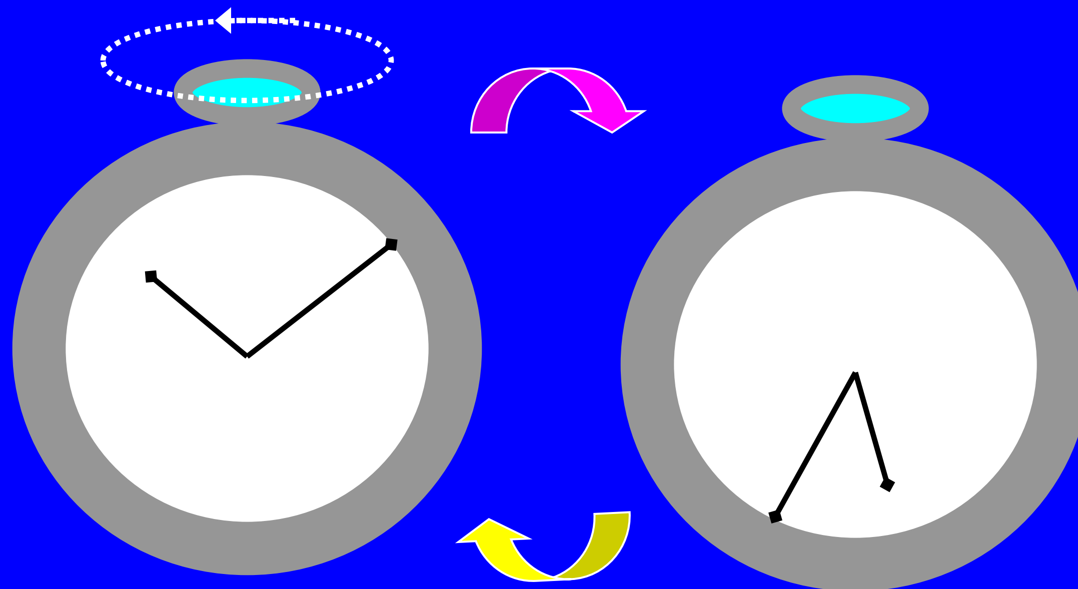
$$AX = B \Rightarrow X = f(A,B)\ ?$$

- Computable solutions do not generally exist for the kind of equations we get with Signal (data types are not restricted). We thus need a specific abstraction mechanism for solving these equations. We study it now in details.

The old fashioned Signal watch

This is an old mechanical watch like the one I have. Turn the spring. The watch goes for some time, and then stops. When it stops, turn again the spring… and so on…

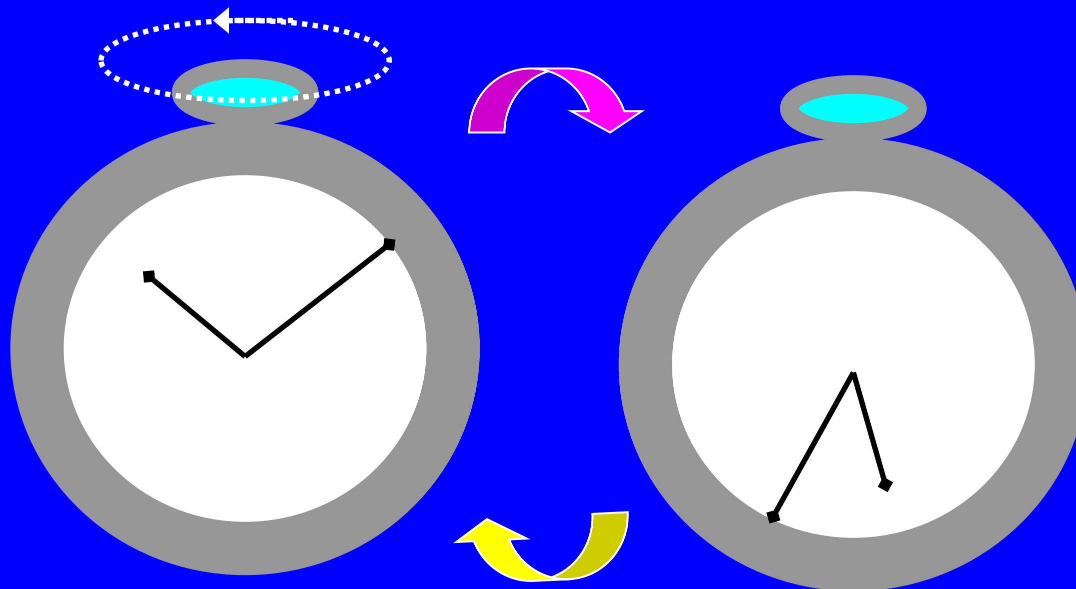
This is an interesting example, as the output up-samples the input. We show how to analyze and execute it.



The old fashioned Signal watch

```
(  X := IN default ZX-1
| ZX := X$1 init 0
| IN ^= when (ZX ≤ 0)
)
```

Signal program
Input **IN**
Returns **X**



The old fashioned Signal watch

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= when B
|  H ^= B ^= X ^= ZX
)
```

Signal program, expanded to facilitate manual analysis

The old fashioned Signal watch

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= when B = [B]
|  H ^= B ^= X ^= ZX
)
```

We show how to represent in a unique data structure both the synchronisation and the dependencies.

Then we show how this data structure can be used to generate schedulings for execution.

Intermezzo: back to solving equations

- Solving equations: which abstraction?

```
X := A+B
```

- We interpret operator + in an abstract way, by rewriting: substitute X for the expression A+B everywhere X occurs in the program

- Rewriting is best encoded using Directed Graphs

$$(A,B) \rightarrow X$$

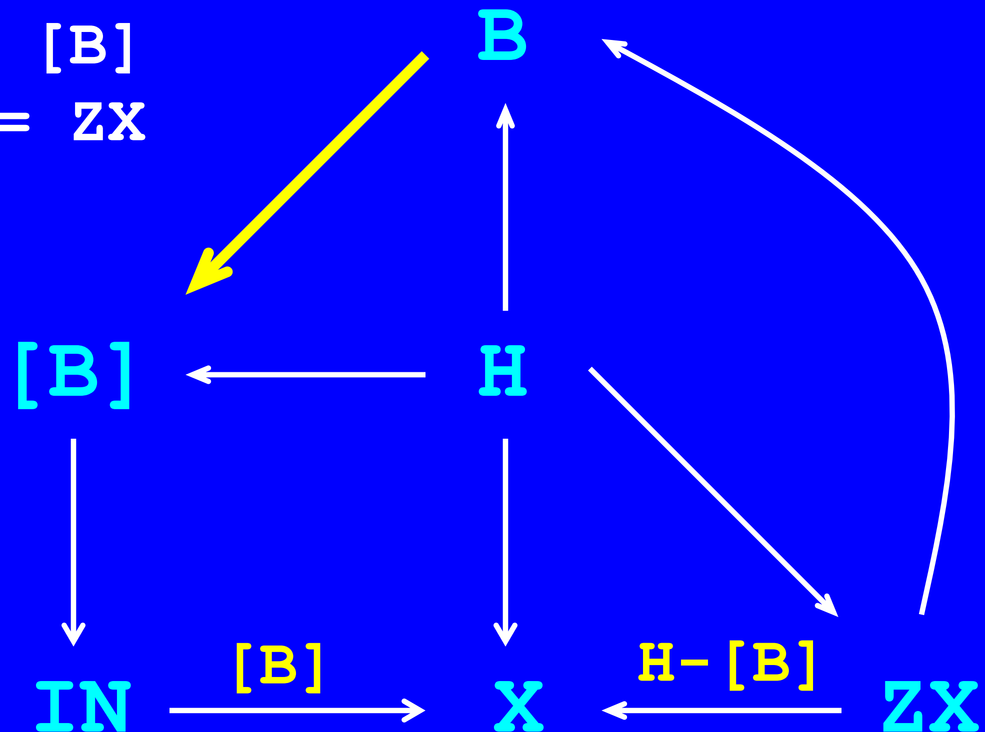
- However, such causality constraints are generally state dependent. We capture this by labeling them with clocks.

The old fashioned Signal watch: scheduling and executing

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= when B = [B]
|  H ^= B ^= X ^= ZX
)
```

Causality

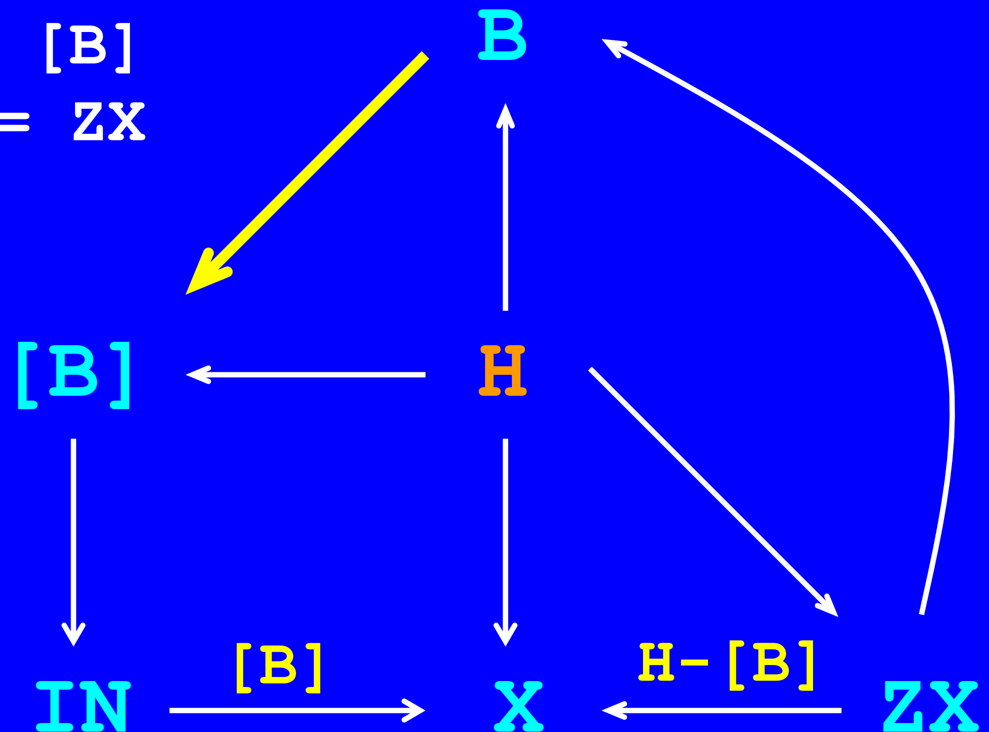
Clock inclusion & causality



The old fashioned Signal watch: scheduling and executing

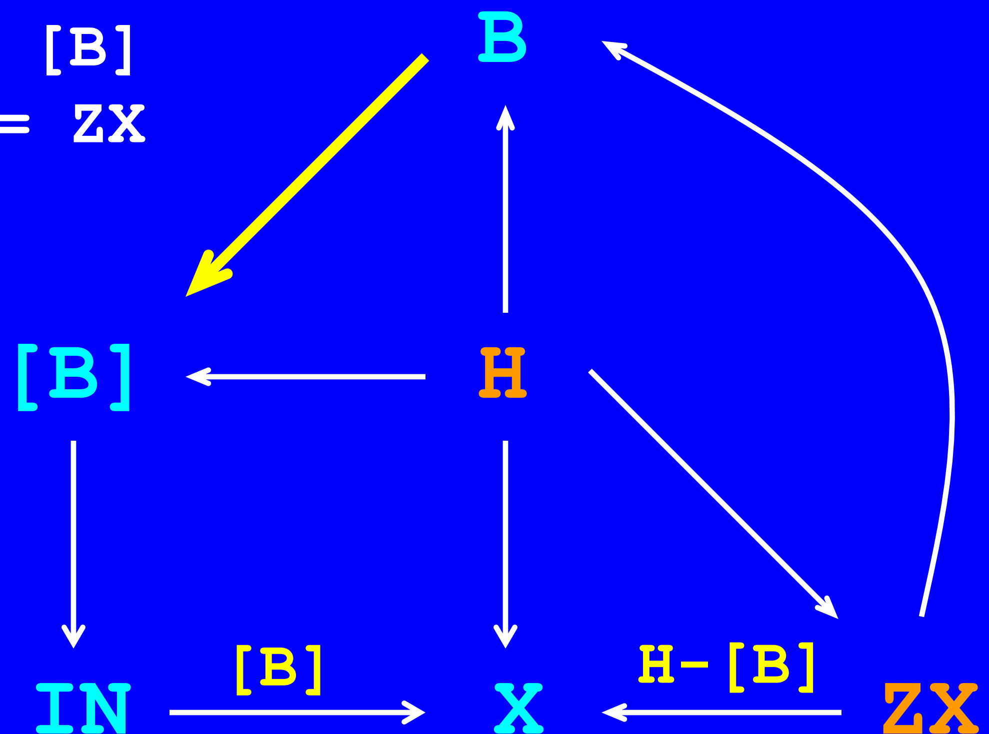
```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= when B = [B]
|  H ^= B ^= X ^= ZX
)
```

In orange: evaluated



The old fashioned Signal watch: scheduling and executing

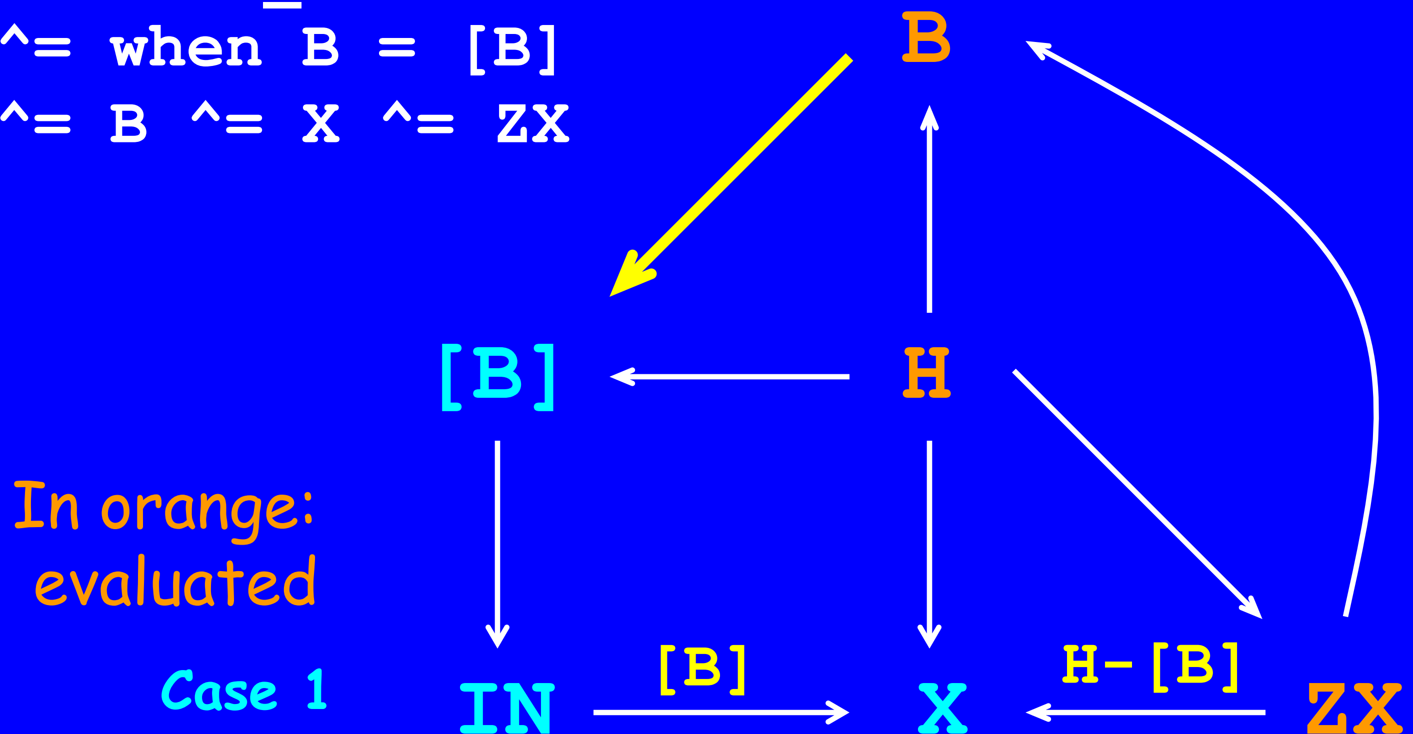
```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= when B = [B]
|  H ^= B ^= X ^= ZX
)
```



In orange: evaluated

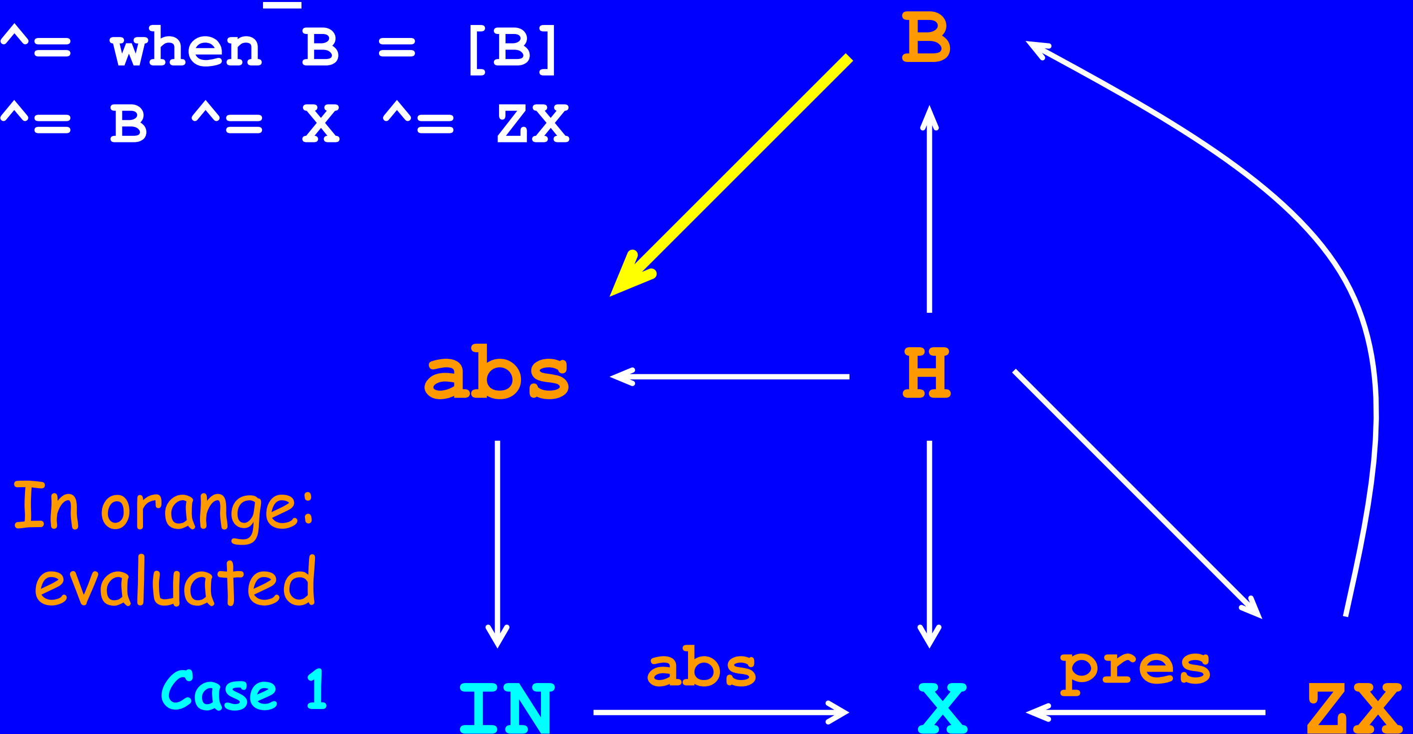
The old fashioned Signal watch: scheduling and executing

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= when B = [B]
|  H ^= B ^= X ^= ZX
)
```



The old fashioned Signal watch: scheduling and executing

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= when B = [B]
|  H ^= B ^= X ^= ZX
)
```

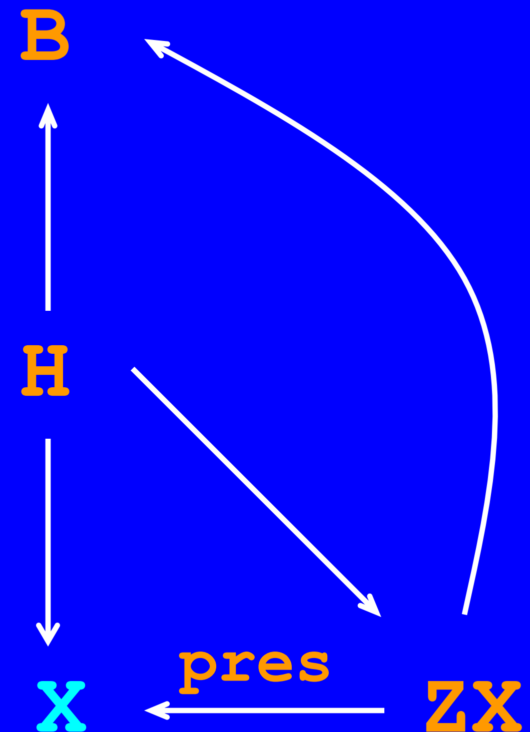


The old fashioned Signal watch: scheduling and executing

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= when B = [B]
|  H ^= B ^= X ^= ZX
)
```

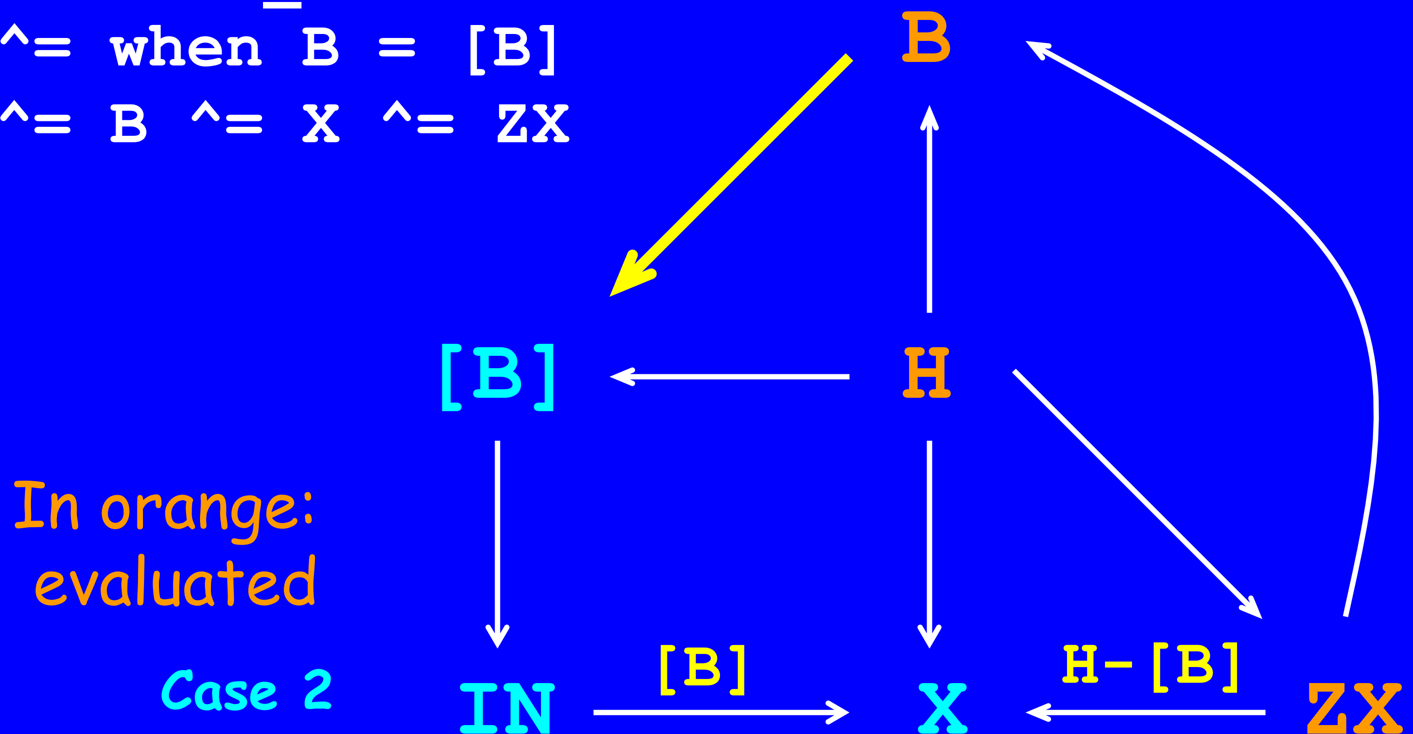
In orange: evaluated

Case 1



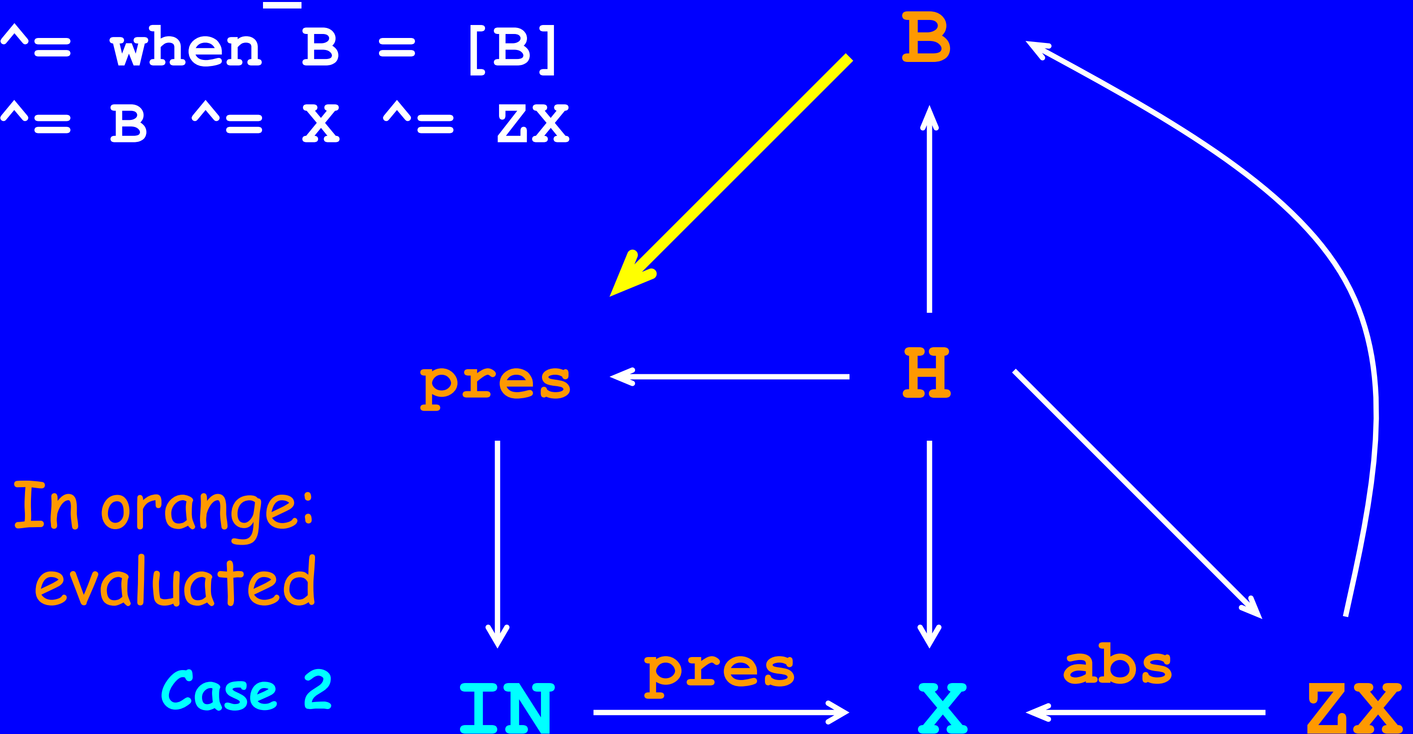
The old fashioned Signal watch: scheduling and executing

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= when B = [B]
|  H ^= B ^= X ^= ZX
)
```



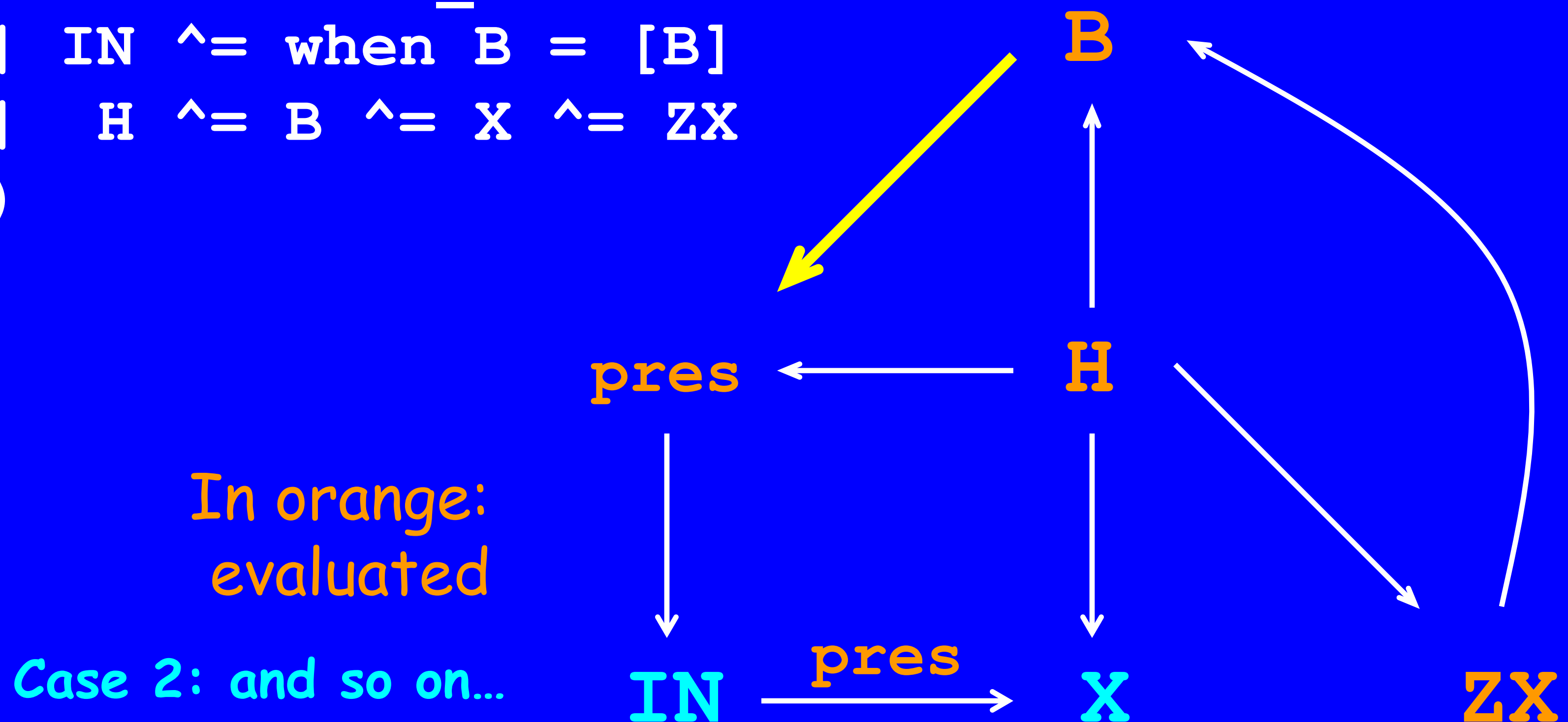
The old fashioned Signal watch: scheduling and executing

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= when B = [B]
|  H ^= B ^= X ^= ZX
)
```



The old fashioned Signal watch: scheduling and executing

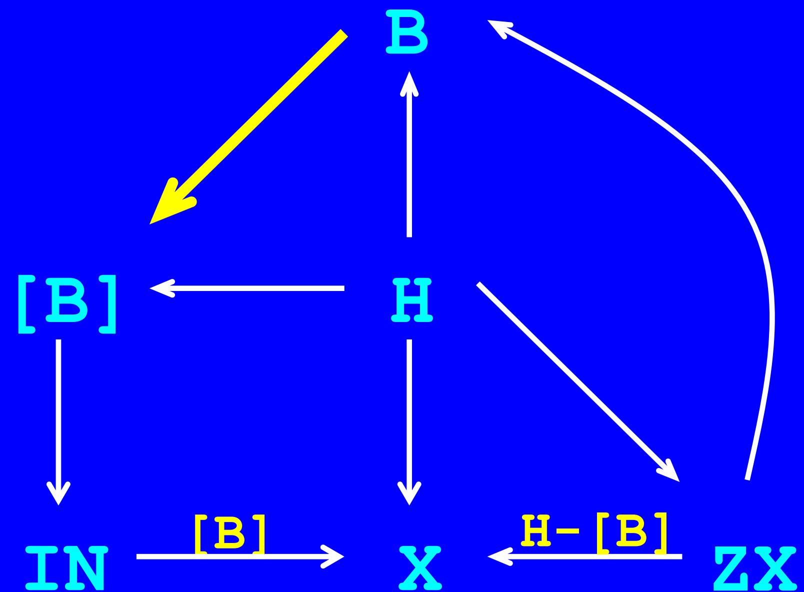
```
(   X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= when B = [B]
|  H ^= B ^= X ^= ZX
)
```



Clock & Causality Analysis: the key data structure

```
(
   (  B := (ZX ≤ 0)
   | IN ^= when B = [B]
   |  H ^= B ^= X ^= ZX )
|
   (  X  ←   IN when B
   |  X  ←   ZX when not B
   |  B  ←  (H,ZX)
   | [B] ←   B
   | ZX  ←   H
   | IN  ←  [B] )
)
```

Part of Signal syntax



Clock & Causality Analysis: the key data structure

```
(
   (  B := (ZX ≤ 0)
   | IN ^= when B = [B]
   |  H ^= B ^= X ^= ZX )
|
   (  X  ←  IN when B
   |  X  ←  ZX when not B
   |  B  ← (H,ZX)
   | [B] ←  B
   | ZX  ←  H
   | IN  ← [B] )
)
```

Part of Signal syntax

- {scheduling + synchronization}
- a calculus of the Signal engine
- <u>an interface model</u>
  - right support for separate compilation
  - islands of fully rigid scheduling can be safely compiled to C

Intermezzo: inputs, outputs, executable, non-determinisic, no-semantics, and more…

- Think again of equations in high school linear algebra, where (a,b) are constants:

$$ax+uy = 0$$
$$bx+vy = 0$$

- Solving for (x,y) amounts to declaring (u,v) as inputs and (x,y) as outputs
- However, we learnt that, depending on the values of constants (a,b), map $(u,v)\rightarrow(x,y)$ may possess:
  - 1 solution "well defined, deterministic, executable"
  - 0 solution "blocking, no semantics"
  - ∞ solution "nondeterministic, no semantics"
- A signal may have a schizophrenic status: partially/sometimes input/output

Intermezzo: inputs and outputs are not just a syntactic, but a semantic notion

```
(
   (  B := (ZX ≤ 0)
   | IN ^= when B = [B]            output: clock(IN)
   |  H ^= B ^= X ^= ZX )          input:  clock(X)
|
   (  X  ←  IN when B              input:  val(IN)
   |  X  ←  ZX when not B          output: val(X)
   |  B  ← (H,ZX)
   | [B] ←  B
   | ZX  ←  H
   | IN  ← [B] )
)
```

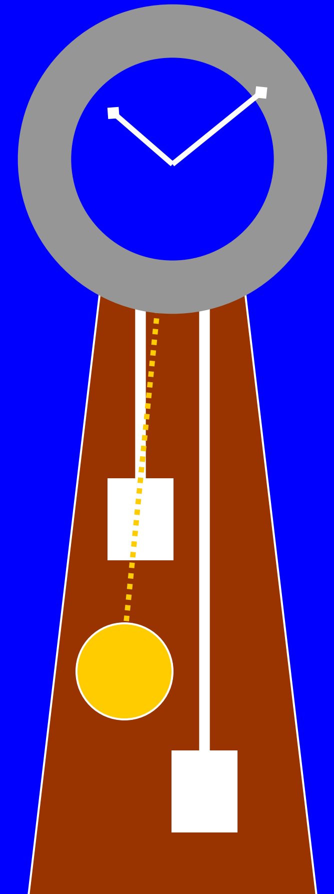
- A signal may have a schizophrenic status: partially/sometimes input/output

The very-old fashioned Signal watch

This is an even older watch! The difference with the previous one is the following: when the watch goes, then the weights move “backward”, back to their initial position.

Thus, for this watch, the weights play alternatively the role of an input (when the user moves the weights), and of an output (when the watch goes).

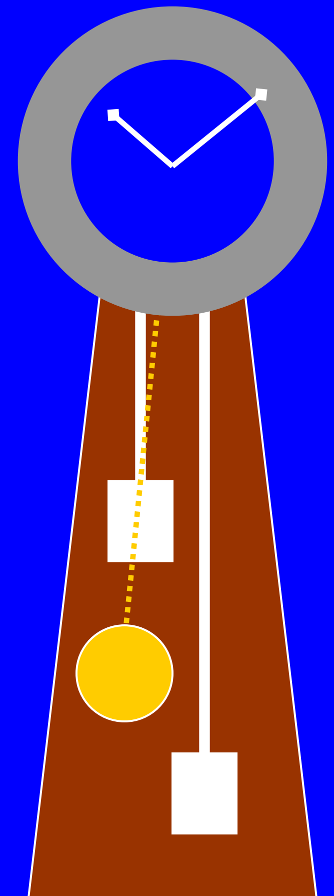
This causes a cycle in dependencies.



The very-old fashioned Signal watch

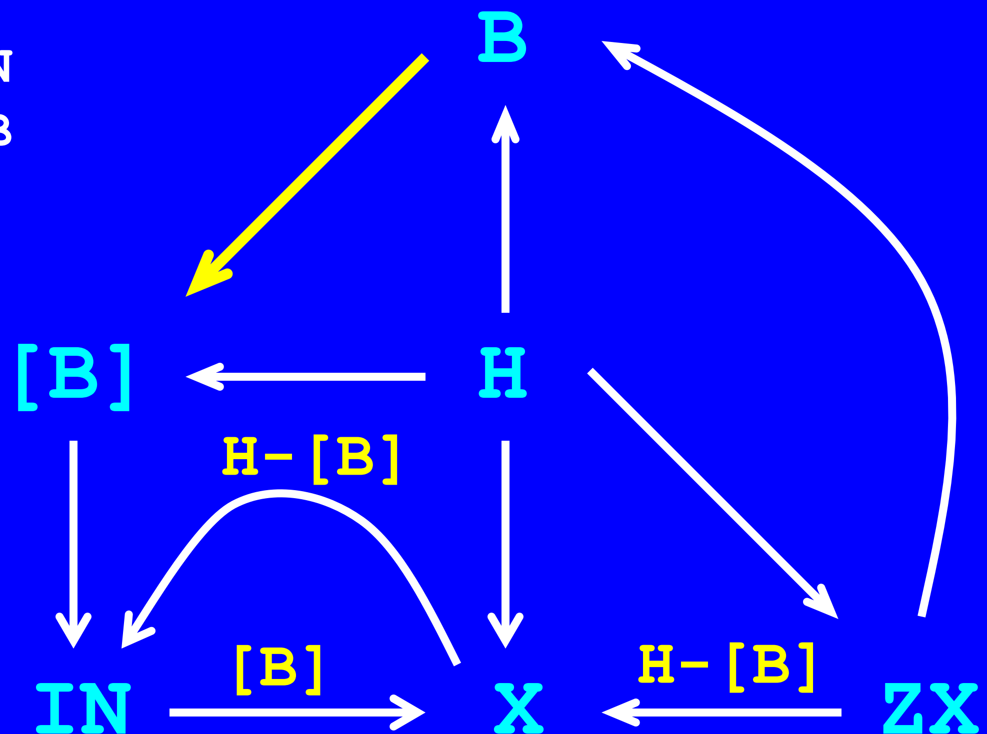
```
(|  X := (IN when B) default ZX-1
 | ZX := X$1 init 0
 |  B := (ZX ≤ 0)
 |  H ^= B ^= X ^= ZX ^= IN
 | IN := X when not B
 |)
```

The backward move of the weights is captured by the yellow statement, where IN becomes an output!



The very-old fashioned Signal watch

```
(|  X := (IN when B) default ZX-1
 | ZX := X$1 init 0
 |  B := (ZX ≤ 0)
 |[B] := when B
 |  H ^= B ^= X ^= IN
 | IN := X when not B
 |)
```

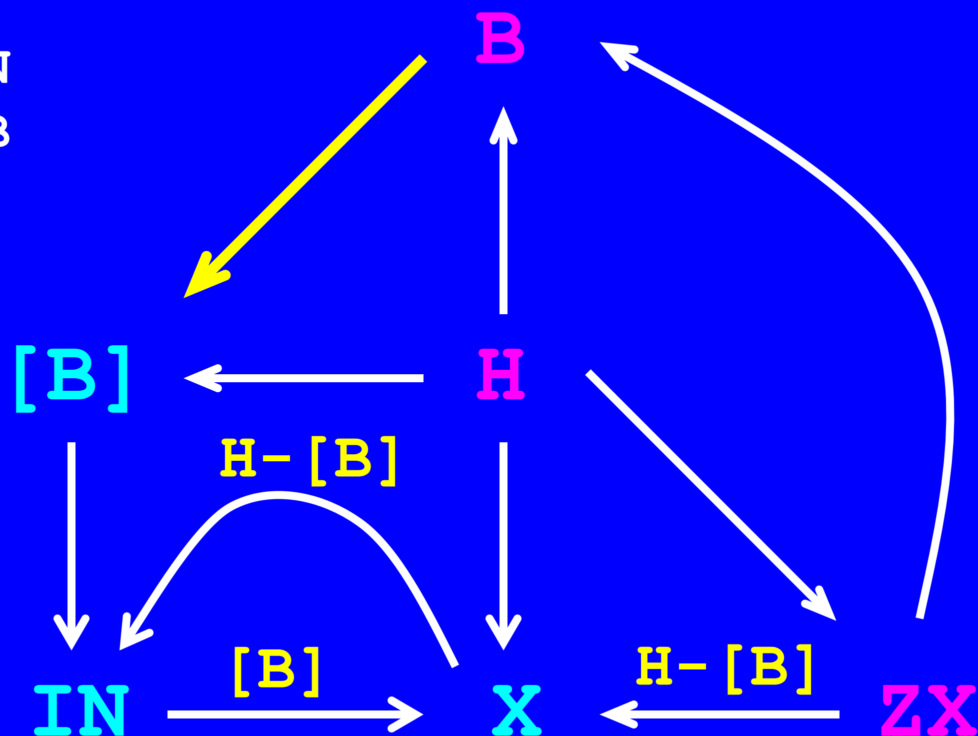


State inclusion & causality

Case 1

The very-old fashioned Signal watch

```
(|  X := (IN when B) default ZX-1
 | ZX := X$1 init 0
 |  B := (ZX ≤ 0)
 |[B] := when B
 |  H ^= B ^= X ^= IN
 | IN := X when not B
 |)
```



State inclusion & causality

Case 1

The very-old fashioned Signal watch

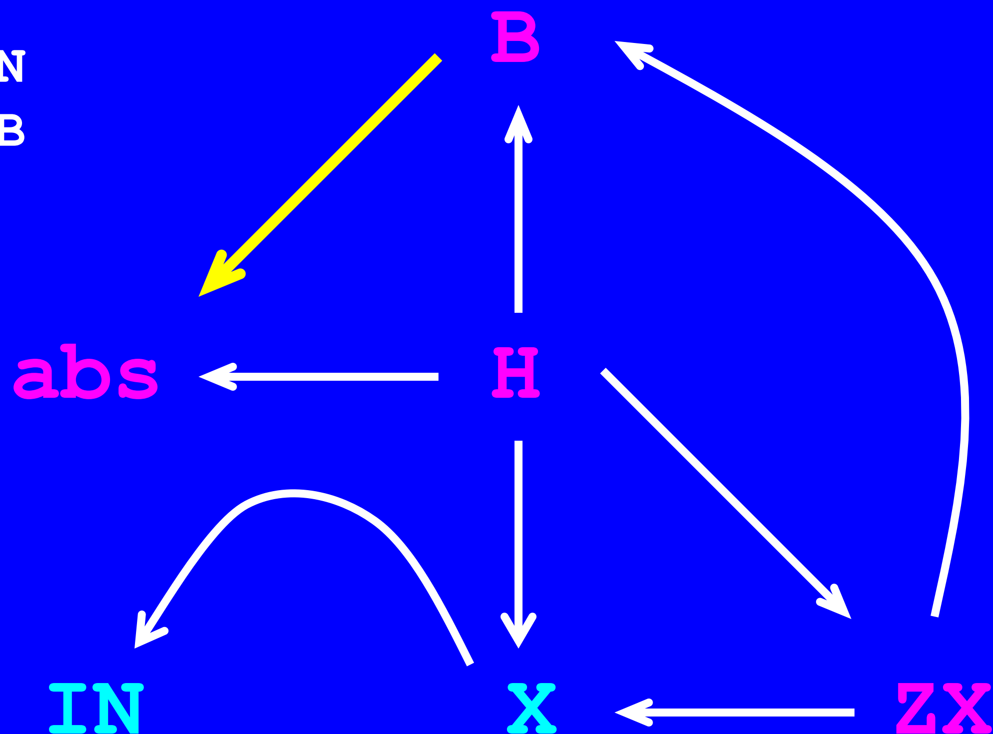
```
(|  X := (IN when B) default ZX-1
 | ZX := X$1 init 0
 |  B := (ZX ≤ 0)
 |[B] := when B
 |  H ^= B ^= X ^= IN
 | IN := X when not B
 |)
```

State inclusion & causality

Case 1

The very-old fashioned Signal watch

```
(|  X := (IN when B) default ZX-1
 | ZX := X$1 init 0
 |  B := (ZX ≤ 0)
 |[B] := when B
 |  H ^= B ^= X ^= IN
 | IN := X when not B
 |)
```



State inclusion & causality

Case 1

Summary: the clock-and-causality calculus, an interface model

Clock-and-causality abstraction

- encodes all valid schedules for each state of the program
- obeys the graph algebra below
- sufficient interface model for single designer component based design
- not enough for distributed multi-team development (contracts neded)

```
(
   (  B  := (ZX ≤ 0)
   | IN ^= when B = [B]
   |  H ^= B ^= X ^= ZX )
|
   (  X   ←   IN when B
   |  X   ←   ZX when not B
   |  B   ←  (H,ZX)
   | [B]  ←   B
   | ZX   ←   H
   | IN   ←  [B] )
)
```

```
(  X  ←  Y when B           ⟹    X  ←  Y when
|  X  ←  Y when C  )                     (when B default when C)

(  X  ←  Y when B           ⟹    X  ←  Z when B when C
|  Y  ←  Z when C  )
```

Summary: the clock-and-causality calculus, a framework for compilation

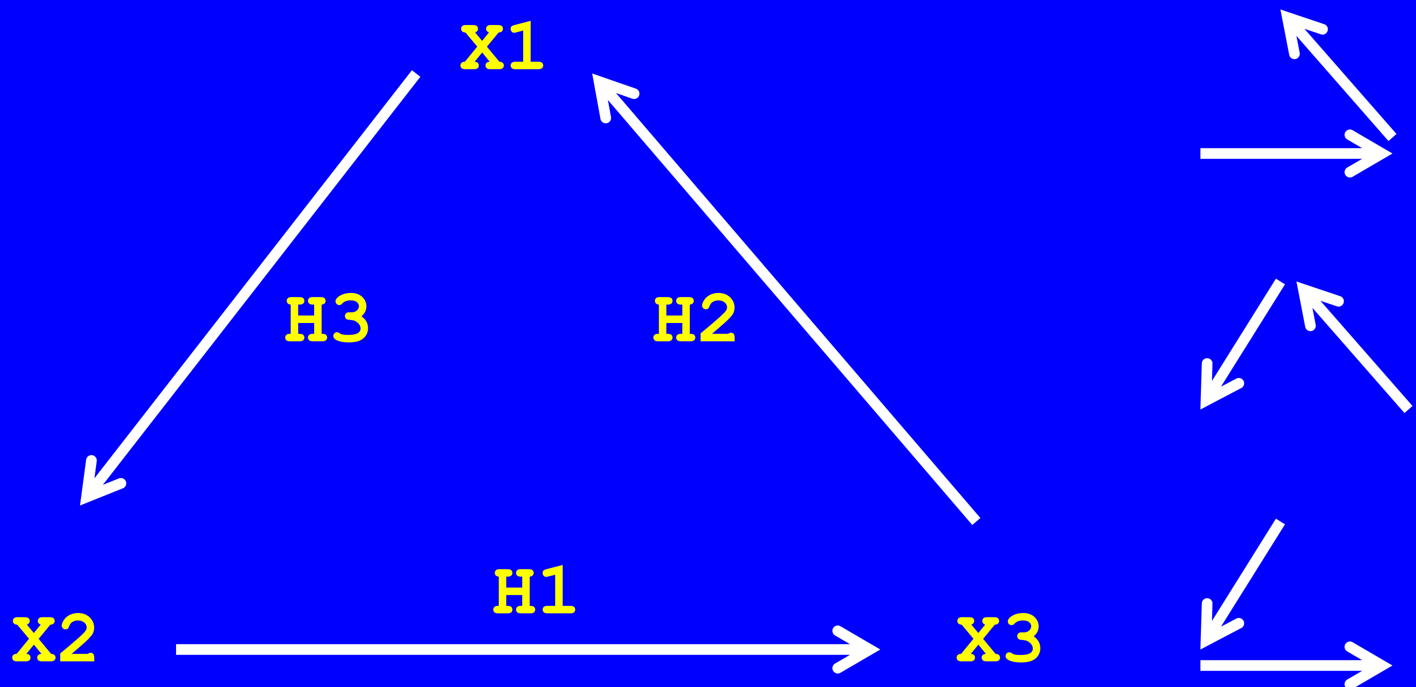
Clock-and-causality abstraction

- encodes all valid schedules for each state of the program
- intermediate code for separate compilation:
  - sequentialize when no room left for concurrency
  - otherwise keep concurrent
- compilation by Signal → Signal program rewriting

```
(
   (  B := (ZX ≤ 0)
   | IN ^= when B = [B]
   |  H ^= B ^= X ^= ZX )
|
   (  X   ←   IN when B
   |  X   ←   ZX when not B
   |  B   ← (H,ZX)
   | [B]  ←   B
   | ZX   ←   H
   | IN   ← [B] )
)
```

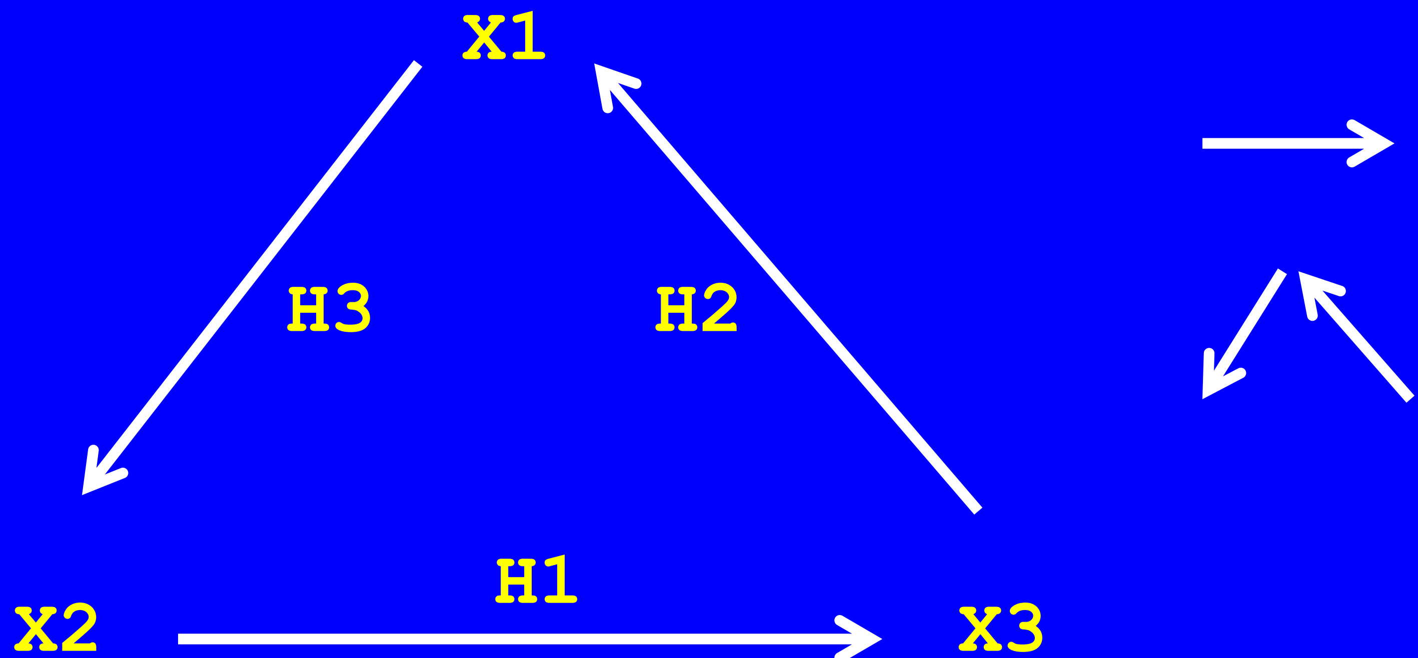
Advanced: getting static scheduling

Assume causal correctness: $H1 \cap H2 \cap H3 = \emptyset$. If $Hi \cap Hj \neq \emptyset$ then three different cuts are required $\Rightarrow$ three replicas are necessary



Advanced: getting static scheduling

If $H1 \cap H2 = \emptyset, H2 \supseteq H3$, then two different cuts are required, thus two replicas only are necessary



Contents

- Motivation
- Signal Model of Computation and Communication
- Key data structure for Signal compilation: clock-and-causality calculus
- Constructive Semantics
- Separate compilation and a notion of interface
- Deploying over distributed (possibly asynchronous) architectures
- Use in architecture modeling and analysis

The essence of constructive semantics

Lustre

- A Lustre program with no zero-delay loop is a *stream function* in Kahn sense
- Execution follows as a Kahn Process Network
- Executions can (but need not) be synchronized as a sequence of global reactions
- Simple, no need for any kind of Constructive Semantics

The essence of constructive semantics

Lustre

- A Lustre program with no zero-delay loop is a *function* in Kahn sense
- Execution follows as a Kahn Process Network
- Executions can (but need not) be synchronized as a sequence of global reactions
- Simple, no need for any kind of Constructive Semantics

Signal

- Due to the possibility of constraints relating clocks, Signal programs can be deterministic, nondeterministic, or "schyzophrenic"
- Difficulties:
  - Non-determinism
  - Causality analysis
- Need special work to
  - *Check if a program is deterministic*
  - Schedule evaluation actions (micro-steps)
- Constructive Semantics takes care of this

The essence of constructive semantics

Esterel

- Due to the possibility of <u>instantaneous dialogs</u>, Esterel programs can be deterministic, nondeterministic, or "schyzophrenic"
- Difficulties:
  - Non-determinism
  - *Causality analysis*
- Need special work to
  - *Check if a program is deterministic*
  - Schedule evaluation actions (micro-steps)
- Constructive Semantics takes care of this

Signal

- Due to the possibility of <u>constraints relating clocks</u>, Signal programs can be deterministic, nondeterministic, or "schyzophrenic"
- Difficulties:
  - Non-determinism
  - *Causality analysis*
- Need special work to
  - *Check if a program is deterministic*
  - Schedule evaluation actions (micro-steps)
- Constructive Semantics takes care of this

The essence of constructive semantics

Esterel

- Due to the possibility of instantaneous dialogs, Esterel programs can be deterministic, nondeterministic, or "schyzophrenic"
- Difficulties:
  - Non-determinism
  - *Causality analysis*
- Need special work to
  - *Check if a program is deterministic*
  - Schedule evaluation actions (micro-steps)
- Constructive Semantics takes care of this

Example [Berry 97]

Logically incorrect

```
module P3:
output O;
present O else emit O end
end module
```

Nondeterministic:

```
module P4:
output O;
present O then emit O end
end module
```

The essence of constructive semantics

Example

Nondeterministic : a mailbox with asynchronous reads and writes

```
   X   := IN default (X$1)
| OUT  :=    X  when event OUT
```

yields clock constraints

```
   X  ^> IN
| OUT ^< X
```

Signal

- Due to the possibility of constraints relating clocks, Signal programs can be deterministic, nondeterministic, or "schyzophrenic"
- Difficulties:
  - Non-determinism
  - Causality analysis
- Need special work to
  - Check if a program is deterministic
  - Schedule evaluation actions (micro-steps)
- Constructive Semantics takes care of this

The essence of constructive semantics

Example

Nondeterministic : a mailbox with asynchronous reads and writes

```
   X  := IN default (X$1)
| OUT := f(X) when event OUT
```

yields clock constraints

```
   X  ^> IN
| OUT ^< X
```

subtle causality analysis

Signal

- Due to the possibility of constraints relating clocks, Signal programs can be deterministic, nondeterministic, or "schyzophrenic"
- Difficulties:
  - *Non-determinism*
  - *Causality analysis*
- Need special work to
  - *Check if a program is deterministic*
  - *Schedule evaluation actions (micro-steps)*
- Constructive Semantics takes care of this

The essence of constructive semantics

Example cont'd

Instantaneous dialog

```
   X := IN default (X$1-1)
| IN ^= when X<0
```

yields additional clock constraint

```
    X ^> IN
```

causality analysis should reveal a causality circuit

Signal

- Due to the possibility of constraints relating clocks, Signal programs can be deterministic, nondeterministic, or "schyzophrenic"
- Difficulties:
  - Non-determinism
  - Causality analysis
- Need special work to
  - *Check if a program is deterministic*
  - Schedule evaluation actions (micro-steps)
- Constructive Semantics takes care of this

The essence of constructive semantics

Esterel: summary

- Due to the possibility of instantaneous dialogs, Esterel programs can be deterministic, nondeterministic, or "schyzophrenic"
- Difficulties:
  - Non-determinism
  - *Causality analysis*
- Need special work to
  - *Check if a program is deterministic*
  - Schedule evaluation actions (micro-steps)
- Constructive Semantics takes care of this

Signal: summary

- Due to the possibility of constraints relating clocks, Signal programs can be deterministic, nondeterministic, or "schyzophrenic"
- Difficulties:
  - Non-determinism
  - *Causality analysis*
- Need special work to
  - *Check if a program is deterministic*
  - Schedule evaluation actions (micro-steps)
- Constructive Semantics takes care of this

The role of constructive semantics

Program

- Each program fragment generates a causality constraint inducing a scheduling constraint

Scheduling constraints

- Combine them with the actual evaluation of variables
- Each variable is evaluated once
- Yields a description of all correct schedules for executing the program (if any)

Yields micro-step semantics for the program

The essence of constructive semantics

Esterel

- Scott domain for signals
  - ⊥, present, absent
- Facts attached to control actions
  - Must
  - Cannot
- Causality relations
  - From control actions to the evaluation of signals and variables
  - From a context to the evaluation of Must and Cannot for a control action
- ⊥: not yet evaluated

Signal

- Scott domain for clocks and signals
  - ⊥, true, false, (any), absent
- Facts: values of
  - Clocks
  - Boolean Signals
- Causality relations
  - From a clock to the signal evaluation it triggers
  - From signals to clocks of the form `when B`, for `B` a predicate involving these signals
  - Between signals in data-flow expressions

The essence of constructive semantics

Causality relations generate Scheduling Constraints

- Scott domain for signals
  - ⊥, present, absent
- Facts attached to control actions
  - Must
  - Cannot
- Causality relations
  - From control actions to the evaluation of signals and variables
  - From a context to the evaluation of Must and Cannot for a control action

- Scott domain for clocks and signals
  - ⊥, true, false, (any), absent
- Facts: values of
  - Clocks
  - Boolean Signals
- Causality relations
  - From a clock to the signal evaluation it triggers
  - From signals to clocks of the form `when B`, for `B` a predicate involving these signals
  - Between signals in data-flow expressions

Constructive Semantics in Signal

Three equivalent representations:

- Clock and causality graph (already seen)
- Scott domain
- Micro-step automata

```
(
   (  B := (ZX ≤ 0)
   | IN ^= when B = [B]
   |  H ^= B ^= X ^= ZX )
|
   (  X   ←   IN when B
   |  X   ←   ZX when not B
   |  B   ← (H,ZX)
   | [B]  ←   B
   | ZX   ←   H
   | IN   ← [B] )
)
```

Constructive Semantics in Signal

Three equivalent representations:

- Clock and causality graph
- Scott domain
- Micro-step automata

Scott domain

$\bot$: not yet evaluated
*: absent

$$D = \{ \bot , * , F , T \}$$

$\bot < * , F , T$: Scott domain

F , T, any subsumed by ▪

Scott domain

⊥: not yet evaluated
*: absent

$$\mathbf{D} = \{ \bot , * , \mathbf{F} , \mathbf{T} \}$$

$\bot < *, \mathbf{F}, \mathbf{T}$: Scott domain

F , T, any subsumed by ▪

This way, we can model both the value of a signal and its current status within the current reaction:

not yet evaluated vs. evaluated

Using this Scott domain, encode all SIGNAL statements (in doing this, abstraction is performed)

$X \rightarrow_B Y$ as a constraint in the Scott domain

B \ X	⊥	*	■
⊥	⊥	⊥	⊥
*			
F			
T	⊥		

Y = X when B

as a constraint in the Scott domain

$(B \rightarrow Y) \mid (X \rightarrow_B Y)$

X / B	⊥	*	▪
⊥	⊥	⊥	⊥
*	⊥/*	⊥/*	⊥/*
F	⊥/*	⊥/*	⊥/*
T	⊥	⊥/*	⊥/▪

Y := f(X)

as a constraint in the Scott domain

(H→X)|(X→Y) where H^=X^=Y

X	⊥	*	▪
Y	⊥	⊥/*	⊥/▪

Results (1)

1. Each Signal reaction is fully abstracted as a system of equations in Scott domain; yields a set Σ of configurations for all variables.

Results (1)

1. Each Signal reaction is fully abstracted as a system of equations in Scott domain; yields a set Σ of configurations for all variables.
2. One step of sequential evaluation consists in changing the value of a single variable, from $\perp$ to $\neq\perp$, while staying within Σ

Results (1)

1. Each Signal reaction is fully abstracted as a system of equations in Scott domain; yields a set Σ of configurations for all variables.
2. One step of sequential evaluation consists in changing the value of a single variable, from $\perp$ to $\neq\perp$, while staying within Σ
3. Executing the reaction amounts to finding a connected path in Σ that starts at $(\perp,\ldots,\perp)$ and ends at $(\neq\perp,\ldots,\neq\perp)$; call such a path an execution path (not unique in general)

Results (2): relating Scott semantics to clock & graph semantics

1. If the clock&causality graph is circuit free, then execution paths exist; each execution path corresponds to a valid sequential execution of the program

2. This way of getting valid executions is compositional: systems of Scott equations can be composed

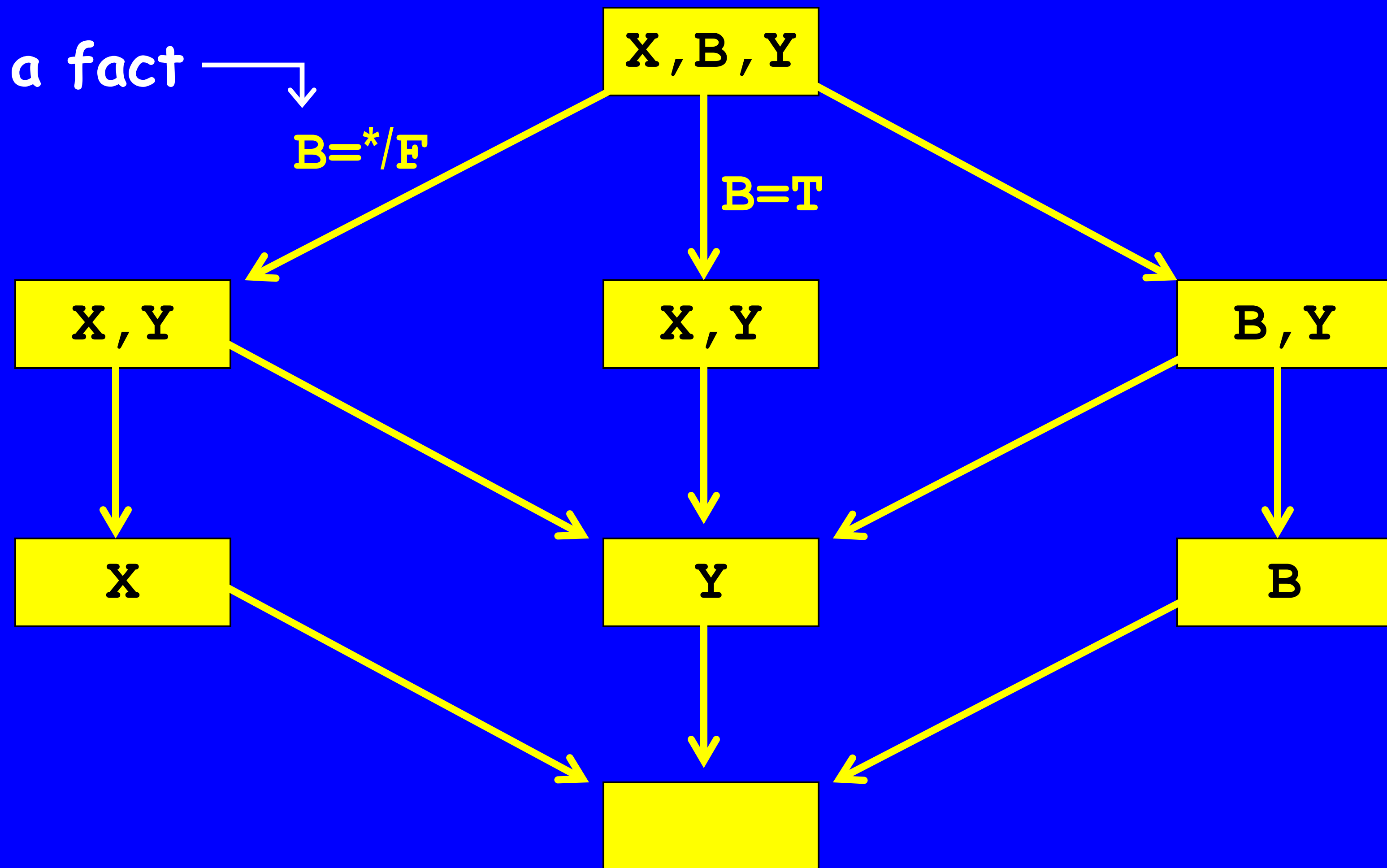
   $\Rightarrow$ supports separate compilation

Constructive Semantics in Signal

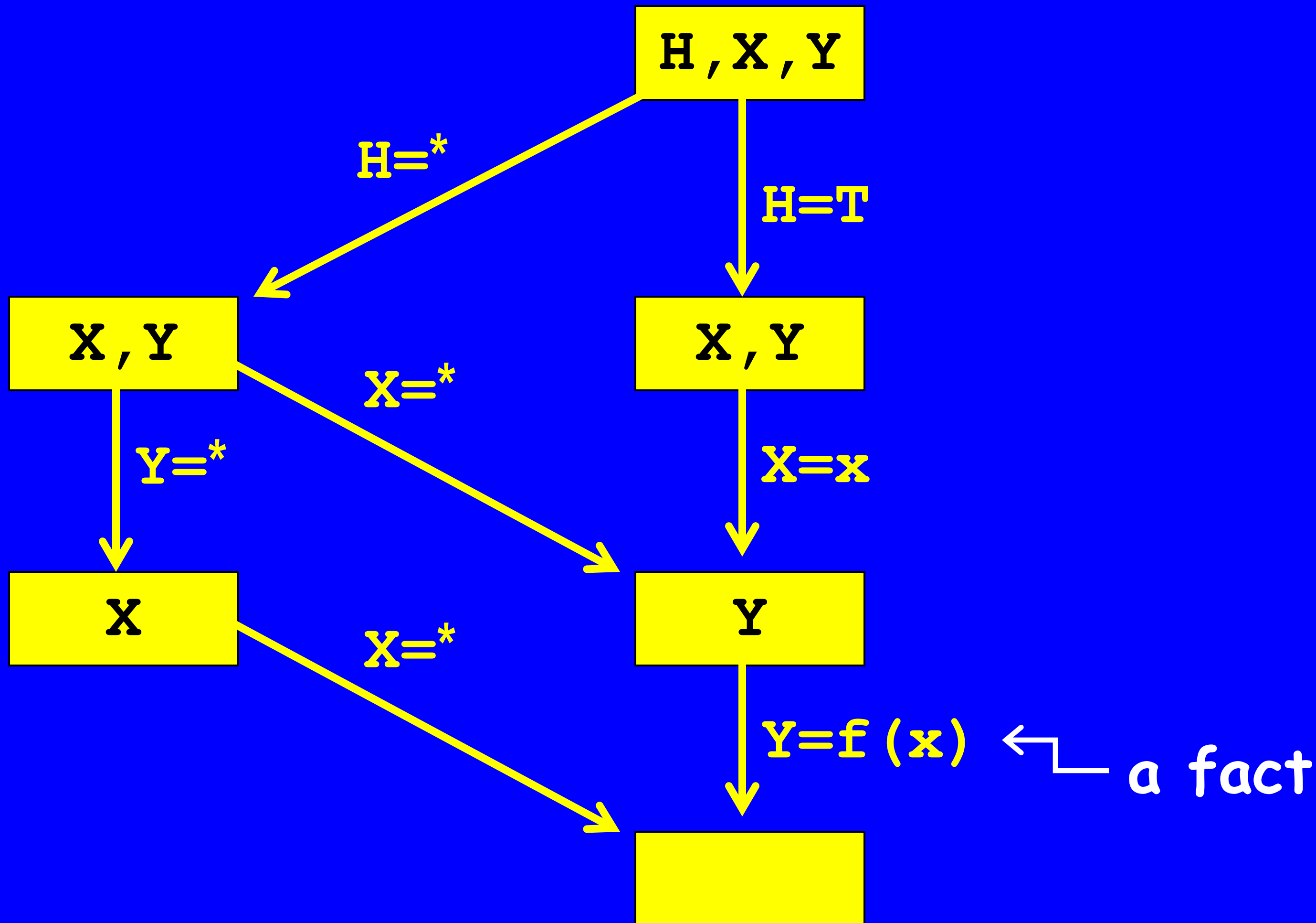
Three equivalent representations:

- Clock and causality graph
- Scott domain
- Micro-step automata

X $\rightarrow_B$ Y as an accepting automaton



Y:=f(X) | H→X | X→Y as an accepting automaton



Micro-step semantics

1. This semantics is obtained by taking the product of the micro-step automata attached to each statement; the product is by synchronizing compatible facts

Micro-step semantics

1. This semantics is obtained by taking the product of the micro-step automata attached to each statement; the product is by synchronizing compatible facts

2. This is modular: we can compose micro-step automata attached to sub-systems

Micro-step semantics

1. This semantics is obtained by taking the product of the micro-step automata attached to each statement; the product is by synchronizing compatible facts
2. This is modular: we can compose micro-step automata attached to sub-systems
3. Such micro-step automata synchronize by exchanging *facts,* i.e., partial information regarding the status of the variables

Micro-step semantics

1. This semantics is obtained by taking the product of the micro-step automata attached to each statement; the product is by synchronizing compatible facts

2. This is modular: we can compose micro-step automata attached to sub-systems

3. Such micro-step automata synchronize by exchanging *facts,* i.e., partial information regarding the status of the variables

4. Since getting facts is monotonic, the exchange of facts can be performed in a distributed and chaotic way; it is confluent

Micro-step semantics

1. This semantics is obtained by taking the product of the micro-step automata attached to each statement; the product is by synchronizing compatible facts

2. This is modular: we can compose micro-step automata attached to sub-systems

3. Such micro-step automata synchronize by exchanging *facts,* i.e., partial information regarding the status of the variables

4. Since getting facts is monotonic, the exchange of facts can be performed in a distributed and chaotic way; it is confluent

5. Warning: this does not solve desynchronized distribution since this still requires knowing the scope of the reaction

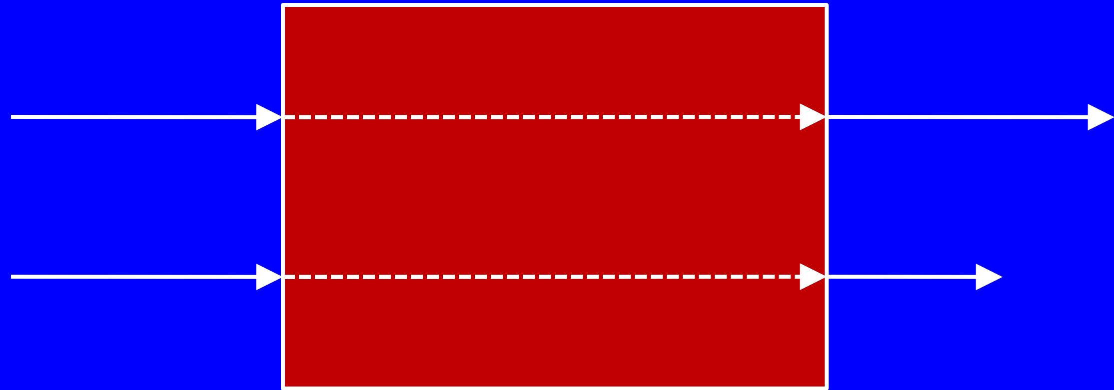
Contents

- Motivation
- Signal Model of Computation and Communication
- Key data structure for Signal compilation: clock-and-causality calculus
- Constructive Semantics
- Separate compilation and a notion of interface
- Deploying over distributed (possibly asynchronous) architectures
- Use in architecture modeling and analysis

Interface theory for separate compilation

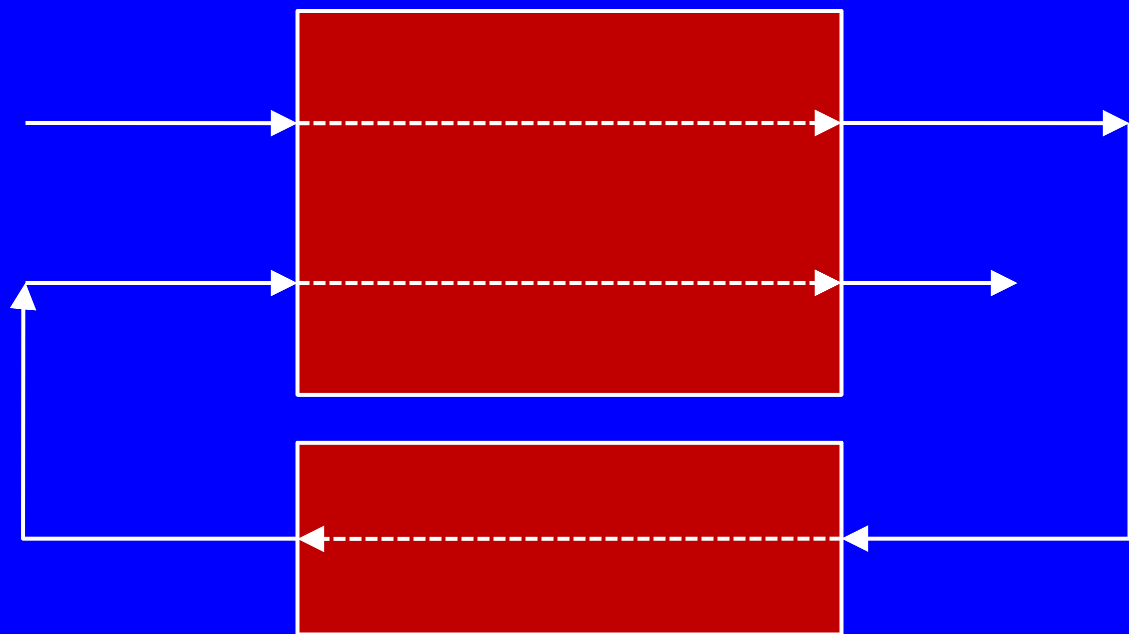
- **Problem:**
  - A synchronous program must be seen as a specification, as it has several possible implementations
  - Separate compilation consists in selecting one possible implementation in a modular way, with the intent of reusing the implementation in contexts that are not known in advance
  - As the Signal based example of the next slide will show, separate compilation must be handled with care
- **Proposed approach:**
  - Adopt the point of view of Interface Theories, where the Component and its Environment are seen as two players in a game
  - Define a *refinement* relation on such interfaces, from which rules for separate compilation will follow

The problem with separate compilation



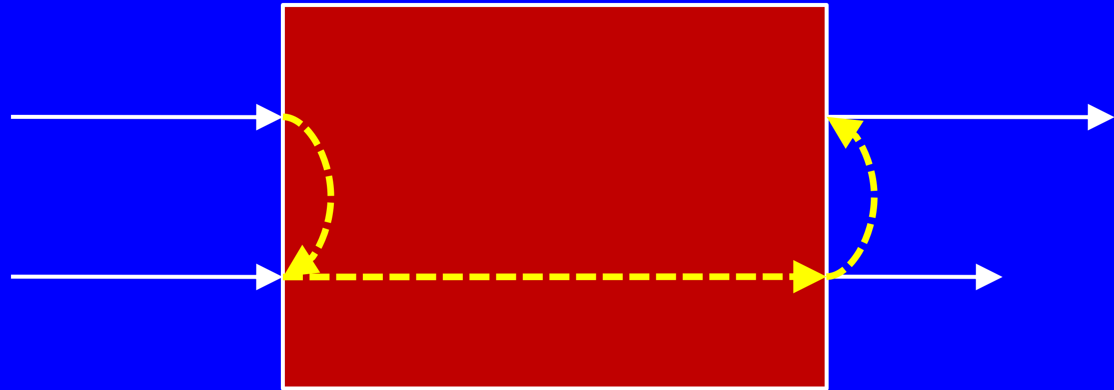
This is a synchronous program (specification)
instantaneous i/o dependencies are shown

The problem with separate compilation



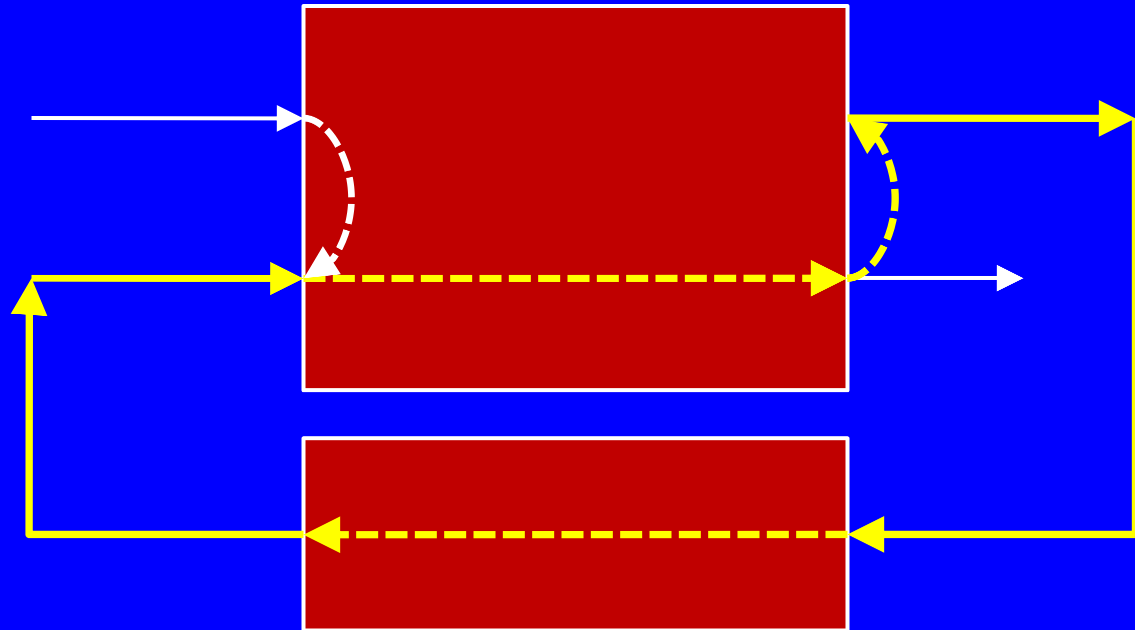
This specification can be safely composed with another program as shown

The problem with separate compilation



Here we show a compilation into sequential code that is compatible with the specification

The problem with separate compilation



Zero-delay circuit: Separate compilation can yield incompatible implementations

The problem with separate compilation

References for this graphical aspect of the problem:

[Raymond88]

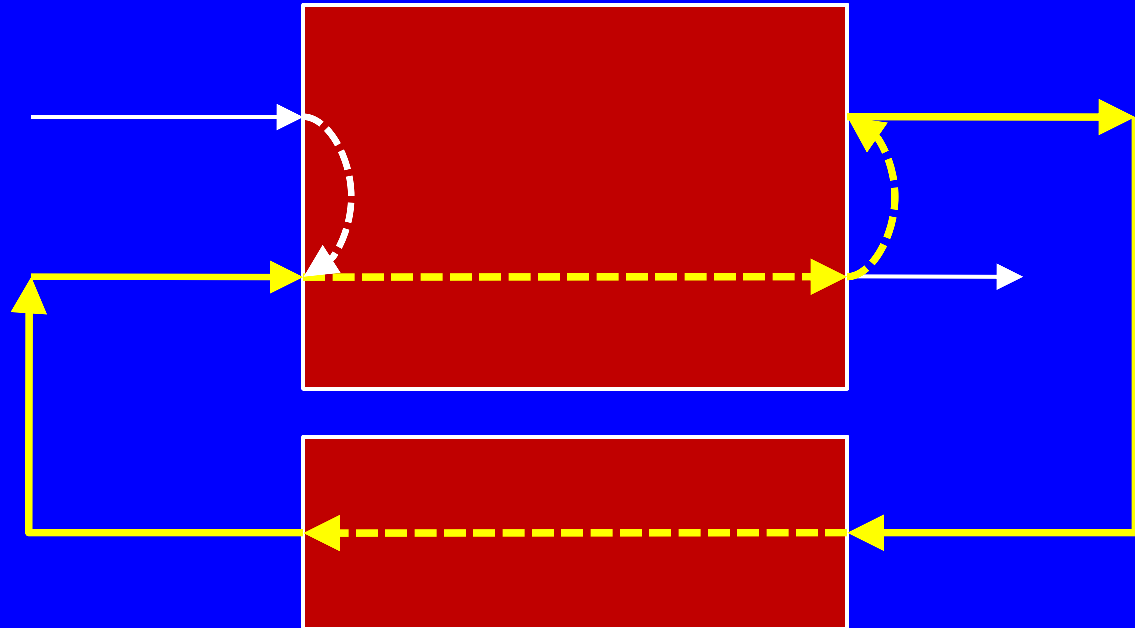
[Maffeis-LeGuernic94]

[Aubry-LeGuernic96]

[Lublinerman-

Tripakis08a,08b,09]

[Pouzet-Raymond09]



What about the general case of programs with multiple clocks?

Separate compilation can yield incompatible implementations

The problem with separate compilation as in Signal

Example:

```
( U ^= V ^= X ^= Y
| X ← U
| Y ← V )
```

The problem with separate compilation as in Signal

Example:

```
 ( U ^= V ^= X ^= Y
 | X ← U
 | Y ← V )
|
 ( U ^= Y
 | U ← Y )
```

yields

```
 ( U ^= V ^= X ^= Y
 | X ← U ← Y ← V
 )
```

which is deadlock-free

The problem with separate compilation as in Signal

Example:

```
( U ^= V ^= X ^= Y
| X ← U
| Y ← V )
|
( U ^= Y
| U ← Y )
```

yields

```
( U ^= V ^= X ^= Y
| X ← U ← Y ← V
)
```

which is deadlock-free

However, implementation

```
( U ^= V ^= X ^= Y
| X,Y ← U,V )
```

The problem with separate compilation as in Signal

Example:

```
( U ^= V ^= X ^= Y
| X ← U
| Y ← V )
|
( U ^= Y
| U ← Y )
```

yields

```
( U ^= V ^= X ^= Y
| X ← U ← Y ← V
)
```

which is deadlock-free

However, implementation

```
( U ^= V ^= X ^= Y
| X,Y ← U,V )
|
( U ^= Y
| U ← Y )
```

causes the deadlock

```
( U ^= Y
| U ←→ Y )
```

The problem with separate compilation as in Signal

Example:

References for this graphical aspect of the problem:

[Raymond88]

[Maffeis-LeGuernic94]

[Aubry-LeGuernic96]

[Lublinerman-

Tripakis08a,08b,09]

[Pouzet-Raymond09]

```
( U ^= V ^= X ^= Y
| X ← U
| Y ← V )
|
( U ^= Y
| U ← Y )
```

yields

```
( U ^= V ^= X ^= Y
| X ← U ← Y ← V
 )
```

which is deadlock-free

However, implementation

```
( U ^= V ^= X ^= Y
| X,Y ← U,V )
|
( U ^= Y
| U ← Y )
```

causes the deadlock

```
( U ^= Y
| U ←→ Y )
```

The problem with separate compilation as in Signal

Example:

```
( U ^= V ^= X ^= Y ^= Z
| X ← U when Z>0
| X ← V whennot Z>0
| Z ← V
| Y ← V )
```

If `Z` is local, `U,V,X,Y` are visible, we can hide `Z`:

```
( U ^= V ^= X ^= Y
| X ← U
| X ← V
| Y ← V )
```

Every implementation of the original program is also a valid implementation of this more abstract interface

Interface theory for separate compilation

- **Problem:**
  - A synchronous program must be seen as a specification, as it has several possible implementations
  - Separate compilation consists in selecting one possible implementation in a modular way, with the intent of reusing the implementation in contexts that are not known in advance

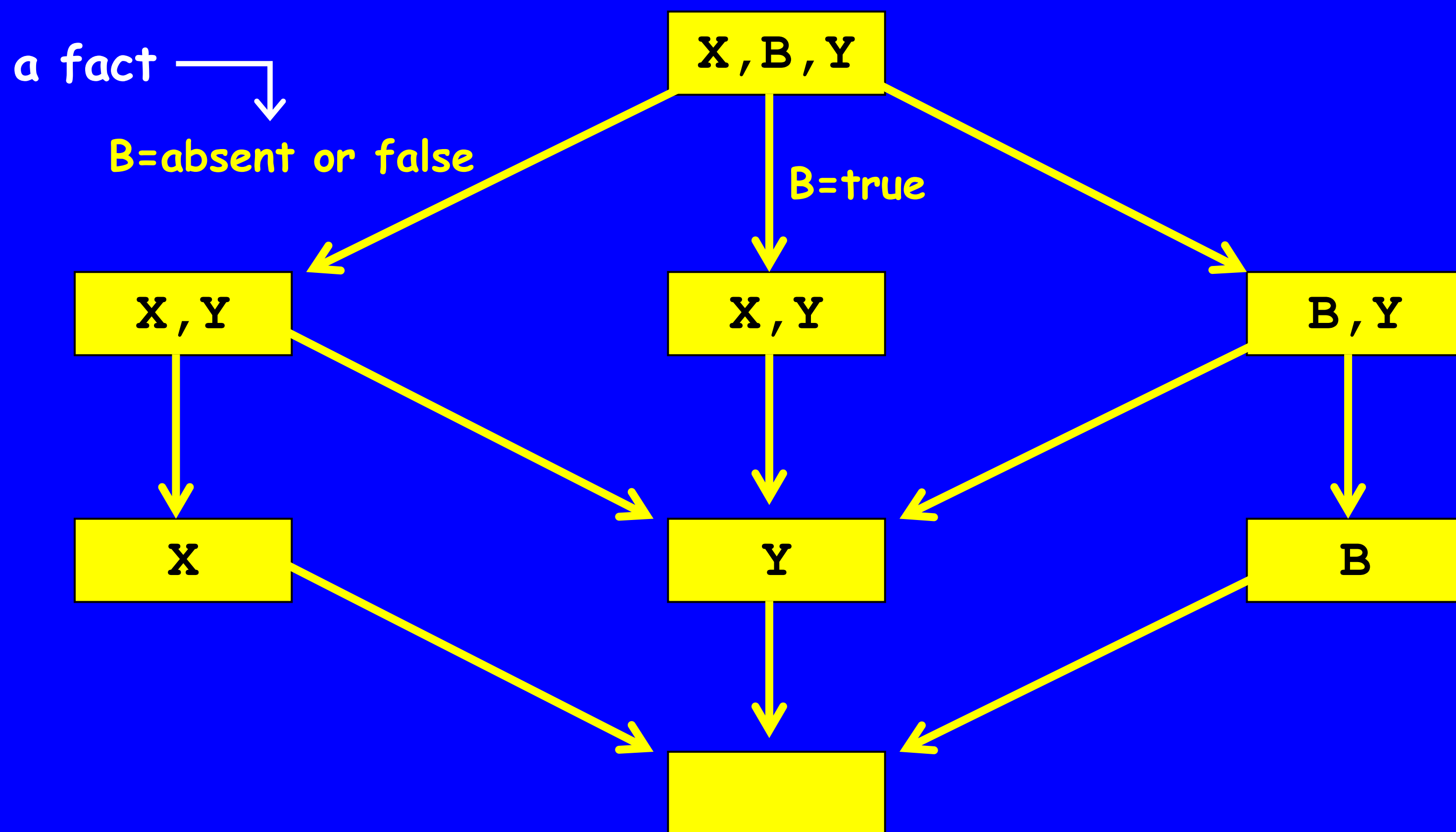
- **Proposed approach:**
  - Adopt the point of view of Interface Theories, where the Component and its Environment are seen as two players in a game
  - Define a *refinement* relation on such interfaces, from which rules for separate compilation will follow

$X \rightarrow_B Y$: scheduling constraint

B \ X	⊥	absent	any
⊥	⊥	⊥	⊥
absent			
false			
true	⊥		
The table specifies the constraints on Y			

Constraint in the Scott domain expressing that Y cannot be evaluated before X when predicate B evaluates to true

$X \to_B Y$: scheduling constraint



Micro-step automaton expressing that
Y cannot be evaluated before X
when predicate B evaluates to true

The essence of constructive semantics

Program

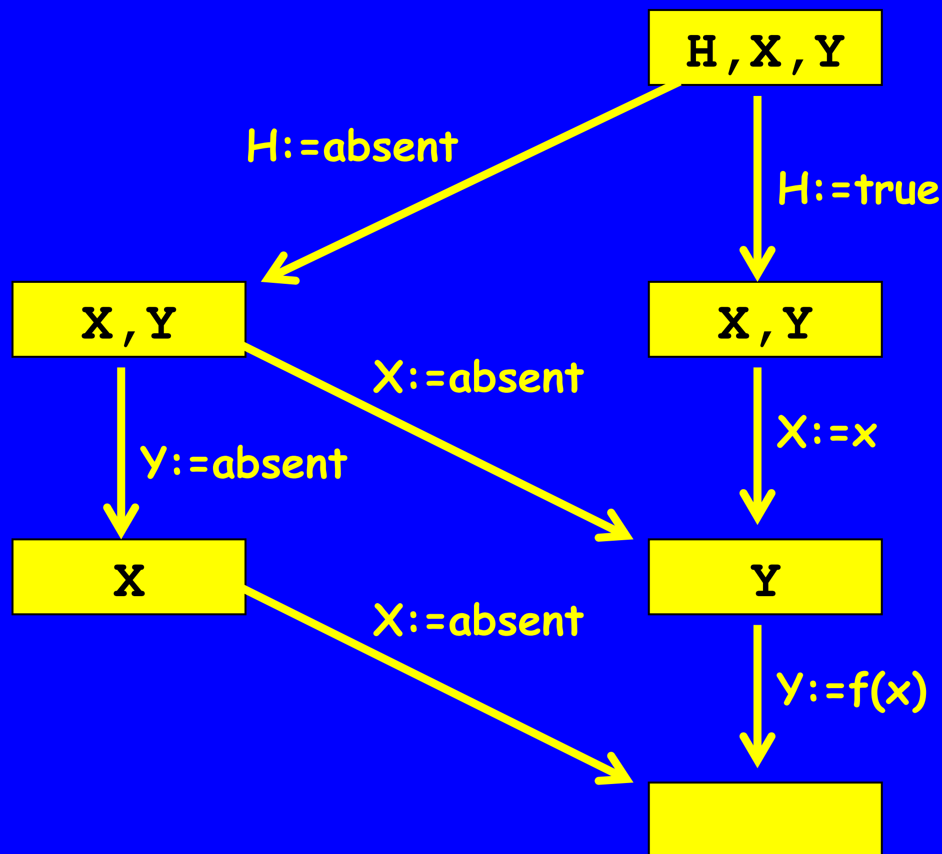
- Each program fragment generates a causality constraint inducing a scheduling constraint

Scheduling constraints

- Combine them with the actual evaluation of variables
- Each variable is evaluated once
- Take product of micro-step automata

Yields micro-step semantics for the program

The essence of constructive semantics

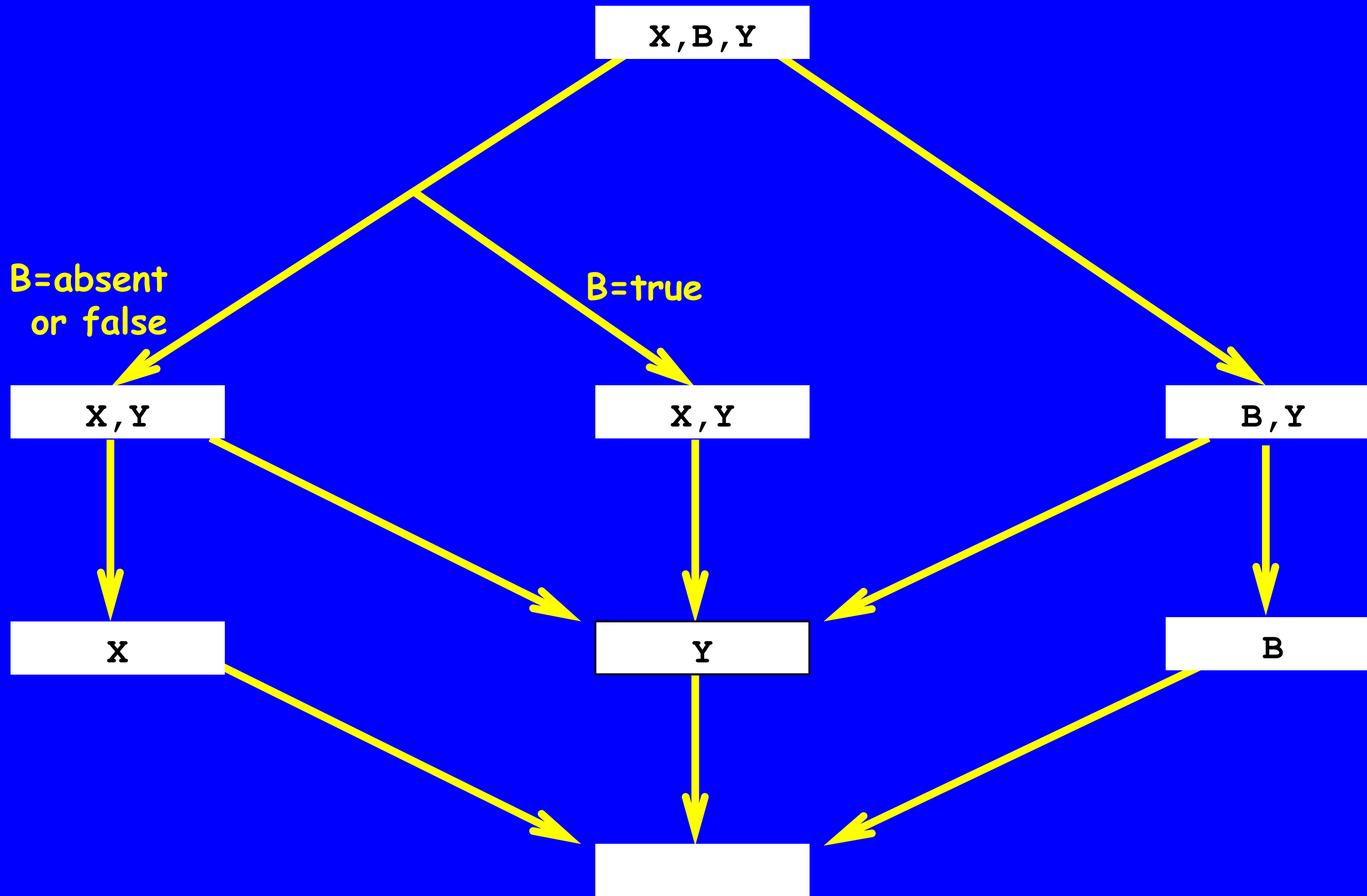


$Y{:=}f(X) \parallel H \rightarrow X \parallel X \rightarrow_H Y$

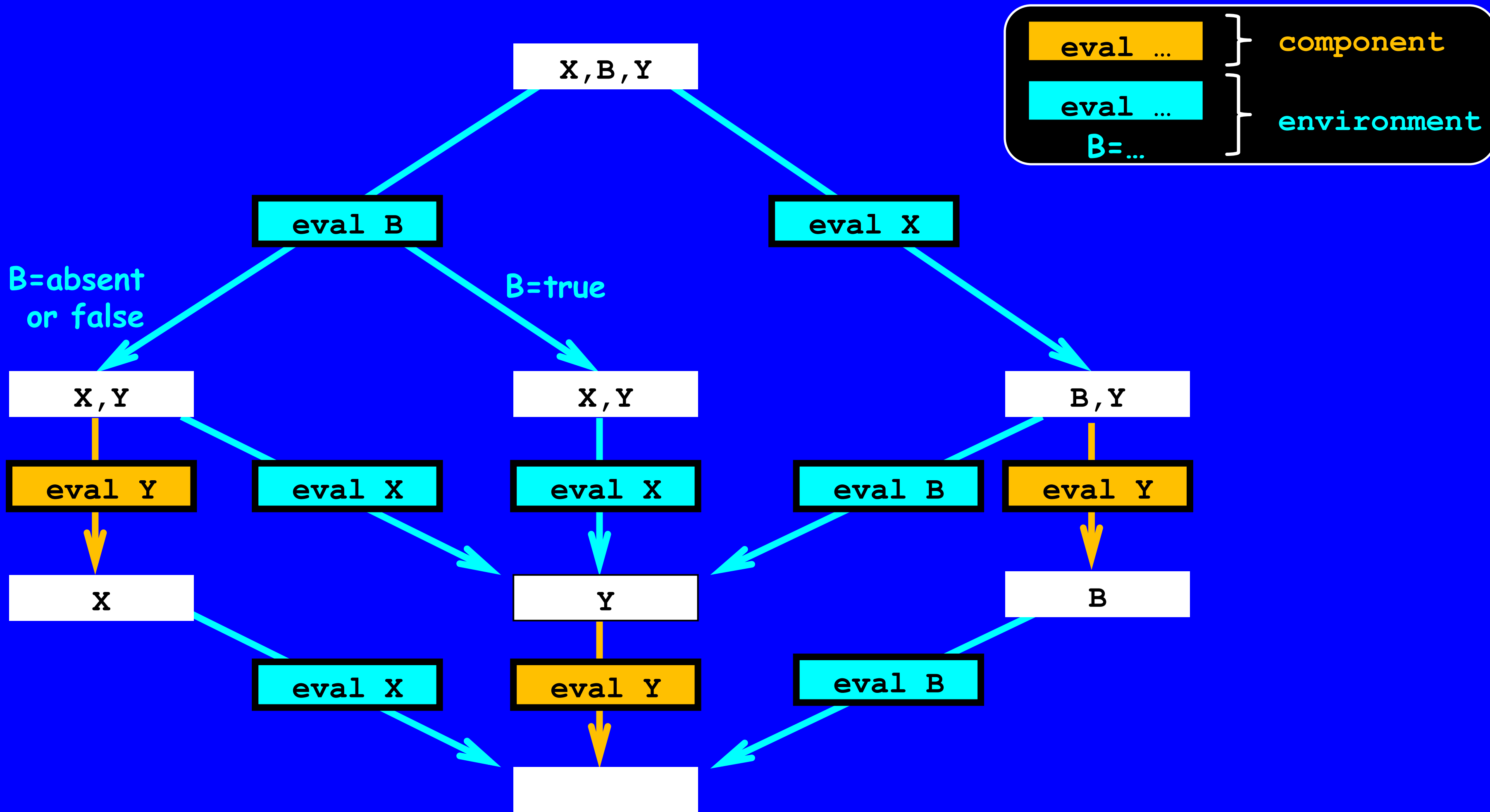
as a micro-step automaton

Back to the scheduling constraint

$X \rightarrow_B Y$

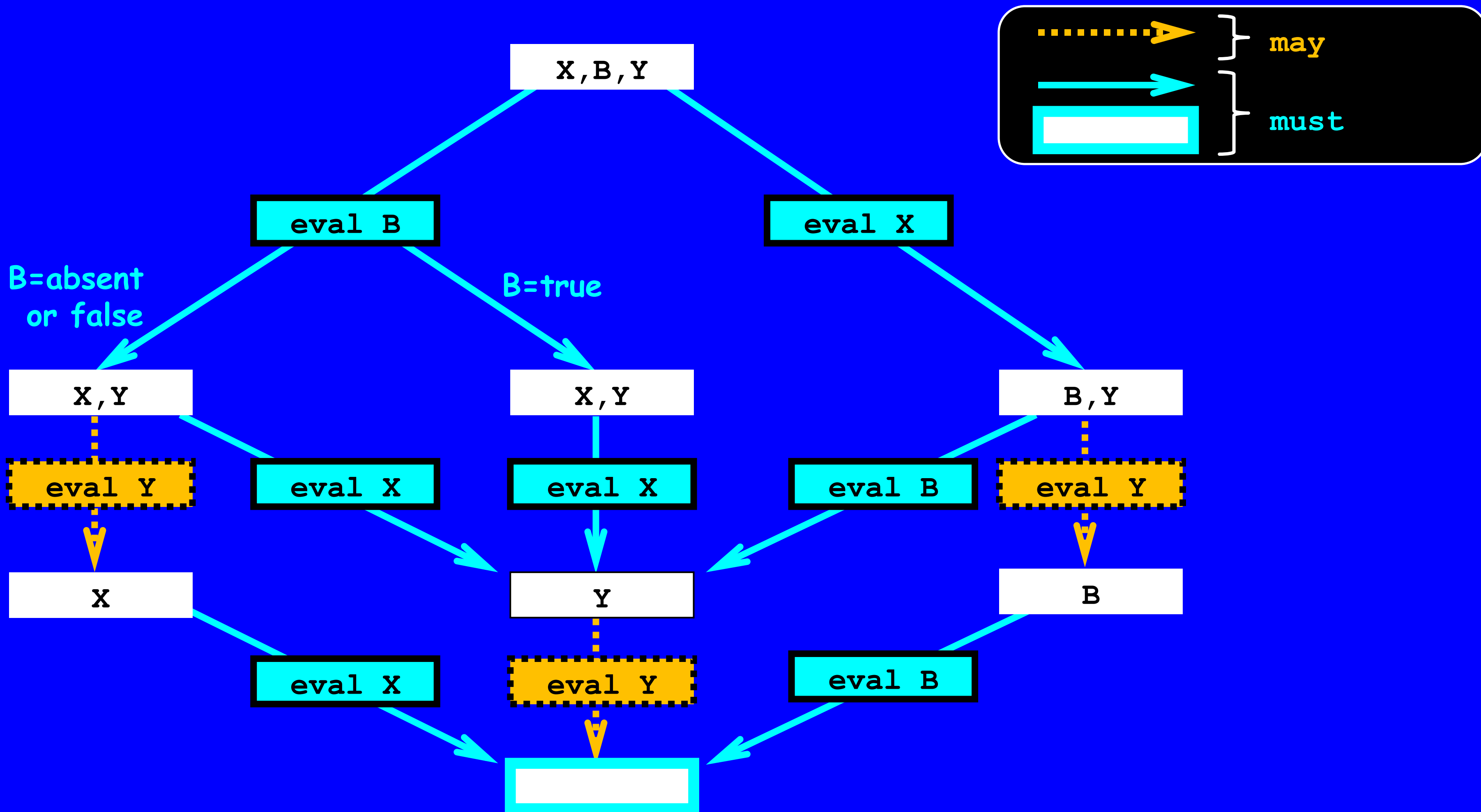


A game theoretic approach for $X \rightarrow_B Y$



Suppose:
Component has control over when to evaluate Y
Environment has control over when to evaluate B,X
No one has control over which value is assigned

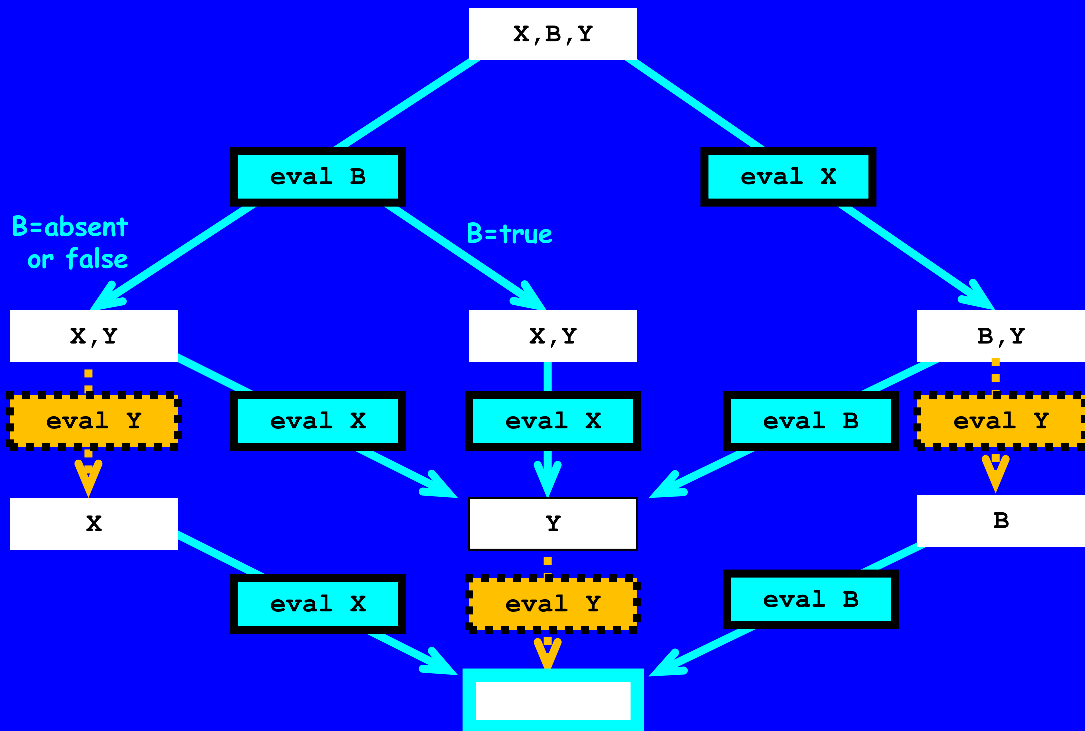
A game theoretic approach: modalities



Component has control over when to evaluate Y
Environment has control over when to evaluate B,X
No one has control over which value is assigned

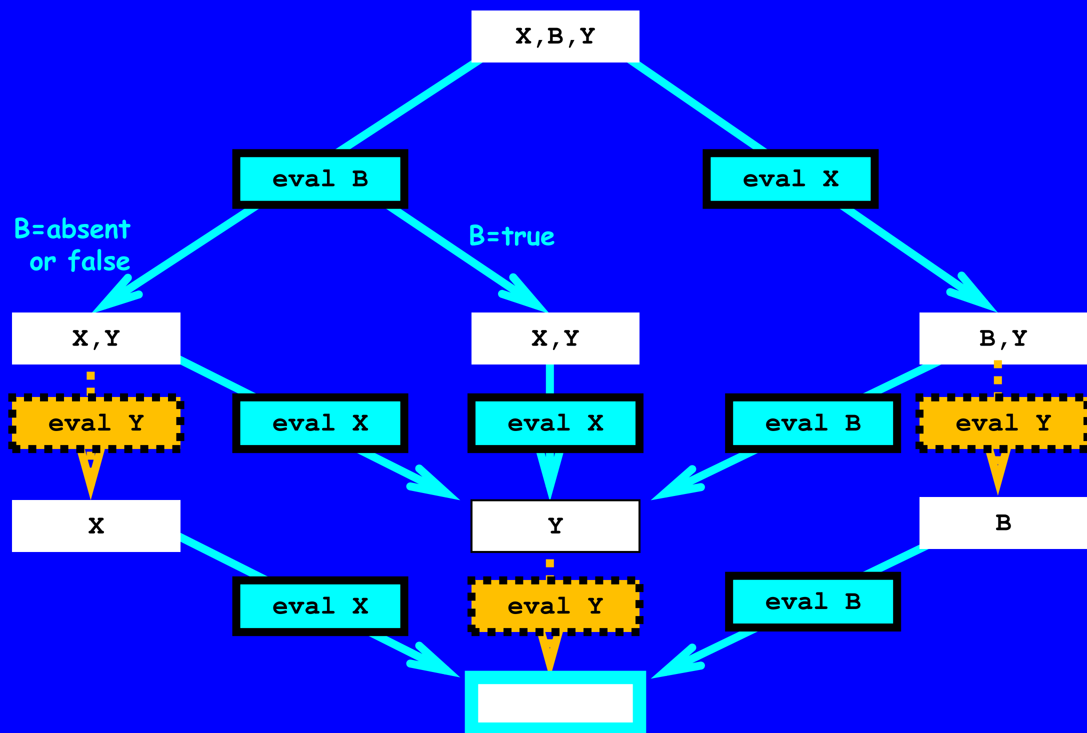
A game theoretic approach: using Marked Modal Interfaces

[Caillaud & Raclet] Marked Modal Interfaces
Transitions labeled *may* / *must*
Some states labeled *must*



- MMI yields an interface theory
- Refinement:
  removing *may*
  adding *must*
- Parallel composition supporting separate implementability
- Residuation
- Separate compil
  ≡
  Computing minimal implementations

Separate compilation followed by KPN execution



- Here we have used micro-step *automata*
- Why not using *nets* ?
- Marked Modal Net Interfaces: MMNI
- Their implementations are safe nets…
- … which execute as KPN
- Facts related to absence must be explicitly transmitted: additional signaling
- Could this simplify distribution?

Contents

- Motivation
- Signal Model of Computation and Communication
- Key data structure for Signal compilation: clock-and-causality calculus
- Constructive Semantics
- Separate compilation and a notion of interface
- Deploying over distributed (possibly asynchronous) architectures
- Use in architecture modeling and analysis

Comparison with other Models of Computation and Communication and Related

- Synchronous paradigms
- Kahn networks, token based Boolean dataflow nets (Ed Lee)
- GALS (Globally Asynchronous Locally Synchronous)
- LTTA (Loosely Time Triggered Architectures)

Synchronous paradigms

- Signal Polychronous MoCC subsumes:
  - Synchronous hardware
  - Synchronous DataFlow (Ed Lee)
  - (hierarchical) state machines
  - Synchronous software
- Non interacting Signal modules progress asynchronously (no relation between their activation clocks)
- Interactions between modules synthesize the necessary synchronizations

Comparison with other Models of Computation and Communication and Related

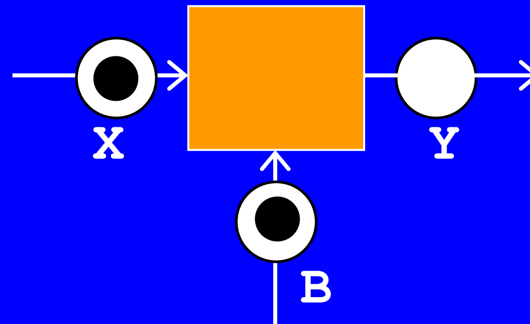
- Synchronous paradigms
- Kahn networks, token based Boolean dataflow nets (Ed Lee)
- GALS (Globally Asynchronous Locally Synchronous)
- LTTA (Loosely Time Triggered Architectures)

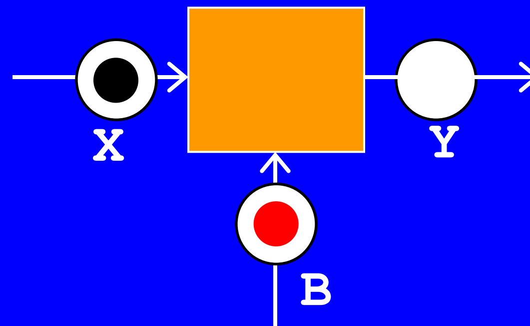
Token based boolean dataflow à la Ed Lee (Kahnification)

```
   Y := X when B
|  X ^= B
```

X and B

have same clock



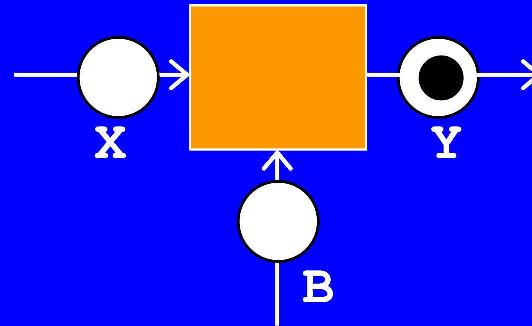


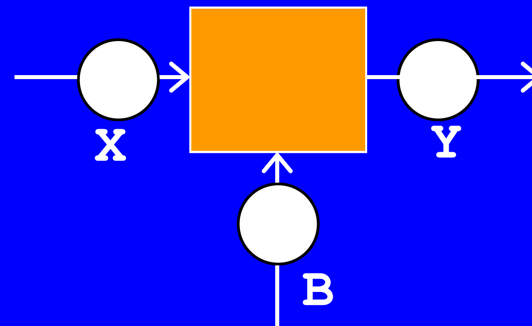
The X-token passes or not depending on whether the boolean token is true (black) or false (red)

Token based boolean dataflow à la Ed Lee (Kahnification)

```
  Y := X when B
| X ^= B
```

X and B have same clock

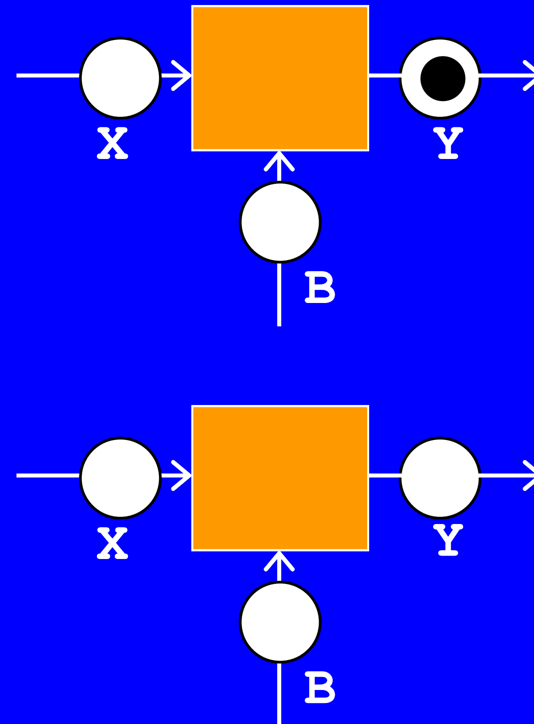




The X-token passes or not depending on whether the boolean token is true (black) or false (red)

Token based boolean dataflow à la Ed Lee (Kahnification)

```
  Y := X when B
| X ^= B
```



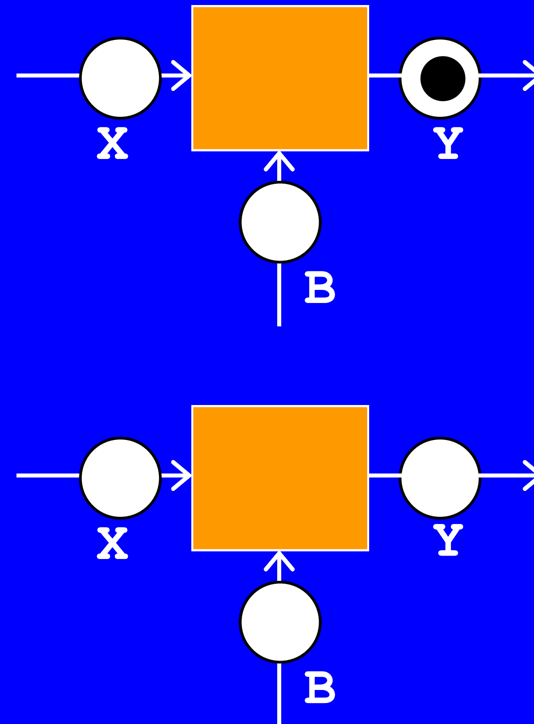
The X-token passes or not depending on whether the boolean token is true (black) or false (red)

That `X` and `B` possess identical clocks is part of the Signal specification

Key difference: the pattern of input tokens is fixed (always a token on `X` and one on `B` for this actor, which makes it robust against asynchrony)

Token based boolean dataflow à la Ed Lee (Kahnification)

```
  Y  := X when B
| X ^= B
```



The X-token passes or not depending on whether the boolean token is true (black) or false (red)

Token based dataflow is isomorphic to synchronous software, i.e., ⊆ Signal MoCC

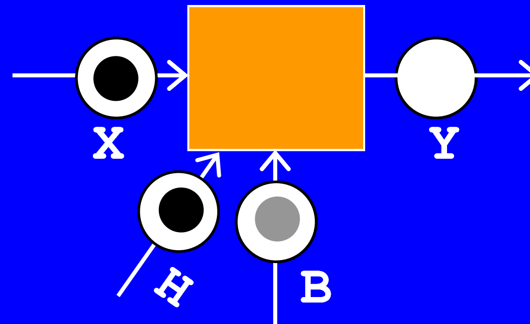
That `X` and `B` possess identical clocks is part of the Signal specification

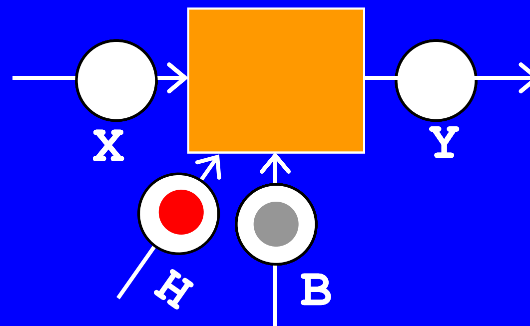
Key difference: the pattern of input tokens is fixed (always a token on `X` and one on `B` for this actor, which makes it robust against asynchrony)

Token based boolean dataflow à la Ed Lee (Kahnification)

```
  Y := X when B
| X ^< B
```

X and B have clocks subject to inclusion

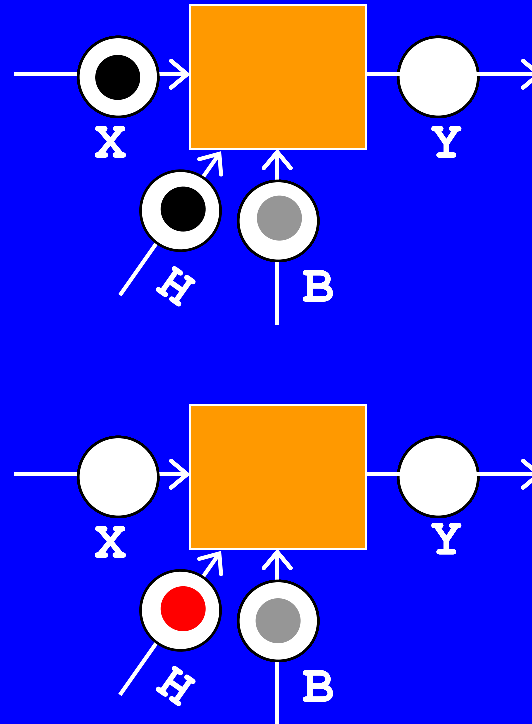




The B-token is always consumed; <u>a H-boolean token is created</u> to indicate whether an X-token is consumed or not

Token based boolean dataflow à la Ed Lee (Kahnification)

```
  Y := X when B
| X ^< B
```



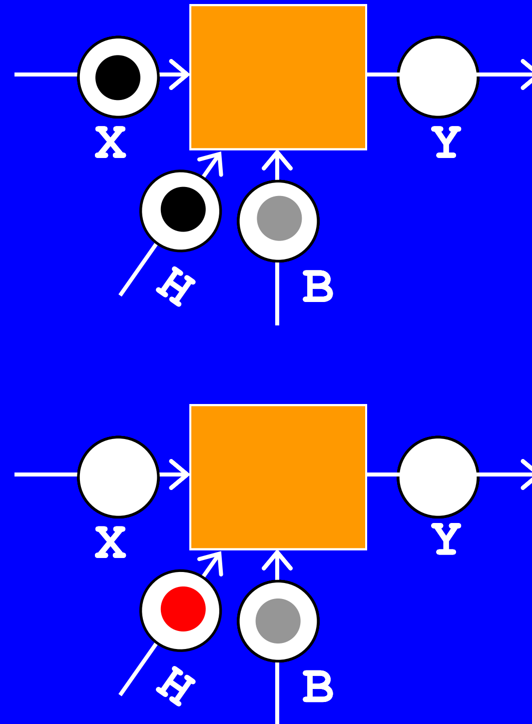
The B-token is always consumed; <u>a H-boolean token is created</u> to indicate whether an X-token is consumed or not

With additional signaling, Signal modules can be mapped to single buffer token based dataflow

Token based boolean dataflow à la Ed Lee (Kahnification)

```
  Y  := X when B
| X  ^< B
```

With additional signaling, Signal modules can be mapped to single buffer token based dataflow



The B-token is always consumed; <u>a H-boolean token is created</u> to indicate whether an X-token is consumed or not

Translation can be structural, but this causes overhead in signaling ⇒ generate automatically minimal signaling

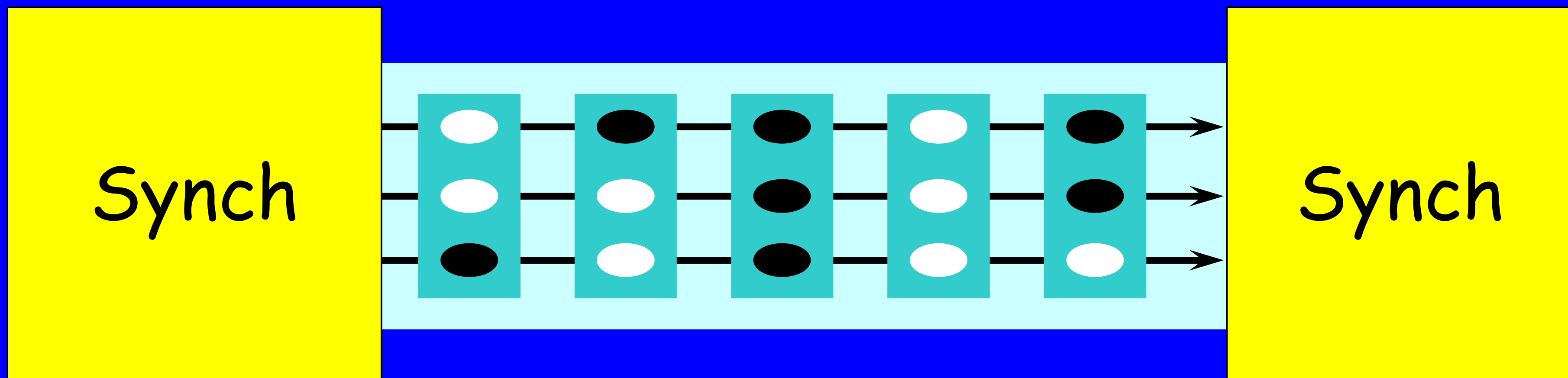
Desynchronisation and mapping Polychrony to GALS

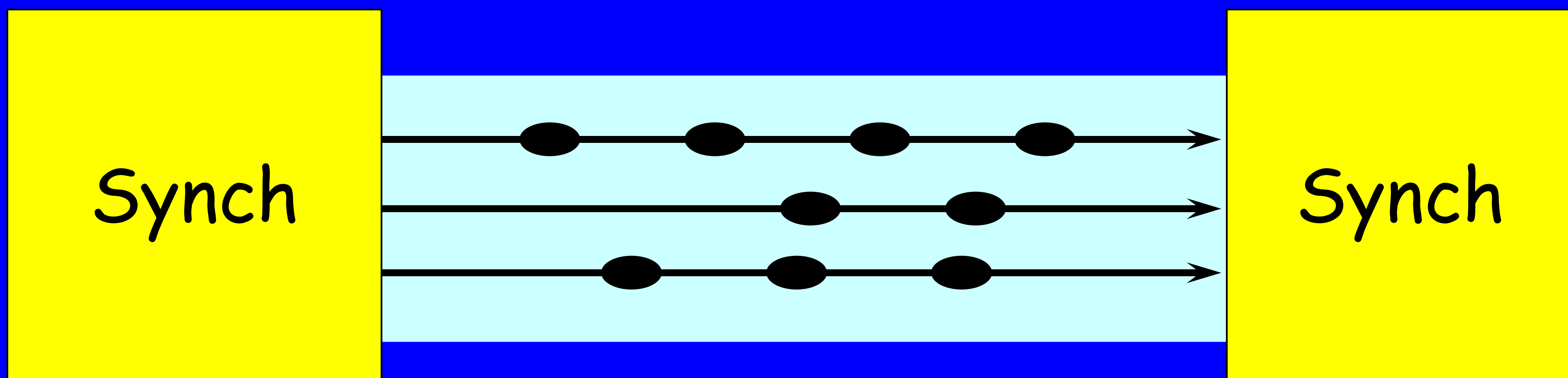
- The above discussion provides hints for automatic correct-by-construction deployment over distributed GALS architectures
- Can also be seen as a way to reduce the number of handshakes in asynchronous hardware
- A large body of results has been developed by
  [Caillaud & Girault 1996]
  [Caillaud, Benveniste, LeGuernic 2000]
  [Caillaud, Potop Butucaru 2005, Potop Butucaru & al 2008]
  to algebraically characterize those synchronous programs equivalent to token based data flow nets
- Synthesis methods follow accordingly

Comparison with other Models of Computation and Communication and Related

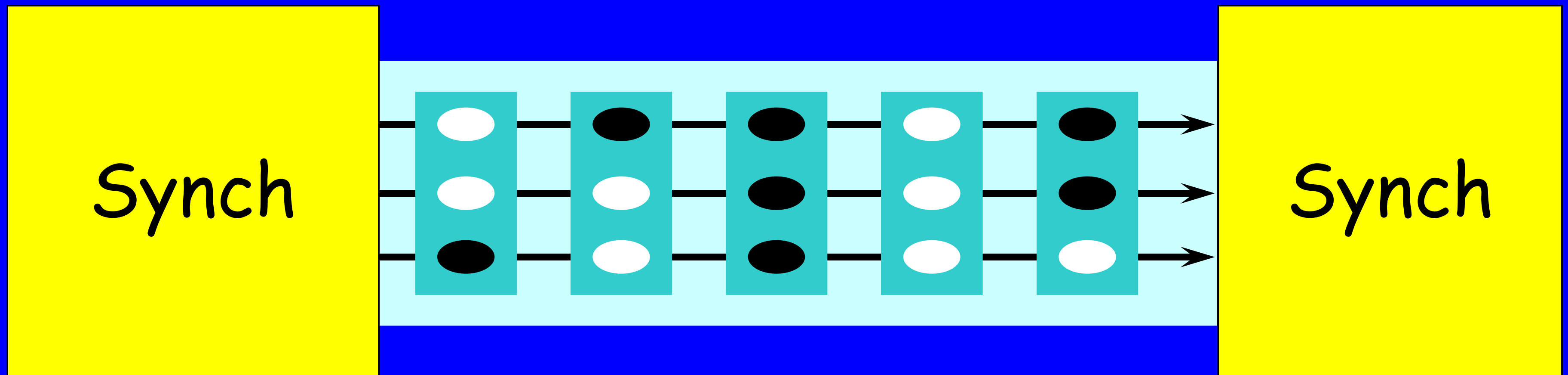
- Synchronous paradigms
- Kahn networks, token based Boolean dataflow nets (Ed Lee)
- GALS (Globally Asynchronous Locally Synchronous)
- LTTA (Loosely Time Triggered Architectures)

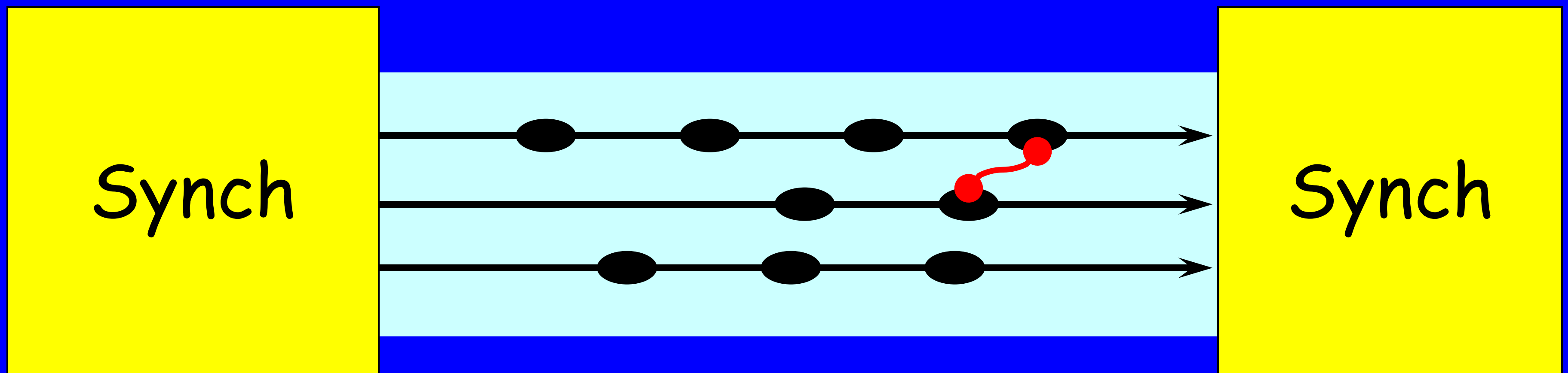
Desynchronisation and mapping Polychrony to GALS: the issues (0)



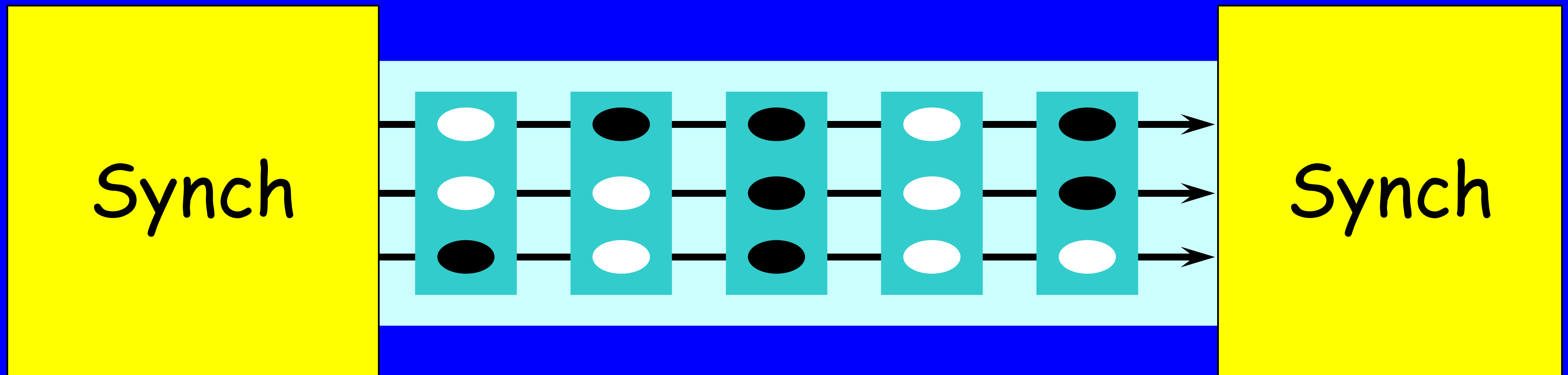


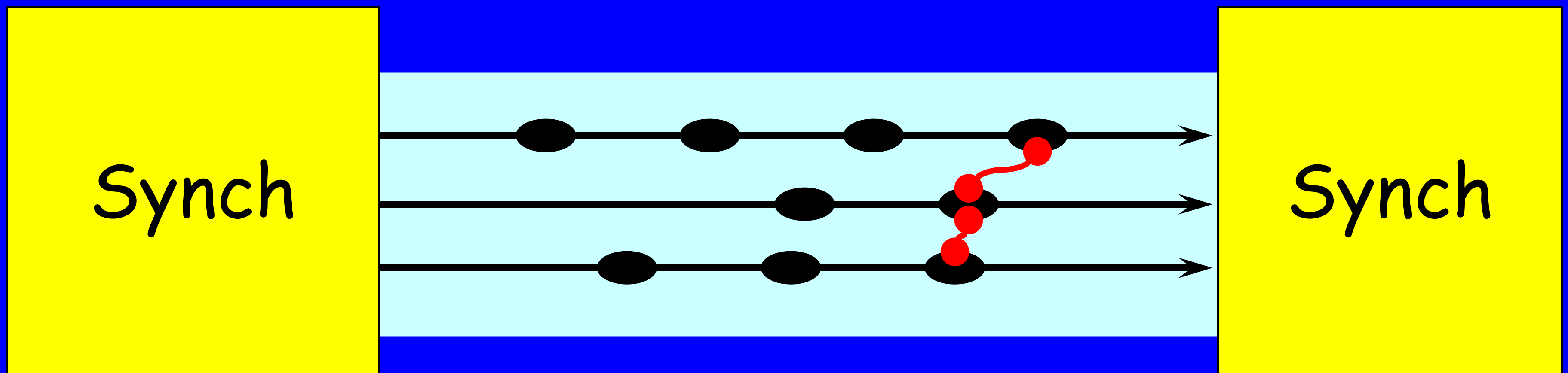
Desynchronisation and mapping Polychrony to GALS: the issues (0)





Desynchronisation and mapping Polychrony to GALS: the issues (0)





Desynchronisation and mapping Polychrony to GALS: the issues (1)

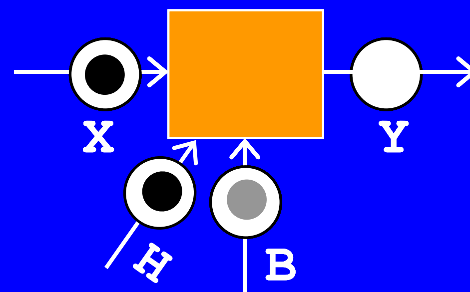
- Local to a synchronous component communicating asynchronously with its environment:
  - reconstruct the successive synchronous reactions without external triggers
- To preserve functional semantics of communication, from synch to asynch:
  - The set of synch runs should be robust against desynchronisation
  - Interblocking properties should not be relaxed by desynchronisation
- Solution: [Potop Caillaud 2005, Potop 2008]

Desynchronisation and mapping Polychrony to GALS: the issues (2)

- Local to a synchronous component communicating asynchronously with its environment:
  - Being able to reconstruct the successive synchronous reactions without external triggers: endochrony (no decision based on absence)

NO

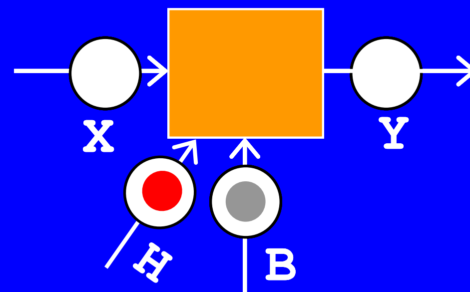
```
  Y := X when B
| X ^< B
```



YES

YES

```
  Y := X when B
| X ^= when H
| H ^= B
```



Desynchronisation and mapping Polychrony to GALS: the issues (3)

To preserve functional semantics of communication, from synch to asynch:

- The set of synch runs should be robust against desynchronisation : **weak endochrony**
  P endochronous ⇒ P weakly endochronous

- **P_1 , P_2 weakly endochronous ⇒**
  $P_1 \mid P_2$ weakly endochronous

- ~~P_1 , P_2 endochronous ⇒~~
  ~~$P_1 \mid P_2$ endochronous~~

Desynchronisation and mapping Polychrony to GALS: the issues (4)

To preserve functional semantics of communication, from synch to asynch:

- Interblocking properties should not be relaxed by desynchronisation (otherwise desynchronisation may remove deadlocks): **weak isochrony**
- Sufficient condition: synchronous composition does not deadlock

Desynchronisation and mapping Polychrony to GALS: the issues (4)

To preserve functional semantics of communication, from synch to asynch:

- Interblocking properties should not be relaxed by desynchronisation (otherwise desynchronisation may remove deadlocks): **weak isochrony**
- Sufficient condition: synchronous composition does not deadlock

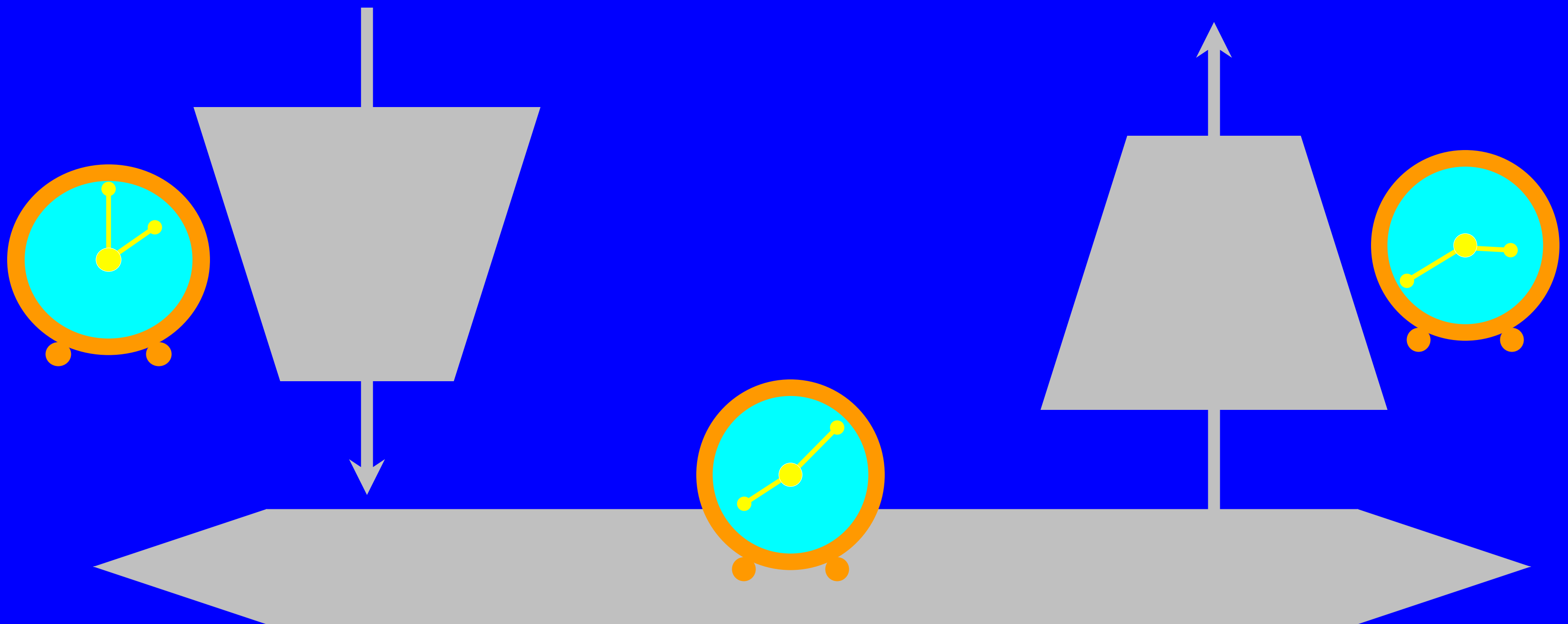
A comprehensive set of results has been obtained by [Caillaud and Potop 05, Potop 2008]; earlier results by Benveniste, Caillaud and le Guernic; also methods and tools by Girault

Comparison with other Models of Computation and Communication and Related

- Synchronous paradigms
- Kahn networks, token based Boolean dataflow nets (Ed Lee)
- GALS (Globally Asynchronous Locally Synchronous)
- LTTA (Loosely Time Triggered Architectures)

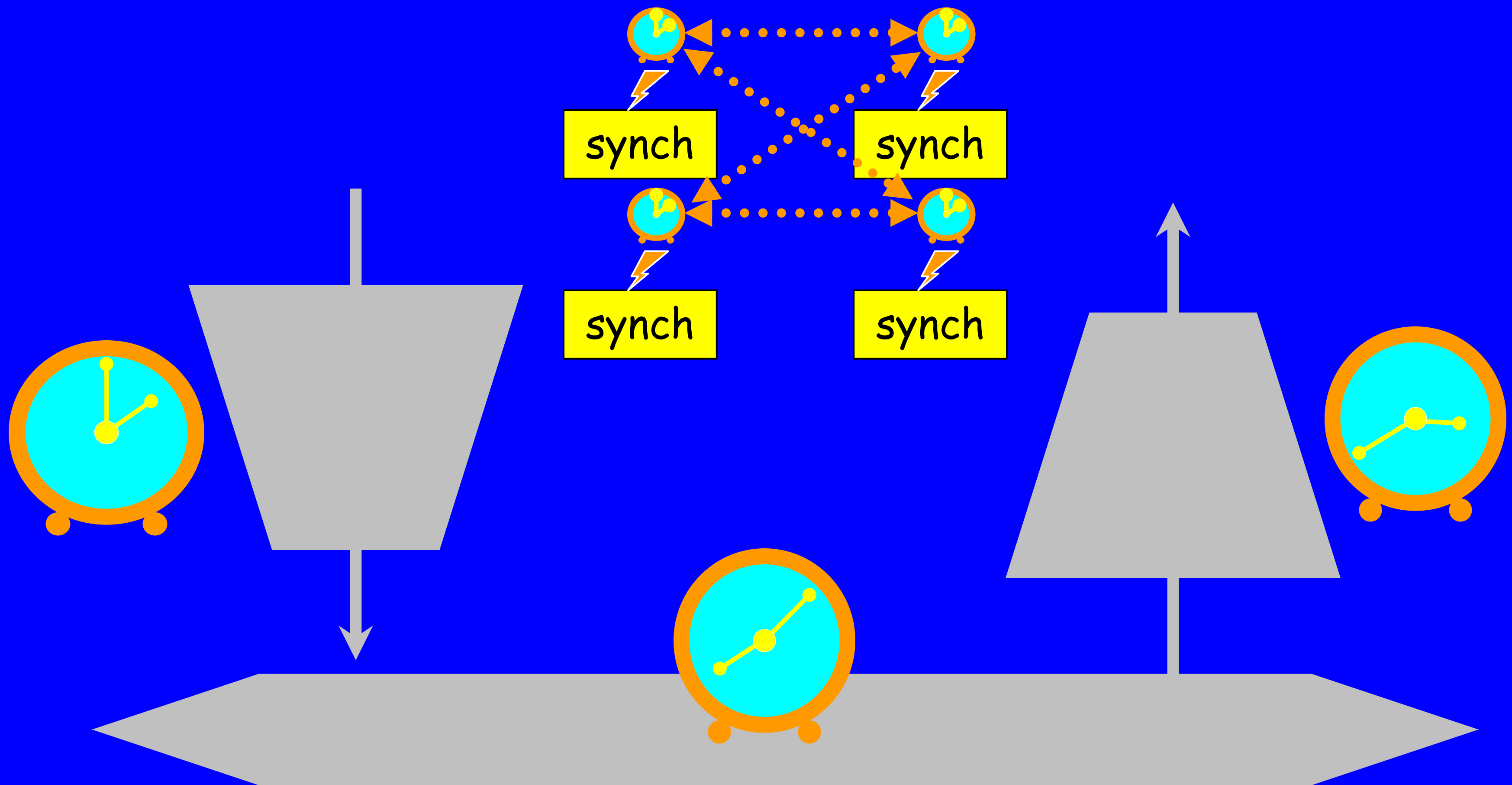
Loosely Time-Triggered Architectures (LTTA) [Benveniste, Caspi, Sangiovanni-Vincentelli & more]

writing/bus/reading are periodic but with non synchronized triggers



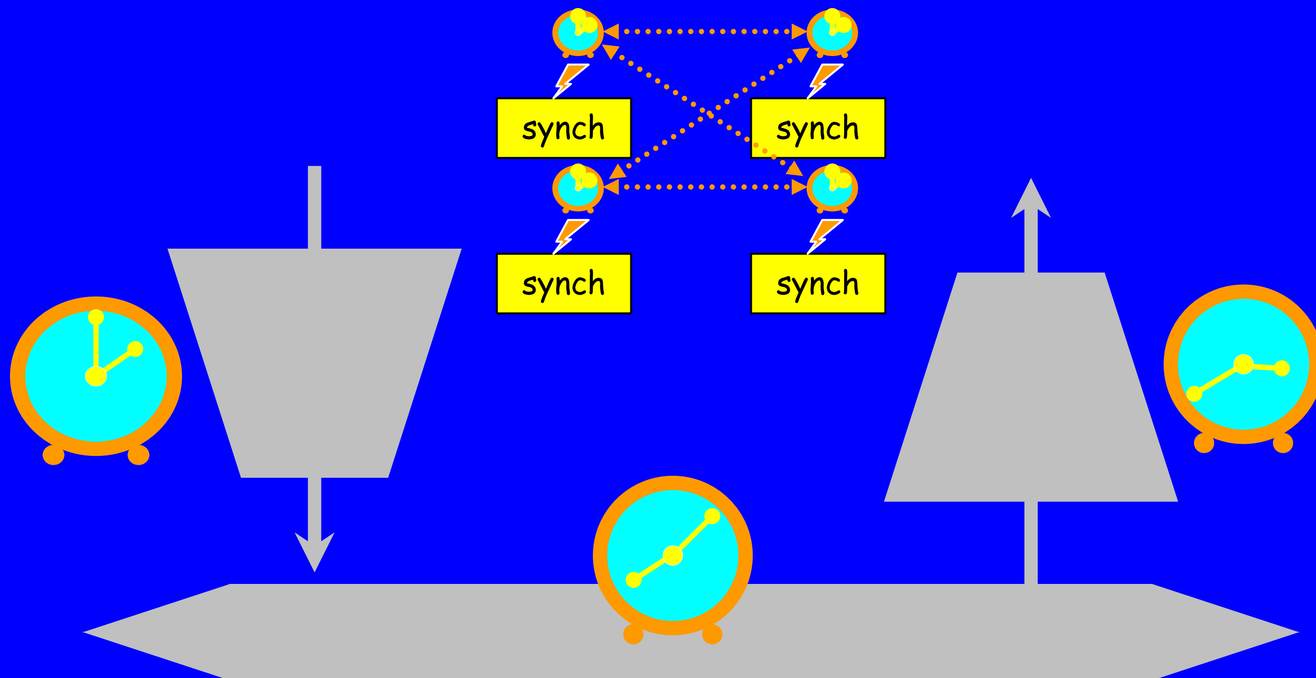
Loosely Time-Triggered Architectures (LTTA) [Benveniste, Caspi, Sangiovanni-Vincentelli & more]

writing/bus/reading are periodic but with non synchronized triggers



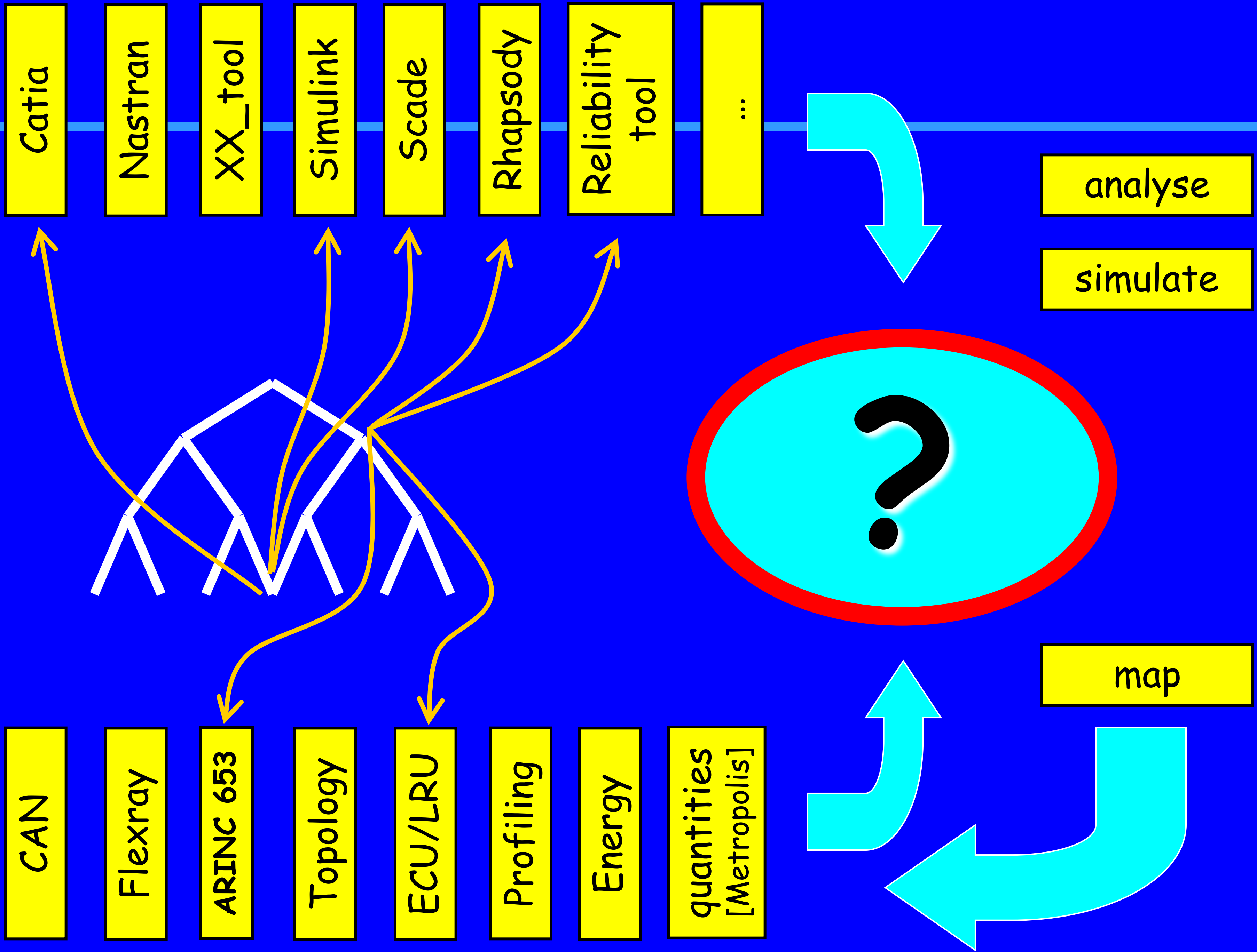
Loosely Time-Triggered Architectures (LTTA) [Benveniste, Caspi, Sangiovanni-Vincentelli & more]

- Deploying on LTTA reduces to:
  1. Upgrading basic LTTA to a GALS infrastructure with bounded inter-clock jitter [Emsoft07]
  2. Using GALS deployment techniques



Contents

- Motivation
- Signal Model of Computation and Communication
- Key data structure for Signal compilation: clock-and-causality calculus
- Constructive Semantics
- Separate compilation and a notion of interface
- Deploying over distributed (possibly asynchronous) architectures
- Use in architecture modeling and analysis



Catia
Nastran
XX_tool
Simulink
Scade
Rhapsody
Reliability tool
...
analyse
simulate
?
map
CAN
Flexray
ARINC 653
Topology
ECU/LRU
Profiling
Energy
quantities [Metropolis]

Enriching Signal models with nonfunctional characteristics

```
(
   (  B := (ZX ≤ 0)
   | IN ^= when B = [B]
   |  H ^= B ^= X ^= ZX )
|
   (  X   ←   IN when B
   |  X   ←   ZX when not B
   |  B   ←  (H,ZX)
   | [B]  ←   B
   | ZX   ←   H
   | IN   ←  [B] )
)
```

Synchro + scheduling

(recall)

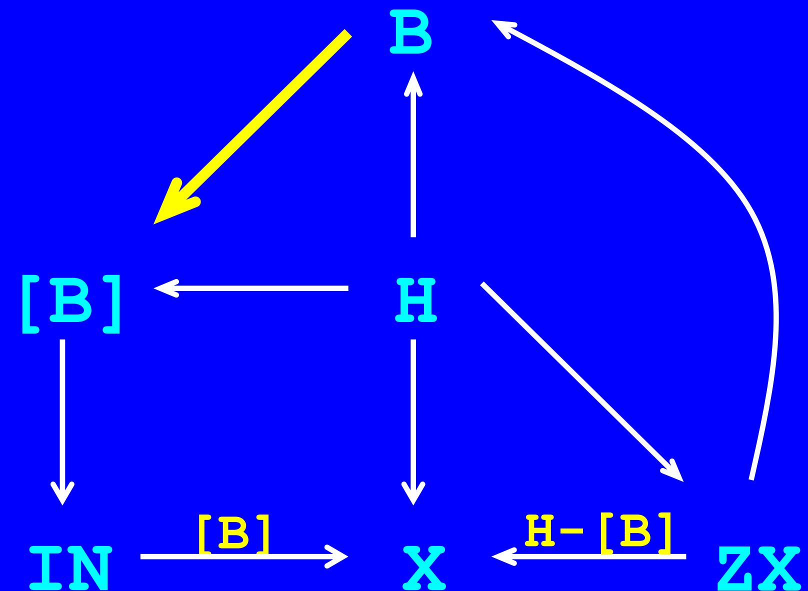
B
[B]
H
IN
[B]
X
H-[B]
ZX

Enriching Signal models with nonfunctional characteristics

```
(
   (  B := (ZX ≤ 0)
   | IN ^= when B = [B]
   |  H ^= B ^= X ^= ZX )
|
     X  ←  IN when B
   | X  ←  ZX when not B
   | B  ←  (H,ZX)
   | [B] ←  B
   | ZX ←  H
   | IN ←  [B] )
)
```

Synchro + scheduling

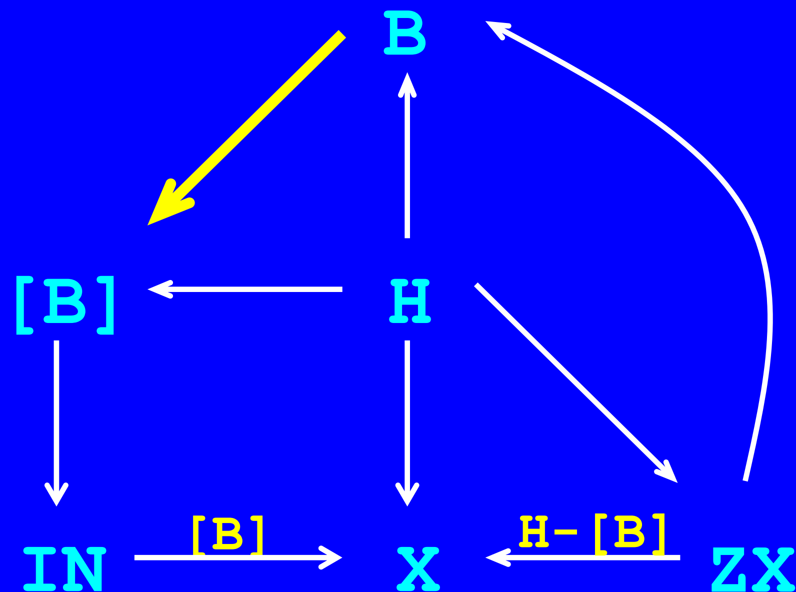
Focus on the two selected statements



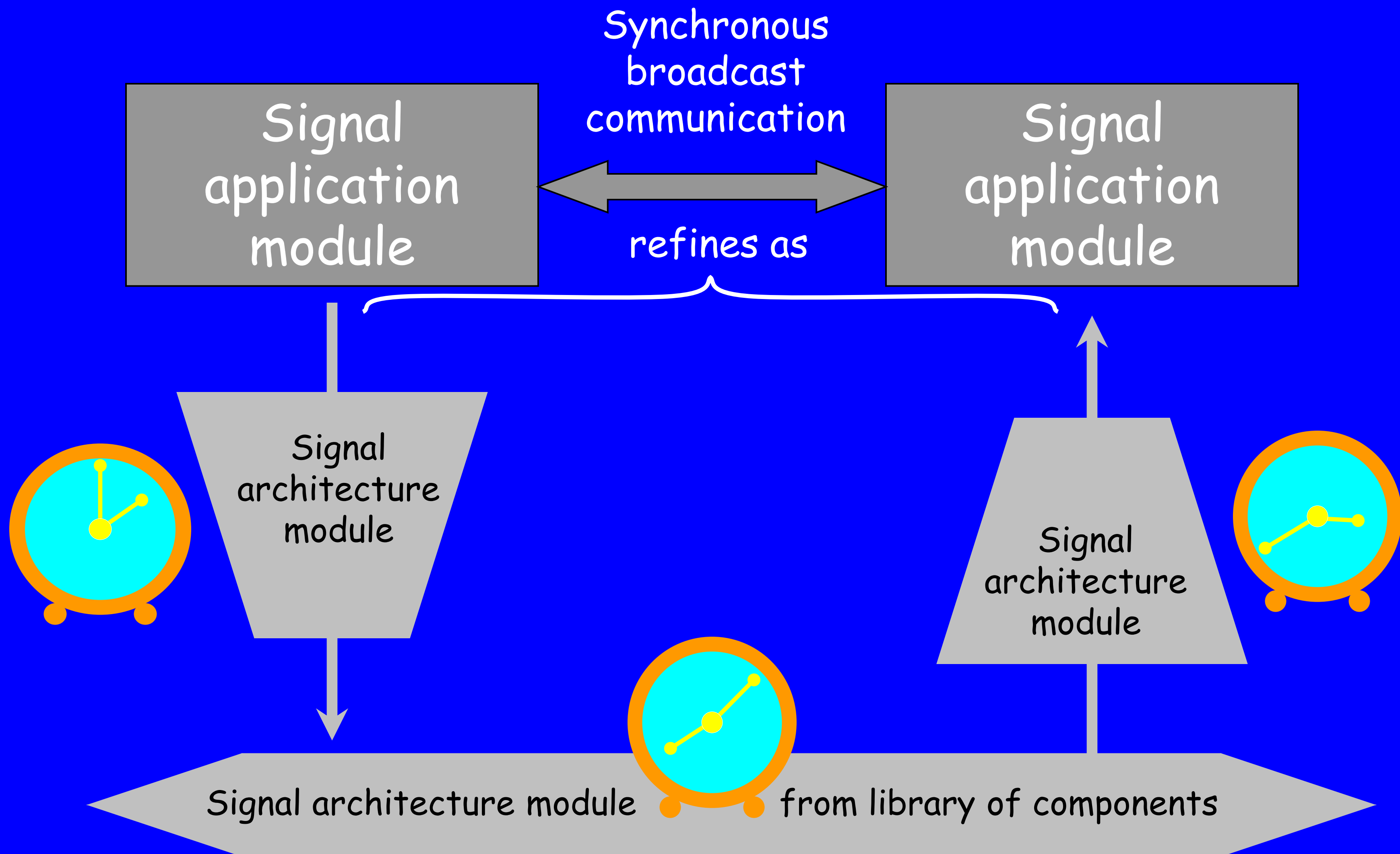
Enriching Signal models with nonfunctional characteristics

```
(
  (  B := (ZX ≤ 0)
  | IN ^= when B = [B]
  |  H ^= B ^= X ^= ZX )
|
     X   ←   IN when B
  |  δ_X  =   (δ_IN +δ′ ) when B
  |  X   ←   ZX when not B
  |  B   ←  (H,ZX)
  |  δ_B  = max(δ_H,δ_ZΞ)
  | [B]  ←   B
  | ZX   ←   H
  | IN   ←  [B] )
)
```

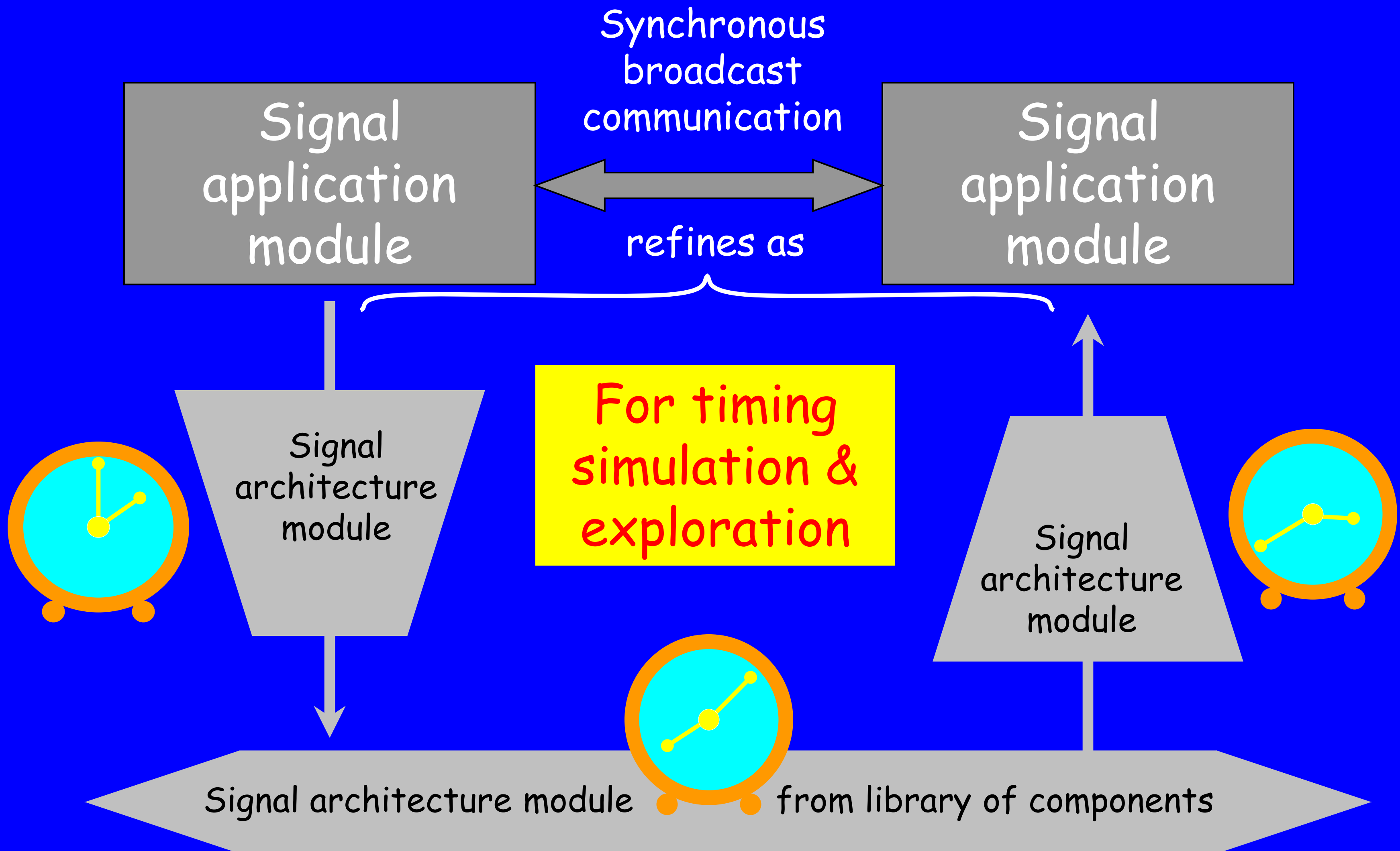
Enriched with timing evaluation; can be generated automatically, for all statements



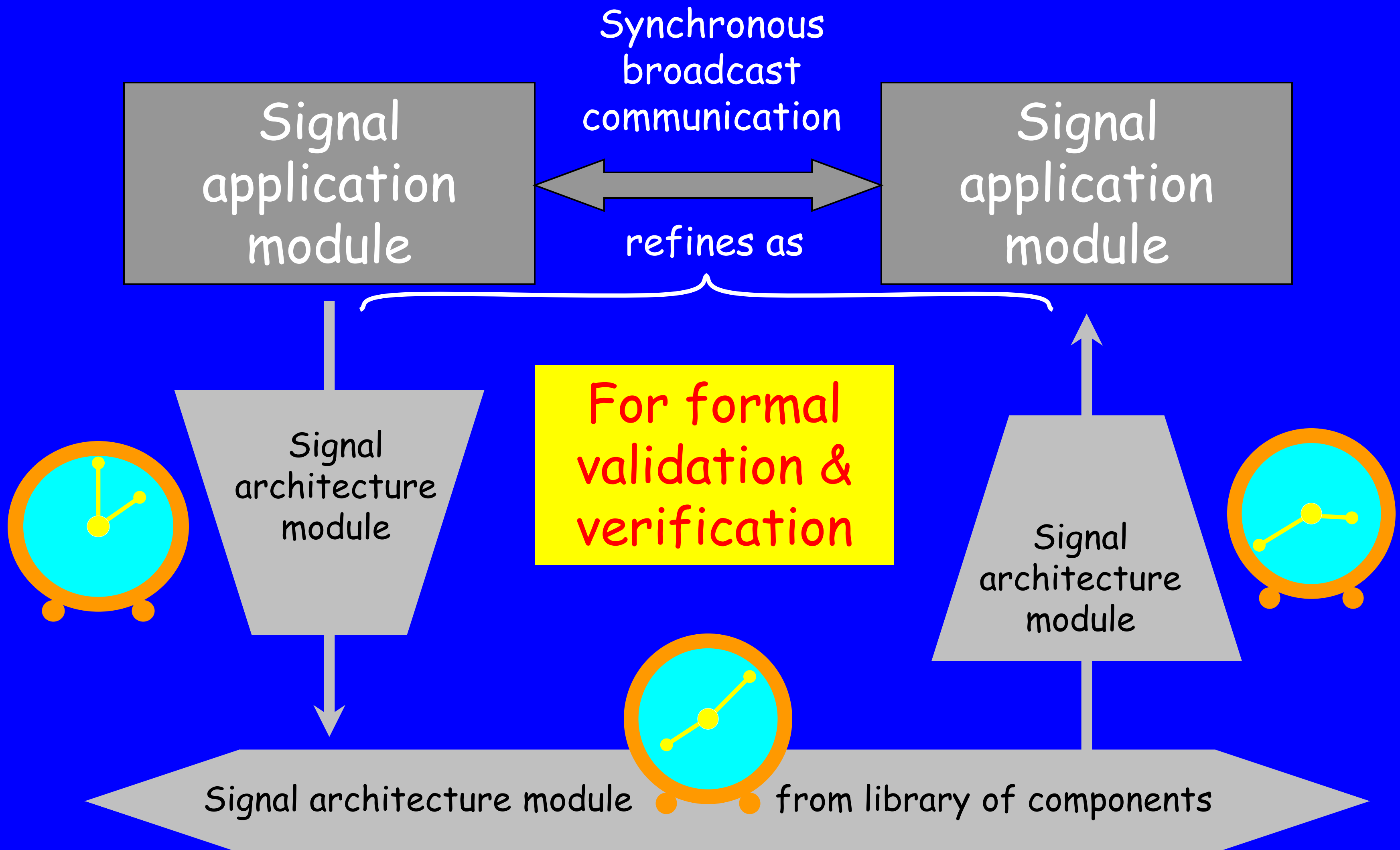
Architecture modeling and analysis (RT-Builder)



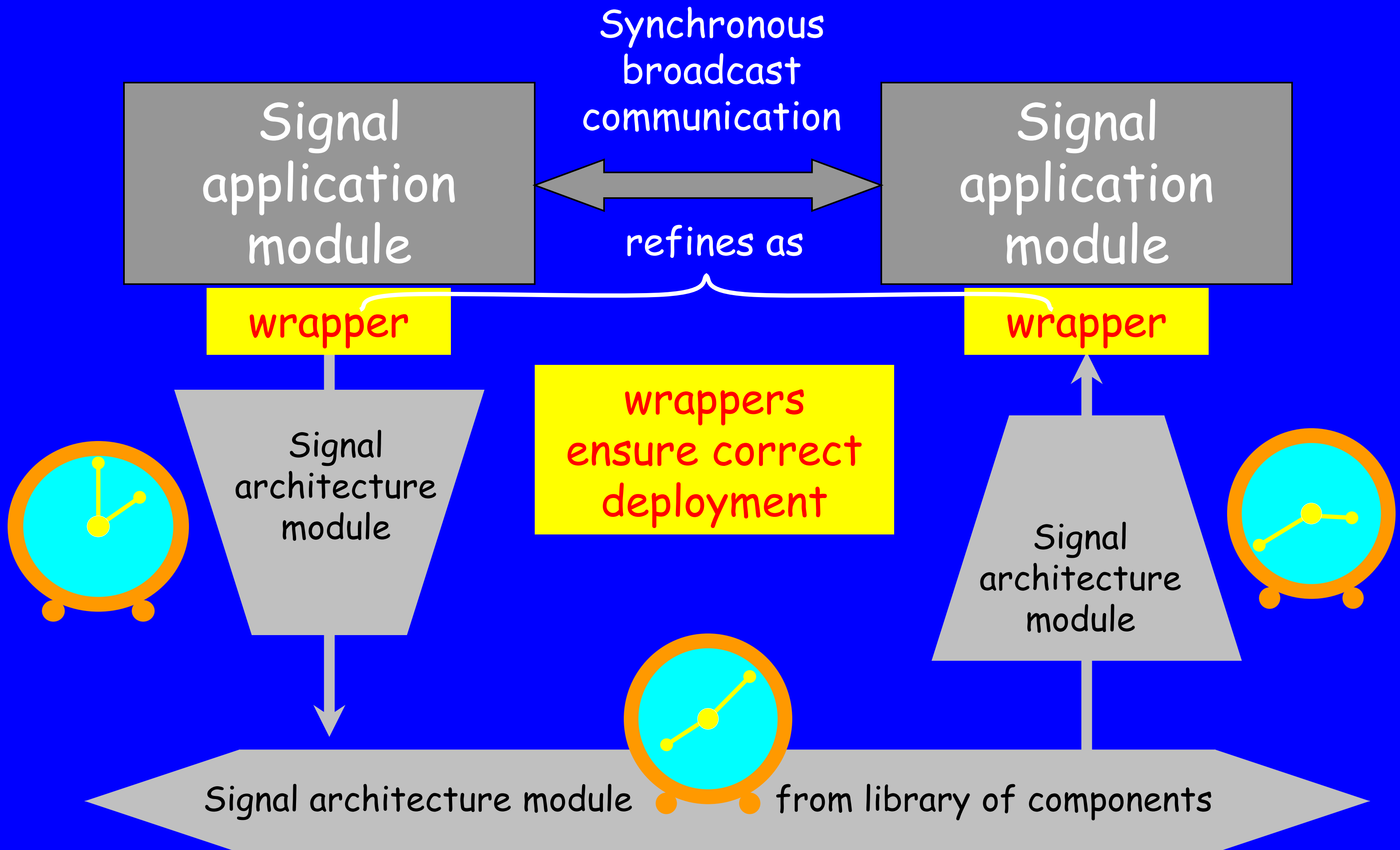
Architecture modeling and analysis (RT-Builder)



Architecture modeling and analysis (RT-Builder)



Architecture deployment: generate wrappers



Conclusion

Signal Polychronous MoCC

- Clear and solid semantics ⇒
  - reproducible simulations
- {synchro+scheduling} calculus ⇒
  - Code generation
  - Interface models
  - Abstract/refine
- Cross-viewpoint models
- Relates and adjusts to other MoCCs ⇒
  - Architecture study

Tools

- *RT-Builder* by Geensys (formerly TNI Software)
- *Polychrony* experimental and academic freeware by INRIA