Synchronous Programming in Control

Paul Caspi

Verimag (CNRS)

A historical perspective based on the observation of several real-world systems during the Crisys Esprit project:

- The Airbus “fly-by-wire” system.
- Schneider’s safety control and monitoring systems for nuclear plants.
- Siemens’ letter sorting machine control,

and many other distributed safety-critical control systems.

Overview

- The Origins of Synchronous Programming
- Synchronous Programming and Real-Time
- Real-Time Validation
- Understanding Synchronous Programming in Control

The Origins of Synchronous Programming

- Basic needs of the domain
- Real-time asynchronous languages
- Synchronous practices
- The formalisation of these practices

Basic Needs of the Domain

- Parallelism:
  - between the controller and the controlled device
  - between the several degrees of freedom to be controlled at the same time
- Guaranteed bounds :
  - on memory
  - on execution times
- Distribution

The Computer Science Answer:
Real-Time Kernels and Languages

Based on the concurrency tradition of operating systems:

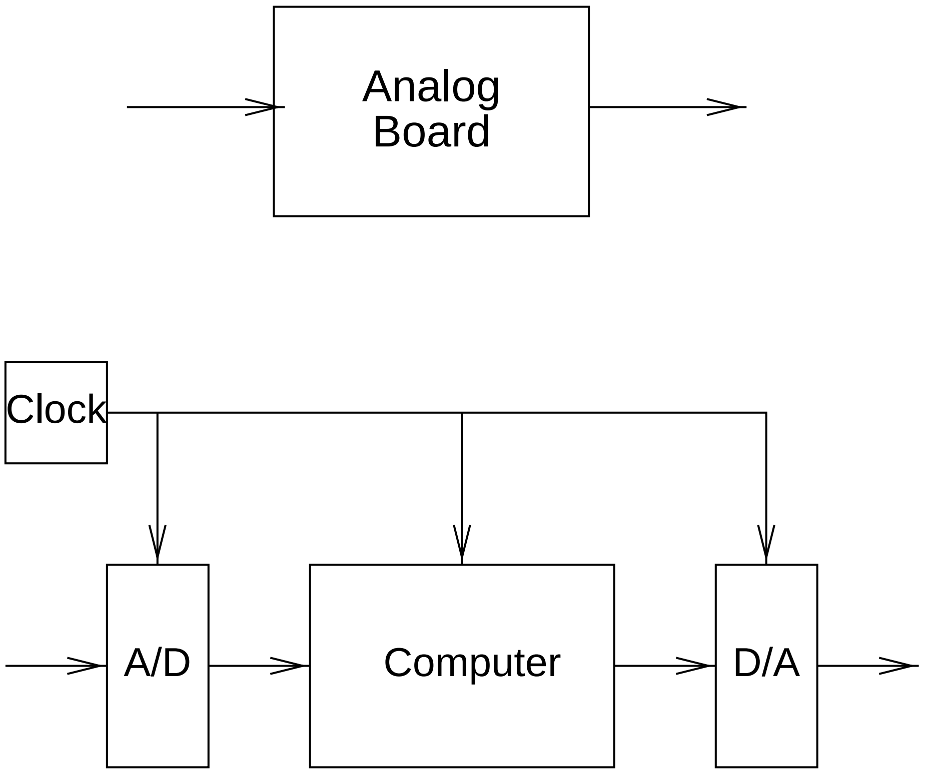
- Synchronisation: semaphores, monitors, sequential processes,
- Communication: shared memory, messages,
- Synchronisation + communication: queues, rendez-vous.

Examples:

- CSP, OCCAM,
- ADA tasking
- real-time OS

The Evolution of Practices

From analog boards to computers:



Analog Board

Clock

A/D

Computer

D/A

periodic clocks

synchronous programs

Periodic Synchronous Programming

```
initialize state;

loop each clock tick

   read other inputs;
   compute outputs and state;
   emit outputs

 end loop
```

Practical Interest

- Perfectly matches:
  - the need for real-time integration of differential equations: forward, fixed step methods,
  - the mathematical theory of sampled control systems,
  - the theory of switching systems.
- Safety, simplicity and efficiency:
  - almost no OS, a single interrupt (the real-time clock), no context saving (the interrupt should occur at idle time)
  - bounded memory, bounded execution time.

⇒ Easier validation, certification

Generalisation: Synchronous Languages

```
initialize state;
loop each input event
   read other inputs;
   compute outputs and state;
   emit outputs
 end loop
```

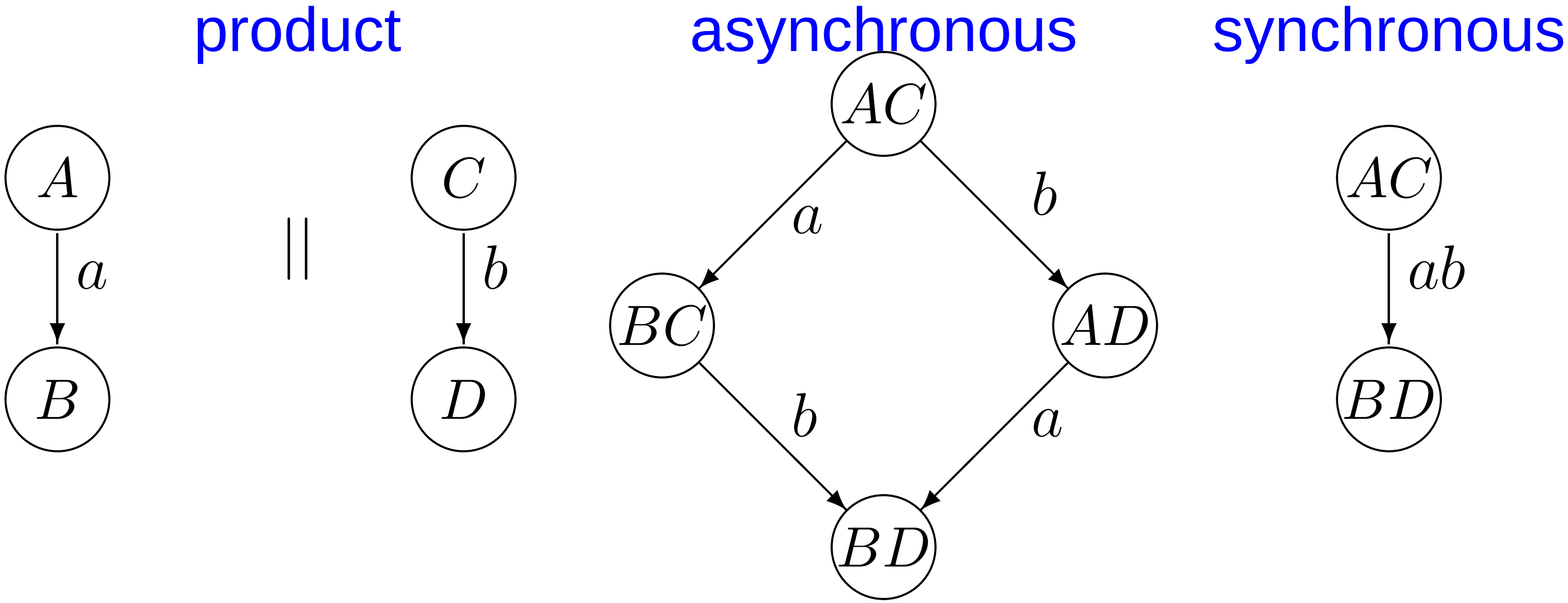
Several styles (imperative, data-flow,...)

Compiled parallelism (instead of concurrent)

most applications of synchronous programming are actually periodic ones.

Theory: SCCS (Milner)

Based on the synchronous product of automata:



product

asynchronous

synchronous

CCS (asynchronous) is a sub-theory of SCCS

Provides a theoretical justification of practice: Synchronous primitives are stronger, programming is easier

Further Justifications (Berry)

- No added non determinism:
  - easier debugging and test
  - less state explosion in formal verification
- Easier temporal reasoning:
  - synchronous steps provide a “natural” notion of logical time: in a concurrency framework `delay 5 seconds` means “a least 5 seconds” and is priority dependent.
  - Easier roll-back and recovery

Conclusion 1:

These advantages seem conclusive and justify the practices.

But . . .

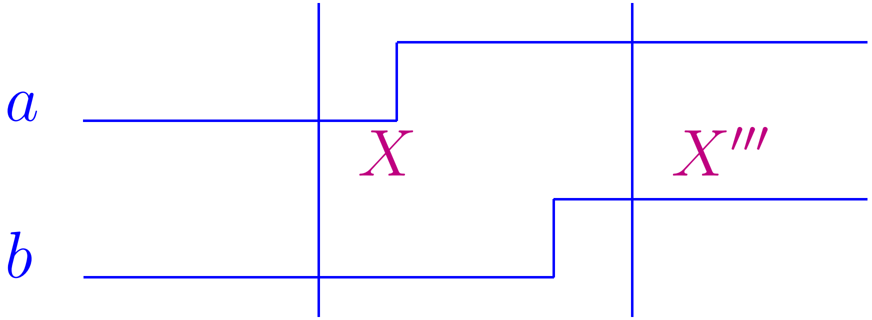
Synchronous Programming and Real-Time

- Real-Time is not Logical Time

- Distribution

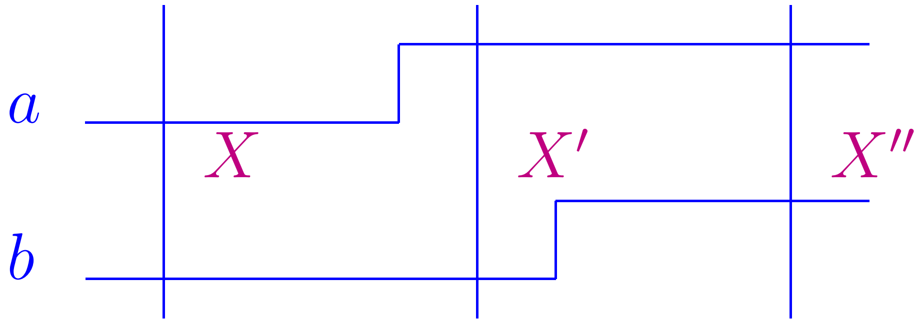
Real-Time is not Logical Time: Sampling Tuples

A possible sampling



. . . Real-Time is not Logical Time: Sampling Tuples

Another possible sampling



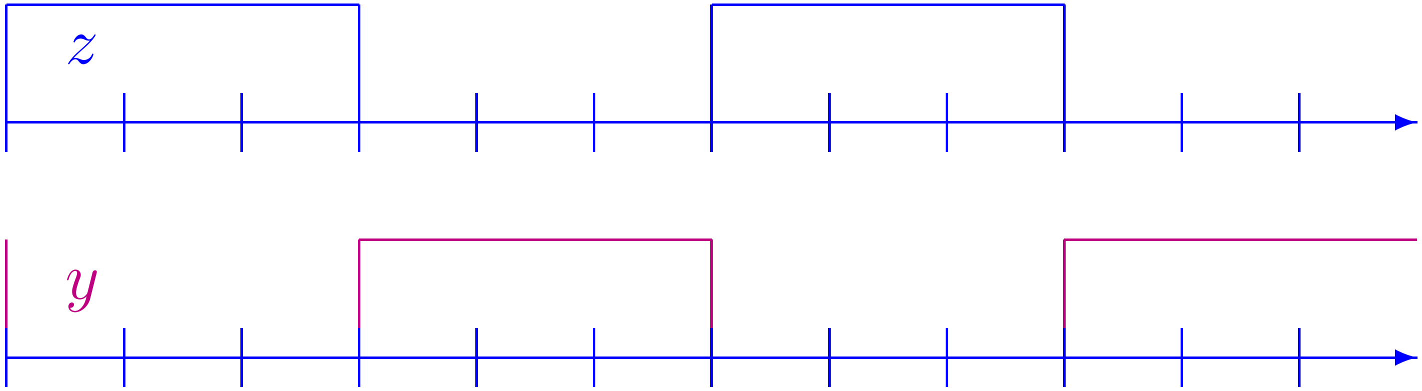
Non determinism, possible race

This was considered a side effect, but practitioners must take it into account.

Real-Time is not Logical Time: Outputs

example : mutual exclusion `always not (y and z)`

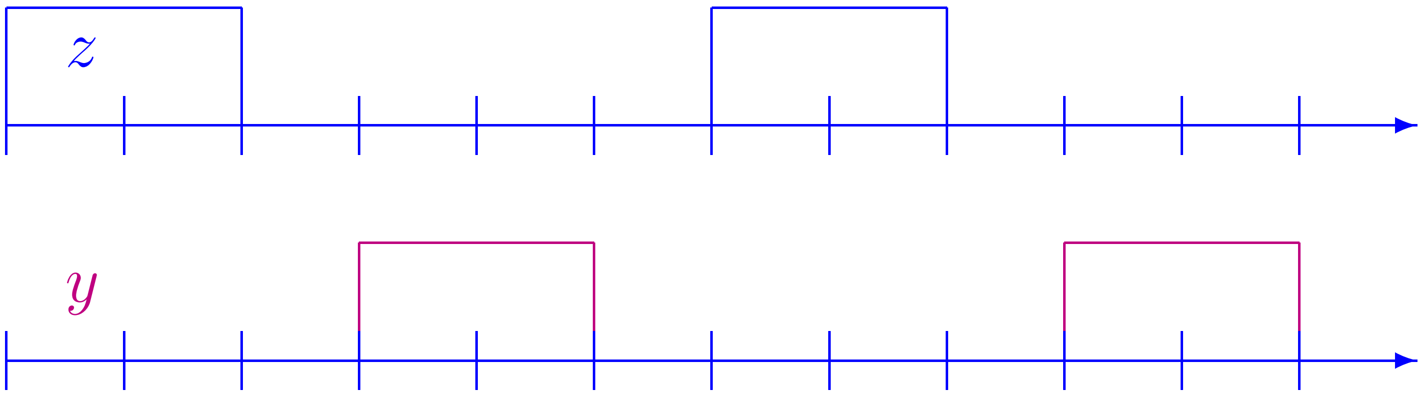
a non robust solution :



. . . Real-Time is not Logical Time: Outputs

example : mutual exclusion `always not (y and z)`

a robust solution :



z waits for y to go down before going up and conversely.

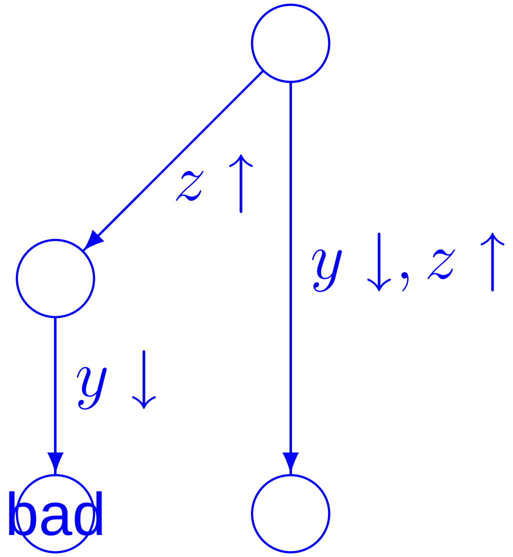
no race !

Races

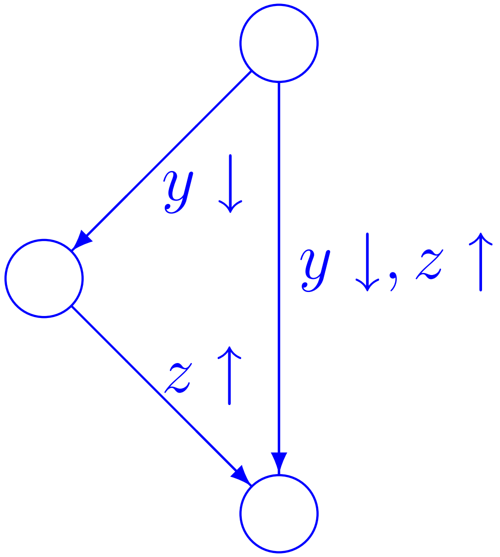
A race takes place when two signals can change at the same time or not, depending on variable delays.

A race is critical if different states can be reached, depending on which signal wins the race.

A critical race

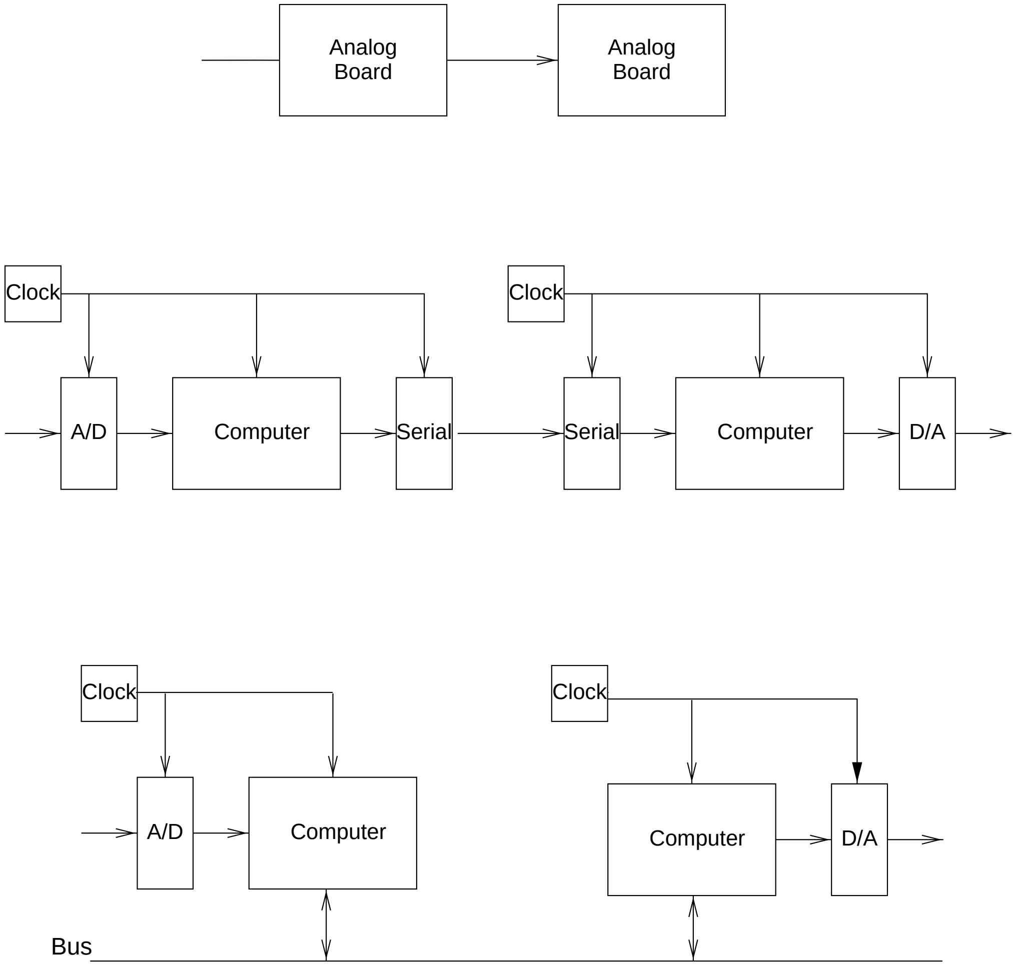


A non critical race



. . . Distribution

From networks of analog boards to local area networks



independent periodic clocks

synchronous programs

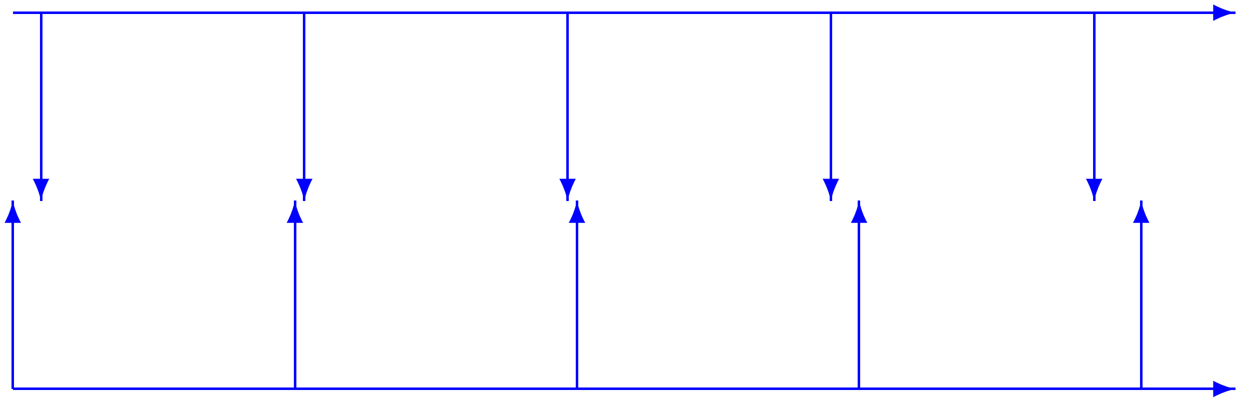
Interest

Autonomy, robustness

- Each computer is a complete one, including its own clock and even possibly its own power supply.

- Communication between computers is non-blocking, based on periodic reads and writes, akin to periodic sampling.

Some Consequences of Quasi Periodicity



Worst situation: reads occur just before writes ⇒ Bounded communication delays

Absolute time is lost: time-outs better than time ???

Sampling errors: data loss or duplication from time to time

Bounded Fairness

Provisional Conclusion 2

For robustness reasons, real-time and distribution require accommodating some asynchrony within the synchronous programming paradigm.

In the sequel we investigate some tracks taken by practioners in this purpose.

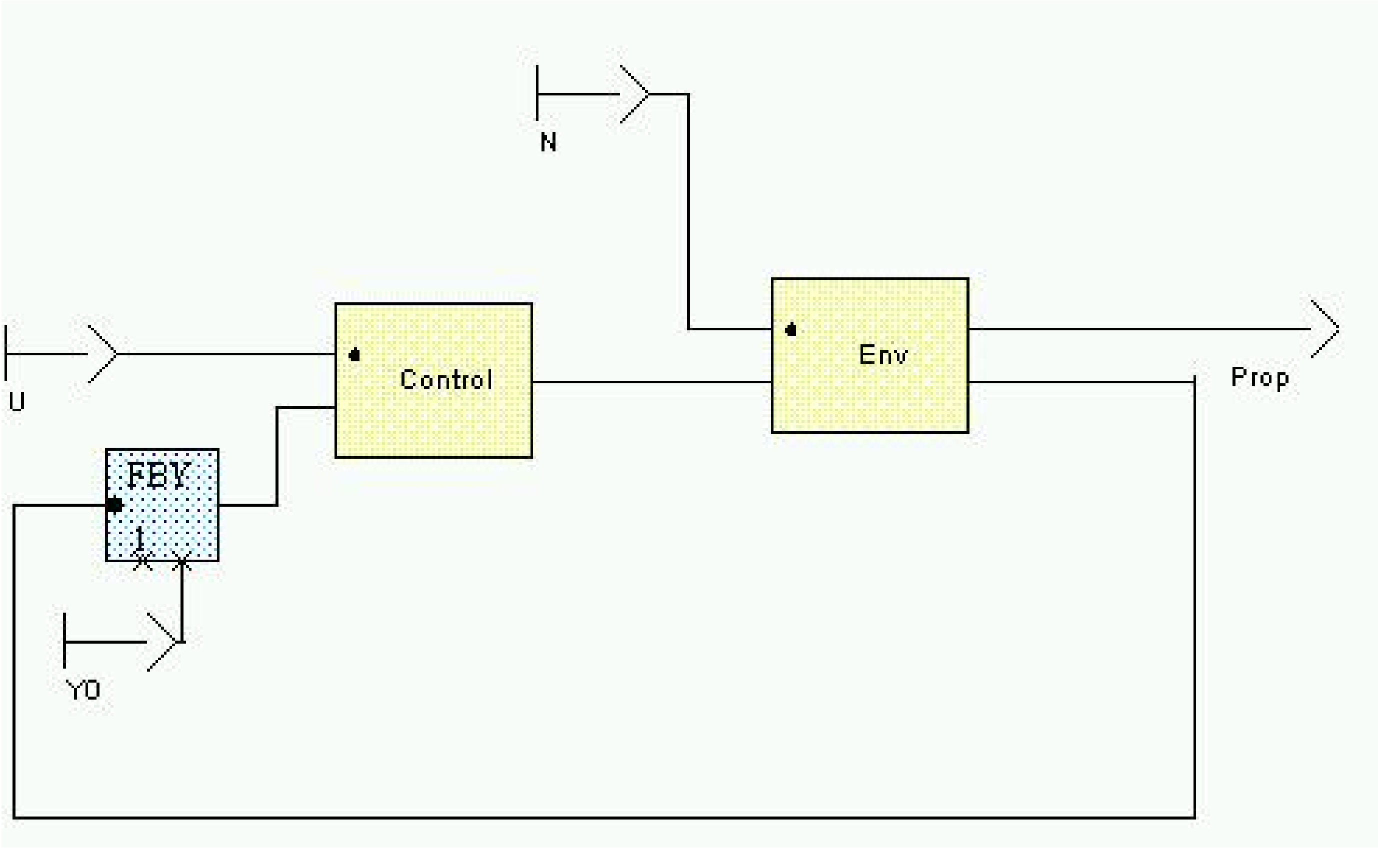
Real-Time Validation

Simulation, test, formal verification

- General Framework
- Centralised case
- Distributed case

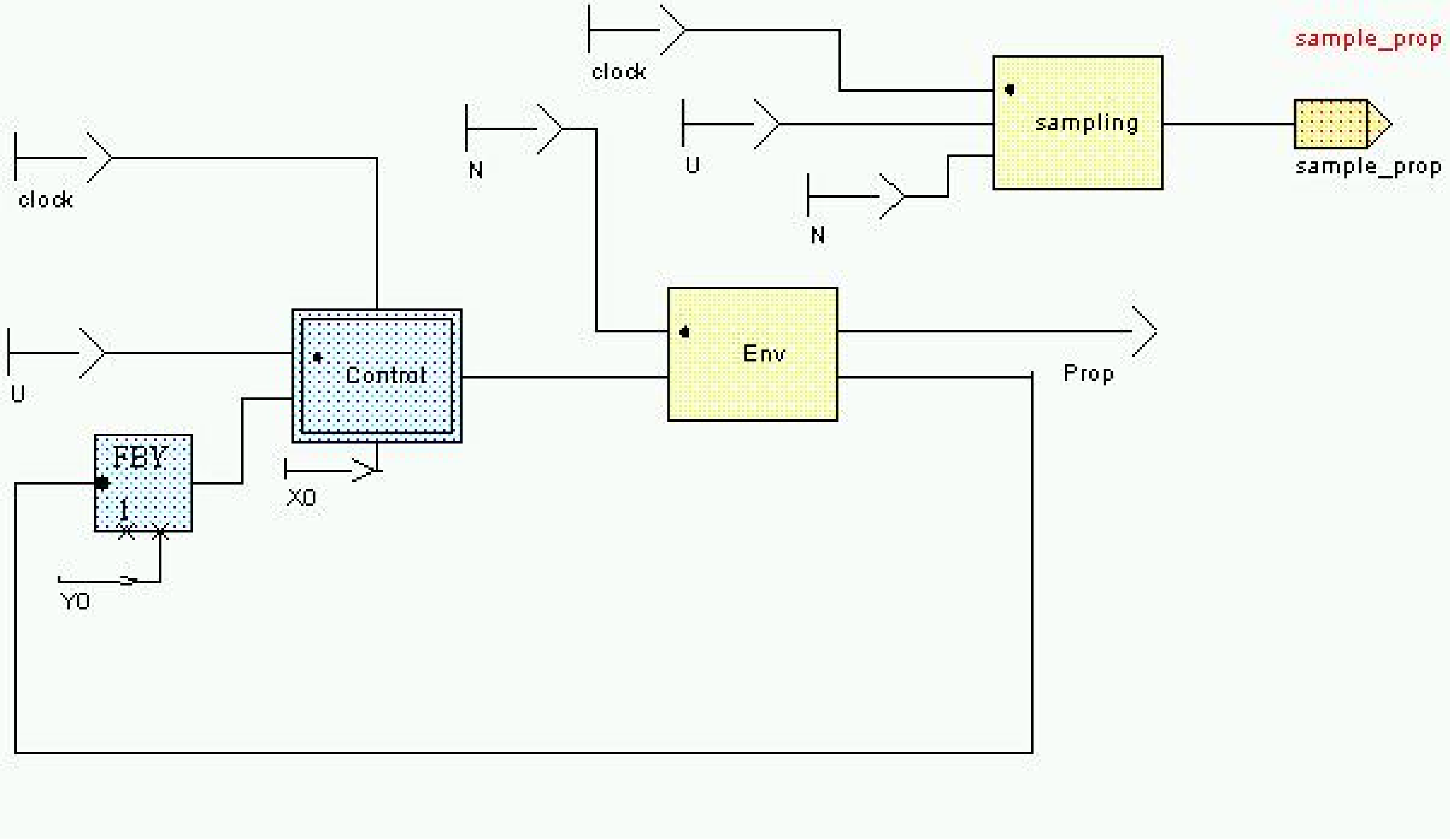
General Framework

Observer theory (Halbwach, Raymond 93) : safety properties can be expressed as synchronous programs outputing the truth value of the property.



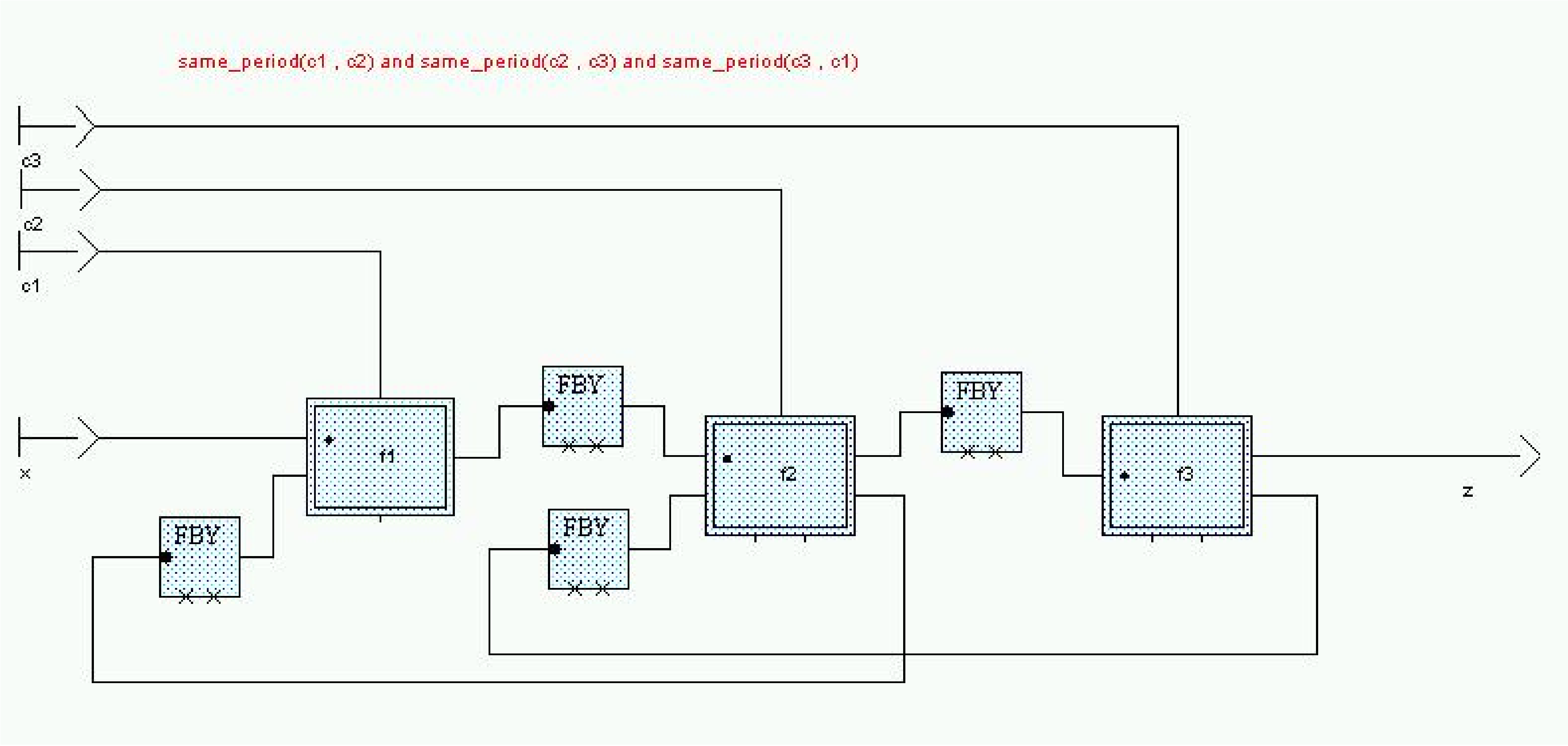
Centralised Case

Take into account the sampling non determinism



Distributed Case

Take into account distribution



same_period(c1 , c2) and same_period(c2 , c3) and same_period(c3 , c1)

Conclusion

Synchronous programming validation tools apply to real-time and distributed control systems.

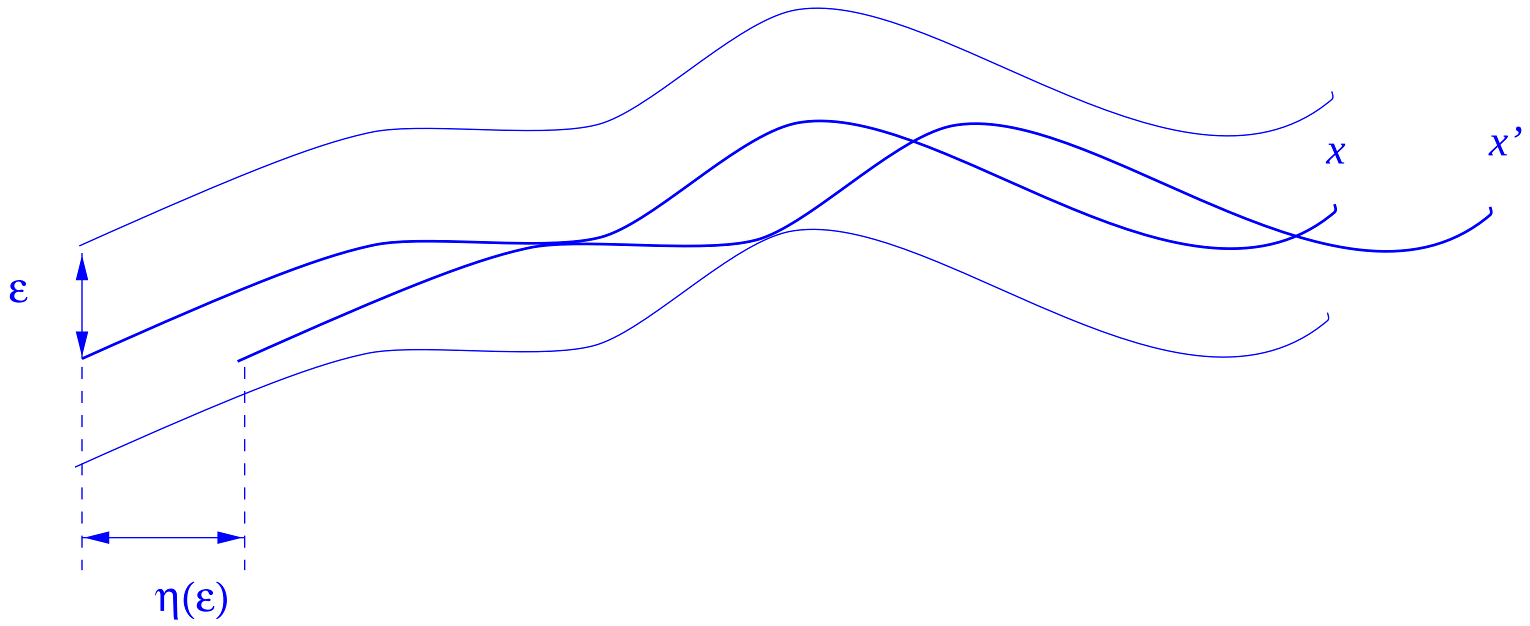
Efficiency issues ?

How to understand and construct robust systems ?

Asynchronous-Synchronous Programming: How to understand it ?

- Continuous Systems
- Non Continuous Systems
- (Mixed Systems)

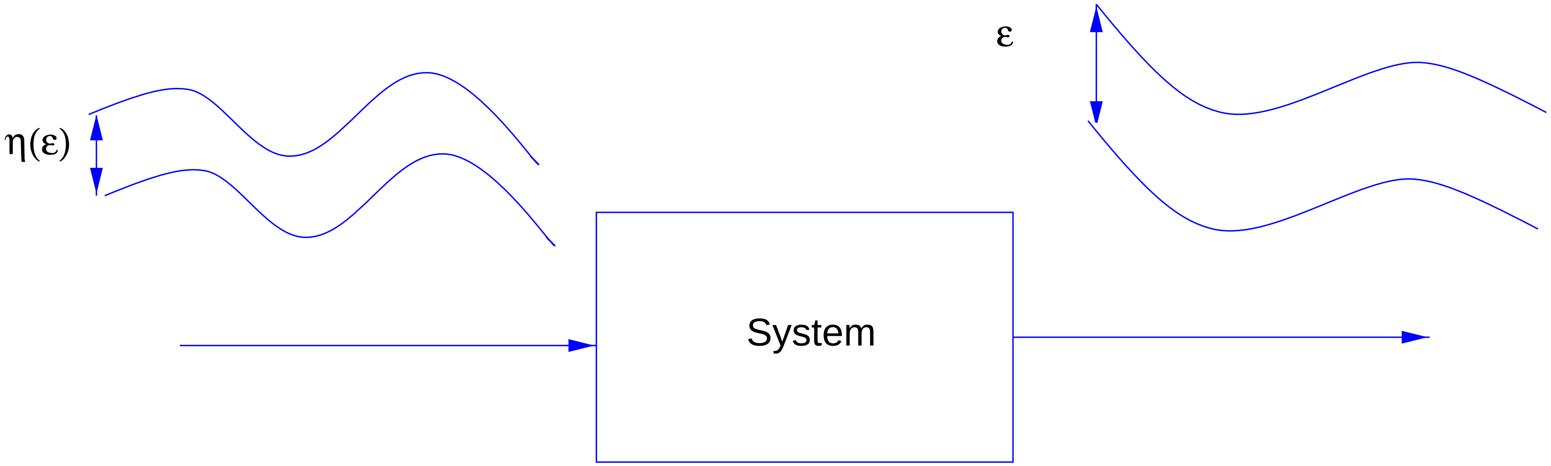
Uniformly Continuous Signals



$$\exists \eta_x > 0, \forall \epsilon > 0, \forall t, t', |t - t'| \leq \eta_x(\epsilon) \Rightarrow |x(t) - x(t')| \leq \epsilon$$

Bounded delays yield bounded errors

Uniformly Continuous Systems



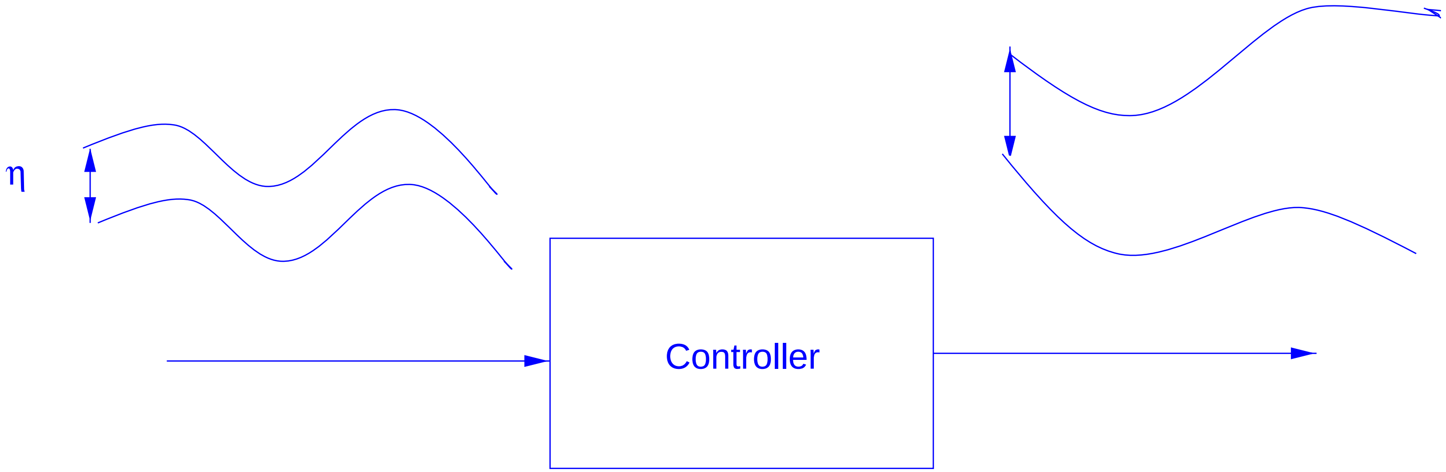
$$\exists \eta_S > 0, \forall \epsilon > 0, \forall x, x', ||x - x'||_\infty \leq \eta_S(\epsilon) \Rightarrow ||f(x) - f(x')||_\infty \leq \epsilon$$

Bounded errors yield bounded errors

But . . .

Even very simple controllers are not uniformly continuous.

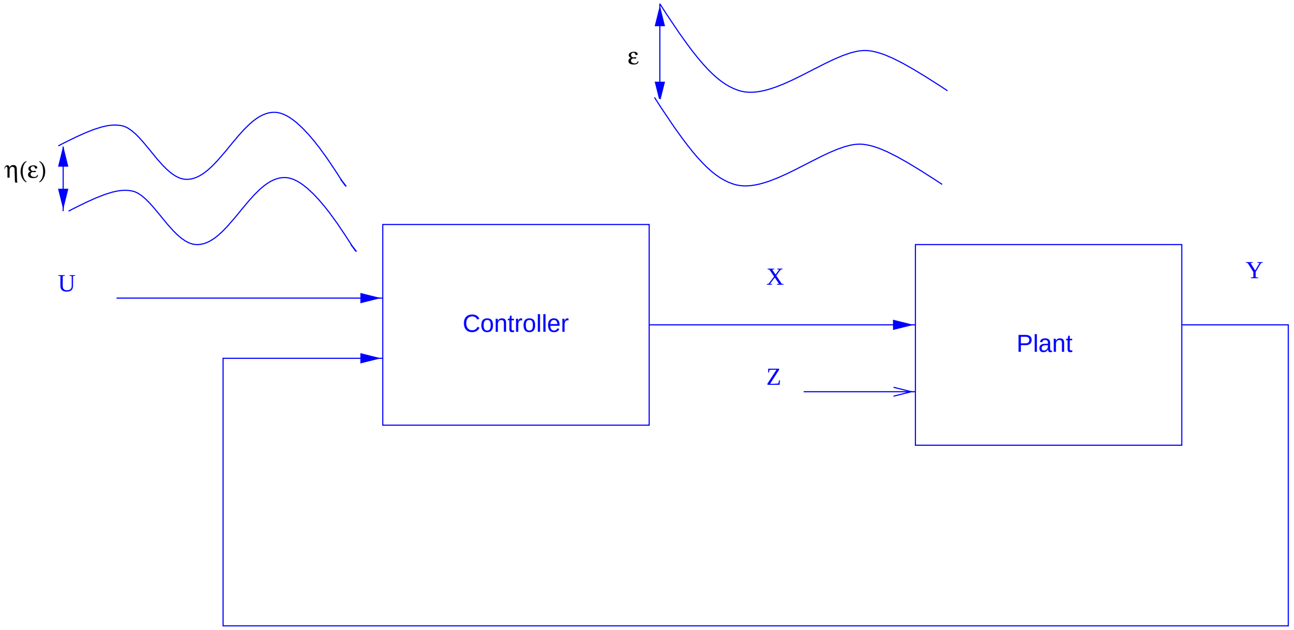
PID for instance



Bounded errors do not yield bounded errors

Stabilized Systems

The closed-loop system computes uniformly continuous signals



Bounded delays yield bounded errors

Doubts . . .

This casts a doubt on two wishful thoughts:

- composability
  - system properties are the mere addition of sub-system ones
- separation of concerns:
  - automatic control people specify
  - computer science people implement

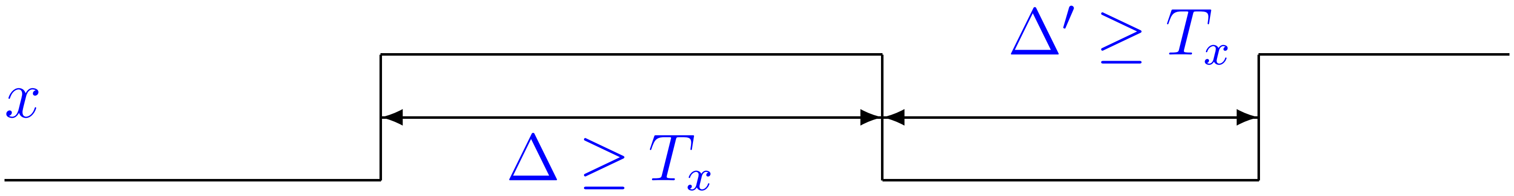
Critical control systems require a tight cooperation between both people

Non Continuous Systems

- Combinational Systems
- Robust Sequential Systems
- Sequential Systems

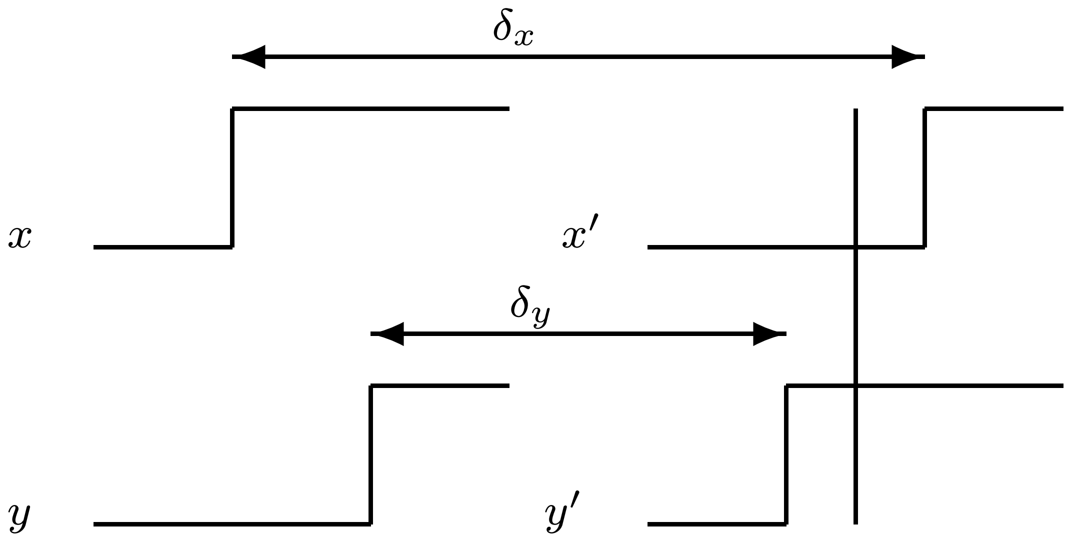
Uniform Bounded-Variability

There exists a minimum stable time T_x associated with a signal x.



But . . .

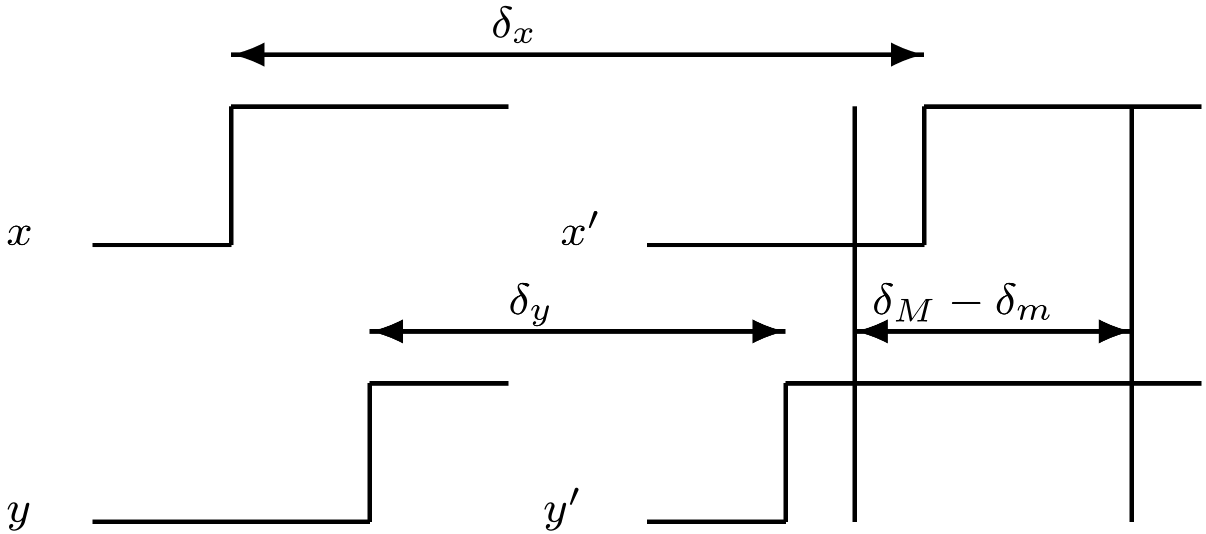
Delays on tuples do not yield delayed tuples



Solution : Confirmation functions

Confirmation Functions

When a component of a tuple changes, wait for some $\delta_M - \delta_m$ time before taking it into account.



If x', y' are (δ_m, δ_M) bounded images of x and y, then $confirm(x', y')$ is a delayed image of (x, y)

allows to retrieve the continuous framework

Robust Sequential Systems

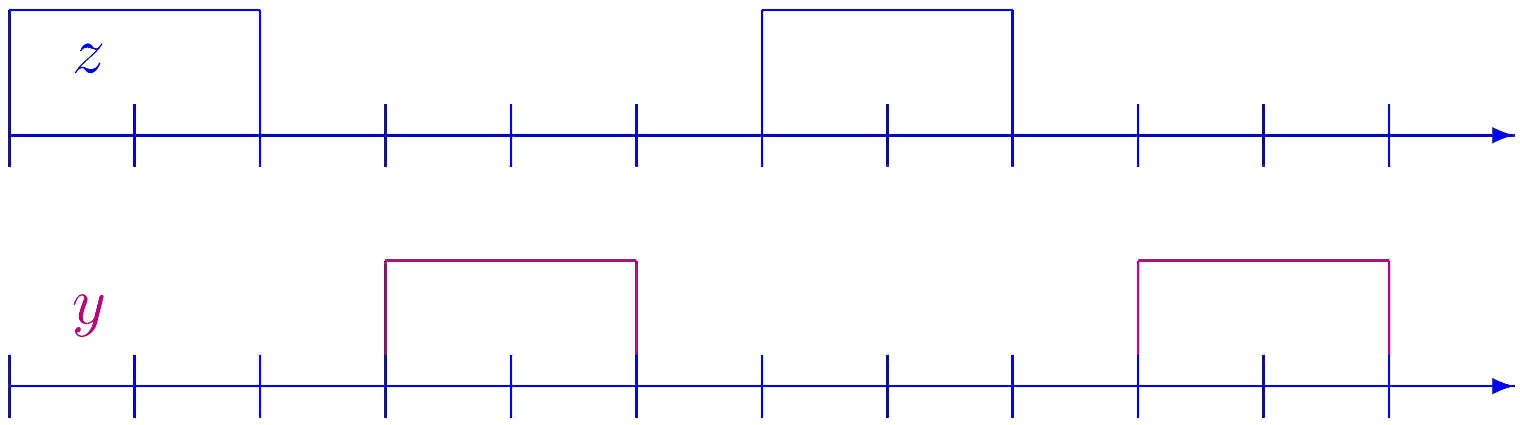
idea : avoid (critical) races

- between state variables : order insensitivity
- between inputs : confluence

Property checking

Asynchronous programming style

Asynchronous Programming Style



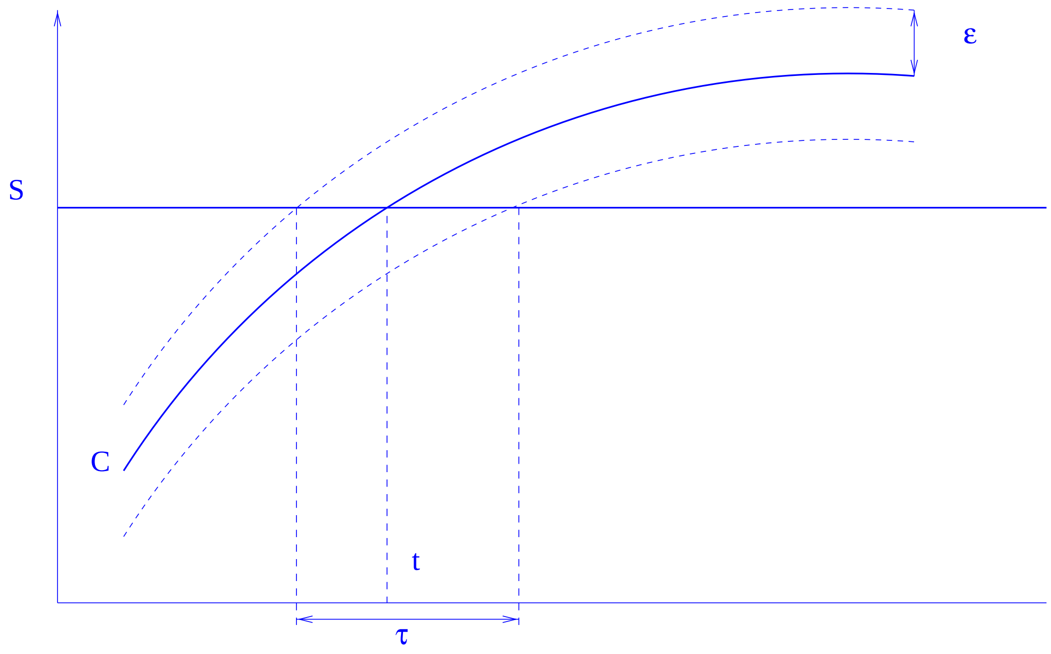
Insert causality chains disallowing races:

z waits for y to go down before going up and conversely.

$$\begin{matrix} not\ y \\ not\ z \end{matrix} \quad ((\rightarrow y \rightarrow not\ y)^*(\rightarrow z \rightarrow not\ z)^*)^*$$

Mixed Systems

Example : Threshold crossing



Relates errors and delays : $\tau = \dfrac{2\epsilon}{|C'(t)|}$

possibly unbounded delays !

Conclusion

- Some insight on techniques used in practice.
- Maybe useful for designers and certification authorities (Crisys Esprit Project)
- An attempt to draw the attention of the Computer Science community on these important problems.

Questions

- Are there linguistic ways to robustness (asynchronous-synchronous languages)?
- How to safely encompass some event-driven computations within the approach?
- Is there a common framework encompassing both theories?

continuous	discrete
uniformly continuous signals	uniform bounded variability
uniformly continuous functions	robust systems
unstable systems	sequential non robust systems