42: *The question of Components, Embedded Systems, and Everything*[1]

Florence Maraninchi, Tayeb Bouhadiba
www-verimag.imag.fr/~maraninx

Verimag-Synchrone / Inst. Nat. Polytechnique de Grenoble

November 17, 2006

[1] *Freely adapted from "The Hitchhiker's Guide to the Galaxy", D. Adams.*

1 42: Motivations and Approach

2 42: Definition

3 First Exercice: the Synchronous MoC

4 2nd Exercice: asynchronous processes on a monoprocessor with shared memory

5 Conclusions

Observing Existing Component-Based Frameworks

- Hardware (synchronous) components, called IPs, really exist. The sequential Boolean abstraction of the electric behavior is sufficient for component-based design.
- Software components really exist (at least in non concurrent frameworks). The central notion is also encapsulation.

Observing Existing Component-Based Frameworks

- Hardware (synchronous) components, called IPs, really exist. The sequential Boolean abstraction of the electric behavior is sufficient for component-based design.
- Software components really exist (at least in non concurrent frameworks). The central notion is also encapsulation.
- Compare with current practise in some domains of concurrent embedded system design: programming with Lustre/SCADE, SystemC/TLM for systems-on-a-chip, virtual prototypes of sensor networks, Ptolemy, the Architecture Analysis and Design Language (AADL), ...

A Remark on ... Using Synchronous Formalisms

Signal or Lustre are very good tools for the modular design and analysis of embedded systems (HW/SW). They are also good candidates for a component-based framework:

- The declarative style is close to the style of ADLs;
- asynchrony may be encoded in a synchronous formalism;
- code generation is well understood;
- time and concurrency are dealt with in a very precise way;
- automatic abstractions and analyses are possible;
- execution platforms and physical environments can be modeled, etc.

Why 42 ?

Isolate the main ideas of a component-based framework, focusing on:

- encapsulation and component protocols (like in OO frameworks) and/or assume-guarantee data specifications
- Conditions for an assemblage of components to be correct (like session types)
- how to build atomicity and to reason in concurrent frameworks?
- hierarchy: (components+wires+director) is a new component

42 is not yet-another-parallel-modeling formalism. It's not meant to be easy to use as a *language*, ...

Approach

- Behaviors are in the components
- The oriented connections are nothing more than wires (no memory, no synchronisation).
- The way components (connected by wires) behave together is defined by a director that characterizes the MoC, as in Ptolemy
- The director is a small "program" in terms of more basic operations

The director may be a model of a physical phenomenon (electricity in synchronous HW, an abstract non-deterministic model of the radio link for sensor networks, ...) or the code of an explicit scheduler in SW, or ...

1 42: Motivations and Approach

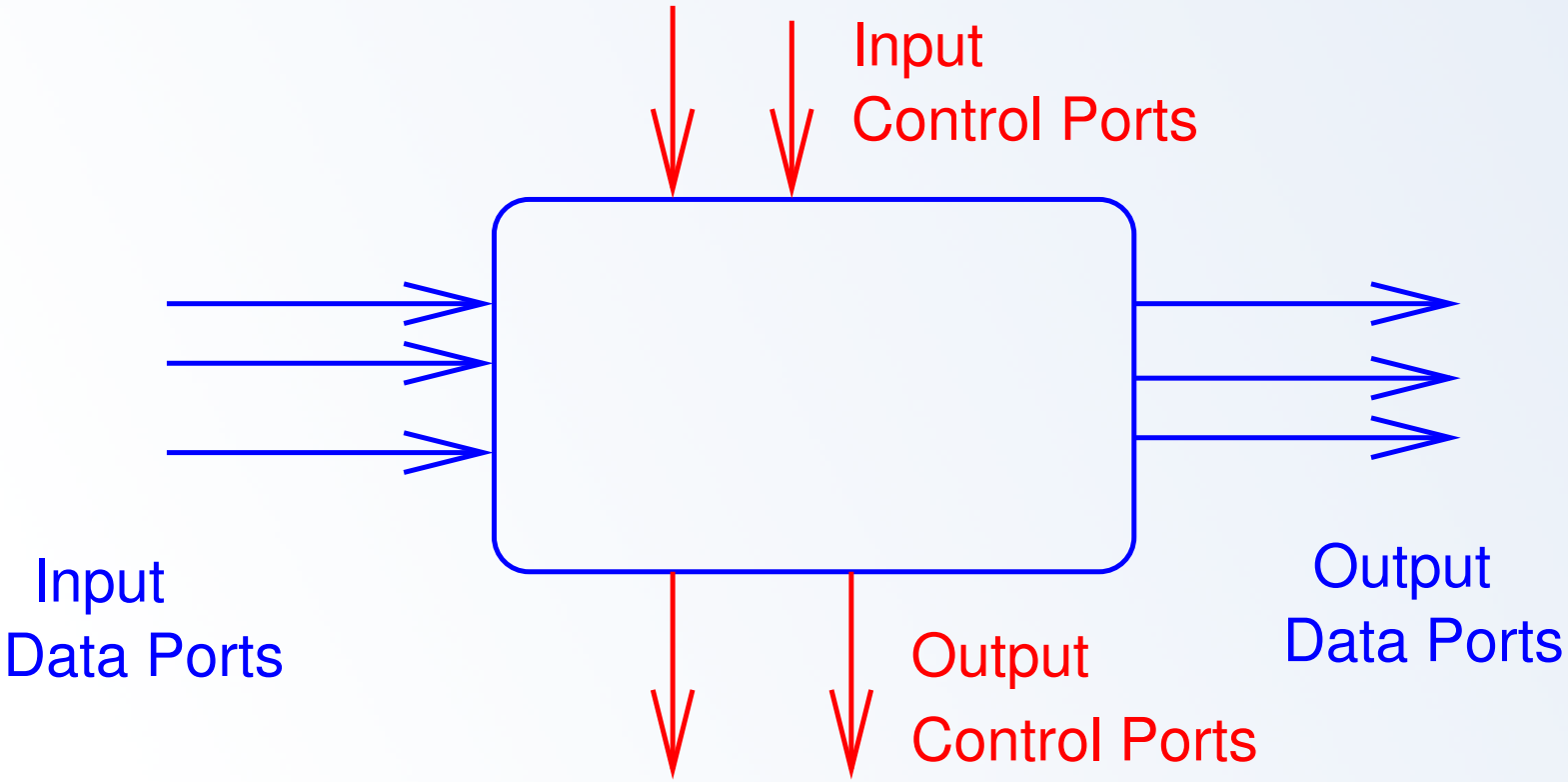
2 42: Definition

3 First Exercice: the Synchronous MoC

4 2nd Exercice: asynchronous processes on a monoprocessor with shared memory

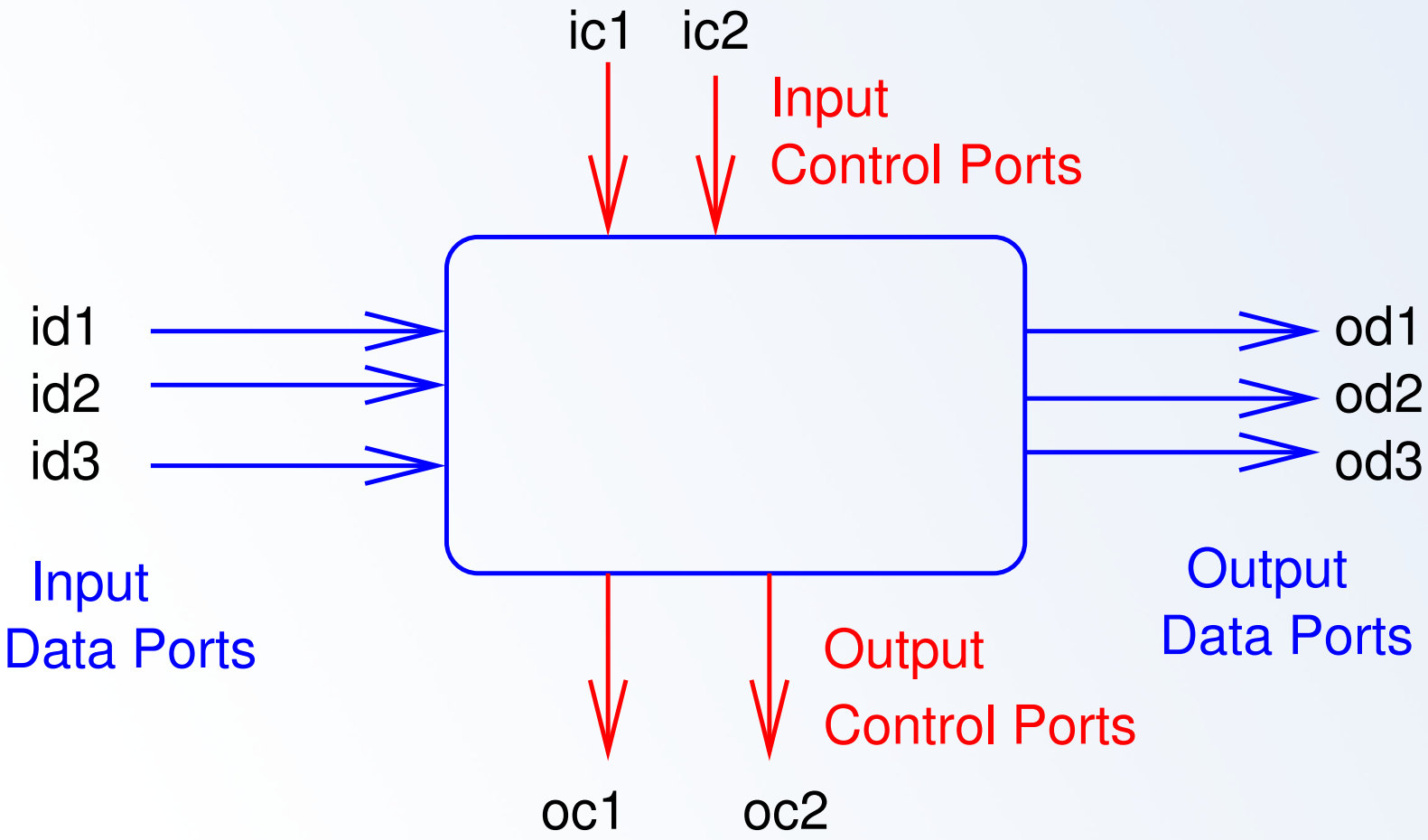
5 Conclusions

A Basic Component with a Self-Defined Notion of Atomicity



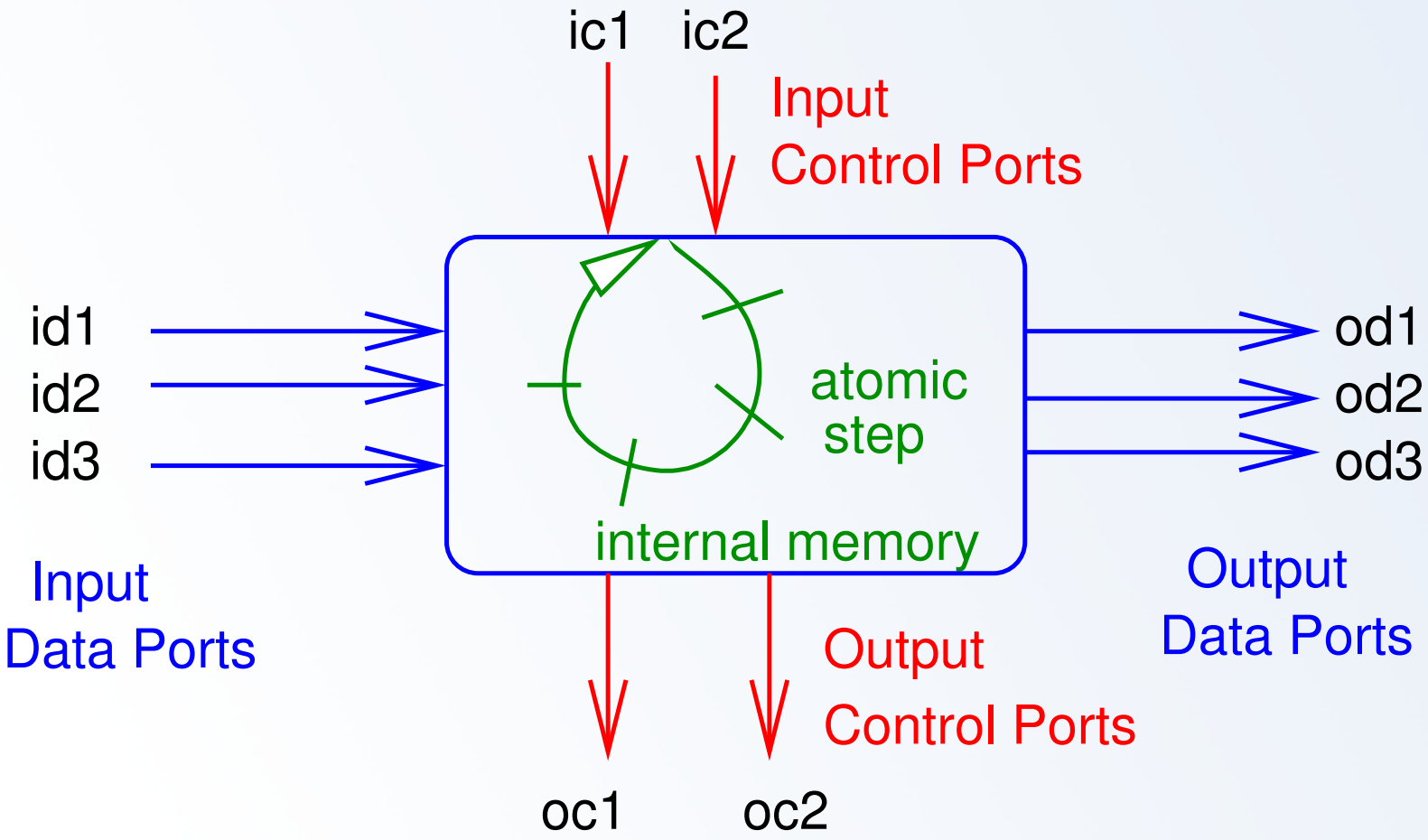
(not necessarily deterministic)

A Basic Component with a Self-Defined Notion of Atomicity



(not necessarily deterministic)

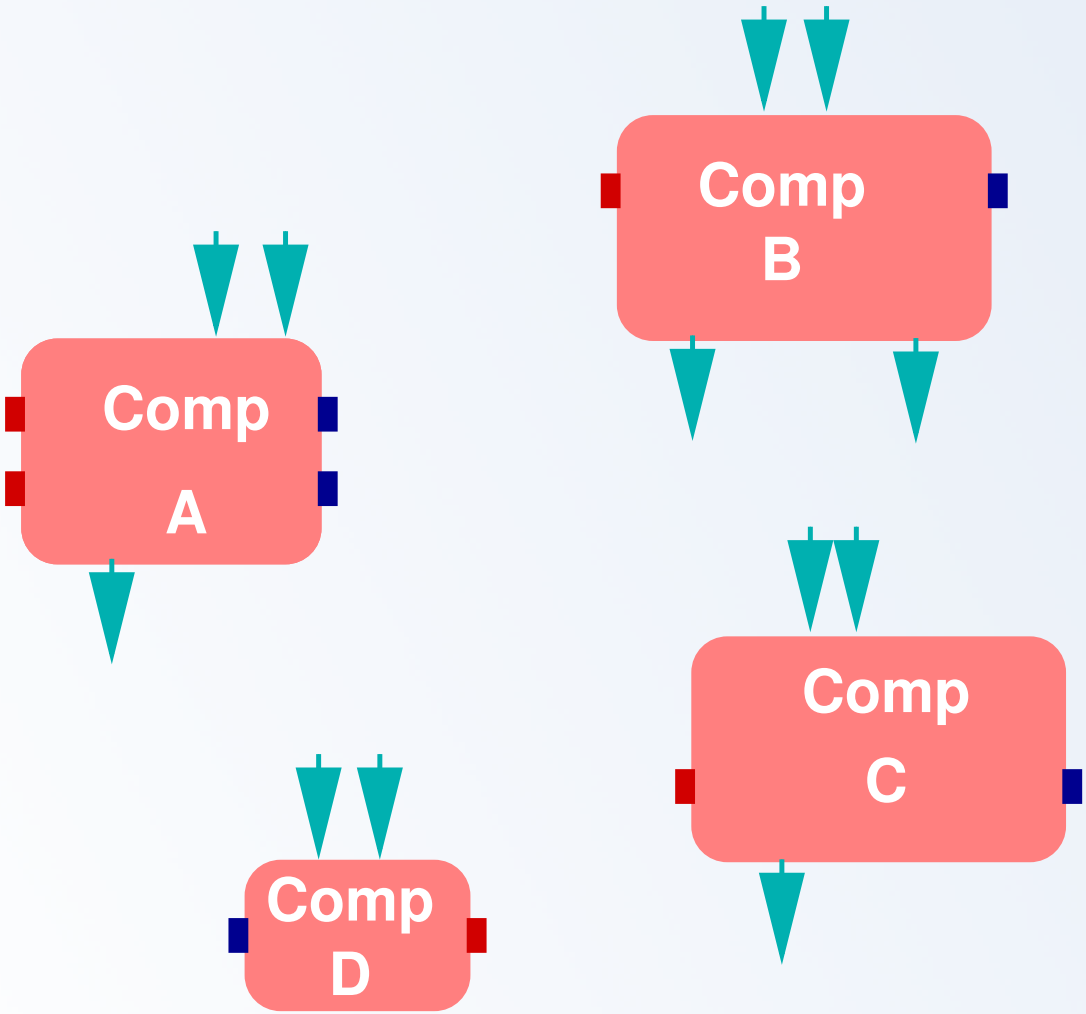
A Basic Component with a Self-Defined Notion of Atomicity



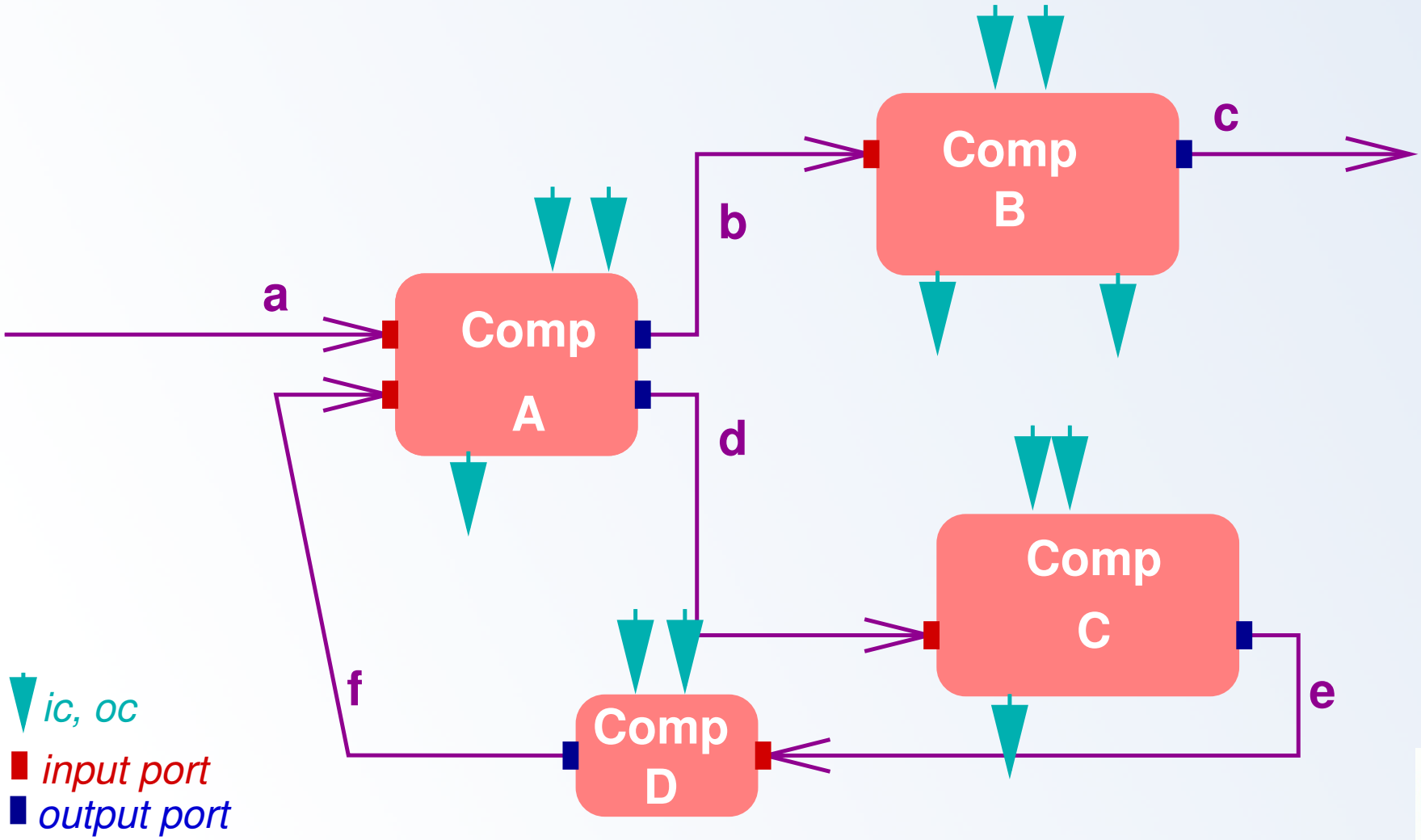
(not necessarily deterministic)

Compositions: The global picture

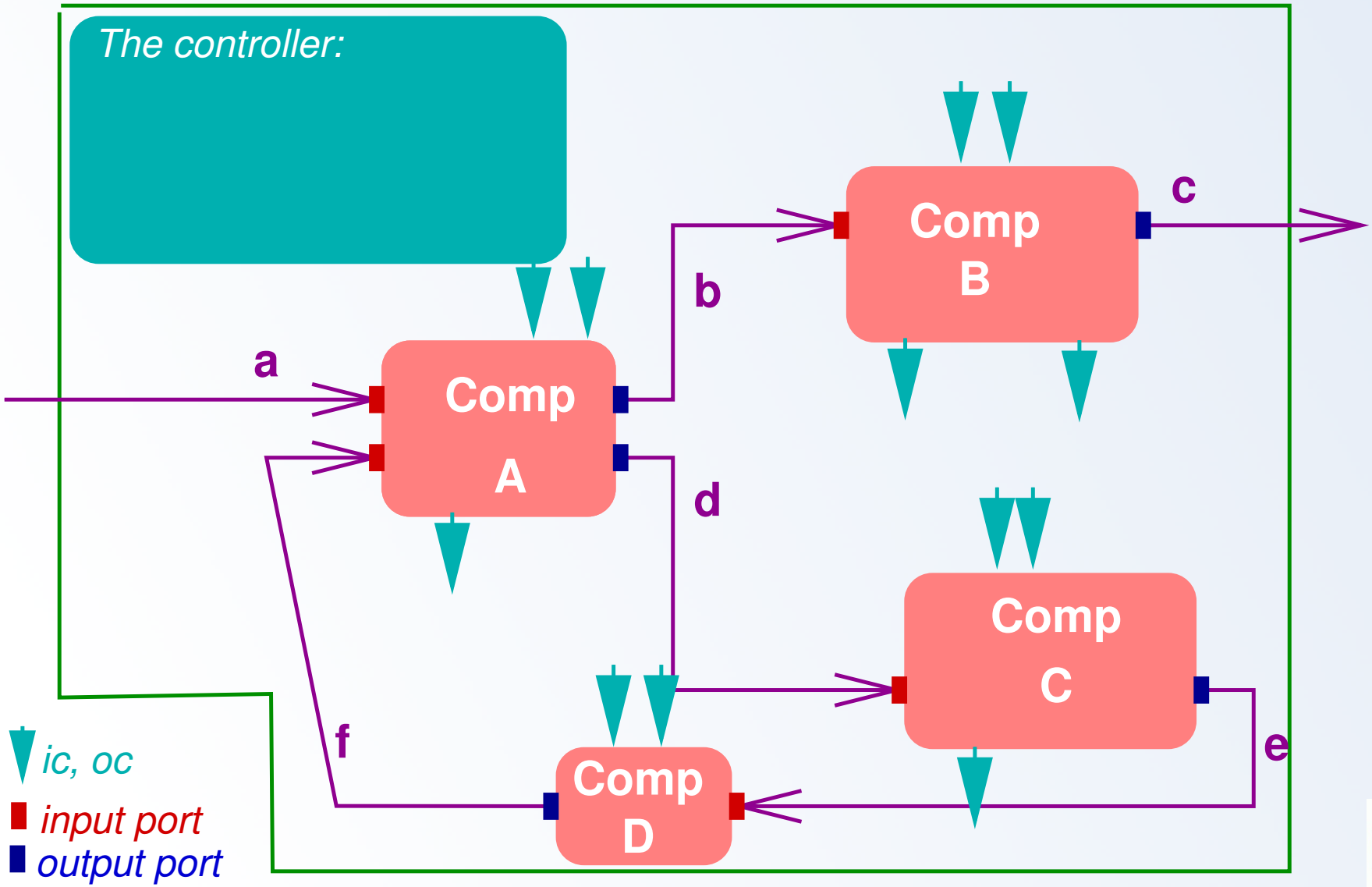
Compositions: The global picture



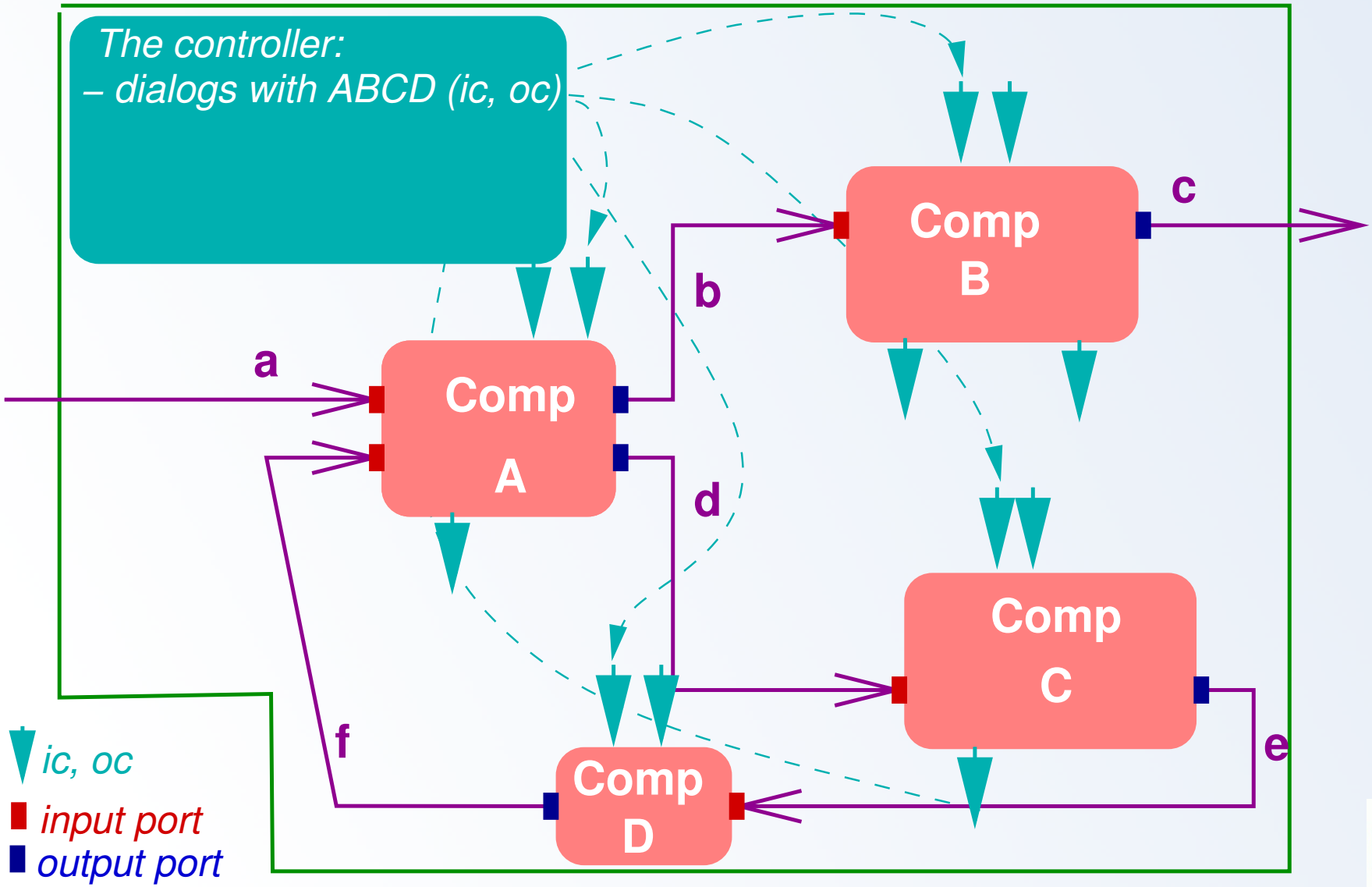
Compositions: The global picture



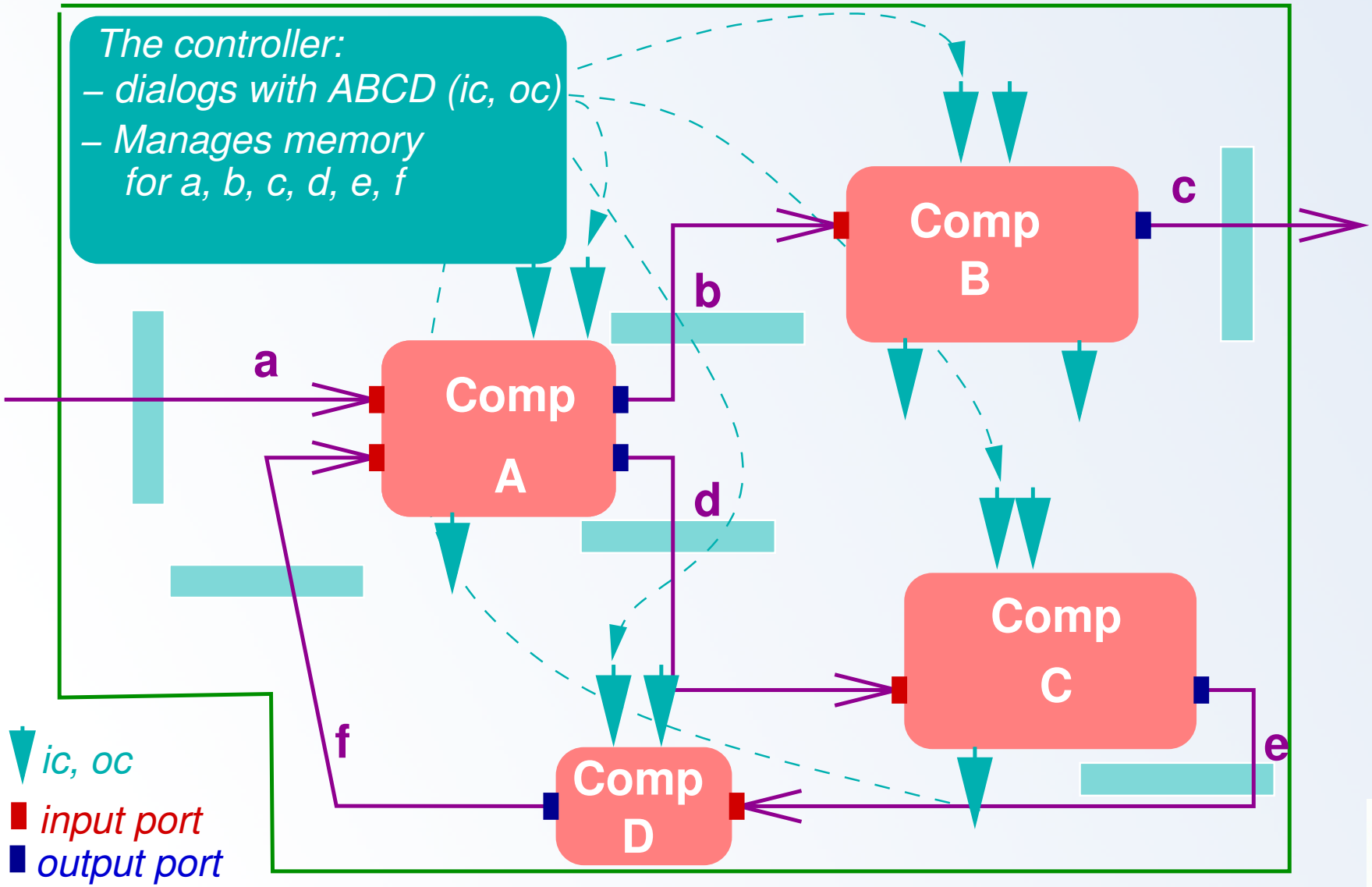
Compositions: The global picture



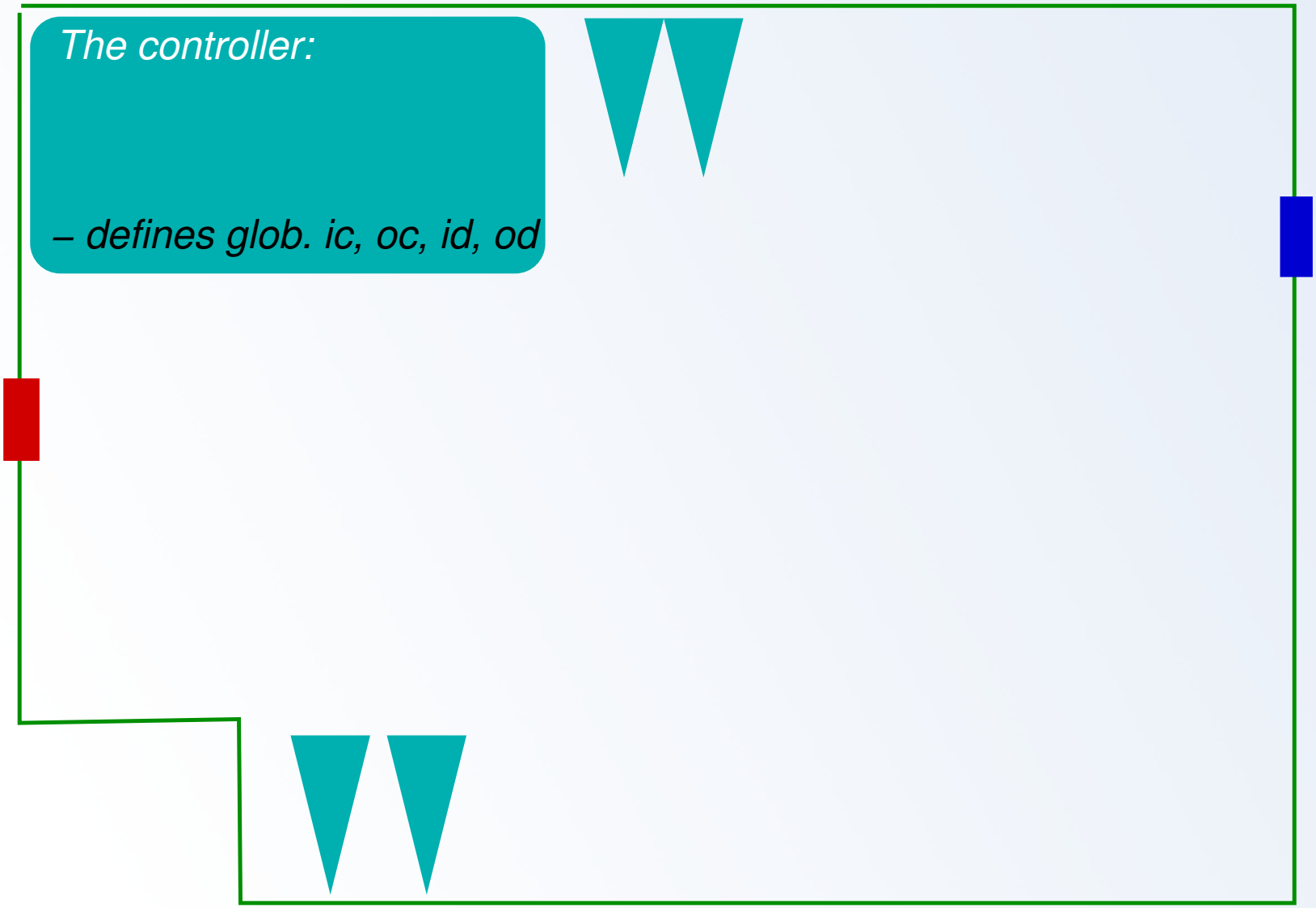
Compositions: The global picture



Compositions: The global picture



Compositions: The global picture



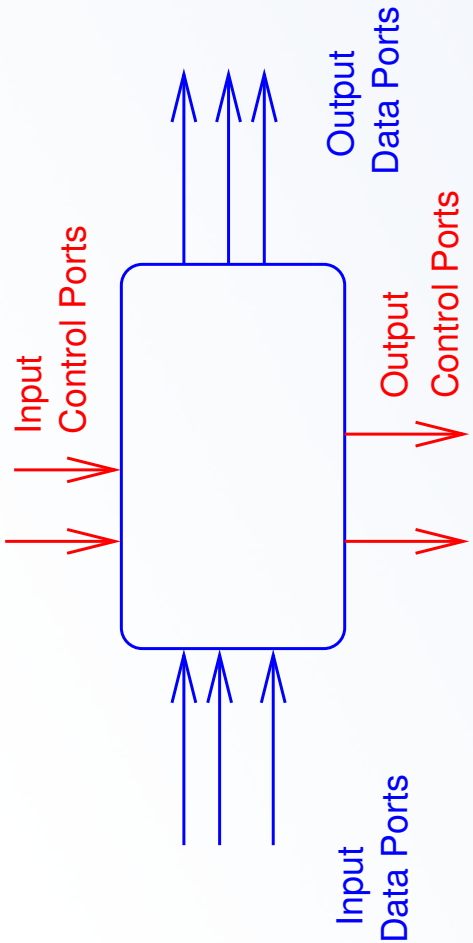
42 Component Protocols, What For?

- Define how components can be used
- Check that an assemblage of components is correct (e.g., in the synchronous MoC, it will enable the detection of instantaneous loops)
- Derive the code of the director from the protocols + other information

Orthogonal to the notion of Assume/Guarantee data constraints

42 Component Protocols, First Ideas

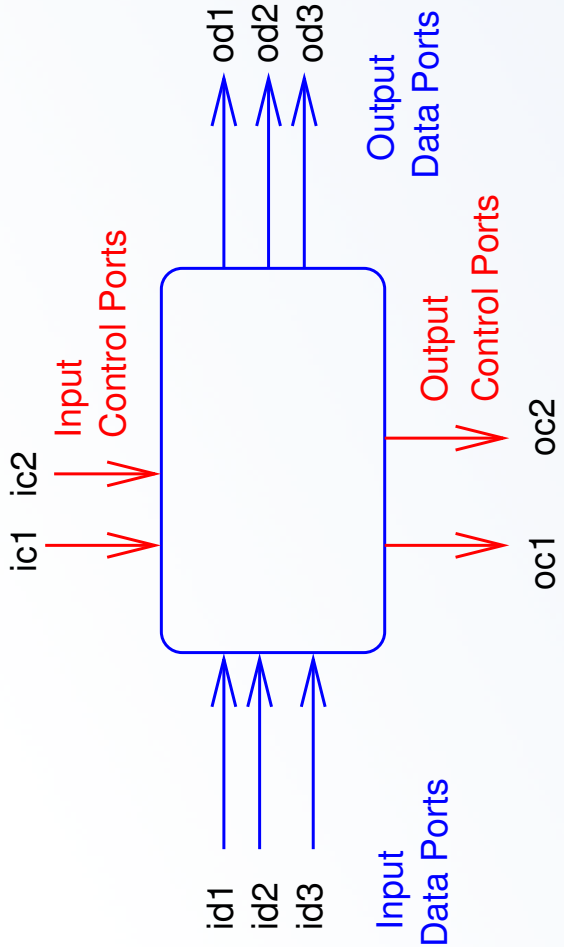
(inspired by multi-clocked synchronous languages)



- Instantaneous constraints to express e.g., *the data output* `od` *is relevant only when the control input* `ic` *is true* or, *the data input* `id` *is required only when the control input* `ic` *is true*
- Logical-time constraints: the data input `id` is required only if asked at the last activation (of this component) with the control output `oc`

42 Component Protocols, First Ideas

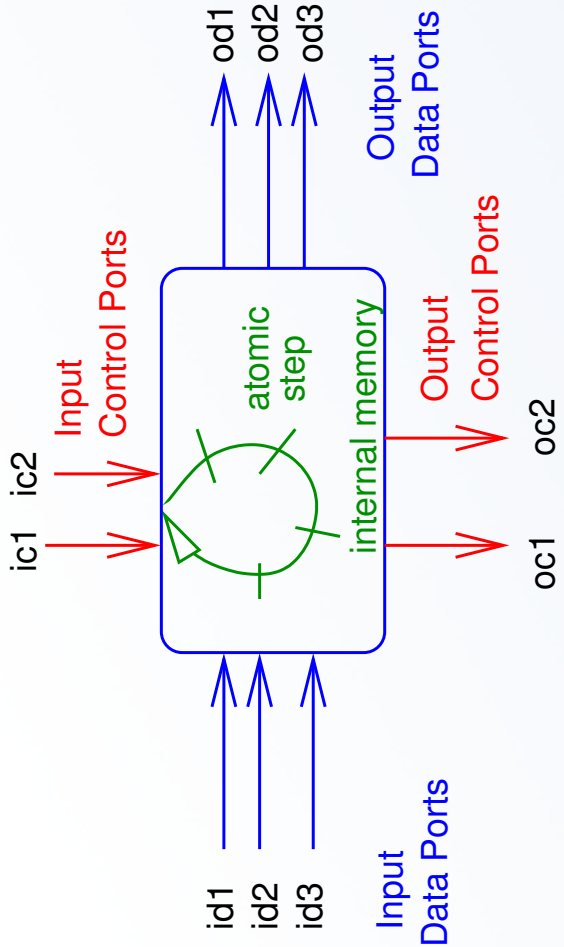
(inspired by multi-clocked synchronous languages)



- Instantaneous constraints to express e.g., *the data output* `od` *is relevant only when the control input* `ic` *is true* or, *the data input* `id` *is required only when the control input* `ic` *is true*
- Logical-time constraints: the data input `id` is required only if asked at the last activation (of this component) with the control output `oc`

42 Component Protocols, First Ideas

(inspired by multi-clocked synchronous languages)



- Instantaneous constraints to express e.g., *the data output* `od` *is relevant only when the control input* `ic` *is true* or, *the data input* `id` *is required only when the control input* `ic` *is true*
- Logical-time constraints: the data input `id` is required only if asked at the last activation (of this component) with the control output `oc`

42 Component Protocols, General Definition

- An automaton structure like in OO protocols, specifying the language of correct sequences of method calls, used here for control inputs
- Accepting states specify what sequences of activations are "complete" w.r.t. the atomicity the component behaviour
- On each transition labeled by a control input, indicate what data inputs it requires and what data or control outputs it produces

Example Object Protocol

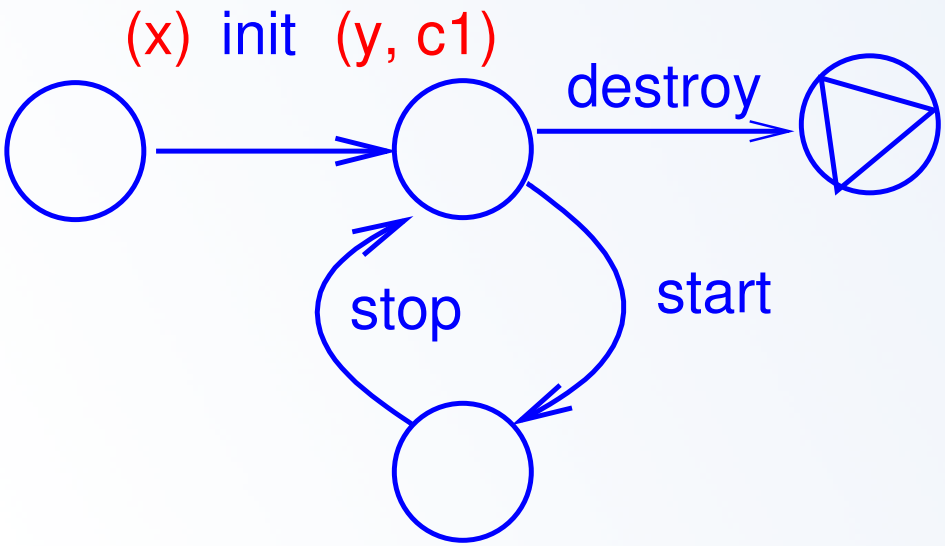
```
package java.applet;
public class Applet {
  //@ public call_sequence
  //          init() : (start() : stop())* : destroy();
  // member declarations ...
}
```

The specification init . (start . stop)* . destroy is meant for the whole life of the object.

http://opuntia.cs.utep.edu/utjml/callseq.html

Specifying and Checking Method Call Sequences of Java Programs

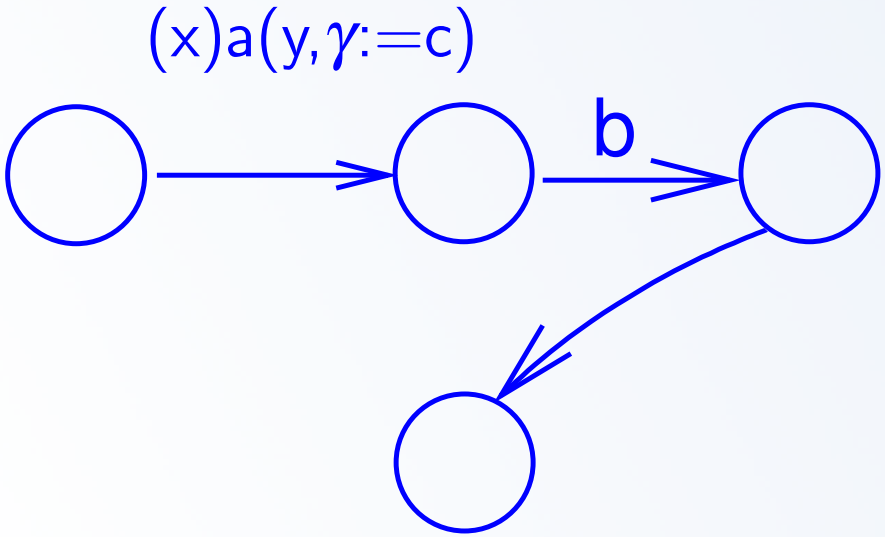
42 Component Protocols, General Definition



control inputs:
init, start, stop,
destroy

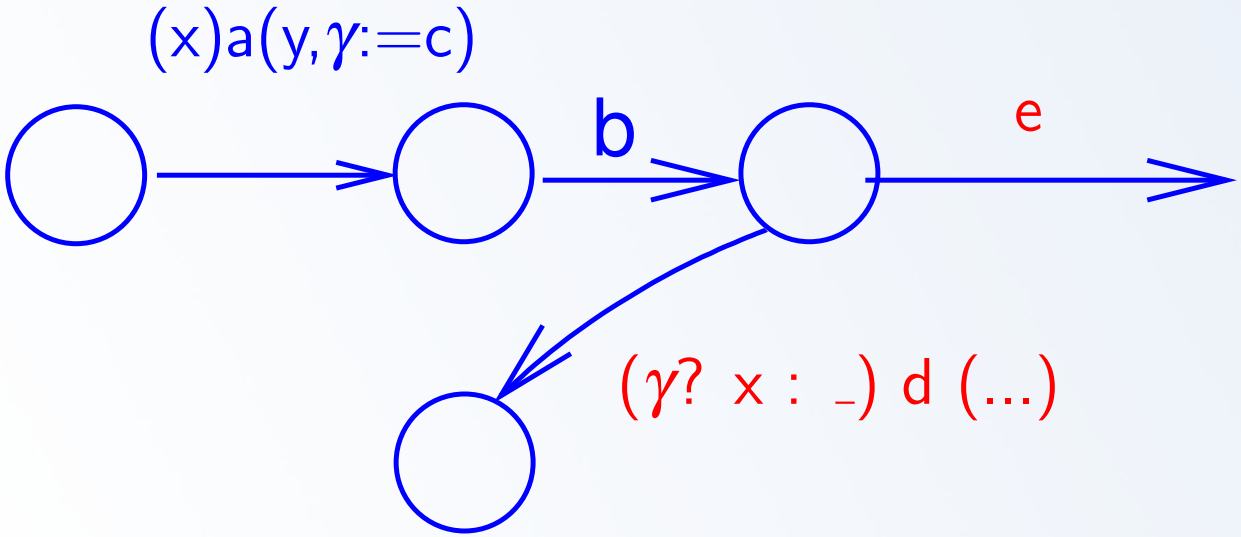
data input : x
data output : y
control output : c1

42 Component Protocols, Intra-Step Sequential Constraints



Idea: we ask the component what it wants to do with command a (that needs the data input x and produces the data output y) and it answers with the control output c, stored in variable γ).
Later, depending on γ, we may need an input x or not.

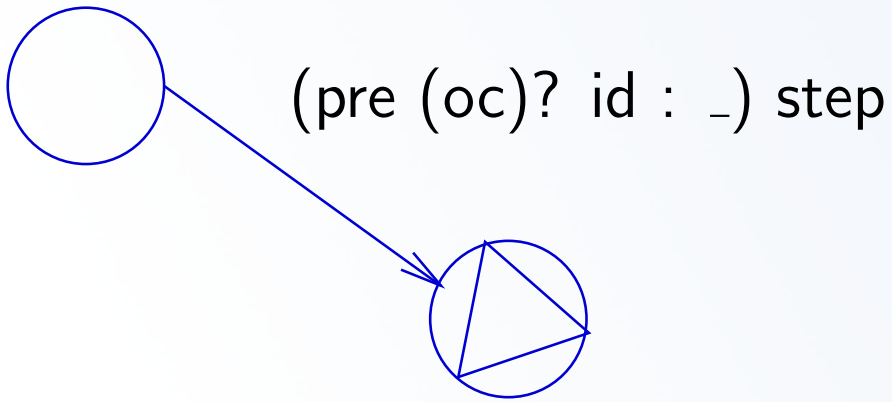
42 Component Protocols, Intra-Step Sequential Constraints



Idea: we ask the component what it wants to do with command a (that needs the data input x and produces the data output y) and it answers with the control output c, stored in variable γ).
Later, depending on γ, we may need an input x or not.

42 Component Protocols, Inter-Step Sequential Constraints

The data input `id` is required for a `step` activation only if asked at the last activation (of this component) with the control output `oc`.



Remarks:

1. `pre` : the needed memory is managed by the director.
2. Does not need a global notion of time: `pre` means "last time" for this component.

1 42: Motivations and Approach

2 42: Definition

3 First Exercice: the Synchronous MoC

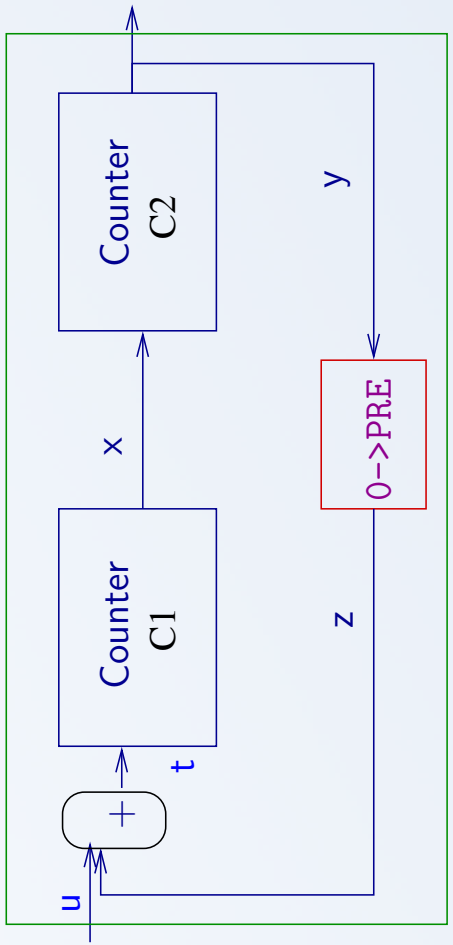
4 2nd Exercice: asynchronous processes on a monoprocessor with shared memory

5 Conclusions

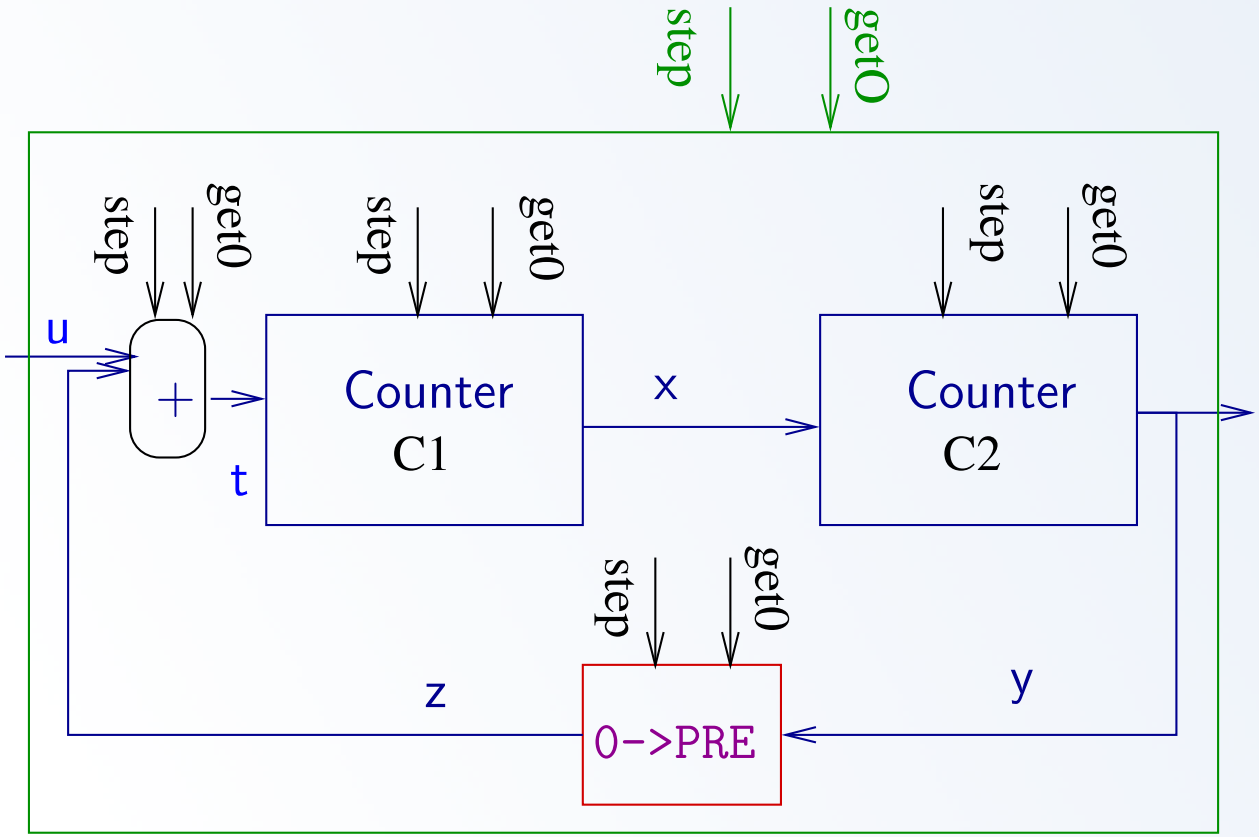
A Circuit Example (in Lustre)

```
node DoubleIntegr (i : int)
     returns (o : int) ;
var x, y, z : int ;
let
    x = Integr (i + (0->pre y)) ;
    y = Integr (x) ;
    o = y ;
tel.

node Integr (i : int)
     returns (o : int) ;
let
    o = i -> pre(o) + i ;
tel.
```



The Component View and the Director Algorithms

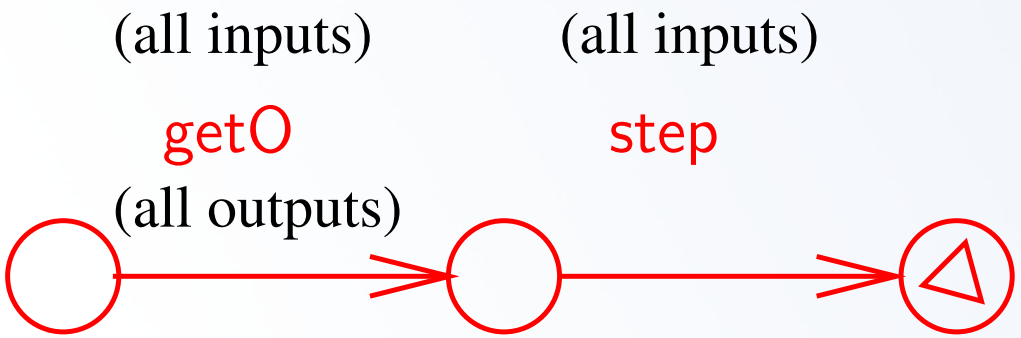



```
global get0:
u.set ;
pre.get0 ;
z.set () ;
plus.get0 ;
t.set ();
c1.get0 ;
x.set () ;
c2.get0 ;
y.set () ;
```

```
global step :c1.step ; c2.step ; pre.step ; plus.step ;
```

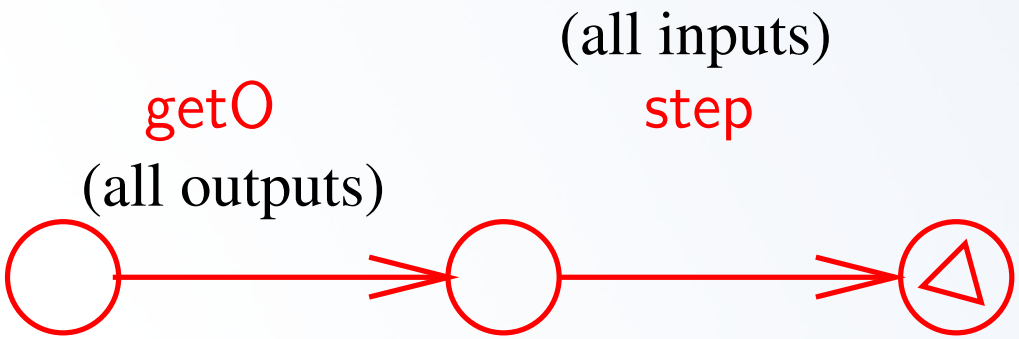
Protocols: All the Mealy Components

one data input, one data output
no control outputs
two control inputs: `getOutput`, `step`



Protocols: The (Moore) PRE component

one data input, one data output
no control outputs
two control inputs: `getOutput`, `step`



Observations on the Synchronous "MoC"

- To be able to implement pure synchrony in a component-based manner, we need to distinguish between `getO` and `step`(s).

Observations on the Synchronous "MoC"

- To be able to implement pure synchrony in a component-based manner, we need to distinguish between `getO` and `step`(s).
- If we get a piece of code with this interface, it can be used as a black box in our component model.

Observations on the Synchronous "MoC"

- To be able to implement pure synchrony in a component-based manner, we need to distinguish between `getO` and `step`(s).
- If we get a piece of code with this interface, it can be used as a black box in our component model.
- The director needs only setting the values of the wires and activating the components.

Observations on the Synchronous "MoC"

- To be able to implement pure synchrony in a component-based manner, we need to distinguish between `getO` and `step`(s).
- If we get a piece of code with this interface, it can be used as a black box in our component model.
- The director needs only setting the values of the wires and activating the components.
- The values on the wires are not meant to be persistent: they are used only during the global step. This is the essence of *synchronous* communication.

Observations on the Synchronous "MoC"

- To be able to implement pure synchrony in a component-based manner, we need to distinguish between `getO` and `step`(s).
- If we get a piece of code with this interface, it can be used as a black box in our component model.
- The director needs only setting the values of the wires and activating the components.
- The values on the wires are not meant to be persistent: they are used only during the global step. This is the essence of *synchronous* communication.
- The director can be deduced from the dataflow graph and the components' protocols (Lustre structural interpreter, electricity in synchronous circuits!)

1 42: Motivations and Approach

2 42: Definition

3 First Exercice: the Synchronous MoC

4 2nd Exercice: asynchronous processes on a monoprocessor with shared memory

5 Conclusions

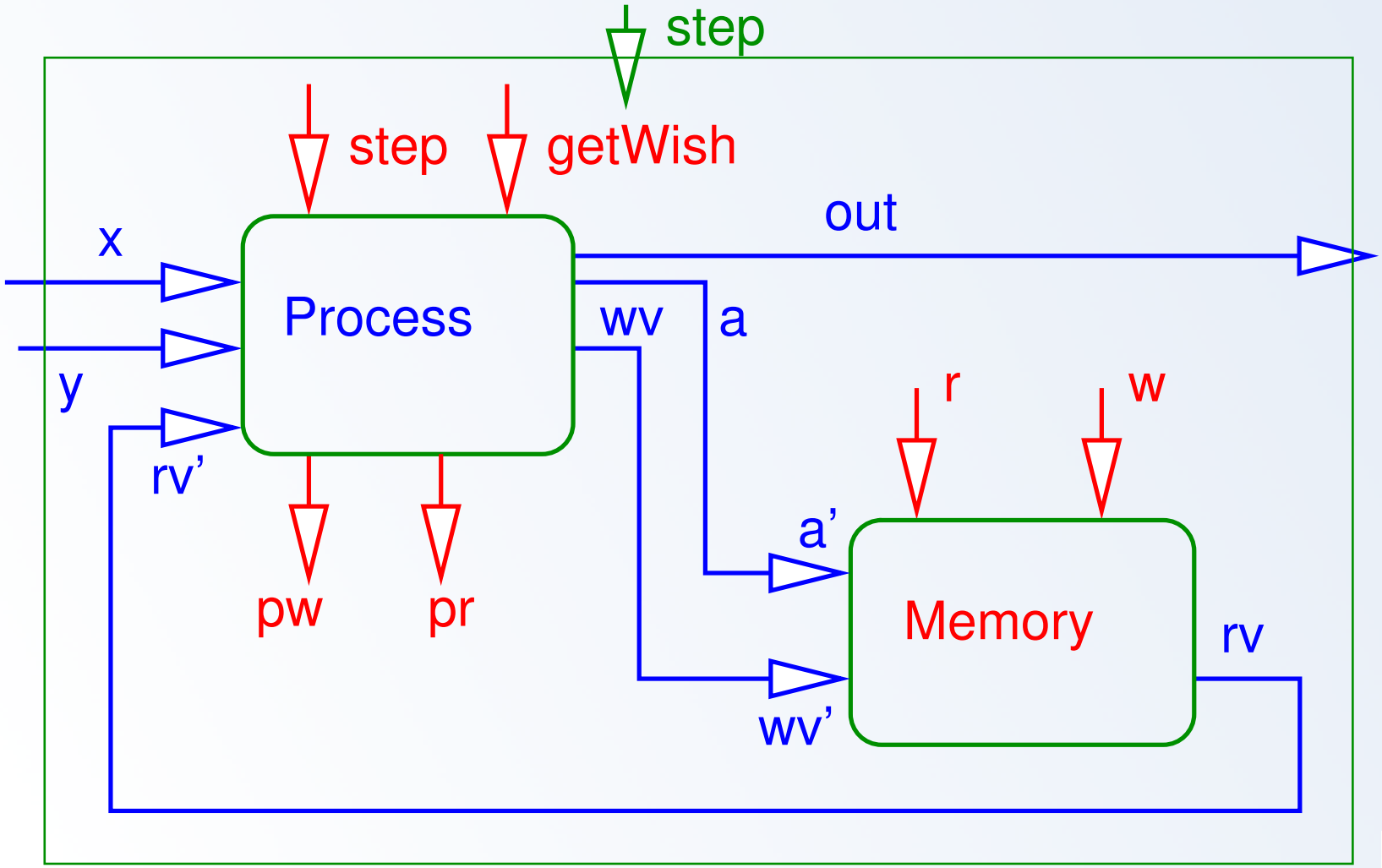
The Execution Platform

A monoprocessor computer, running multiple processes or threads thanks to a time-sharing scheduler. All processes (or threads) access the same memory.

Reading or writing a word from/to memory is made atomic by the HW.

Assume the processes are programmed in a language with an explicit `yield` instruction.

The Component Picture



The global step, informally

If we encapsulate several processes, the shared memory, and a scheduler, we get a global component whose global step corresponds to:

- Either a step of process 1 and a step of the memory
- Or a step of process 2 and a step of the memory
- ...

But never a step of process 1 and a step of process 2.

More important: a step of process i that writes to memory, and the corresponding step of the memory, are no longer distinguishable.
The director defines the atomicity of the global step.

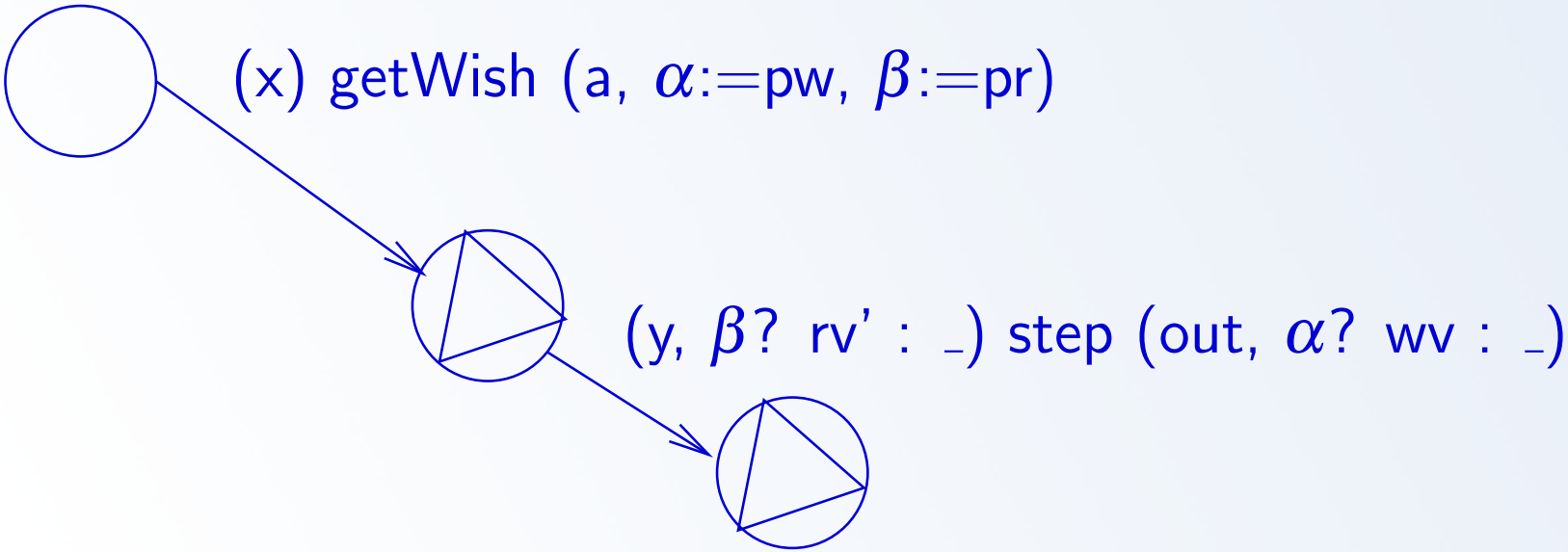
A Remark on masters and slaves

The steps of the memory are not triggered by the global step directly. They are required by the processes.

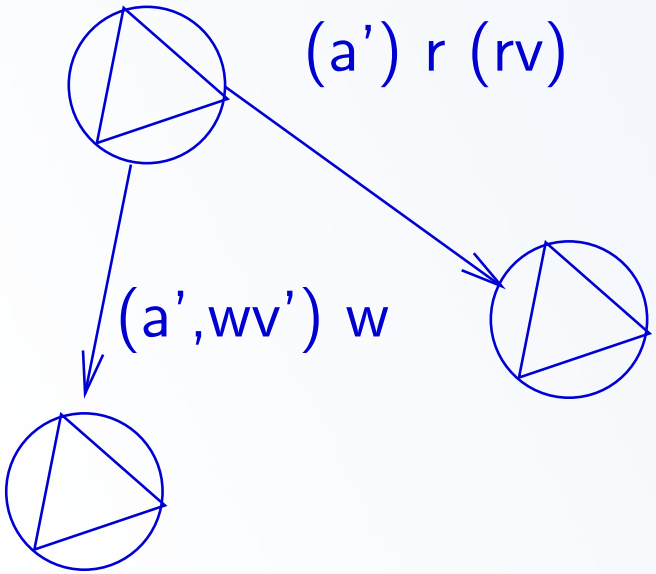
This leeds to the idea of a constraint between the control outputs of the processes (the masters), and the control inputs of the memories (the slaves).

`oc (process)` $\Longrightarrow$ `ic (memory)`

Component Protocols: a Process



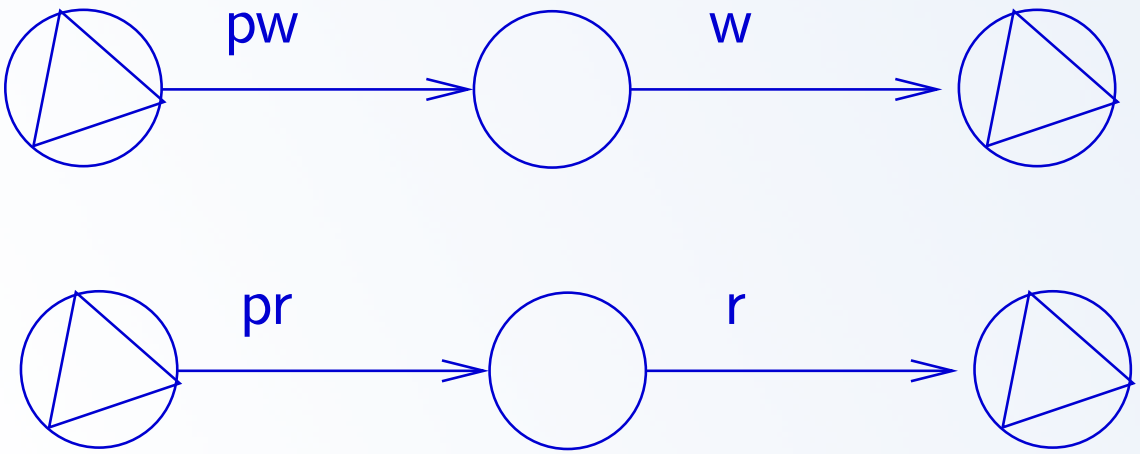
Component Protocols: the Memory



Remark: such a memory may accept several `writes` and/or several `reads` "at the same time" provided they use distinct addresses.

A "Test-and-Set" instruction may be described by a `read-write` activation of the memory, or by an unbreakable sequence of a `read` and a `write` at the same address.

The Master/Slave constraints



A write (resp. read) request should be followed (within the same global step) by a write (resp. read) activation of the memory.

1 42: Motivations and Approach

2 42: Definition

3 First Exercice: the Synchronous MoC

4 2nd Exercice: asynchronous processes on a monoprocessor with shared memory

5 Conclusions

Constraints from which `step` could be defined

- The processes' protocols
- The connections *(cannot consume the value on a wire before it has been produced)*
- The master/slave constraints, if any
- A Global indication:
  - Synchronous MoC: a global step should be exactly one step of each component
  - Asynchronous MoC: a global step should be one step of Process 1 (and its consequences) XOR one step of Process 2 (and its consequences)

Coordination language: connections + M/S constraints + global indication

42: a framework for reasoning about atomicity

An important point: concurrent system development is about building atomicity (for reasoning on a component in isolation) from elementary atomicity mechanisms given by the execution platform.

- 42 can describe a lot of existing concurrent paradigms (currently: 42'ization of the SystemC/TLM MoC)
- *Semantics? Probably in terms of TAG machines, or parameterized products of automata.*
- Languages?
  - A lot of programming styles are available for the components;
  - the ADL is just a set of connections;
  - the director can be described as an imperative program, or as a set of constraints, ...