

Switch to full screen

Logical-Time Contracts for Reactive Embedded Components

Lionel Morel,
Florence Maraninchi

Verimag

Grenoble, France

<http://www-verimag.imag.fr/~lmorel>

Introduction

Context: Development of reactive critical systems.

Our proposal :

- ▶ Help programmer give **local specifications** of composite systems
⇒ Apply **Design-By-Contracts** to reactive systems

Our goal :

- ▶ Reduce specification effort (through **reusability**)
- ▶ **Exploit local specs** in a development environment plugged to validation tools
- ▶ Allow **early execution** of partially specified programs

Summary

- ▶ **Reactive Systems**
- ▶ **A Synchronous Language**
- ▶ **Design-by-Contract**
- ▶ **Contracts for Reactive Systems**
- ▶ **Exploiting Contracts**
- ▶ **Related Works**

Summary

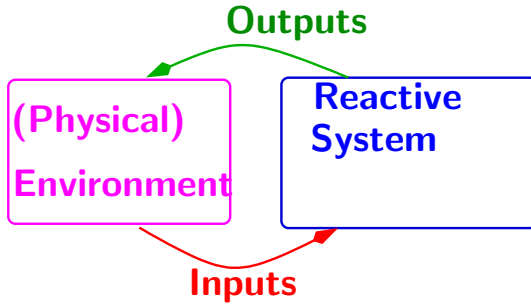
- ▶ **Reactive Systems**
- ▶ **A Synchronous Language**
- ▶ Design-by-Contract
- ▶ Contracts for Reactive Systems
- ▶ Exploiting Contracts
- ▶ Related Works

Reactive systems and synchronous approach

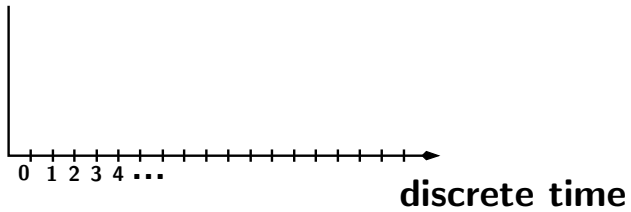
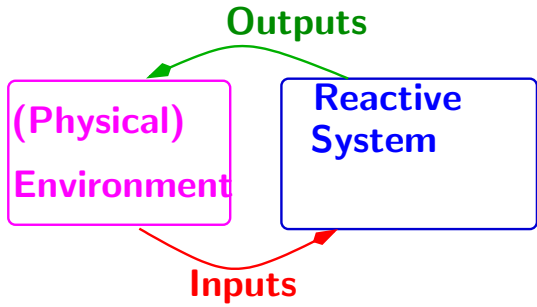
(Physical)
Environment

Reactive
System

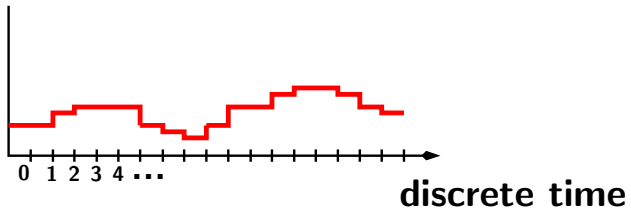
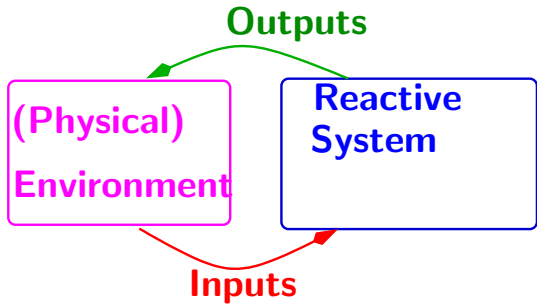
Reactive systems and synchronous approach



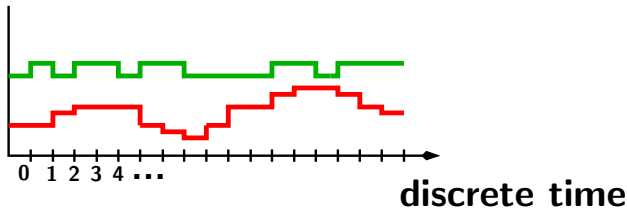
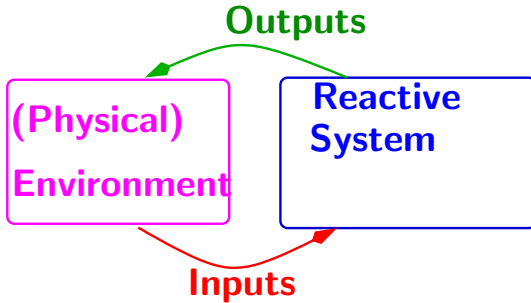
Reactive systems and synchronous approach



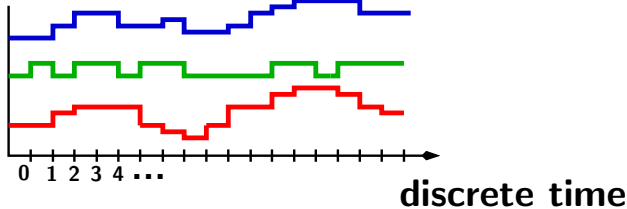
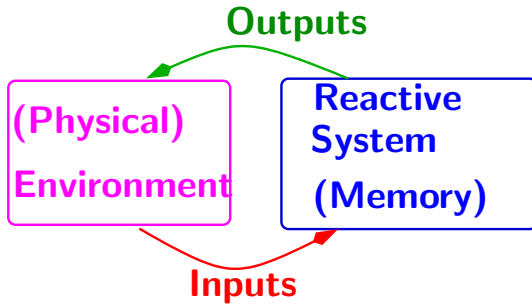
Reactive systems and synchronous approach



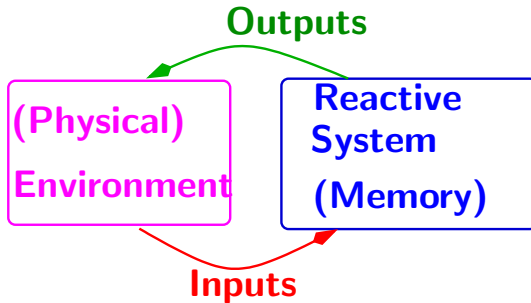
Reactive systems and synchronous approach



Reactive systems and synchronous approach

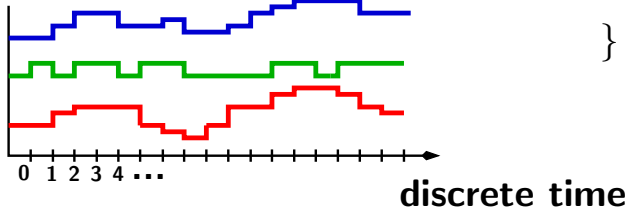


Reactive systems and synchronous approach

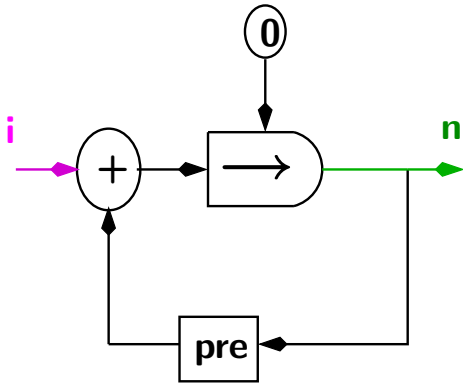


Standard Execution Scheme:

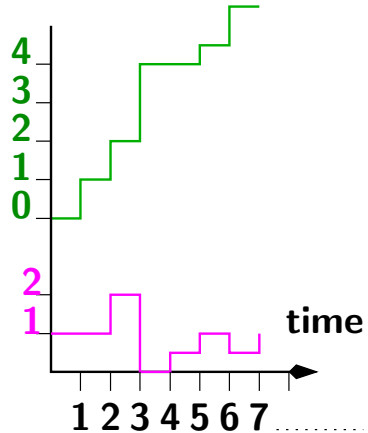
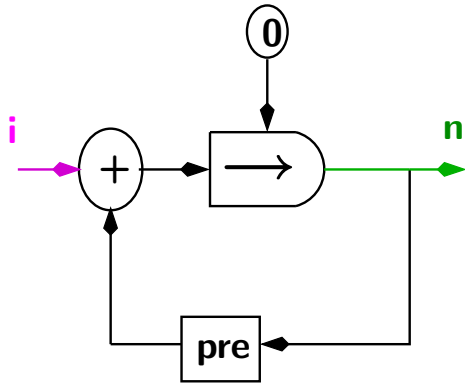
```
while(true){
    Read Inputs();
    Compute Outputs();
    Update Memory();
}
```



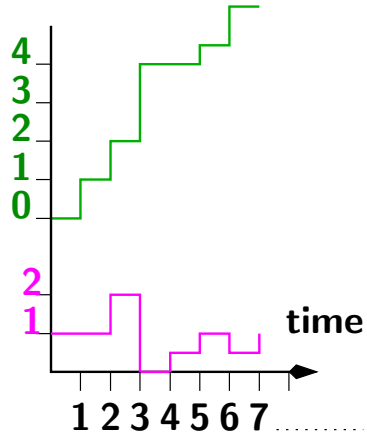
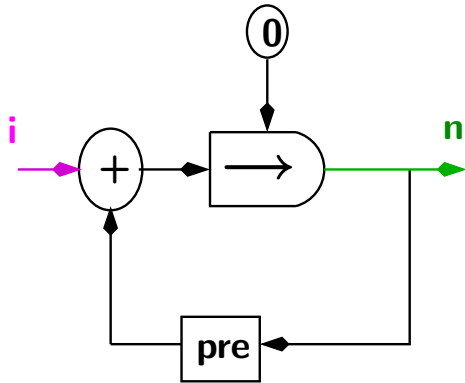
The synchronous language Lustre



The synchronous language Lustre

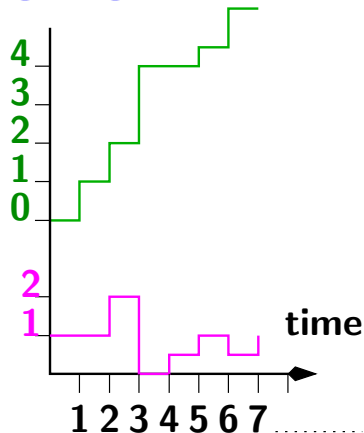
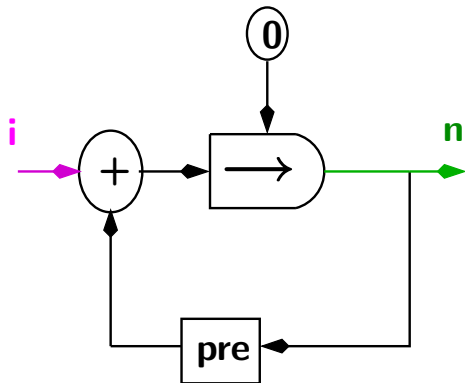


The synchronous language Lustre



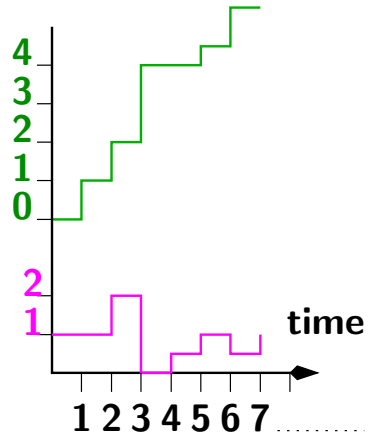
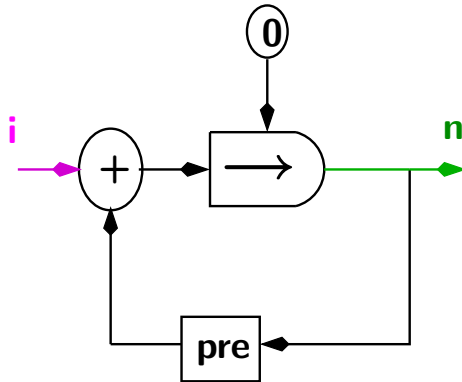
▶ formal definition \Rightarrow model-checker, test tools, ...

The synchronous language Lustre



- ▶ formal definition \Rightarrow model-checker, test tools, ...
- ▶ **Scade** = commercial IDE based on Lustre

The synchronous language Lustre



- ▶ formal definition \Rightarrow model-checker, test tools, ...
- ▶ **Scade** = commercial IDE based on Lustre
- ▶ Used by : Airbus, Schneider, CS-Transport, etc...

Summary

- ▶ Reactive Systems
- ▶ A Synchronous Language
- ▶ **Design-by-Contract**
- ▶ Contracts for Reactive Systems
- ▶ Exploiting Contracts
- ▶ Related Works

Design-by-Contract

Consider **behavioral contracts** (aka **functional contracts**) as introduced in OOP :

- ▶ separation of **assumption** (pre) and **guarantee** (post) conditions (for each method),
 - ▶ pre = at beginning of a call to the method
 - ▶ post = at the end of a call to the method
- ▶ use of **invariance** condition = before/after each call to **any** method in the class

Design-by-Contract - Example

Stack component (with limited nb of elements):

```
class Stack{
  private int nbElements;
  /** invariant nbElements >= 0
    and nbElements <= MaxNbElements */
  void push(int element){
    /** assume nbElements < MaxNbElements */
    /** guarantee nbElements != 0
      && topOfStack() == element; */
    ...
  }
}
```

Summary

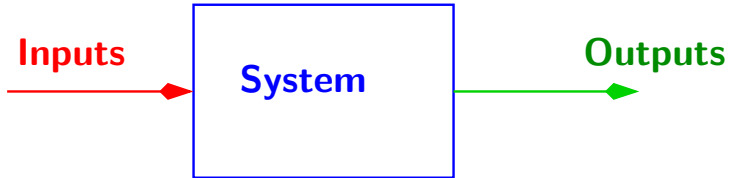
- ▶ Reactive Systems
- ▶ A Synchronous Language
- ▶ Design-by-Contract
- ▶ **Contracts for Reactive Systems**
- ▶ Exploiting Contracts
- ▶ Related Works

Contracts for Reactive Systems

Successive Executions of the same piece of code

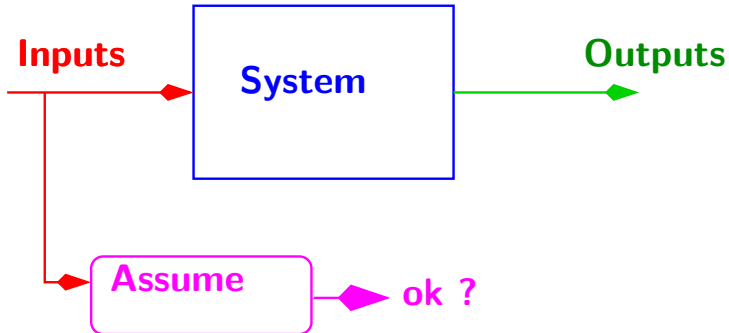
Contracts for Reactive Systems

Successive Executions of the same piece of code



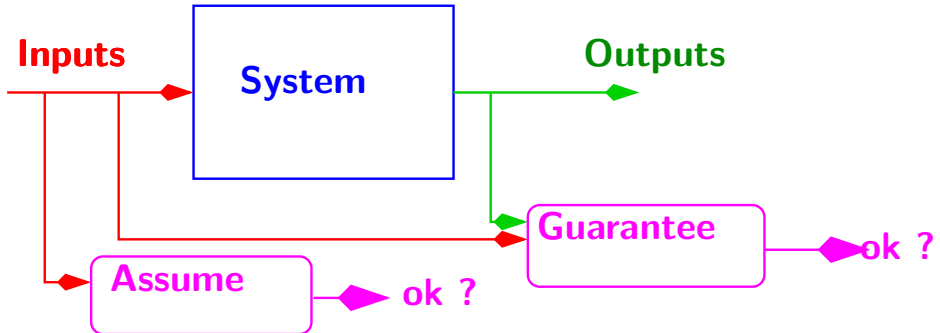
Contracts for Reactive Systems

Successive Executions of the same piece of code



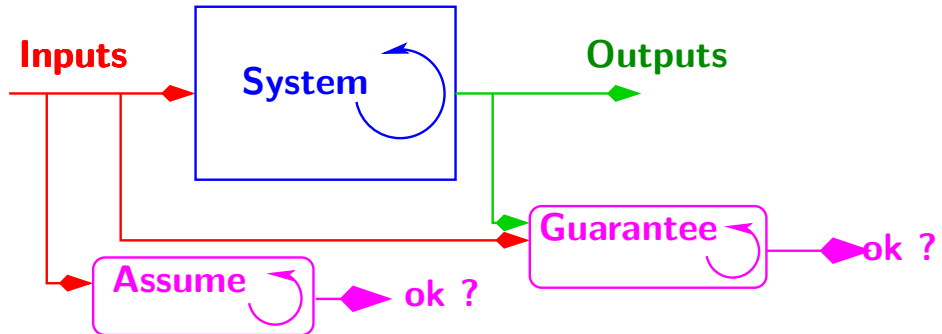
Contracts for Reactive Systems

Successive Executions of the same piece of code



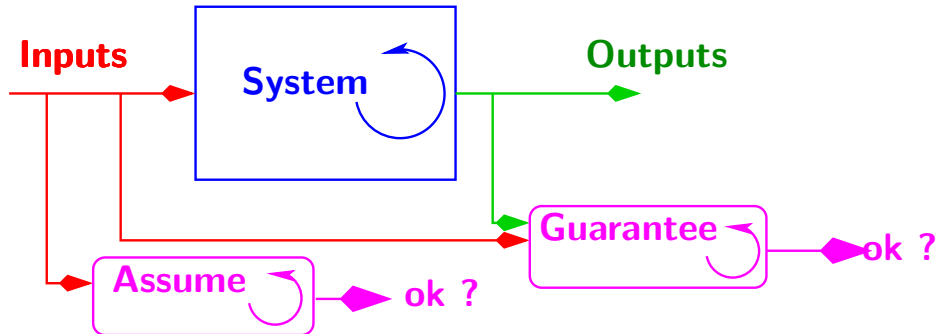
Contracts for Reactive Systems

Successive Executions of the same piece of code



Contracts for Reactive Systems

Successive Executions of the same piece of code

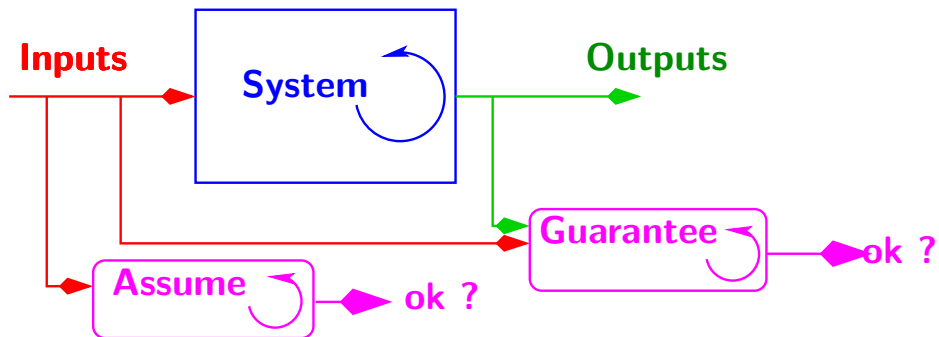


Meaning:

As long as input flows satisfy A , output flows satisfy G .

Contracts for Reactive Systems

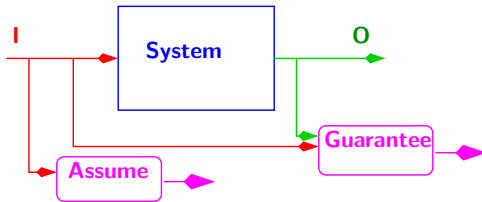
Successive Executions of the same piece of code



Meaning:

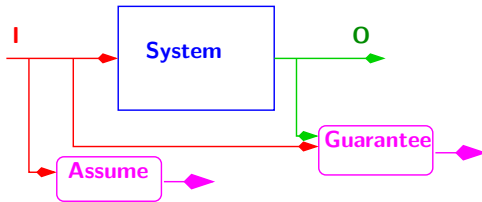
As long as input flows satisfy A , output flows satisfy G .
 A and G are **synchronous observers**

Contracts for Reactive Systems - Example



- ▶ A : I is always increasing
- ▶ G : O is not true more than twice in a row

Contracts for Reactive Systems - Example



- ▶ A : I is always increasing
- ▶ G : O is not true more than twice in a row

- ▶ A and G need local memory
- ▶ Need same power of expression for contracts as for the system (use **the same language**)

Industrial Case Study

Airbus case-study

Airplane's **Electric Load Management Unit**:

- ▶ **big** application ≈ 350 SCADE pages (a lot!)
- ▶ Library of reusable components :
 - ▶ 30 very basic components used "**everywhere**"
 - ▶ many bigger components reused

Industrial Case Study

Airbus case-study

Airplane's **Electric Load Management Unit**:

- ▶ **big** application ≈ 350 SCADE pages (a lot!)
- ▶ Library of reusable components :
 - ▶ 30 very basic components used "**everywhere**"
 - ▶ many bigger components reused

reusability \Rightarrow Contracts very useful **during specification**

ELMU - Example component

We want to describe a component such that :



- ▶ **assume** : a not true more than twice in a row
- ▶ **guarantee** : b not true more than three times in a row

ELMU - Example component

We want to describe a component such that :



- ▶ **assume** : a not true more than twice in a row
- ▶ **guarantee** : b not true more than three times in a row

Spec given in natural language while you can describe it with a contract (2 observers).

Summary

- ▶ Reactive Systems
- ▶ A Synchronous Language
- ▶ Design-by-Contract
- ▶ Contracts for Reactive Systems
- ▶ **Exploiting Contracts**
- ▶ Related Works

Exploiting contracts

Contracts help **specifying** systems.

We can also exploit them for :

- ▶ formal **verification** of components
- ▶ **early execution** of under-specified systems

Verification

Many questions can be asked:

- ▶ **Is "this" contract implementable at all?**

Verification

Many questions can be asked:

- ▶ Is "this" contract implementable at all?
- ▶ Does a given implementation satisfy "this" contract?

Verification

Many questions can be asked:

- ▶ Is "this" contract implementable at all?
- ▶ Does a given implementation satisfy "this" contract?
- ▶ Can I plug 2 components together?

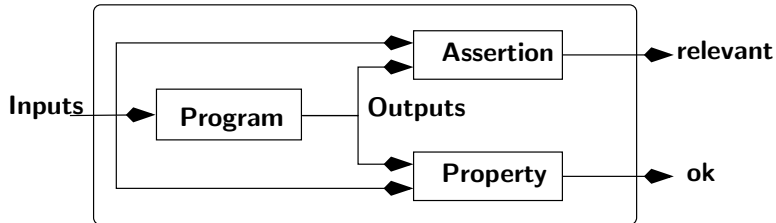
Verification

Many questions can be asked:

- ▶ Is "this" contract implementable at all?
- ▶ Does a given implementation satisfy "this" contract?
- ▶ Can I plug 2 components together?
- ▶ What is the contract of a composition of 2 components?

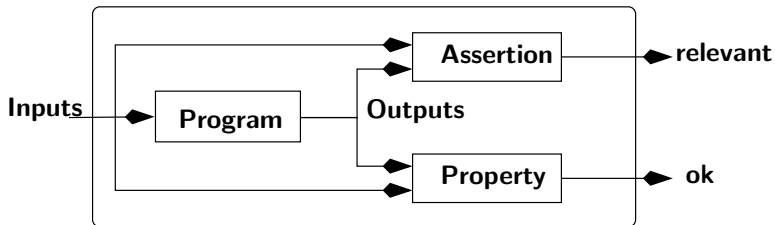
Verification - cont'd

All these questions can be seen as **instances of the classical verification problem based on observers:**



Verification - cont'd

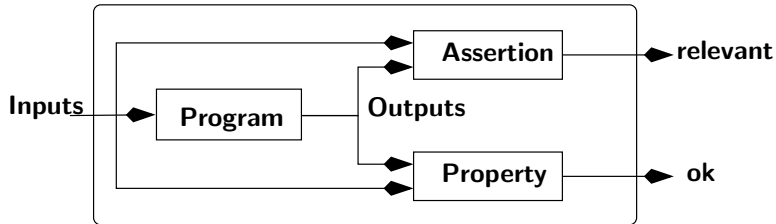
All these questions can be seen as **instances of the classical verification problem based on observers**:



Ex: Taking A as **assertion** and G as **property** answers question "Does a given implementation satisfy this contract ?"

Verification - cont'd

All these questions can be seen as **instances of the classical verification problem** based on **observers**:



Ex: Taking A as **assertion** and G as **property** answers question "Does a given implementation satisfy this contract ?"

⇒ **use standard validation tools** (test, model-checking, etc...) **for free.**

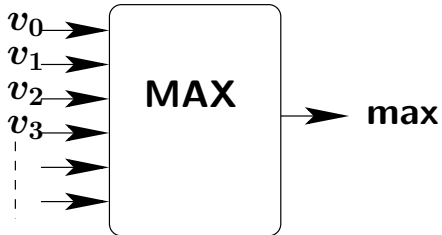
Early execution (work in progress...)

Don't need to wait **full implementation** for **simulation**.

Early execution (work in progress...)

Don't need to wait **full implementation** for **simulation**.

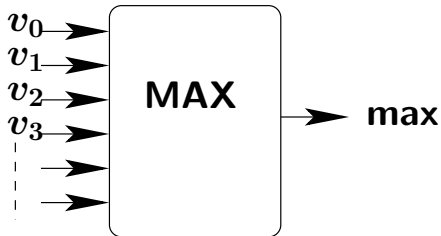
Example :



Early execution (work in progress...)

Don't need to wait **full implementation** for **simulation**.

Example :



If wanting to simulate
MAX:

- ▶ Don't need to know **how** max is computed
- ▶ Just need to know that $\forall i. max \geq v_i$

Summary

- ▶ Reactive Systems
- ▶ A Synchronous Language
- ▶ Design-by-Contract
- ▶ Contracts for Reactive Systems
- ▶ Exploiting Contracts
- ▶ **Related Works**

Related Works

- ▶ On **description of contracts**
 - ▶ OOP : contracts for Eiffel[Meyer], Java[JAssert]
 - ▶ Hardware systems : don't cares[Brayton]
- ▶ On **formal verification of components with contracts** (**assume/guarantee reasoning**) :
 - ▶ for concurrent systems in general:
[Misra/Chandy-81], [Abadi/Lamport-93],
[McMillan-97]
 - ▶ for data flow networks :
[Stølen-95], [Broy-95]

Conclusion

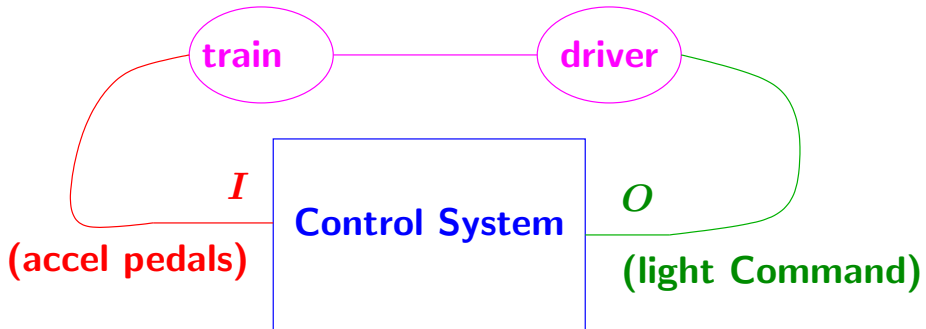
- ▶ Express contracts for reactive embedded components (in the same language as components themselves)
- ▶ Exploit them during development and validation

Perspectives

- ▶ Early execution is work in progress
- ▶ Contracts for asynchronous components

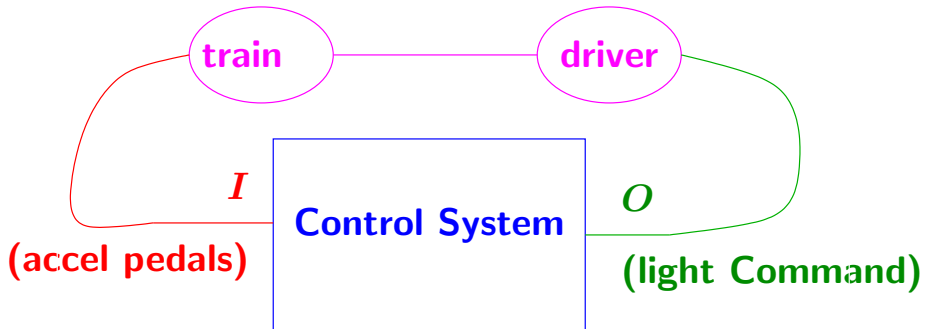
Bonus track :

Why *A* doesn't talk about outputs



Bonus track :

Why A doesn't talk about outputs



A : train stops at red light !

$$O_n = true \Rightarrow \neg I_{n+1}$$

Why A doesn't talk about outputs

allowing O in A :

- ▶ is more complex to write : you have to make sure to always talk about previous values of O s !
- ▶ is not more expressive : Outputs ultimately depend on inputs ! If you really need O s you can always re-define them in A