Embedded software design with Polychrony

DATE'09 tutorial on Correct-by-Construction Embedded Software Synthesis: Formal Frameworks, Methodologies, and Tools

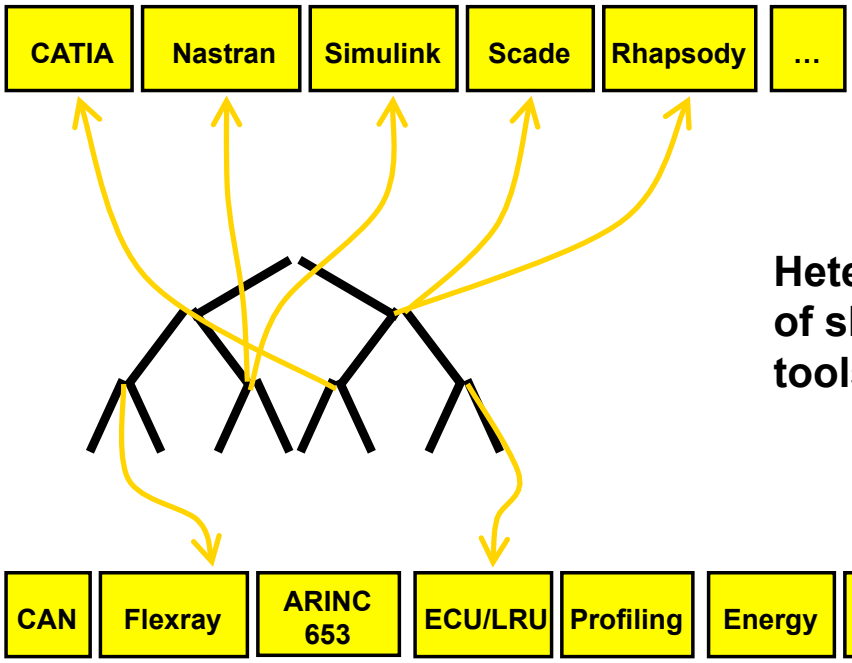
Jean-Pierre Talpin, INRIA

List of contributors Albert Benveniste, Paul Le Guernic, Thierry Gautier, Loïc Besnard, Dumitru Potop, Benoît Caillaud

Embedded software design with Polychrony

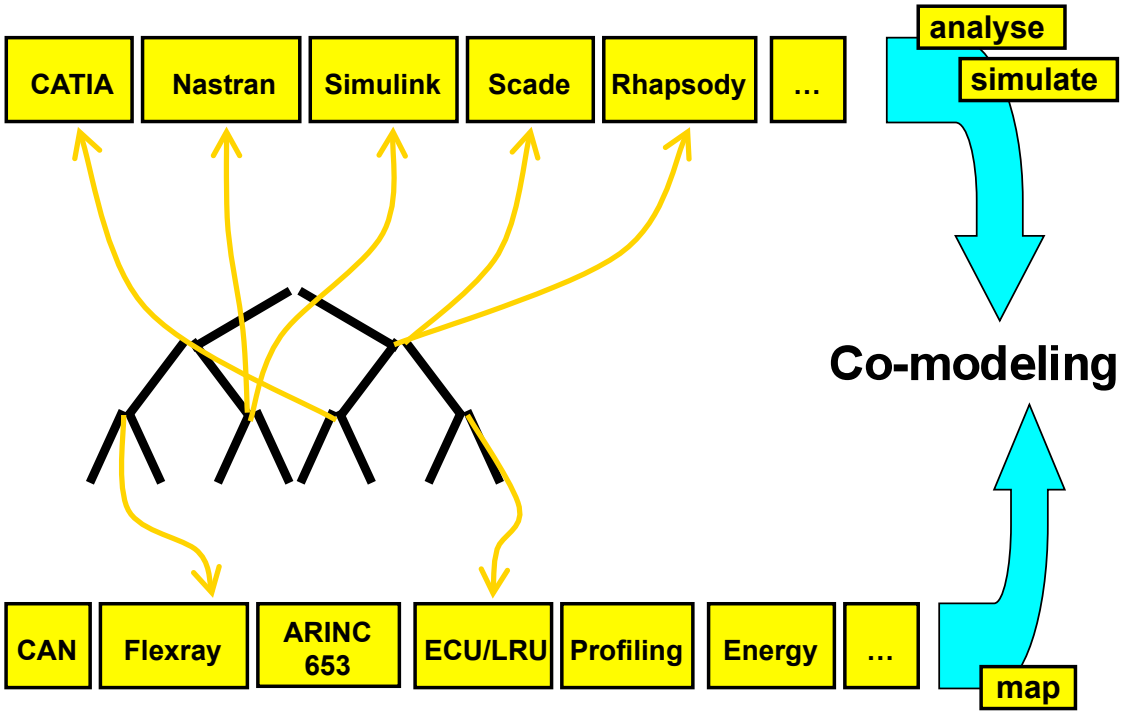
- **Motivations**
  - **Key challenge in system design**
  - **Engineering or mathematics ?**
- Polychronous model of computation
  - The essence of polychrony
  - The old-fashioned watch
- Data structures and code generation
  - From equations to programs
  - Desynchronization and mapping
- Use for architecture modeling and analysis
- Conclusive remarks

Key challenge in system design

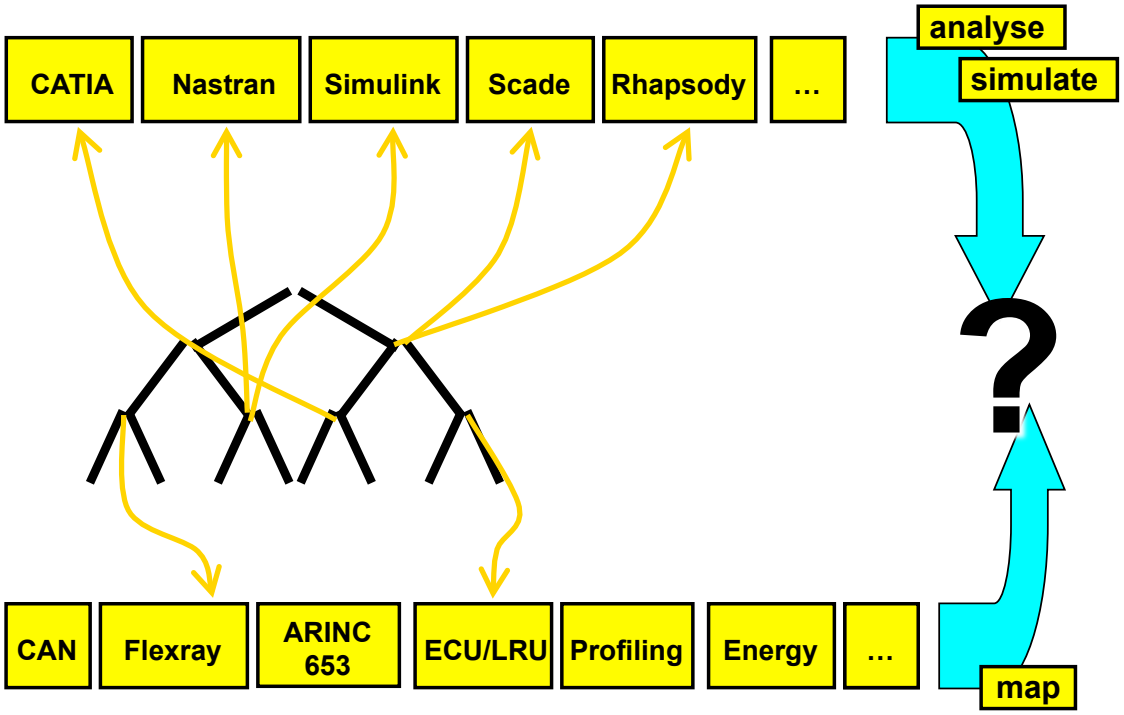


Heterogeneity of skills, teams, tools, methods

Key challenge in system design



Key challenge in system design



Engineering or mathematics ?

FORGET

- RTOS, RME, scheduling
- COTS, reuse
- ARINC 653, architactures
- C, Java, programming
- Simulink, UML, modeling
- ...

REMEMBER

- Specifying sets and surfaces by equations
- Intersecting them by systems of equations
- Equations with no, one or many solutions
- Fixed-point and differential equations
- Solving them may be difficult

Engineering or mathematics ?

ENGINEERING

- You want to reuse components
- Design system by composing modules
- Composition may be blocking or non deterministic
- Embedded systems have real-time behaviors
- Generate and execute code for components

MATHEMATICS

- Specifying sets and surfaces by equations
- Intersecting them by systems of equations
- Equations with no, one or many solutions
- Fixed-point and differential equations
- Solving them may be difficult

Engineering or mathematics ?

MATHEMATICS

Composition is easy

Execution is hard

ENGINEERING

Composition is hard

Execution is easy

The essence of polychrony

MATHEMATICS

Composition is easy
Execution is hard

POLYCHRONY

Synchronous composition is easier
Code generation is harder

The theory amounts to solving equations in a specific model of computation and communication

The essence of polychrony

What it is not

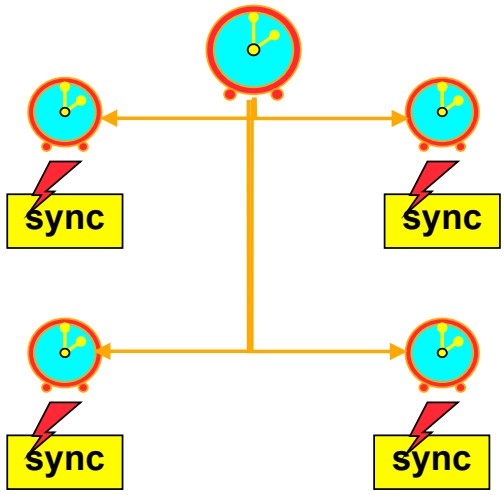
Synchronous hardware

Synchronous dataflow

Simulink diagrams (simple ones)

Each module has a single clock that triggers all signals

All different clocks are derived and sampled from a global master clock via a frequency or phase mechanism



The essence of polychrony

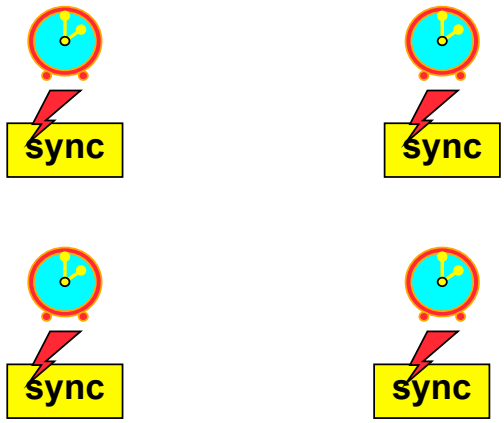
What it is not

Synchronous hardware

Synchronous dataflow

Simulink diagrams (simple ones)

Execution is easy



The essence of polychrony

What it is not

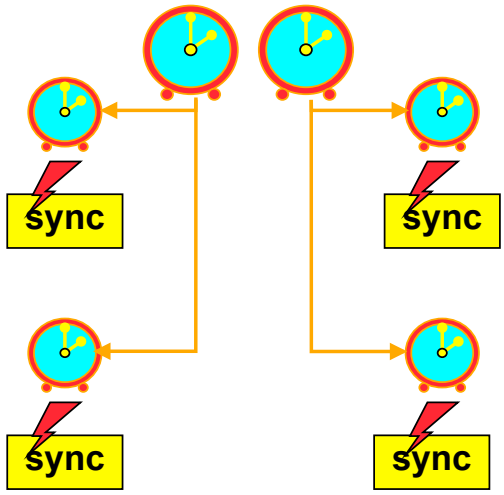
Synchronous hardware

Synchronous dataflow

Simulink diagrams (simple ones)

Execution is easier

but ....



The essence of polychrony

What it is not

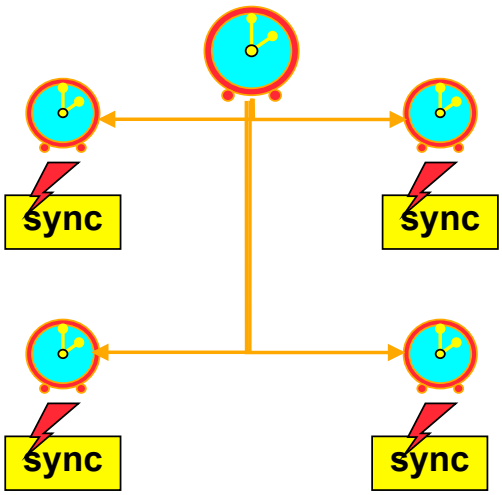
Synchronous hardware

Synchronous dataflow

Simulink diagrams (simple ones)

Execution is easier

Composition is harder



The essence of polychrony

What it is

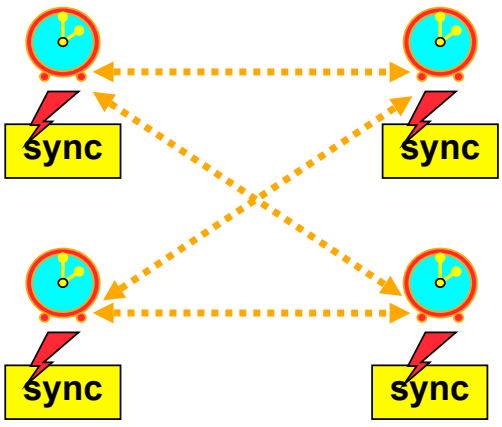
Specification of open systems with synchronous software components: polychrony

Prepared to accept more components

Each module is a synchronous software

There is no global master clock

The different clocks can be related by synchronization constraints



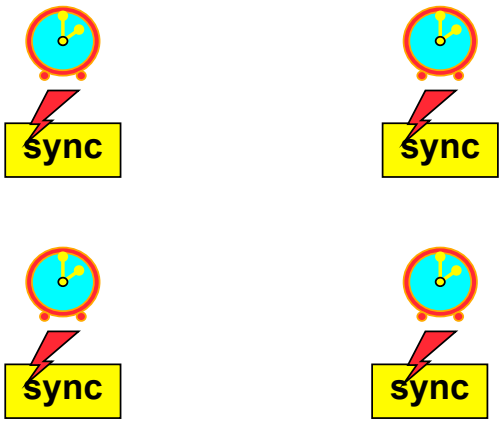
The essence of polychrony

What it is

Specification of open systems with synchronous software components: polychrony

Prepared to accept more components

Execution is harder



The essence of polychrony

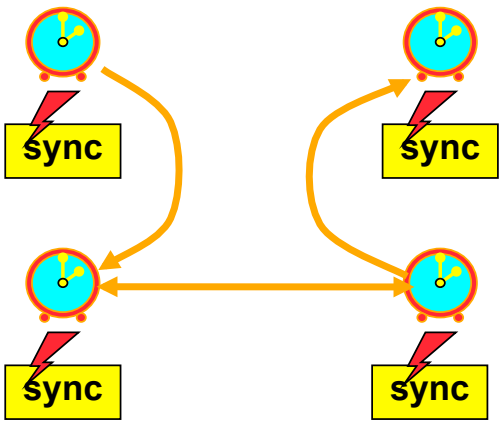
What it is

Specification of open systems with synchronous software components: Polychronous

Clocks can be related by synchronization constraints

Execution is harder

Composition is simpler



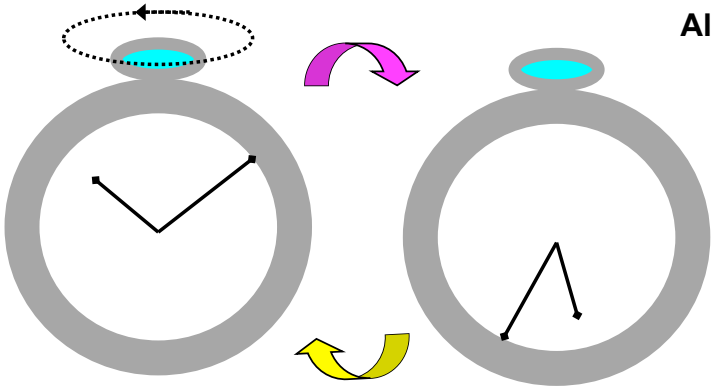
Embedded software design with Polychrony

- Motivations
  - Key challenge in system design
  - Engineering or mathematics ?
- **Polychronous model of computation**
  - **The essence of polychrony**
  - **The old-fashioned watch**
- Data structures and code generation
  - From equations to programs
  - Desynchronization and mapping
- Use for architecture modeling and analysis
- Conclusive remarks

The old-fashioned 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…*"

Albert Benveniste



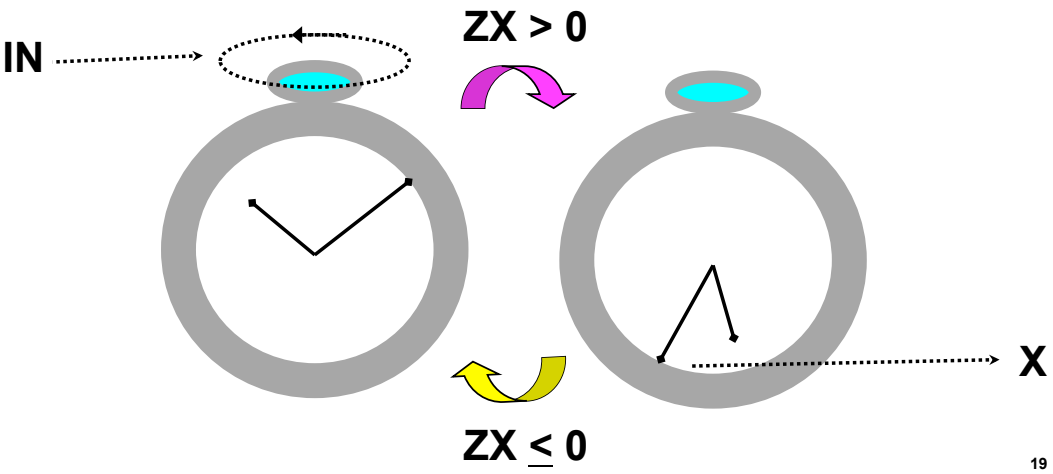
- This is an interesting example:
  - the output up-samples the input,
  - hence it is not a data-flow function.
- We show how to analyze and execute it.

The old-fashioned watch

In equational style

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

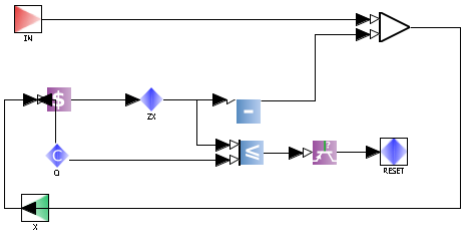
Input IN
Decrement X
Return X if positive
Input IN is negative ....



The old-fashioned watch

In diagrammic style (Eclipse)

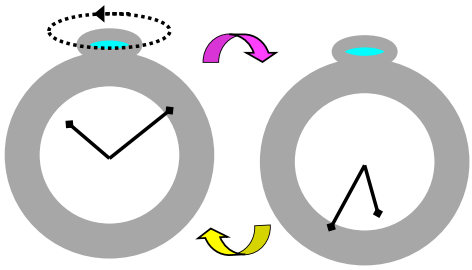
```
(|   X := IN default ZX-1
 | ZX := X$ init 0
 | IN ^= when (ZX ≤ 0)
 |)
```



The old-fashioned watch

A few definitions

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

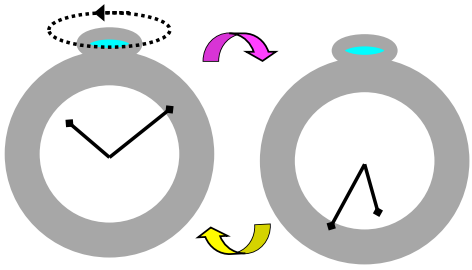


$(t_0, 0)$	**An event is a time tag and a value**
$\{(t_0, 0), (t_1, 3), (t_2, 2), (t_3,1), (t_4, 0)\}$	**A signal is a set of events**
$(t_0, 0)<(t_1, 3)<(t_2, 2)<(t_3,1)<(t_4, 0)$	**Its time tags are totally ordered**

The old-fashioned watch

A few definitions

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

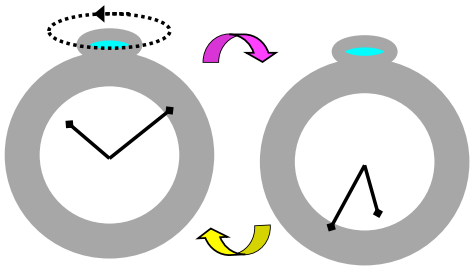


IN $(t_0, 3)$	$(t_4, 3)$	**A trace is a set of signals**
ZX $(t_0, 0)$ $(t_1, 3)$ $(t_2, 2)$ $(t_3,1)$ $(t_4, 0)$		**Its time tags are partially ordered**
X $(t_0, 3)$ $(t_1, 2)$ $(t_2, 1)$ $(t_3,0)$ $(t_4, 3)$		**A process is a set of trace**

The old-fashioned watch

A few definitions

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



IN $(u_0, 3)$ $(u_4, 3)$

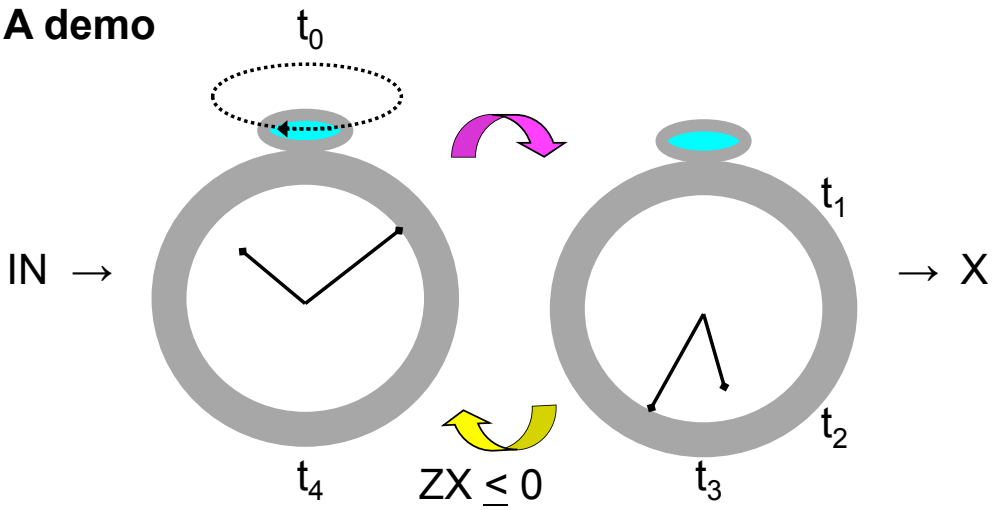
ZX $(t_0, 0)$ $(t_1, 3)$ $(t_2, 2)$ $(t_3, 1)$ $(t_4, 0)$

Tags are related when events are synchronized and equations are composed

```
IN ^= when (ZX ≤ 0)
```


The old-fashioned watch

A demo



IN $(t_0, 3)$ $(t_4, 3)$

ZX $(t_0, 0)$ $(t_1, 3)$ $(t_2, 2)$ $(t_3, 1)$ $(t_4, 0)$

X $(t_0, 3)$ $(t_1, 2)$ $(t_2, 1)$ $(t_3, 0)$ $(t_4, 3)$

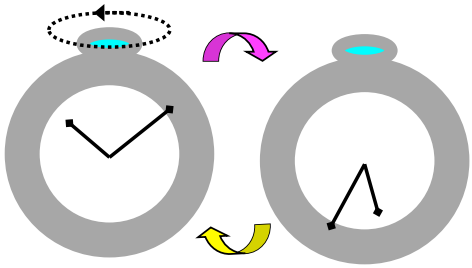
Embedded software design with Polychrony

- Motivations
  - Key challenge in system design
  - Engineering or mathematics ?
- Polychronous model of computation
  - The essence of polychrony
  - The old-fashioned watch
- **Data structures and code generation**
  - **From equations to programs**
  - **Desynchronization and mapping**
- Use for architecture modeling and analysis
- Conclusive remarks

Scheduling and execution

From equations to programs

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



IN $(u_0, 3)$ $(u_4, 3)$

ZX $(t_0, 0)$ $(t_1, 3)$ $(t_2, 2)$ $(t_3, 1)$ $(t_4, 0)$

Synchronization (BDDs)

```
IN ^= when (ZX ≤ 0)
```

IN $(u_0, 3)$ $(u_4, 3)$

X $(t_0, 0)$ $(t_1, 3)$ $(t_2, 2)$ $(t_3, 1)$ $(t_4, 0)$

Causality (scheduling graphs)

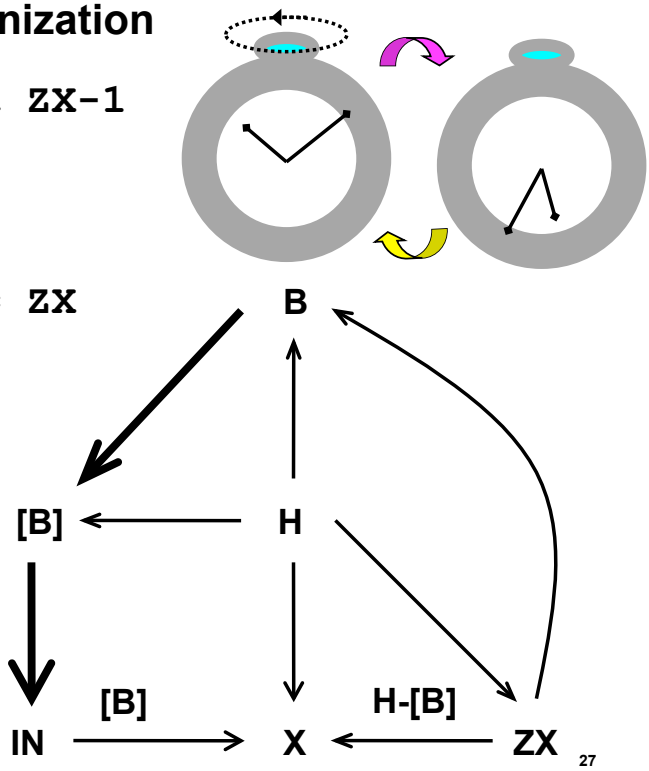
```
X := IN default ZX-1
```

Scheduling and execution

Equations for synchronization

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

Scheduling graph



Causality

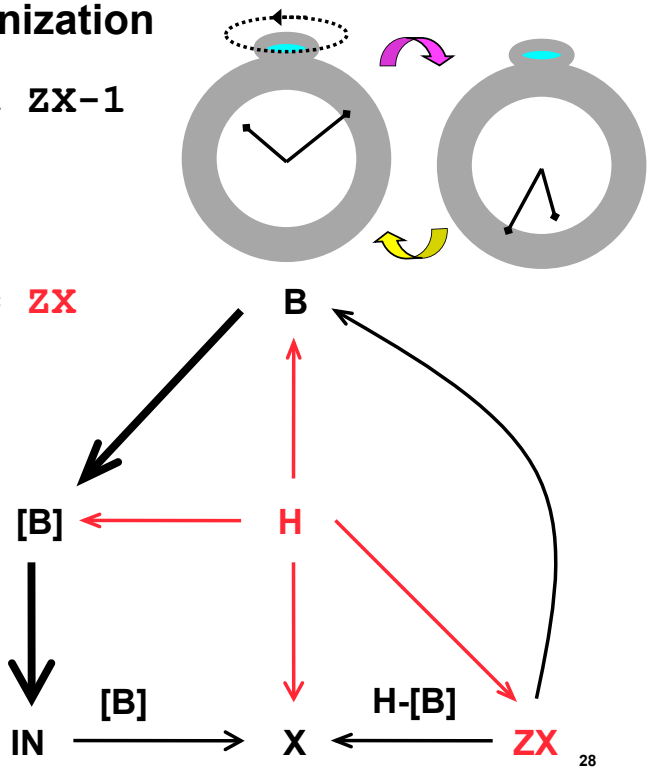
Clock causality

Scheduling and execution

Equations for synchronization

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

Scheduling graph



Evaluated nodes in red

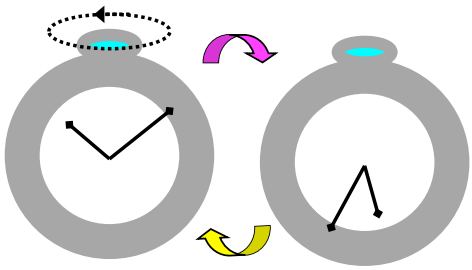
- Root(s) of the graph
- State variable(s)

Enabled transitions in red

Scheduling and execution

Equations for synchronization

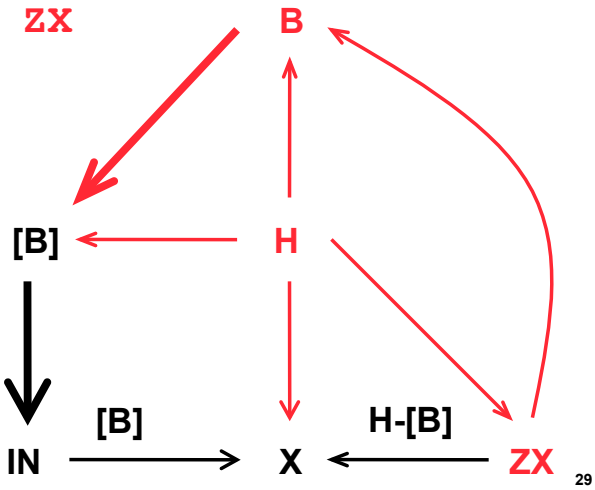
```
   X := IN default ZX-1
 | ZX := X$ init 0
 | B  ^= (ZX ≤ 0)
 | IN ^= [B]
 | H  ^= B ^= X ^= ZX
```



Scheduling graph

Evaluation in red

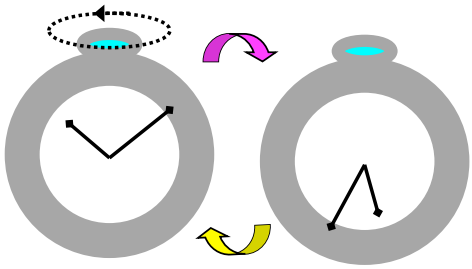
- Immediate successors
- All predecessors of B are enabled



Scheduling and execution

Equations for synchronization

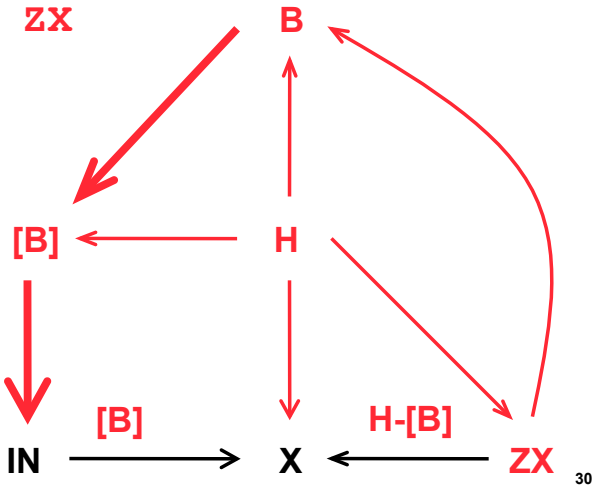
```
   X := IN default ZX-1
 | ZX := X$ init 0
 | B  ^= (ZX ≤ 0)
 | IN ^= [B]
 | H  ^= B ^= X ^= ZX
```



Scheduling graph

Evaluation in red

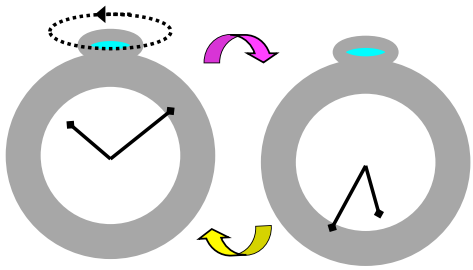
- Value [B] can be determined from its predecessors



Scheduling and execution

Equations for synchronization

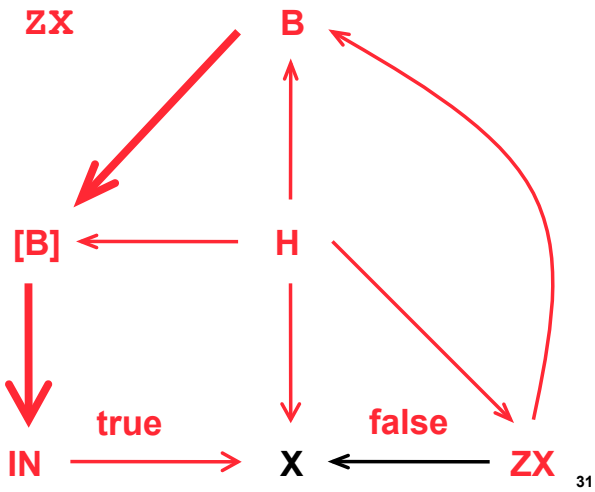
```
   X := IN default ZX-1
 | ZX := X$ init 0
 | B  ^= (ZX ≤ 0)
 | IN ^= [B]
 | H  ^= B ^= X ^= ZX
```



Scheduling graph

Evaluation in red

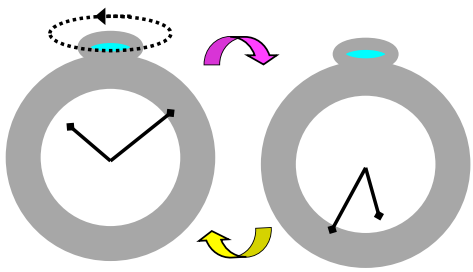
- Initially B is true
- IN is active
- Its transition is active



Scheduling and execution

Equations for synchronization

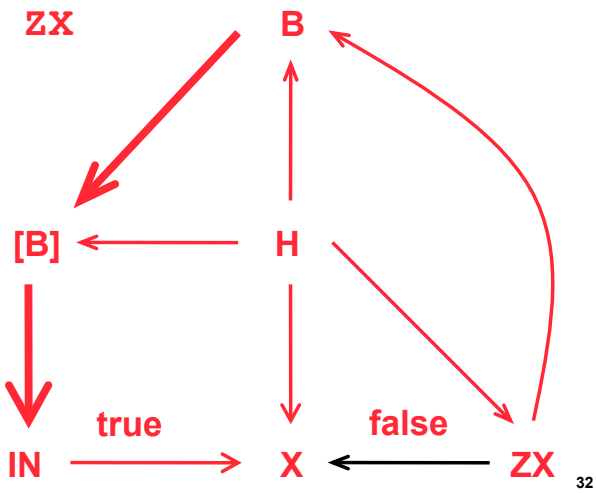
```
   X := IN default ZX-1
 | ZX := X$ init 0
 | B  ^= (ZX ≤ 0)
 | IN ^= [B]
 | H  ^= B ^= X ^= ZX
```



Scheduling graph

Evaluation in red

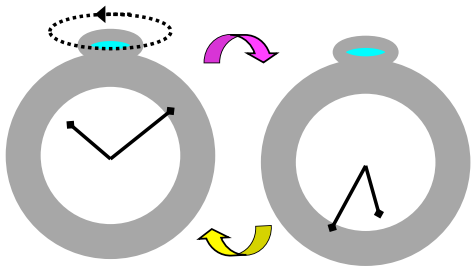
- X can be computed

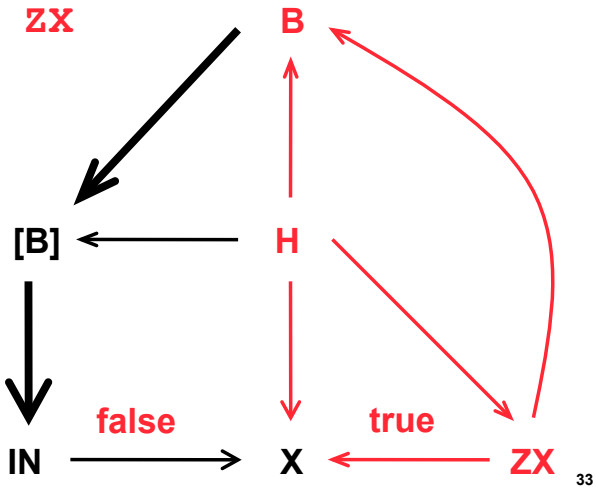


Scheduling and execution

Equations for synchronization

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





Scheduling graph

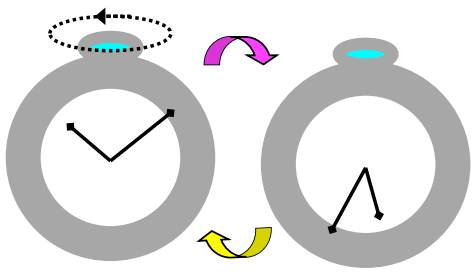
Evaluation in red

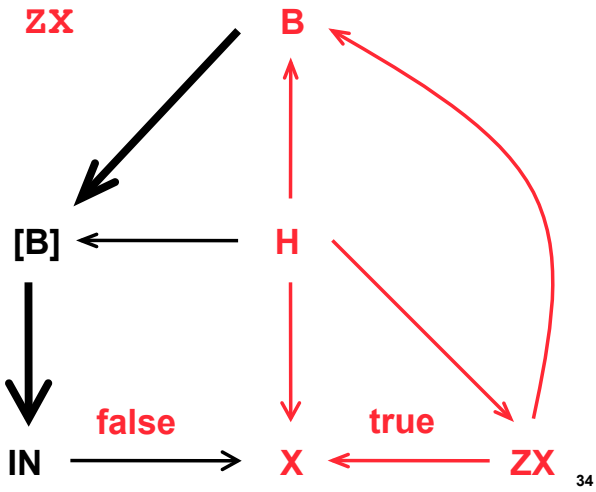
- Second time, B is false
- IN is inactive

Scheduling and execution

Equations for synchronization

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





Scheduling graph

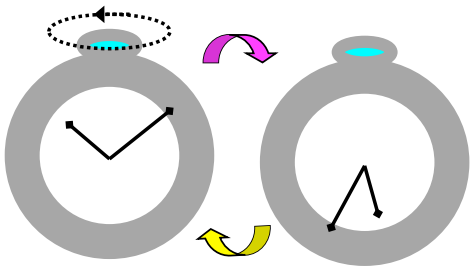
Evaluation in red

- X is computed from ZX

Scheduling and execution

Equations for synchronization

```
   B := (ZX ≤ 0)
|  IN ^= [B]
|   H ^= B ^= X ^= ZX
```



Scheduling graph

```
    X  <-  IN when B
|   X  <-  ZX when ¬B
|   B  <- (H,ZX)
|  [B] <-  B
|  ZX  <-  H
|  IN  <- [B]
```

Clock and scheduling relations are part of the Signal syntax

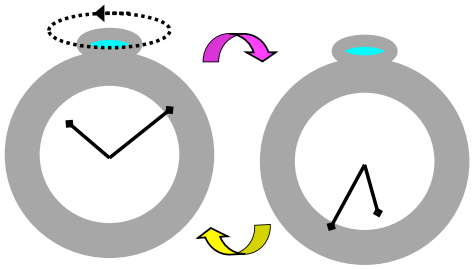
Scheduling and execution

Equations for synchronization

```
   B := (ZX ≤ 0)
|  IN ^= [B]
|   H ^= B ^= X ^= ZX
```

Scheduling graph

```
    X  <-  IN when B
|   X  <-  ZX when ¬B
|   B  <- (H,ZX)
|  [B] <-  B
|  ZX  <-  H
|  IN  <- [B]
```



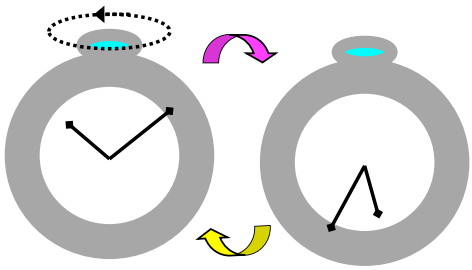
Clock and scheduling relations define an interface model used to :

- represent a module in its environment
- separately compile a module
- map a set of modules on an architecture

Scheduling and execution

Equations for synchronization

```
   B := (ZX ≤ 0)
|  IN ^= [B]
|   H ^= B ^= X ^= ZX
```



Scheduling graph

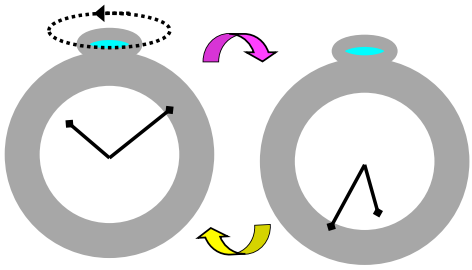
```
    X  <-  IN when B
|   X  <-  ZX when ¬B
|   B  <- (H,ZX)
|  [B] <-  B
|  ZX  <-  H
|  IN  <- [B]
```

Clock and scheduling relations define a calculus to :

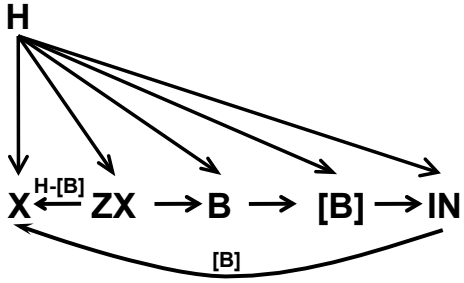
- give an operational semantics and interpret a system of equations
- define correctness-preserving transformations and code generation functionalities

Scheduling and execution

Code generation



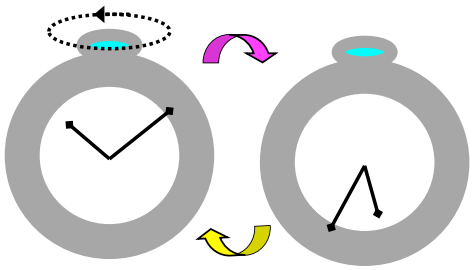
```
if (!read_watch_H(&H))
  return FALSE;
B = X < 0;
if (B) {
  if (!read_watch_IN(&IN))
    return FALSE;
  X = IN;
} else X = X - 1;
write_watch_X(X);
return TRUE;
```



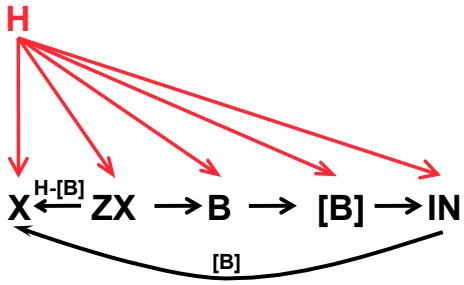
Clocks and scheduling relations

Scheduling and execution

Code generation



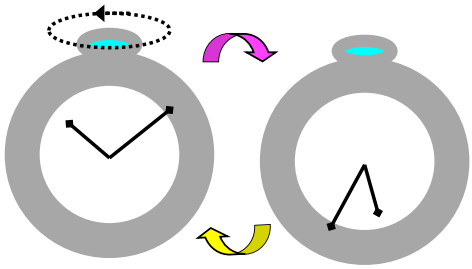
```
if (!read_watch_H(&H))
  return FALSE;
B = X < 0;
if (B) {
  if (!read_watch_IN(&IN))
    return FALSE;
  X = IN;
} else X = X - 1;
write_watch_X(X);
return TRUE;
```



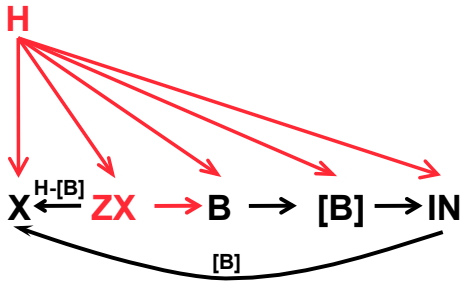
H is available

Scheduling and execution

Code generation



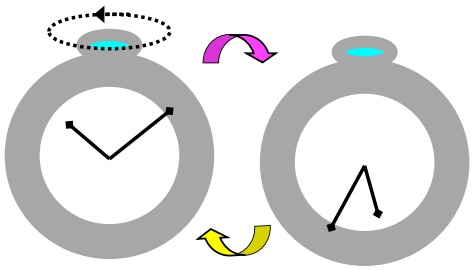
```
if (!read_watch_H(&H))
  return FALSE;
B = X < 0;
if (B) {
  if (!read_watch_IN(&IN))
    return FALSE;
  X = IN;
} else X = X - 1;
write_watch_X(X);
return TRUE;
```



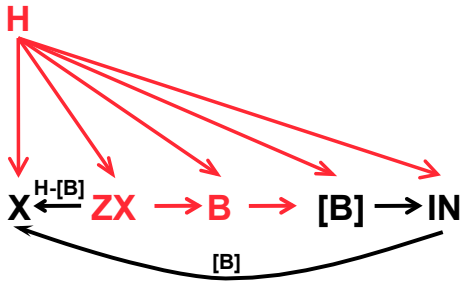
The state variable X is fetch

Scheduling and execution

Code generation



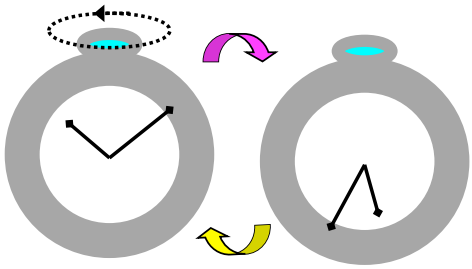
```
if (!read_watch_H(&H))
  return FALSE;
B = X < 0;
if (B) {
  if (!read_watch_IN(&IN))
    return FALSE;
  X = IN;
} else X = X - 1;
write_watch_X(X);
return TRUE;
```



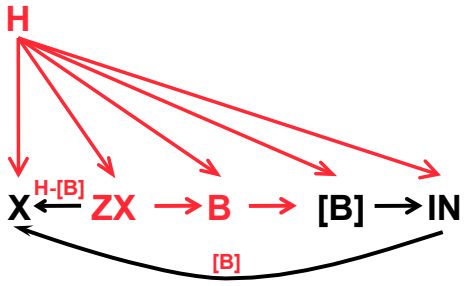
B is computed

Scheduling and execution

Code generation



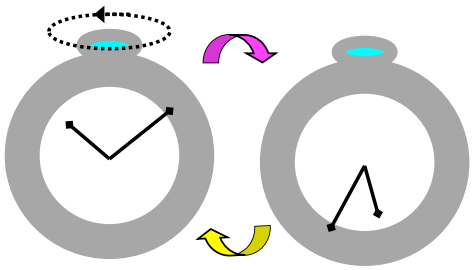
```
if (!read_watch_H(&H))
  return FALSE;
B = X < 0;
if (B) {
  if (!read_watch_IN(&IN))
    return FALSE;
  X = IN;
} else X = X - 1;
write_watch_X(X);
return TRUE;
```



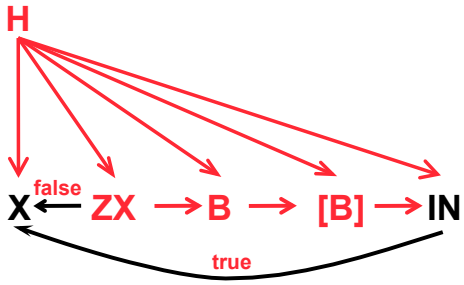
B is tested

Scheduling and execution

Code generation



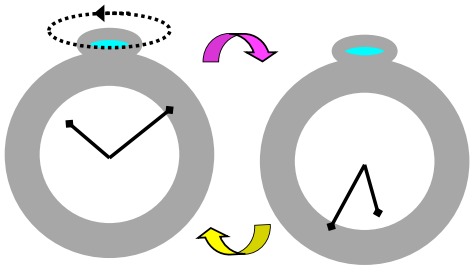
```
if (!read_watch_H(&H))
  return FALSE;
B = X < 0;
if (B) {
  if (!read_watch_IN(&IN))
    return FALSE;
  X = IN;
} else X = X - 1;
write_watch_X(X);
return TRUE;
```



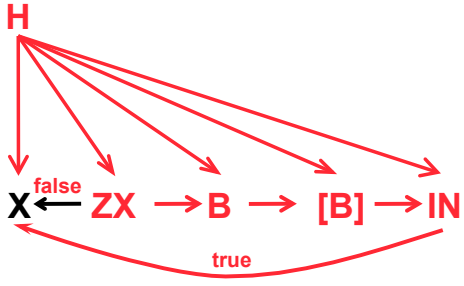
Initially, B is true

Scheduling and execution

Code generation



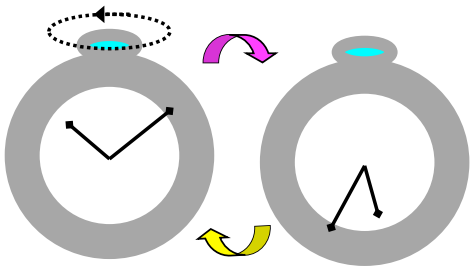
```
if (!read_watch_H(&H))
  return FALSE;
B = X < 0;
if (B) {
  if (!read_watch_IN(&IN))
    return FALSE;
  X = IN;
} else X = X - 1;
write_watch_X(X);
return TRUE;
```



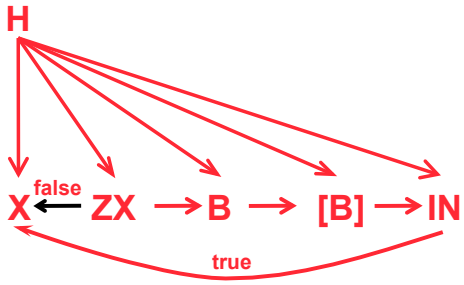
IN is fetch

Scheduling and execution

Code generation



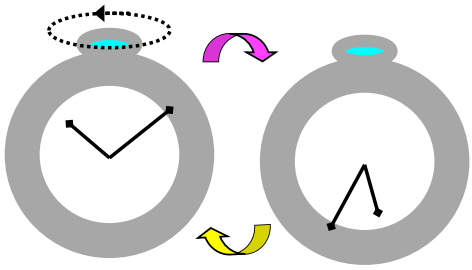
```
if (!read_watch_H(&H))
  return FALSE;
B = X < 0;
if (B) {
  if (!read_watch_IN(&IN))
    return FALSE;
  X = IN;
} else X = X - 1;
write_watch_X(X);
return TRUE;
```



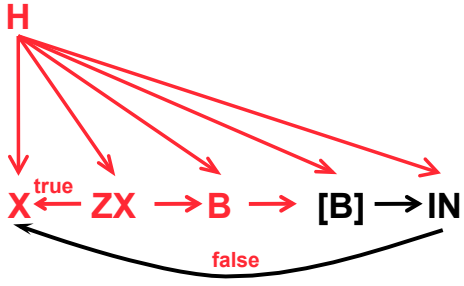
X is computed and sent

Scheduling and execution

Code generation



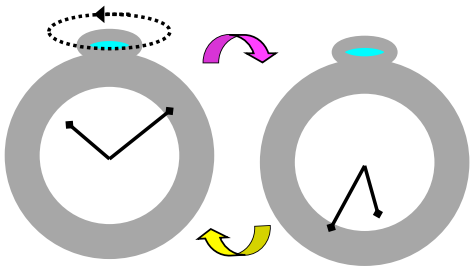
```
if (!read_watch_H(&H))
  return FALSE;
B = X < 0;
if (B) {
  if (!read_watch_IN(&IN))
    return FALSE;
  X = IN;
} else X = X - 1;
write_watch_X(X);
return TRUE;
```



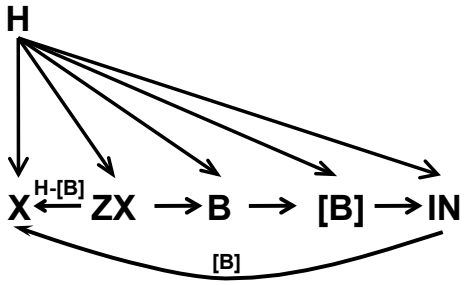
Next time, B will be false

Scheduling and execution

Model transformation and code generation based on clocks and scheduling relations

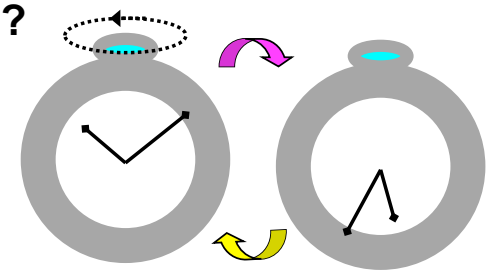


- **Serialization** reinforcement of a graph for sequential code generation
- Input-driven or output-driven **clustering** for modular code generation
- Distributed code generation
- GALS architecture **mapping**



Correctness

What if IN is not synchronized ?

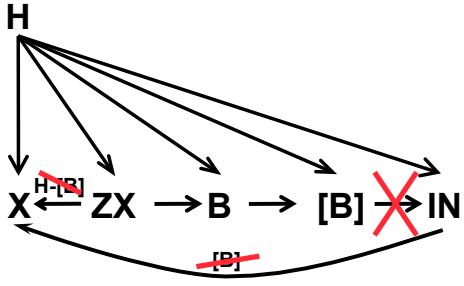


```
   X := IN default ZX-1
 | ZX := X$ init 0
 | B  ^= (ZX ≤ 0)
 | IN ^= [B]
 | H  ^= B ^= X ^= ZX
```

Consequence

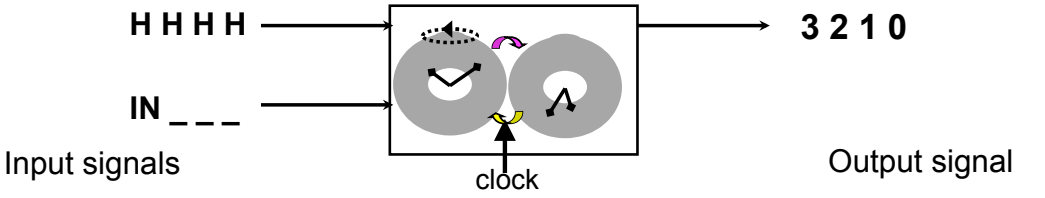
X is still deterministically defined by IN or ZX

Is this harmless ?



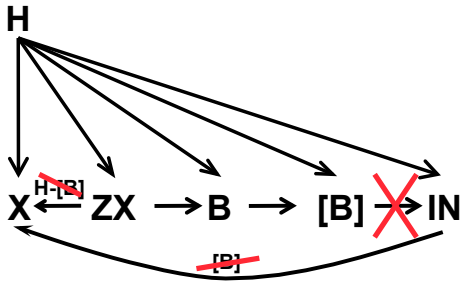
Correctness

In a synchronous environment IN still has its own clock



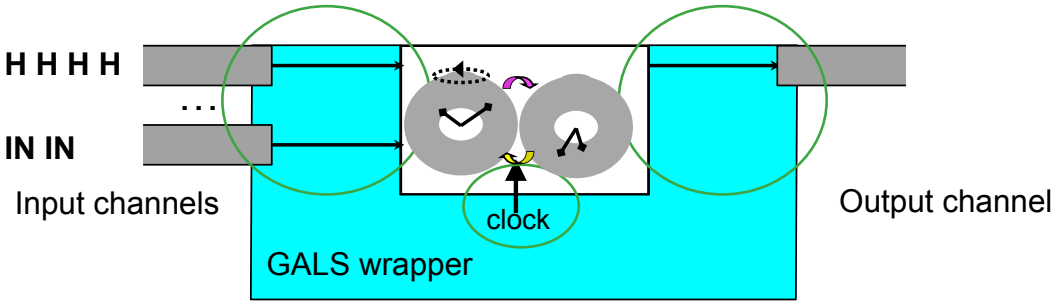
The clock still decides when to schedule H and IN

The clock is deterministic



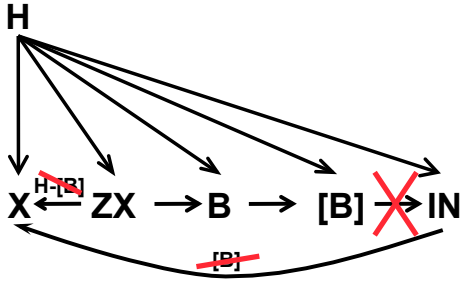
Correctness

In an asynchronous environment, timing is elastic



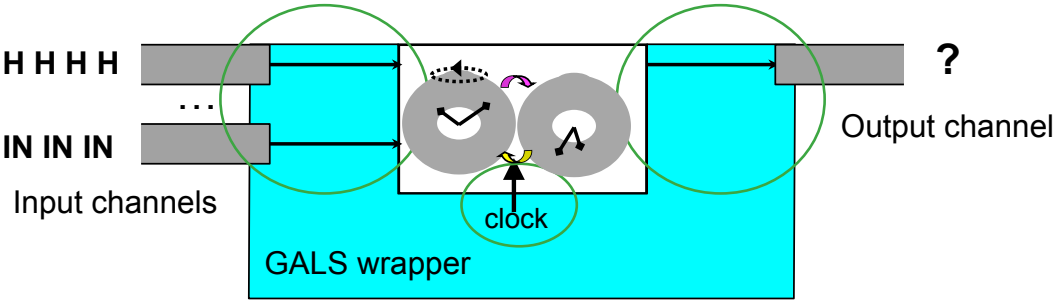
Synchronous signals are interfaced to asynchronous channels

Reaction triggering defines module activation



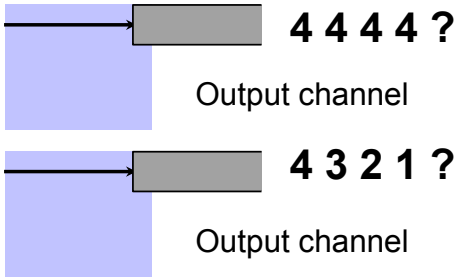
Correctness

But the core relies on the absence or presence of IN



The watch can be reset at any time

The intended specification is broken

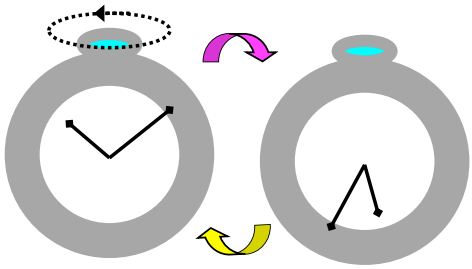


Correctness

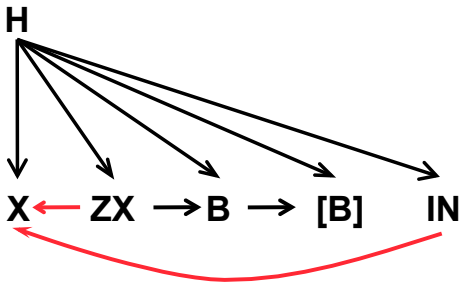
Endochrony

The capability of a module to internally compute the presence or absence of all signals

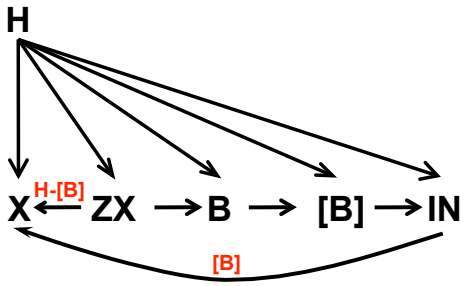
... so that it can be interfaced with asynchronous channels



This is deterministic



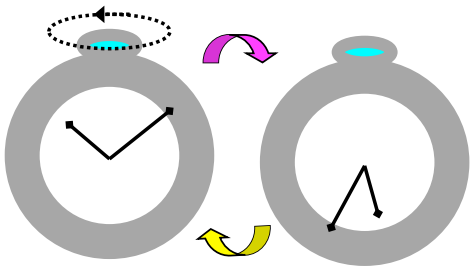
This is endochronous



Correctness

Problem : endochrony is not compositional

If two modules p and q are endochronous then p | q not necessarily is



Solution : weakening endochrony [Potop et al.,'05-'07]

Existence of stuttering states

From any execution point (e.g. in the scheduling graph) one reaches a stuttering state

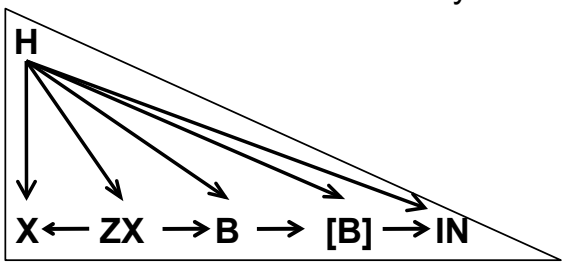
Confluence

Two compatible reactions (e.g. two sub-graphs) can be scheduled in any order and yield to the same stuttering state

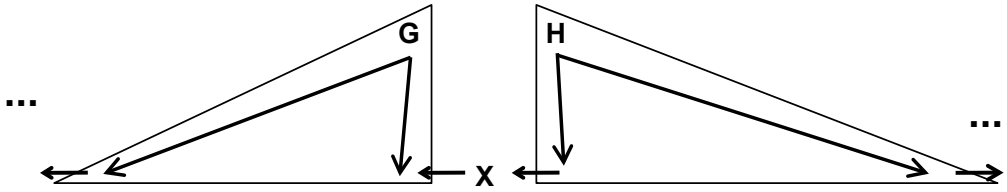
Correctness

Weak endochrony is compositional

If a module p is endochronous then it is weakly endochronous



If two modules p and q are weakly endochronous
if there composition p | q is non blocking
then p | q is weakly endochronous



A conservative, static and compositional decision procedure

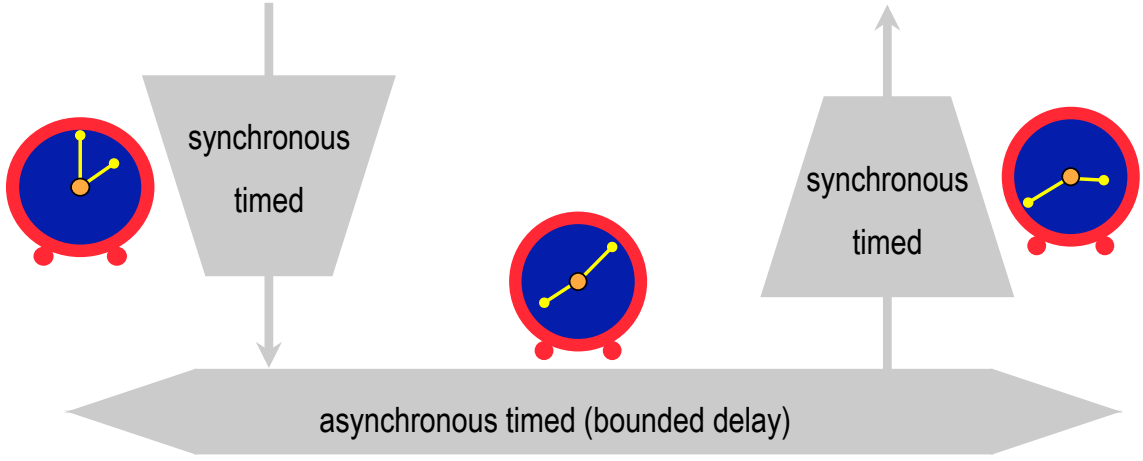
Embedded software design with Polychrony

- Motivations
  - Key challenge in system design
  - Engineering or mathematics ?
- Polychronous model of computation
  - The essence of polychrony
  - The old-fashioned watch
- Data structures and code generation
  - From equations to programs
  - Desynchronization and mapping
- **Use for architecture modeling and analysis**
- **Conclusive remarks**

Architecture modeling and analysis

Example of the loosely time-triggered architecture (Benveniste et al.)

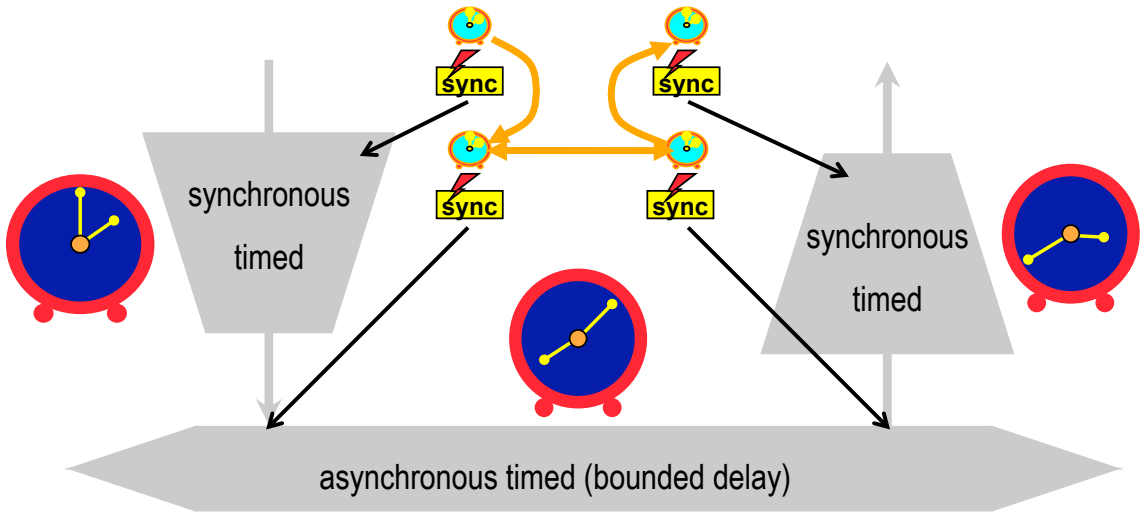
Writer, bus and reader are periodic
They have non-synchronized triggers



Architecture modeling and analysis

The loosely time-triggered protocol (EMSOFT'02)

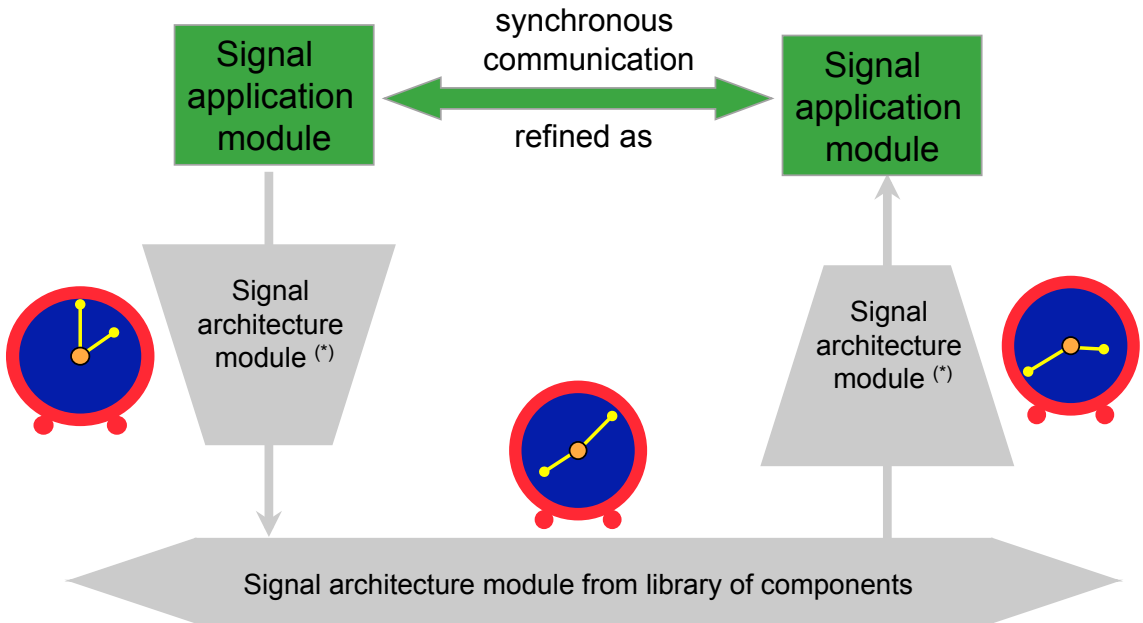
Upgrade LTTA with protocol for bounded inter-clock jitter
Use GALS mapping techniques



Architecture modeling and analysis

RT-Builder (Geensys)

Real-time simulation and architecture exploration, verification, validation

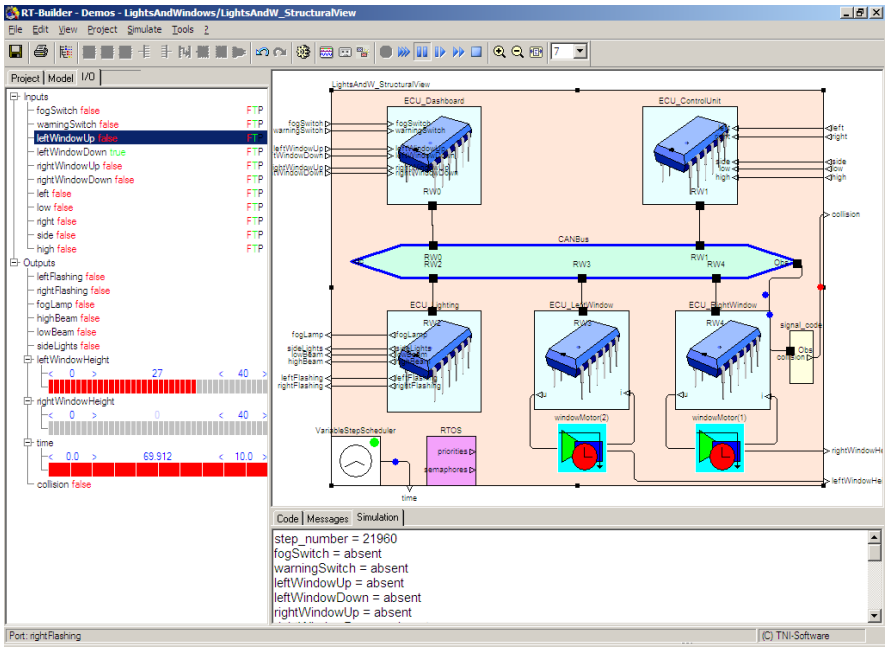


(*) example : library of Signal models for the APEX ARINC-653 real-time operating system services

Architecture modeling and analysis

RT-Builder (Geensys)

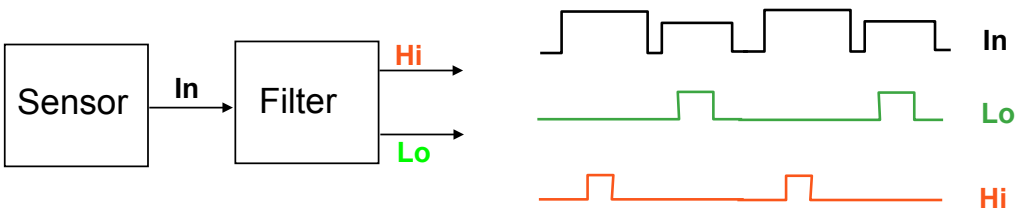
Real-time, hardware in-the-loop, simulation of electronic equipments



Conclusions - methodology

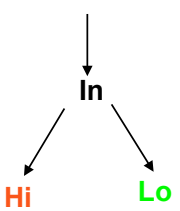
Synchronous abstraction of heterogeneous functionalities

Continuous and/or discrete real-time processing



Abstraction of control by partially ordered scheduling relations

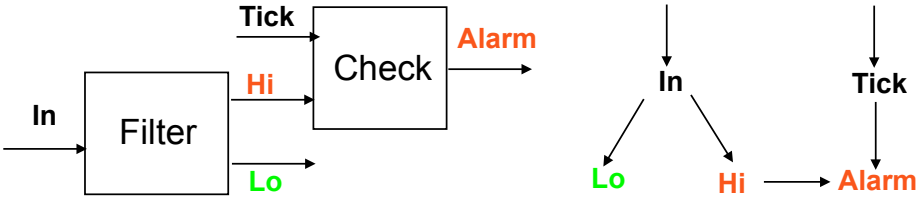
In is present iff either **Hi** or **Lo** is present
Hi and **Lo** are never present simultaneously
Hi and **Lo** cannot happen before **In**



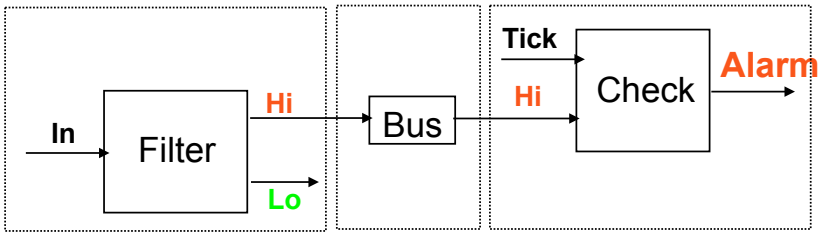
Conclusions - methodology

A refinement-based design process

Specification of multi-clocked systems by partially related symbolic clocks



Transformation and code generation for mapping on GALS architectures

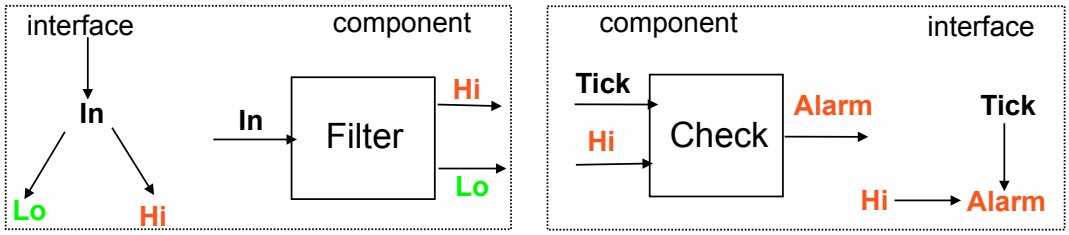


Bridge between functional specification and architecture modeling

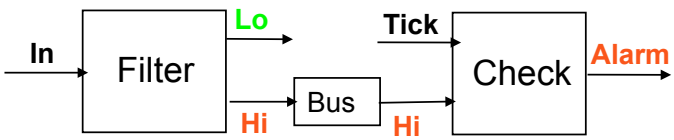
Conclusions - methodology

Component-based design process

Encapsulation of heterogeneous functionalities with interface descriptions



Transport and integration of encapsulated functions and services



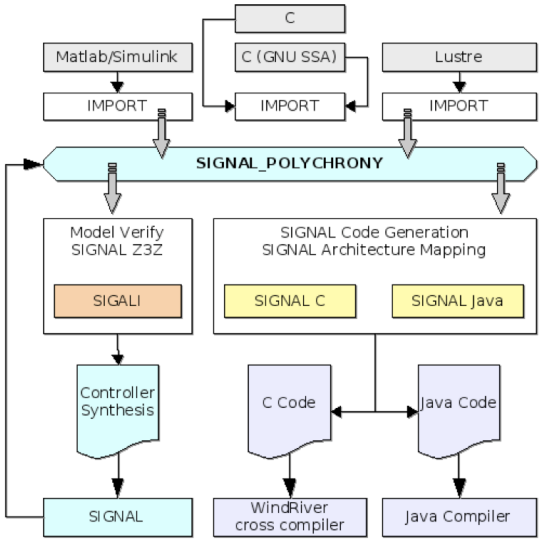
Pivot formalism for architecture exploration

Conclusions - tools

An experimental toolset

- Signal data-flow language
- Libraries (RTOS services)
- Analysis engine
- Transformation algorithms
- Model checker
- Controller synthesis
- Import functionalities: GCC-SSA, StateCharts, Simulink, Lustre, Scade …
- Code generators: C, C++, Java

=> SME, an Eclipse-TopCased plug-in



Conclusions - tools

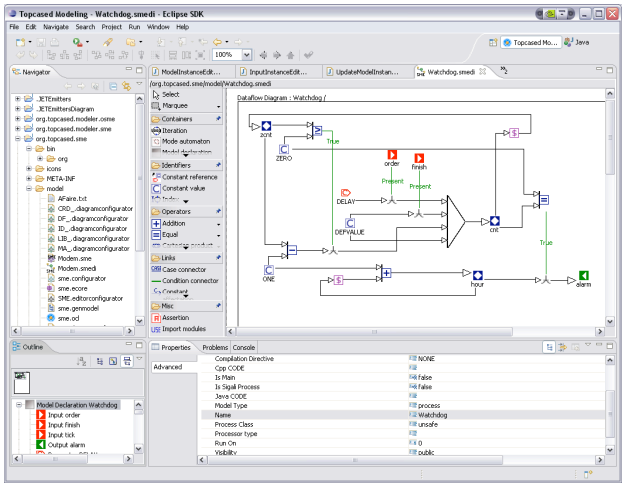
SME, synchronous modeling environment
An open-source Eclipse plugin for Polychrony

A unified model of computation for architecture exploration of integrated modular avionics

- Data-flow for computation
- Mode automata for control
- Libraries for services

An eclipse interactive interface

- Open import functionalities
- High-level visual editor
- Analysis and transformation visualization and traceability



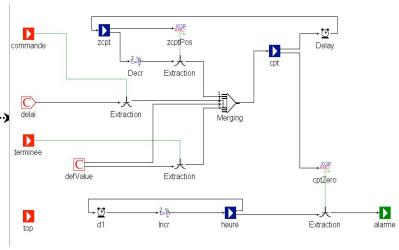
Component of the TopCased and OpenEmbeDD project

Conclusions - tools

Principle

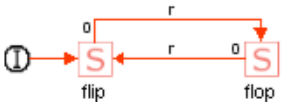
One diagram per conceptual view (SLAP'06) :

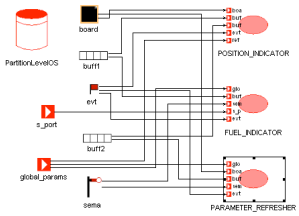
- Untimed data-flow diagram
- A relational timing model

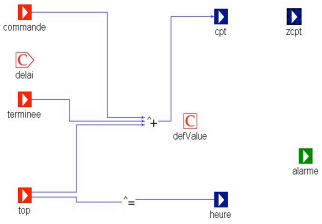


An open and extensible framework

- multi-clocked mode automata (EMSOFT'06)
- integrated modular avionics (SEAA'06)

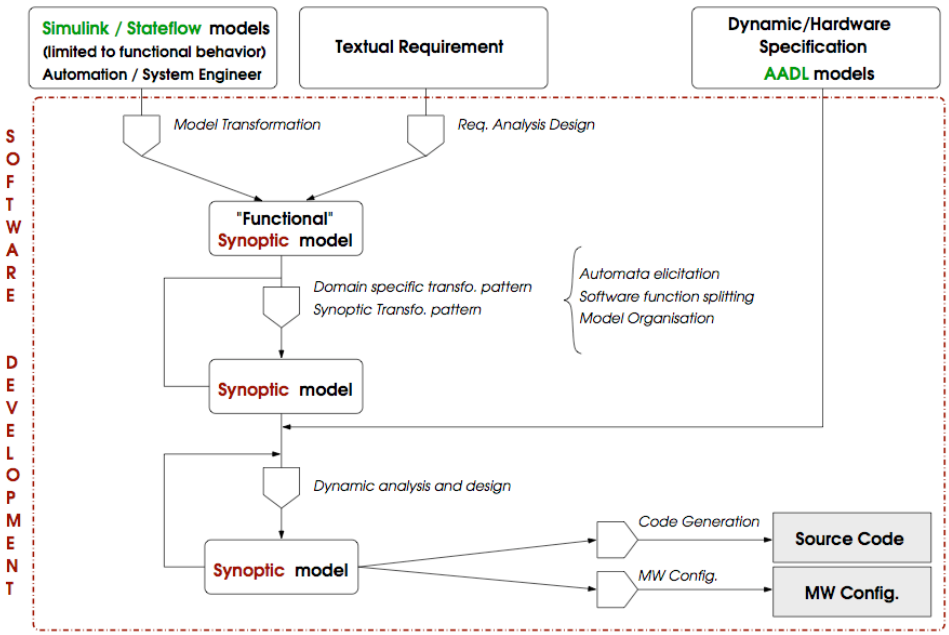






Conclusions - tools

Synoptic - domain-specific design language for space application software



Conclusion

The polychronous model of computation and communication

- A clear and solid semantics
  - Compositional correctness criteria
- A calculus of synchronization and scheduling
  - Sequential, modular, distributed code generation
  - Interface models, abstraction and refinement
- Model (oCC) transformations
  - Heterogeneous architecture modeling and analysis
  - Virtual prototyping

Tools

- RT-Builder by Geensys
- Polychrony, experimental and academic freeware by INRIA

Bibliography

On the model of computation

"Polychrony for system design" Le Guernic, P., Talpin, J.-P., Le Lann, J.-C. Journal for Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design. World Scientific, August 2003.

On desynchronization

“Correct-by-construction asynchronous implementation of modular synchronous specifications”. D. Potop-Butucaru, B. Caillaud. In Fundamenta Informaticae. IOS Press, 2006.

On architecture modeling

"Polychronous design of embedded real-time systems" Gamatié, A., Gautier, T., Le Guernic, P., Talpin, J.-P. ACM Transactions on Software Engineering and Methodology. ACM Press, 2006.

On virtual prototyping

"Formal refinement checking in a system-level design methodology" Talpin, J.-P., Le Guernic, P., Shukla, S. K., Gupta, R., Doucet, F. Special Issue of Fundamenta Informaticae on Applications of Concurrency to System Design. IOS Press, 2004.

On model-driven engineering

"A metamodel for the design of polychronous systems" Brunette, C., Talpin, J.-P., Gamatié, A., Gautier, T. Journal of Logic and Algebraic Programming, Special Issue on Applying Concurrency Research to Industry. Elsevier, 2008.

Website http://www.irisa.fr/espresso