

Working with CSP

ATTENTION!!!

- Please get involved in the lectures
- When I ask apparently rhetoric questions it's because I'm not sure of your background understanding
- I will keep asking increasingly stupid questions until I've found the lower bound of knowledge!

An example of the dangers of Threads

- Monitors (as implemented in Java) should be the simplest solution to program threads
- The following example shows a very small example of undesired behavior

“Wot, No Chickens!”

- Peter Welch, University of Kent
- Five Philosophers (consumers)
 - Think
 - Go to Canteen to get Chicken for dinner
 - Repeat
- Chef (producer)
 - produces four chickens at a time and delivers to canteen

“Wot, No Chickens!”

- Philosopher 0 is greedy -- never thinks
- Other philosophers think 3 time units before going to eat
- Chef takes 2 time units to cook four chickens
- Chef takes 3 time units to deliver chickens
 - occupies canteen while delivering
- Simplified code follows -- leaves out exception handling try-catch

```
class Canteen {  
  
    private int n_chickens = 0;  
  
    public synchronized int get(int id) {  
        while (n_chickens == 0) {  
            wait(); // Wot, No Chickens!  
        }  
        n_chickens--; // Those look good...one please  
        return 1;  
    }  
  
    public synchronized void put(int value) {  
        Thread.sleep(3000); // delivering chickens..  
        n_chickens += value;  
        notifyAll (); // Chickens ready!  
    }  
}
```

```
class Chef extends Thread {  
    private Canteen canteen;  
  
    public Chef (Canteen canteen) {  
        this.canteen = canteen;  
        start ();  
    }  
  
    public void run () {  
        int n_chickens;  
        while (true) {  
            sleep (2000); // Cooking...  
            n_chickens = 4;  
            canteen.put (n_chickens);  
        }  
    }  
}
```

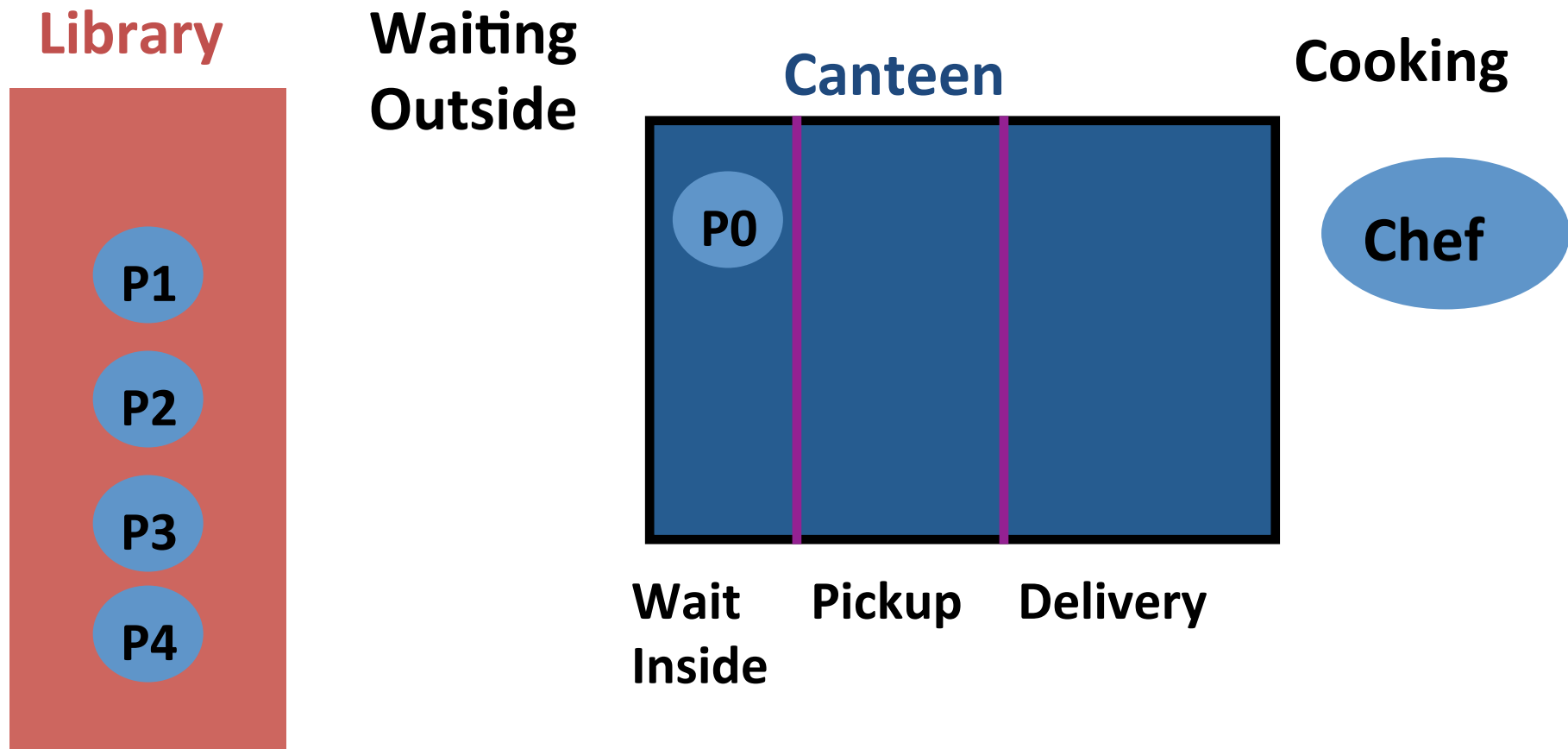
```
class Phil extends Thread {
    private int id;
    private Canteen canteen;

    public Phil(int id, Canteen canteen) {
        this.id = id;
        this.canteen = canteen;
        start ();
    }
    public void run() {
        int chicken;
        while (true) {
            if (id > 0) {
                sleep(3000);                // Thinking...
            }
            chicken = canteen.get(id); // Gotta eat...
        } // mmm...That's good
    }
}
```

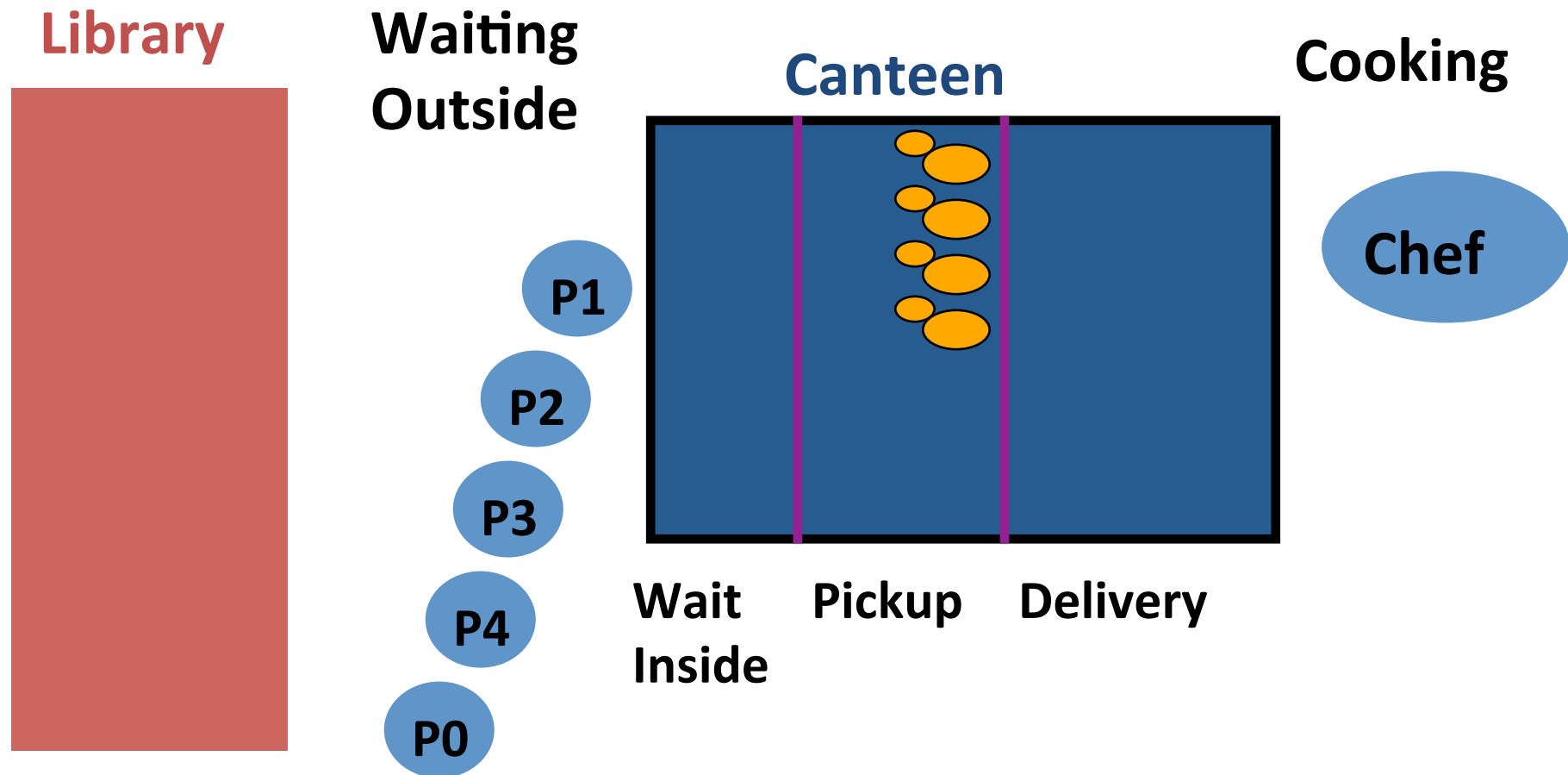


```
class College {  
  
    public static void main (String argv[]) {  
  
        int n_philosophers = 5;  
        Canteen canteen = new Canteen ();  
        Chef chef = new Chef (canteen);  
        Phil[] phil = new Phil[n_philosophers];  
  
        for (int i = 0; i < n_philosophers; i++) {  
            phil[i] = new Phil (i, canteen);  
        }  
    }  
}
```

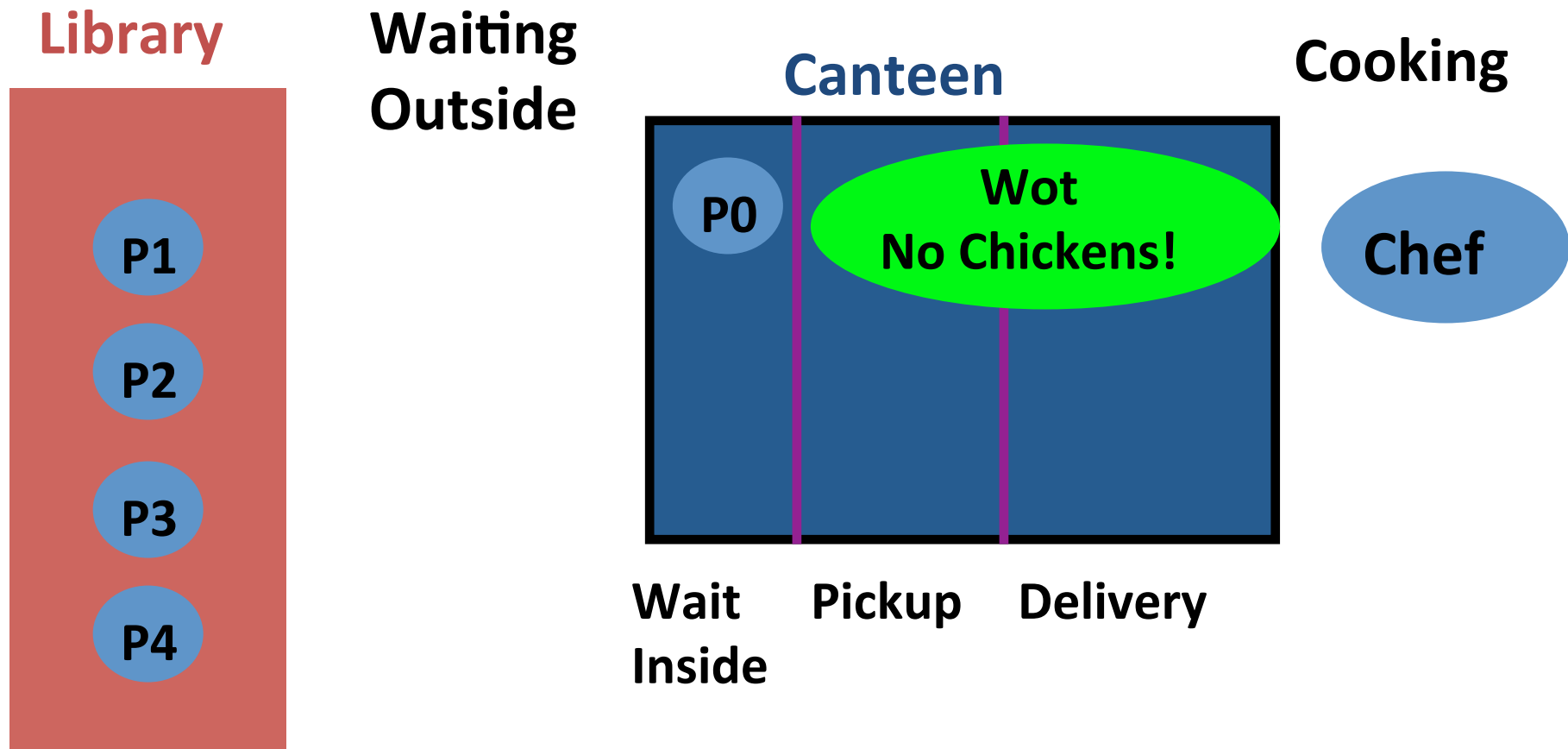
“Wot, No Chickens!”



“Wot, No Chickens!”



“Wot, No Chickens!”



Problems with Java Monitors

- **Semantics of *Notify* / *NotifyAll* ???**
- **Breaks O-O model**
 - Threads are controlled by calling functions inside monitor
 - Depend on notification by other threads
 - Multiple monitors can easily lead to deadlock
 - Monitors can easily lead to starvation
 - Global ! knowledge needed to adequately coordinate multiple monitors

Working with CSP

No algebra required
(mostly from Peter Welch)

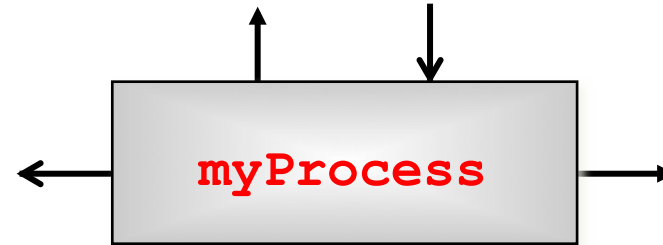
Processes



myProcess

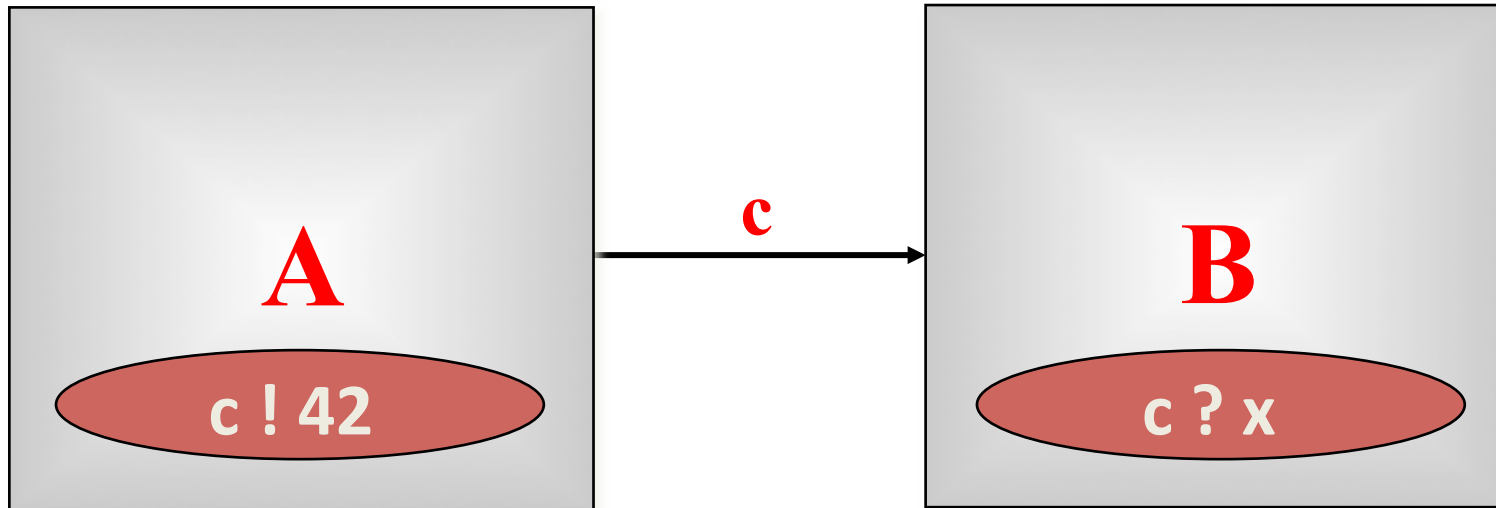
- A **process** is a component that encapsulates some data structures and algorithms for manipulating that data.
- Both its data and algorithms are **private**. The outside world can neither see that data nor execute those algorithms! [They are not *objects*.]
- The algorithms are executed by the process in its own thread (or threads) of control.
- So, how does one process interact with another?

Processes



- The simplest form of interaction is *synchronised* message-passing along **channels**.
- The simplest forms of channel are **zero-buffered** and **point-to-point** (i.e. *wires*).
- But, we can have **buffered** channels (*blocking/overwriting*).
- And **any-1**, **1-any** and **any-any** channels.
- And structured multi-way synchronisation (e.g. **barriers**) ...
- And high-level (e.g. **CREW**) *shared-memory* locks ...

Synchronized Communication

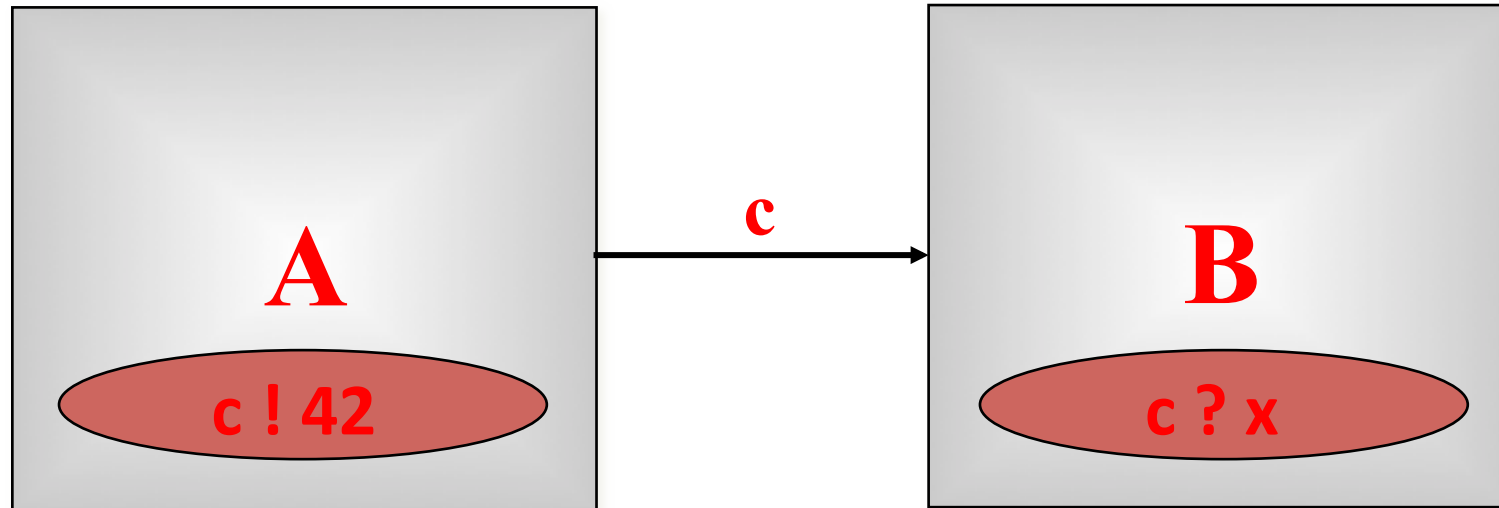


A may write on c at any time, but has to wait for a *read*.

B may read from c at any time, but has to wait for a *write*.

$(A(c) \parallel B(c)) \setminus \{c\}$

Synchronised Communication



Only when both A and B are ready can the communication proceed over the channel c .

$$(A(c) \parallel B(c)) \setminus \{c\}$$

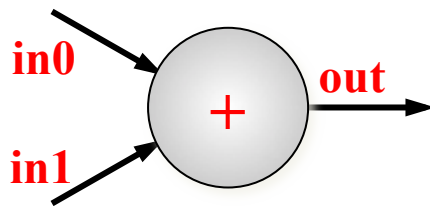
'Legoland' Catalog



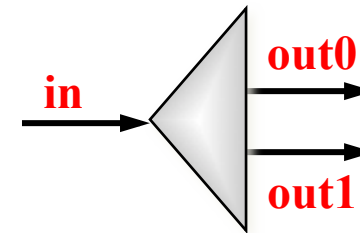
IdInt (in, out)



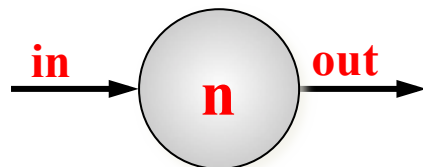
SuccInt (in, out)



PlusInt (in0, in1, out)



Delta2Int (in, out0, out1)



PrefixInt (n, in, out)



TailInt (in, out)

‘Legoland’ Catalog

- This is a catalog of fine-grained processes - think of them as pieces of hardware (e.g. chips). They process data (**ints**) flowing through them.
- They are presented not because we suggest working at such fine levels of granularity ...
- They are presented in order to build up fluency in working with parallel logic.

‘Legoland’ Catalog

- Parallel logic should become just as easy to manage as serial logic.
- This is not the traditionally held view ...
- But that tradition is **wrong**.
- **CSP/occam** people have always known this.

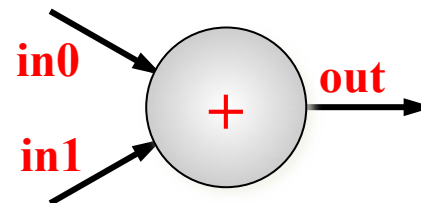
Let's look at some **CSP** *pseudo-code* for these processes ...



```
IdInt (in, out) = in?x --> out!x --> IdInt (in, out)
```

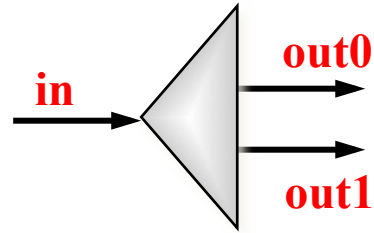


```
SuccInt (in, out) = in?x --> out!(x + 1) --> SuccInt (in, out)
```



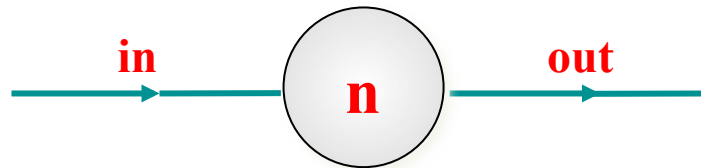
Note the parallel input

```
PlusInt (in0, in1, out) =  
  (in0?x0 --> SKIP || in1?x1 --> SKIP);  
  out!(x0 + x1) --> PlusInt (in0, in1, out)
```



Note the parallel output

```
Delta2Int (in, out0, out1) =
  in?x --> (out0!x --> SKIP || out1!x --> SKIP);
Delta2Int (in, out0, out1)
```

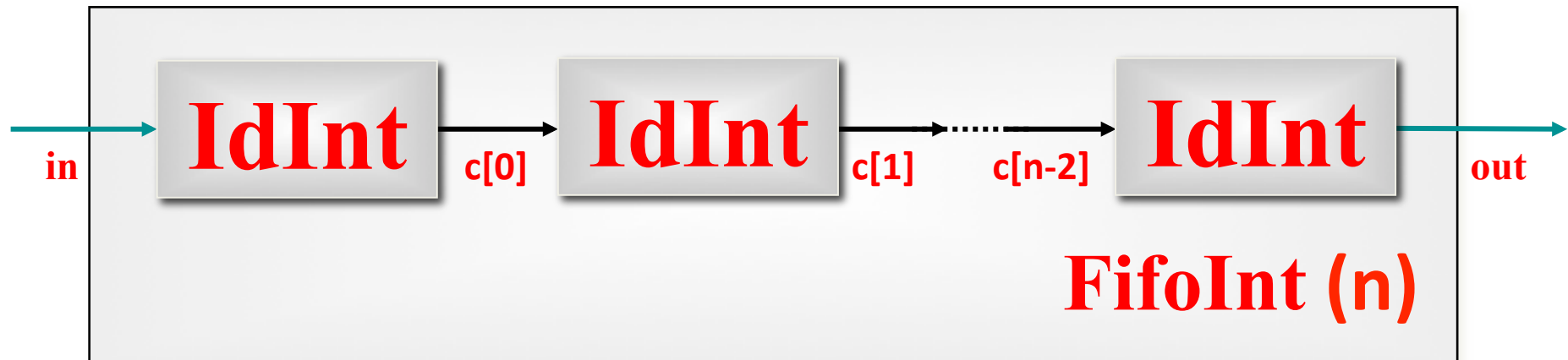


```
PrefixInt (n, in, out) = out!n --> IdInt (in, out)
```



```
TailInt (in, out) = in?x --> IdInt (in, out)
```

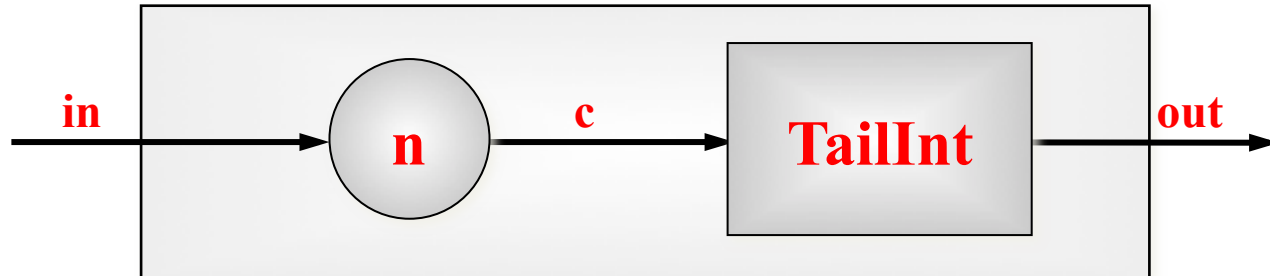
A Blocking FIFO Buffer



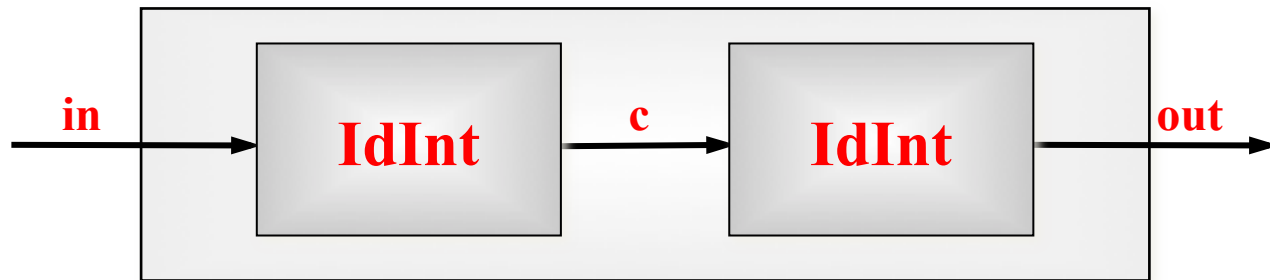
```
FifoInt (n, in, out) =  
  IdInt (in, c[0]) ||  
  ([||i = 0 FOR n-2] IdInt (c[i], c[i+1])) ||  
  IdInt (c[n-2], out)
```

Note: this is such a common idiom that it is provided as a (channel) primitive in some implementations.

A Simple Equivalence



`(PrefixInt (n, in, c) || TailInt (c, out)) \ {c}`



`(IdInt (in, c) || IdInt (c, out)) \ {c}`

The outside world can see no difference between these two 2-place FIFOs ...

Good News!

The good news is that we can 'see' this semantic equivalence with just one glance.

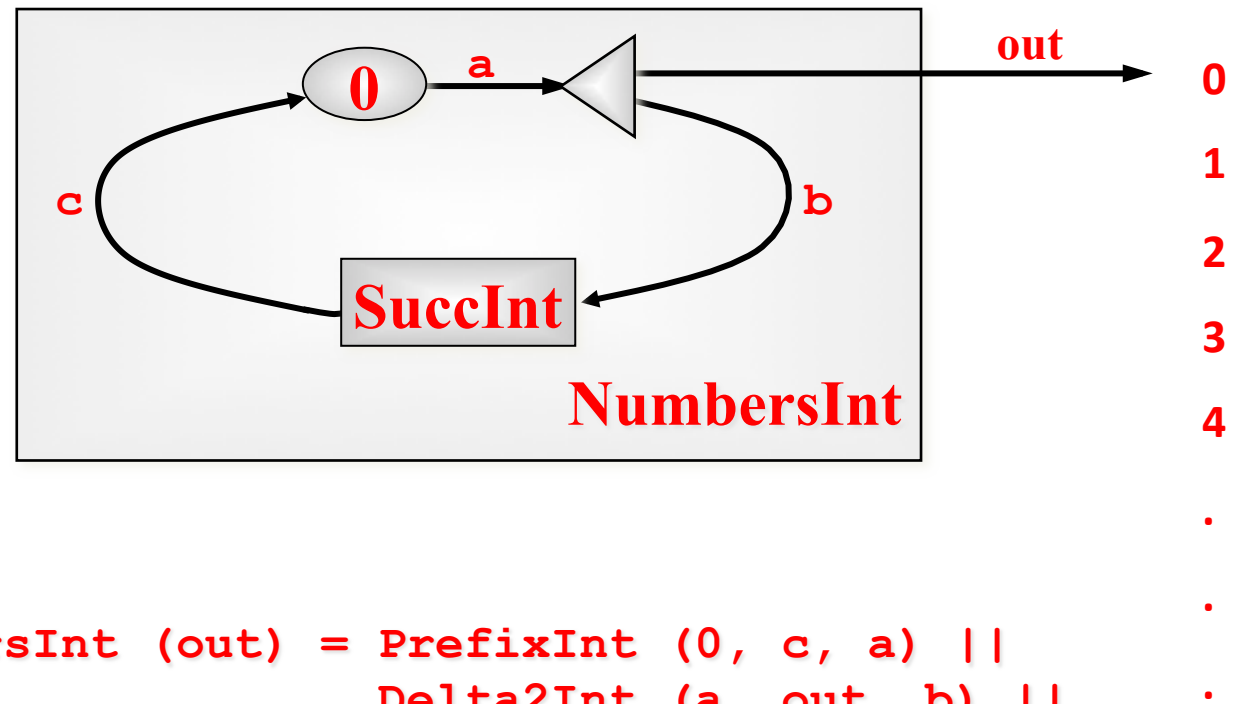
[CLAIM] **CSP** semantics cleanly reflects our intuitive feel for interacting systems.

This quickly builds up confidence ...



Wot - no chickens?!!

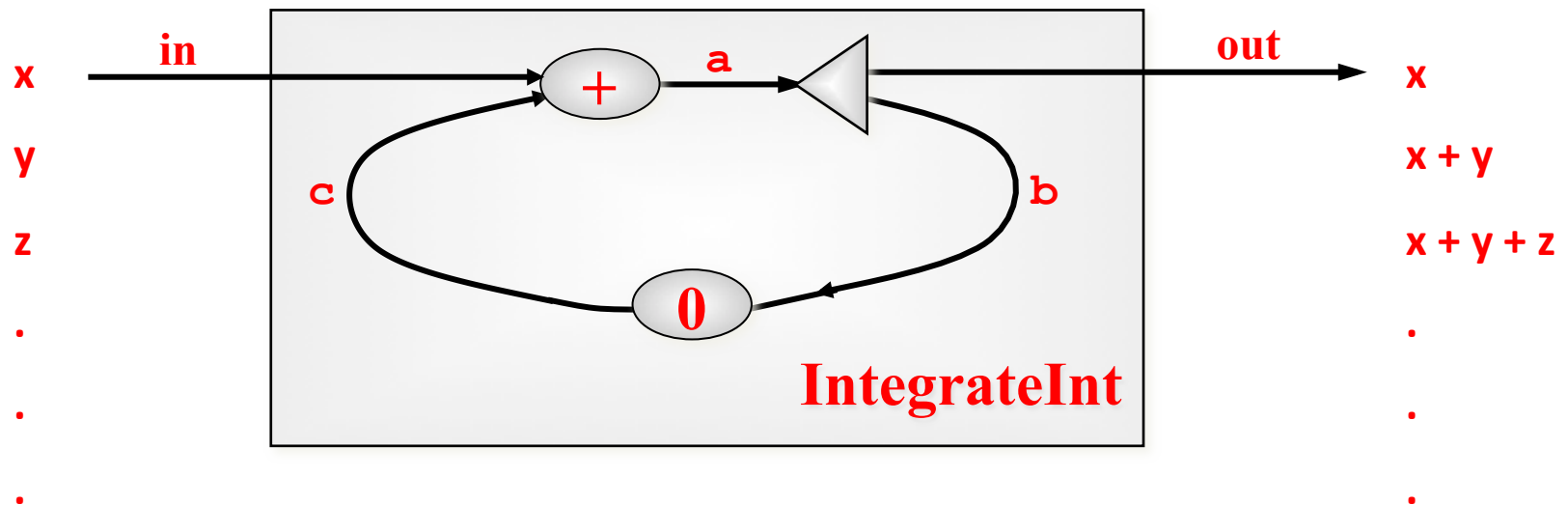
Some Simple Networks



```
NumbersInt (out) = PrefixInt (0, c, a) ||
                  Delta2Int (a, out, b) ||
                  SuccInt (b, c)
```

Note: this pushes numbers out so long as the receiver is willing to take it.

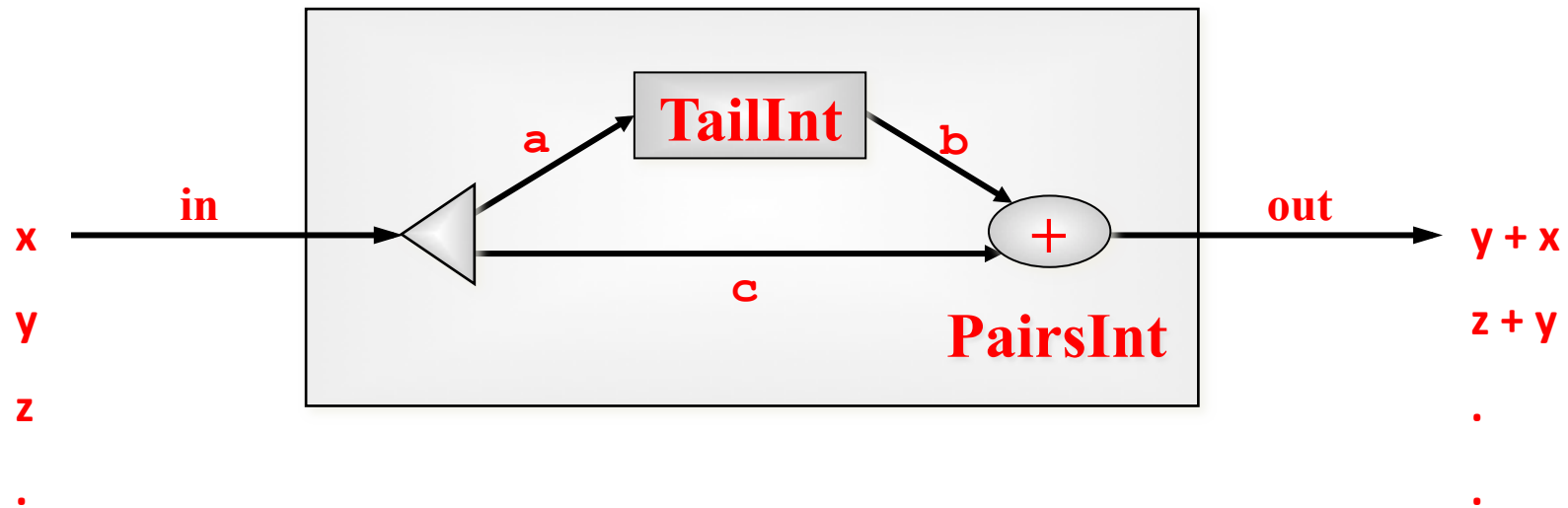
Some Simple Networks



```
IntegrateInt (out) = PlusInt (in, c, a) ||
                    Delta2Int (a, out, b) ||
                    PrefixInt (0, b, c)
```

Note: this outputs one number for every input it gets.

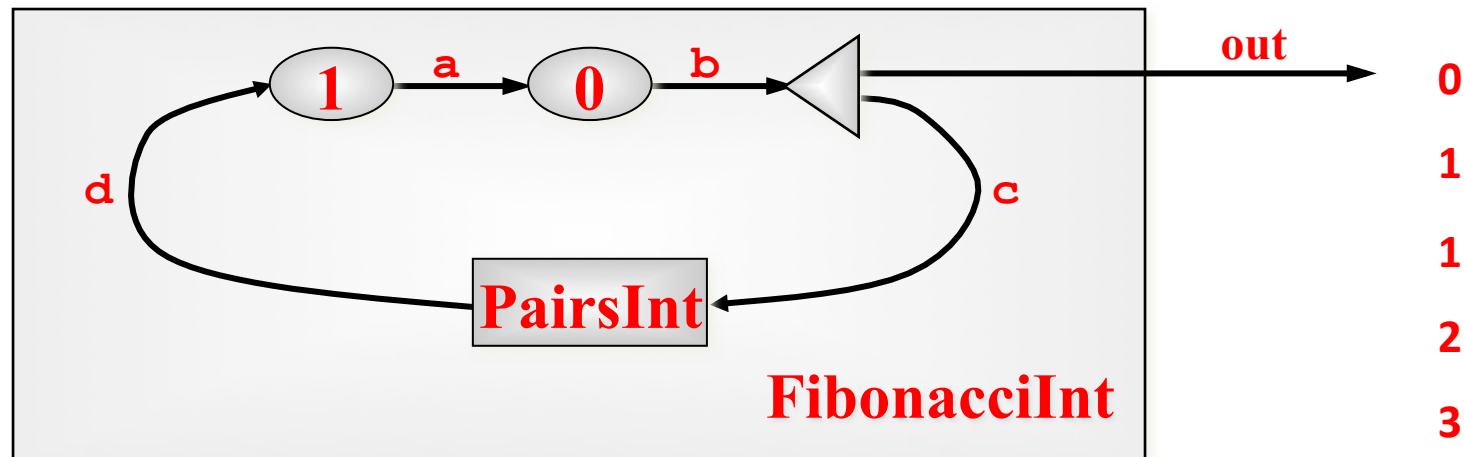
Some Simple Networks



```
PairsInt (in, out) = Delta2Int (in, a, c) ||
                    TailInt (a, b) ||
                    PlusInt (b, c, out)
```

Note: this needs two inputs before producing one output. Thereafter, it produces one number for every input it gets.

Some Layered Networks

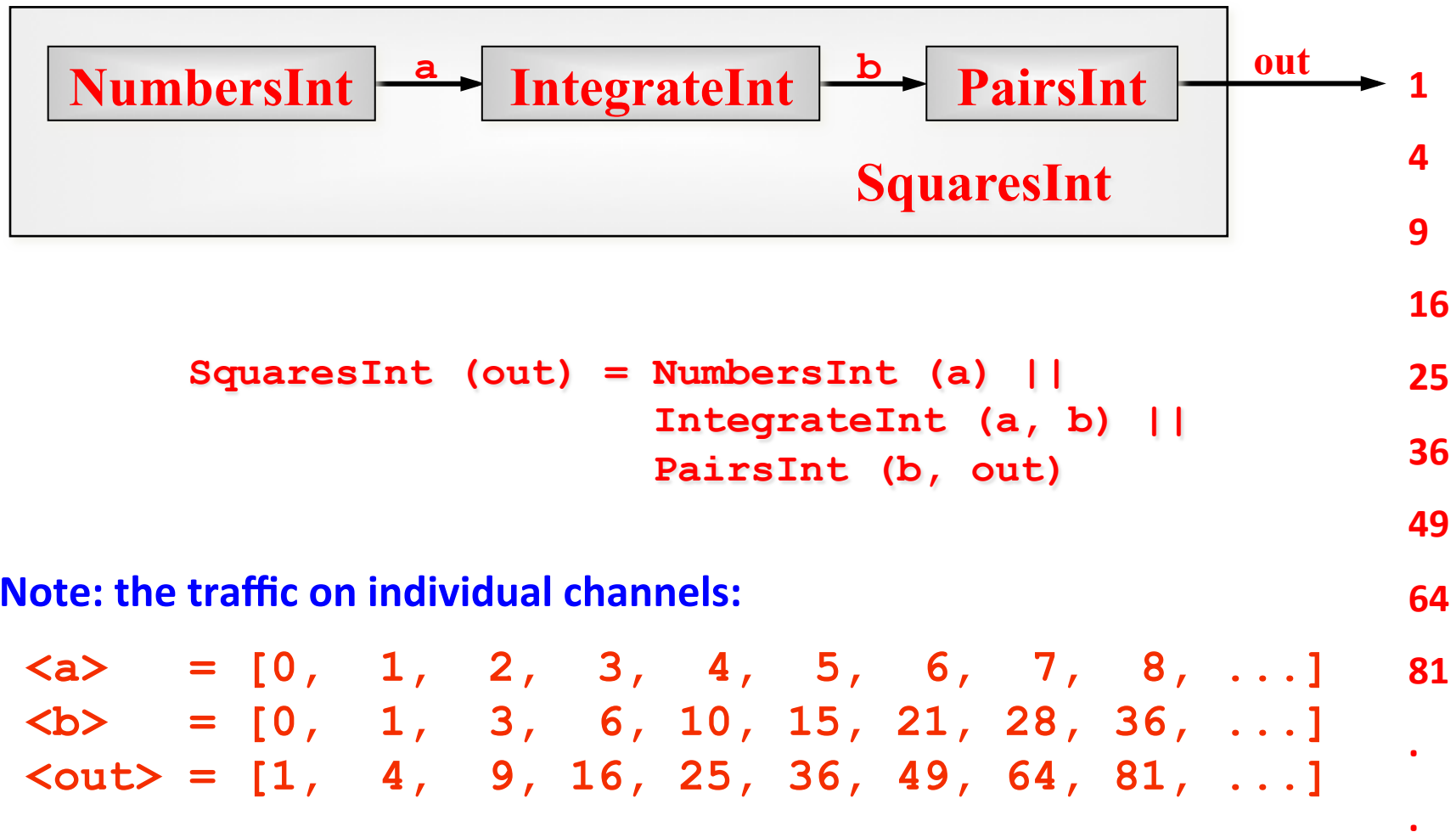


```

FibonacciInt (out) = PrefixInt (1, d, a) ||
                    PrefixInt (0, a, b) ||
                    Delta2Int (b, out, c) ||
                    PairsInt (b, c)
    
```

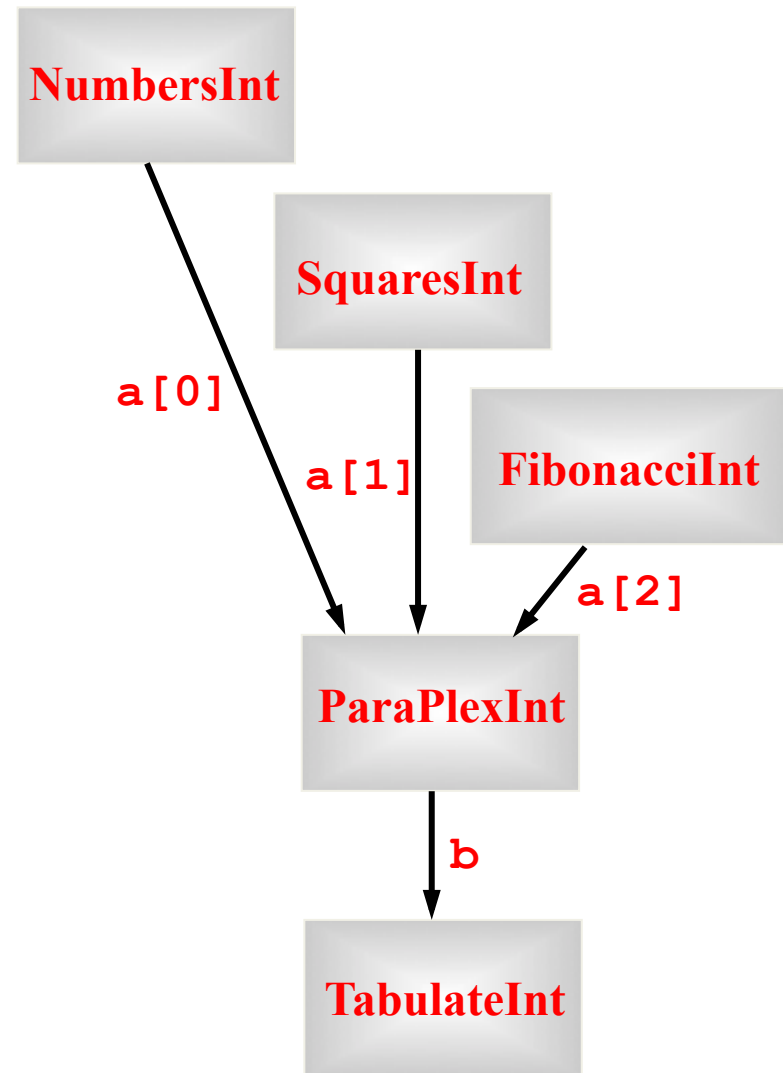
Note: the two numbers needed by **PairsInt** to get started are provided by the two **PrefixInts**. Thereafter, only one number circulates on the feedback loop. If only one **PrefixInt** had been in the circuit, deadlock would have happened (with each process waiting trying to input).

Some Layered Networks



Quite a Lot of Processes

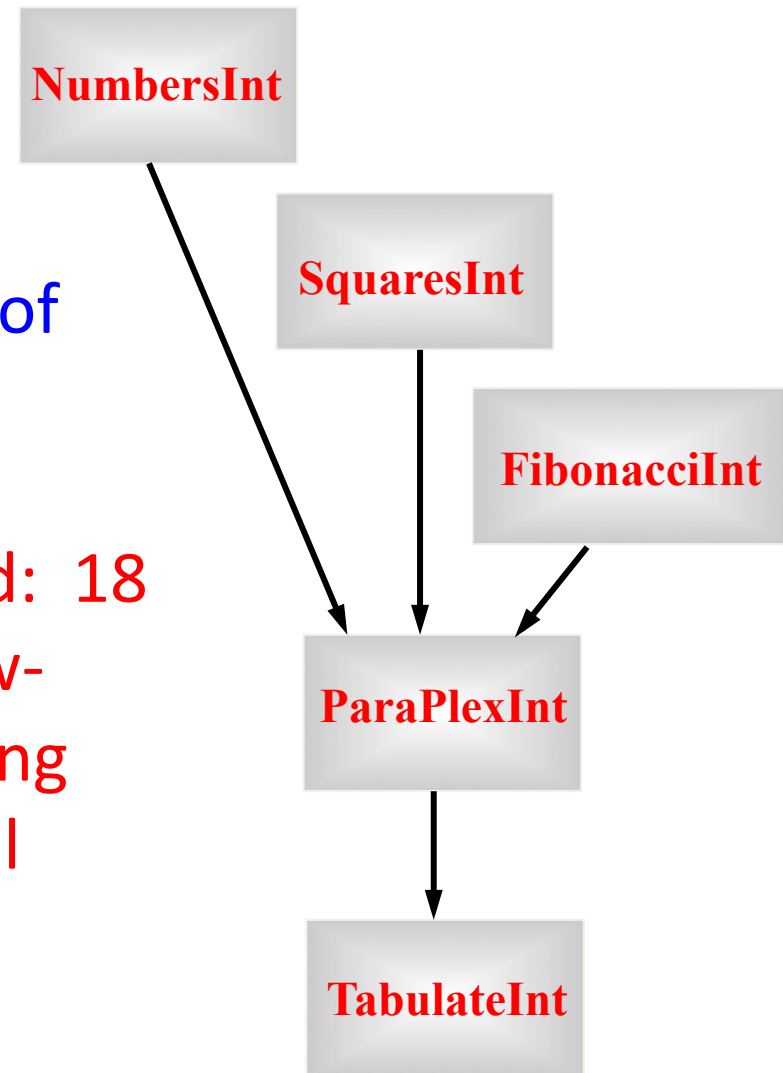
```
NumbersInt (a[0]) ||  
SquaresInt (a[1]) ||  
FibonacciInt (a[2]) ||  
ParaPlexInt (a, b) ||  
TabulateInt (b)
```



Quite a Lot of Processes

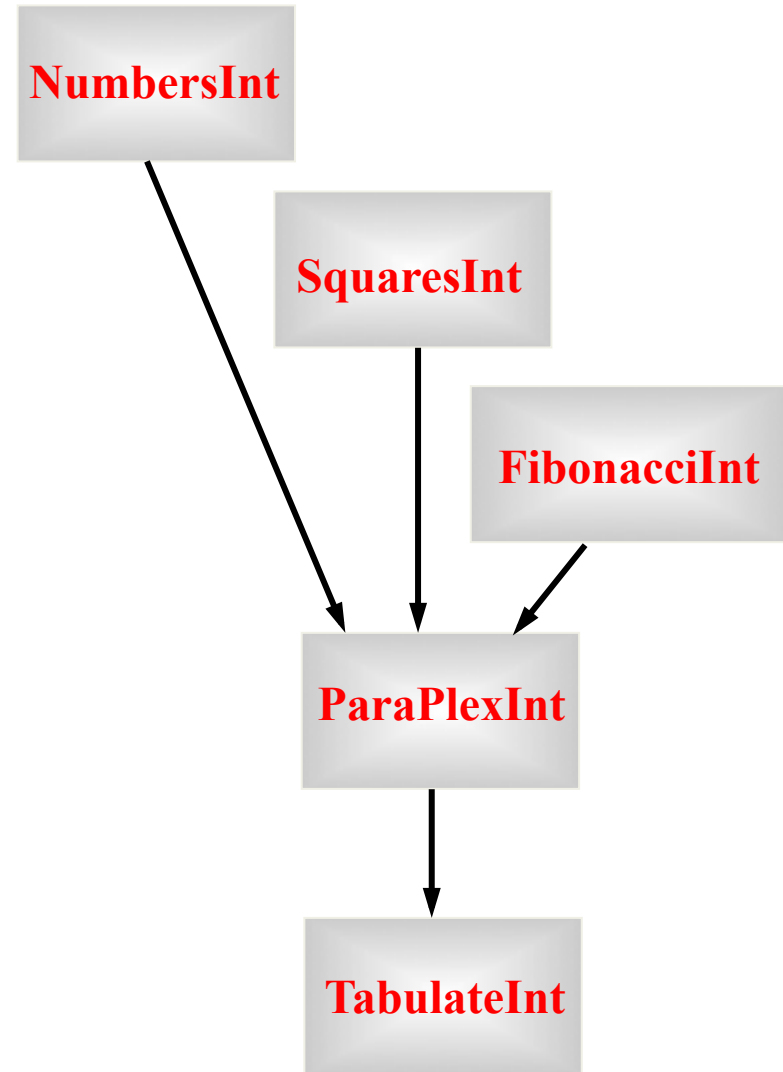
At this level, we have a network of 5 communicating processes.

In fact, 28 processes are involved: 18 non-terminating ones and 10 low-level transients repeatedly starting up and shutting down for parallel input and output.



Quite a Lot of Processes

Fortunately, CSP semantics are ***compositional*** - which means that we only have to reason at each layer of the network in order to design, understand, code, and maintain it.



Deterministic Processes

So far, our parallel systems have been ***deterministic***:

- the values in the output streams depend only on the values in the input streams;
- the semantics is scheduling independent;
- no race hazards are possible.

CSP parallelism, on its own, ***does not introduce non-determinism***.

This gives a firm foundation for exploring real-world models which cannot always behave so simply.

Non-Deterministic Processes

In the real world, it is sometimes the case that things happen as a result of:

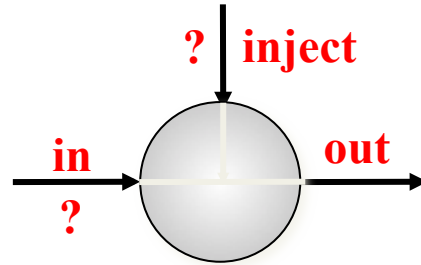
- what happened in the past;
- when (or, at least, in what order) things happened.

In this world, things are scheduling dependent.

CSP addresses these issues *explicitly*.

Non-determinism does not arise by default.

A Control Process



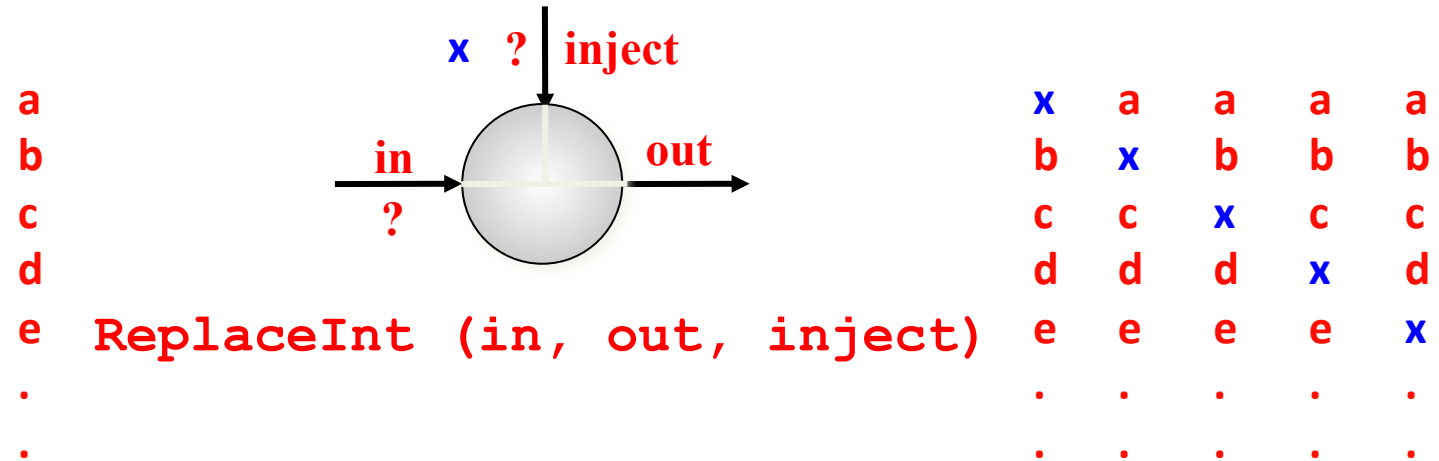
`ReplaceInt (in, out, inject)`

Coping with the real world - making choices ...

In **ReplaceInt**, data normally flows from **in** to **out** unchanged.

However, if something arrives on **inject**, it is output on **out** - *instead of* the next input from **in**.

A Control Process

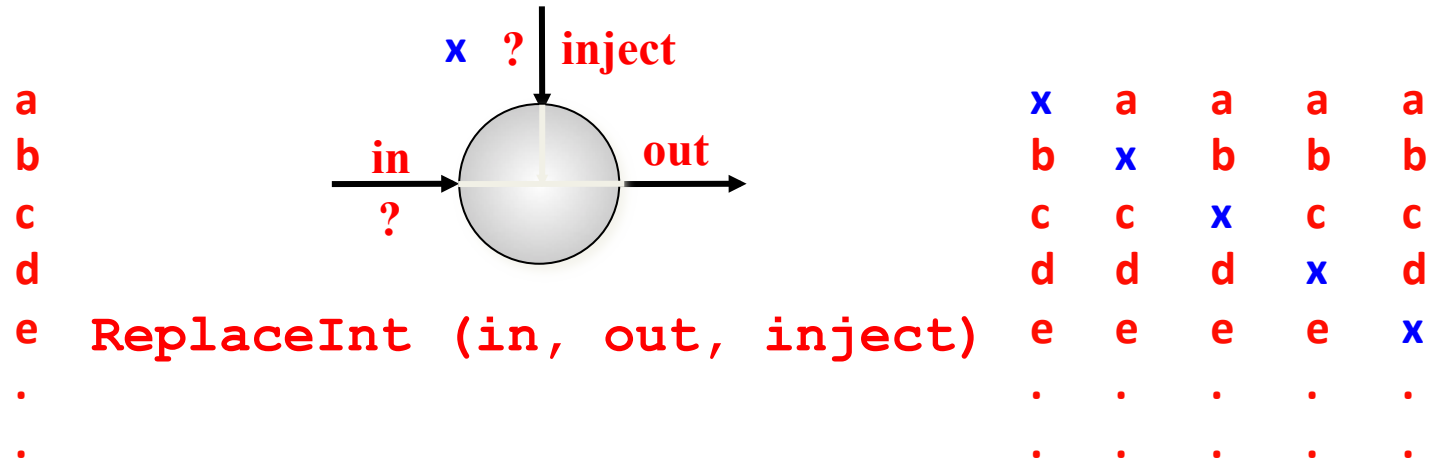


The **out** stream depends upon:

- The values contained in the **in** and **inject** streams;
- the **order** in which those values arrive.

The **out** stream is **not** determined just by the **in** and **inject** streams - it is **non-deterministic**.

A Control Process



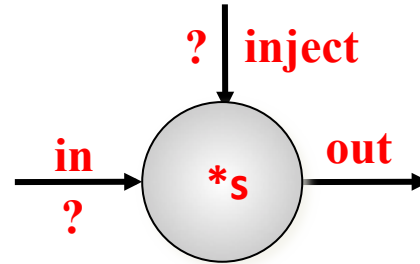
```

ReplaceInt (in, out, inject) =
  (inject?x --> ((in?a --> SKIP) || (out!x --> SKIP))
    [PRI]
    in?a --> out!a --> SKIP
  );
ReplaceInt (in, out, inject)
  
```

Note: `[]` is the (external) choice operator of CSP.

`[PRI]` is a prioritised version - giving priority to the event on its left.

Another Control Process



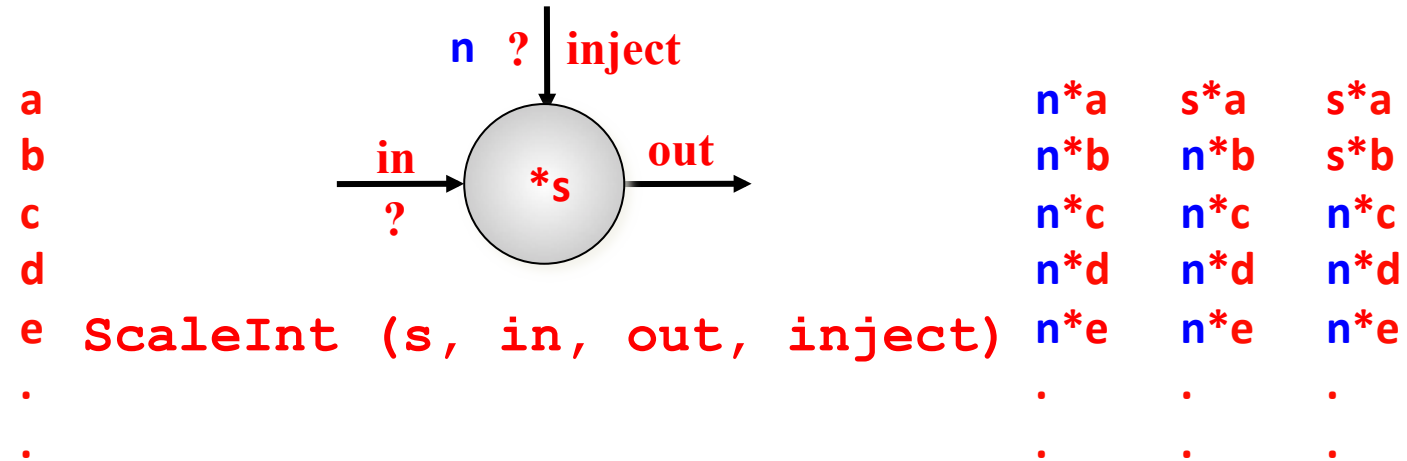
`ScaleInt (s, in, out, inject)`

Coping with the real world - making choices ...

In **ScaleInt**, data flows from **in** to **out**, getting scaled by a factor of **s** as it passes.

Values arriving on **inject**, reset that **s** factor.

Another Control Process

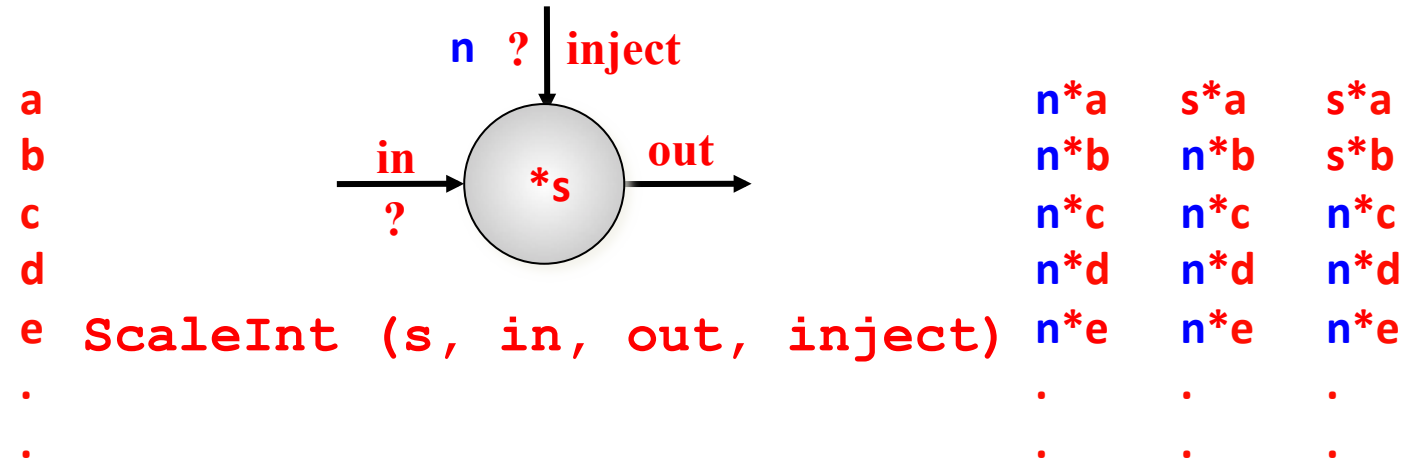


The **out** stream depends upon:

- The values contained in the **in** and **inject** streams;
- the **order** in which those values arrive.

The **out** stream is **not** determined just by the **in** and **inject** streams - it is **non-deterministic**.

Another Control Process

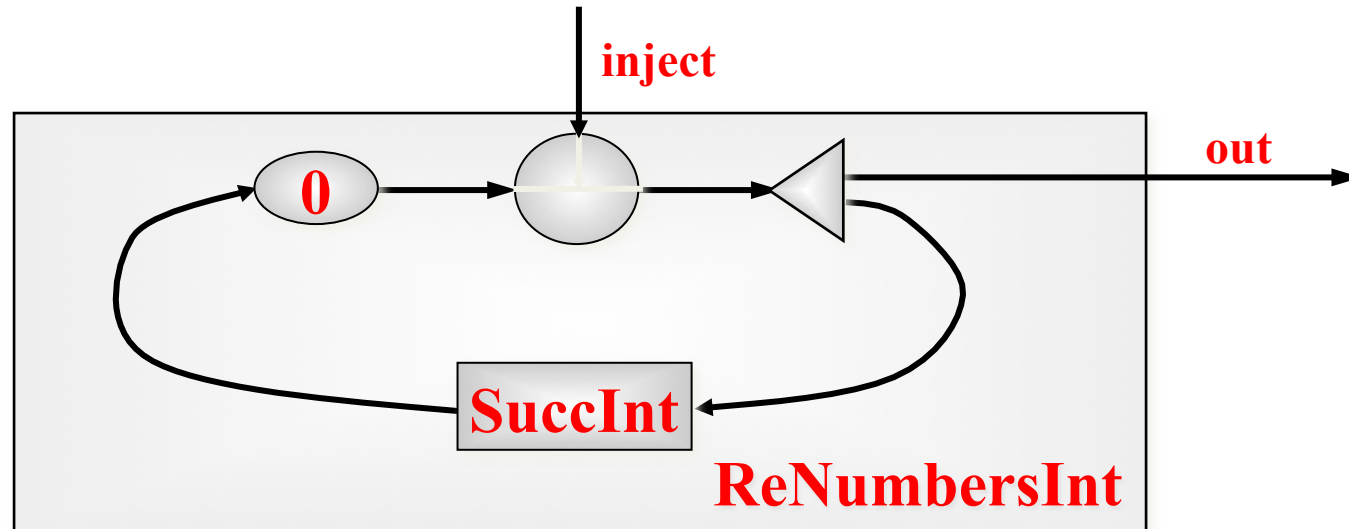


```
ScaleInt (s, in, out, inject) =
  (inject?s --> SKIP
   [PRI]
   in?a --> out!s*a --> SKIP
  );
ScaleInt (s, in, out, inject)
```

Note: **[]** is the (external) choice operator of CSP.

[PRI] is a prioritised version - giving priority to the event on its left.

Some Resettable Networks

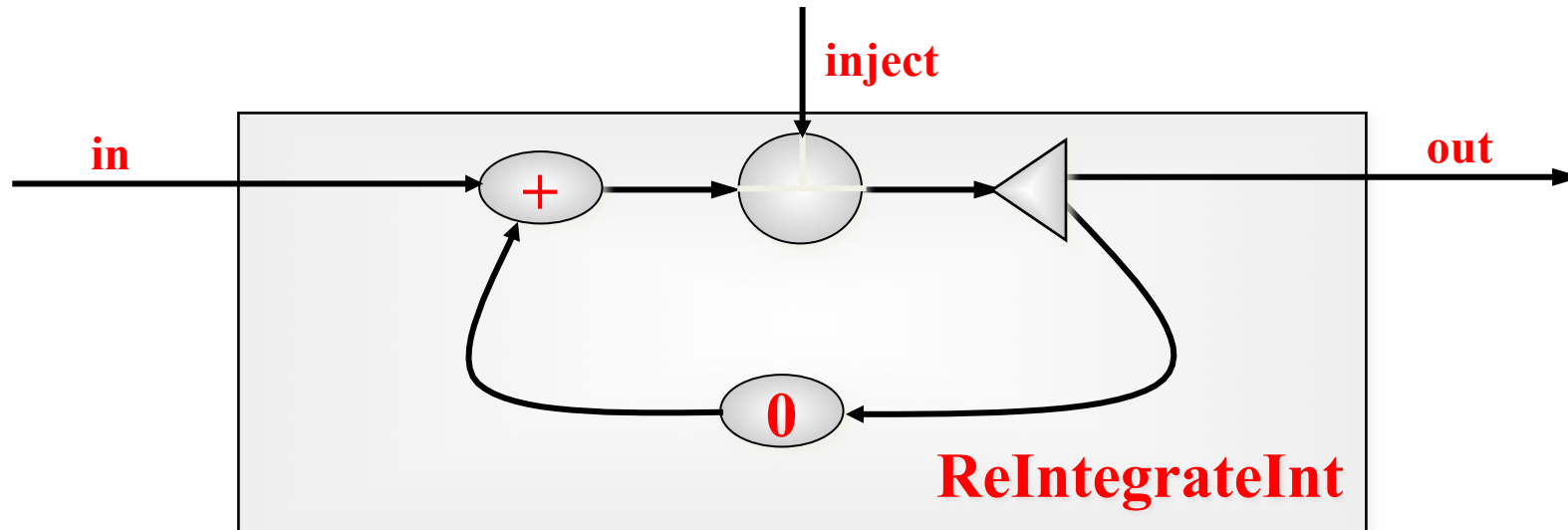


This is a *resettable* version of the **NumbersInt** process.

If nothing is sent down **inject**, it behaves as before.

But it may be reset to count from *any* number at *any* time.

Some Resettable Networks

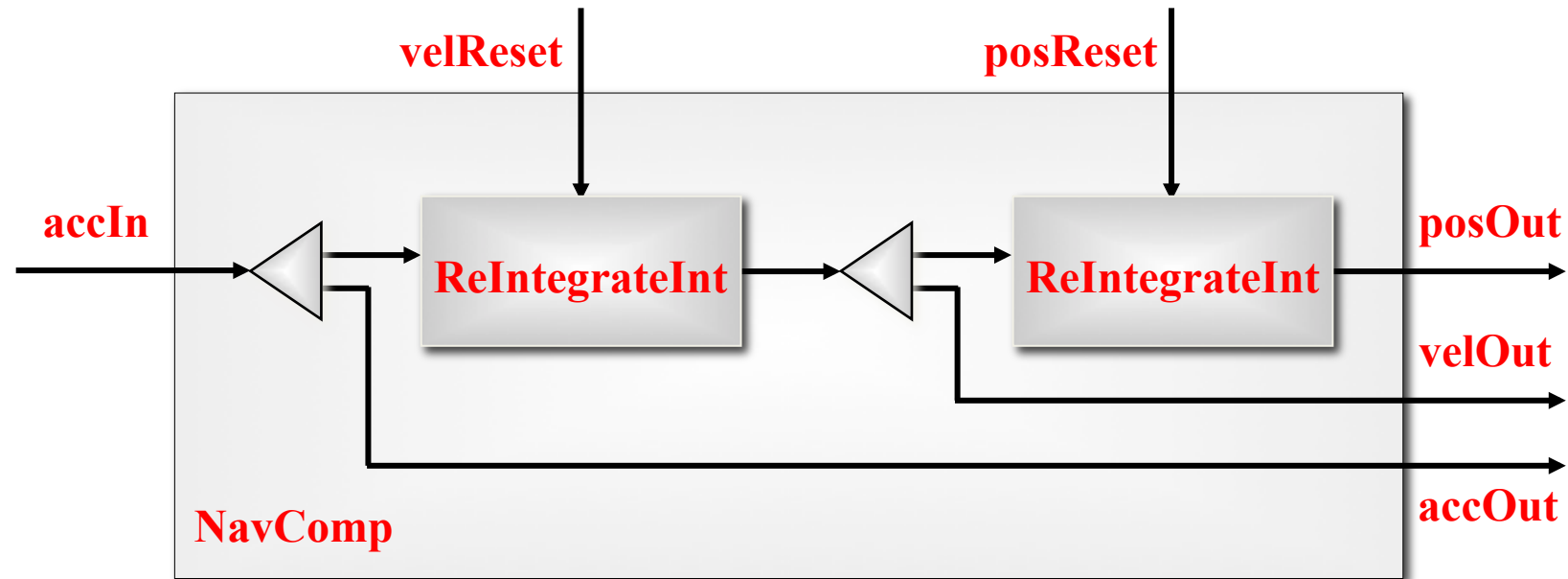


This is a *resettable* version of the **IntegrateInt** process.

If nothing is sent down **inject**, it behaves as before.

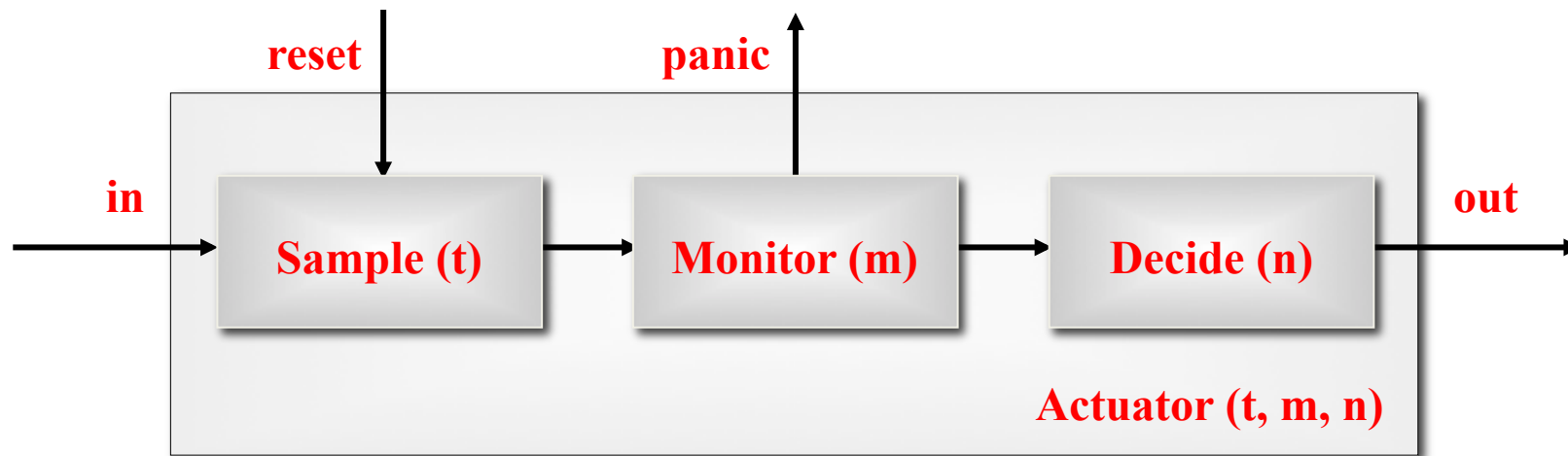
But its running sum may be reset to *any* number at *any* time.

An Inertial Navigation Component



- **accIn**: carries *regular* accelerometer samples;
- **velReset**: velocity *initialisation* and *corrections*;
- **posReset**: position *initialisation* and *corrections*;
- **posOut/velOut/accOut**: *regular* outputs.

Final Stage Actuator



- **Sample (t)** : every **t** time units, output *latest* input (or **null** if none); the value of **t** may be **reset**;
- **Monitor (m)** : copy input to output counting **nulls** - if **m** in a row, send panic message and terminate;
- **Decide (n)** : copy non-**null** input to output and *remember* last **n** outputs - convert **nulls** to a *best guess* depending on those last **n** outputs.

Use figures...

- Hoare did not use figures in his book
 - He does now!!!
- Figures helps you design your application
 - And helps us understand what you want to do
- A tool for graphical CSP programming will be introduced later...

Languages

- You will be introduced to a number of languages for programming with CSP
 - Occam
 - C/C++
 - Java
 - Python
 - Haskell
 - Go

CSPBuilder

CSPBuilder Motivation

- Scientists as programmers have issues
- Simplest approach often used, even if inefficient
 - or some use the most efficient approach possible, even if unmaintainable
 - Heavy usage of global values
- Coding practices are often out-dated
 - Poor style, little documentation, incomplete testing
 - Difficult to convert to modern programming techniques
 - Learning curve, poor training, and legacy code are issues.
- Common programming tools that helped make large-scale software efforts such as GNU/Linux successful are not uniformly utilized
 - minimal use of software configuration management, build systems
- **Result: Efficient parallel programs are not written by scientists**

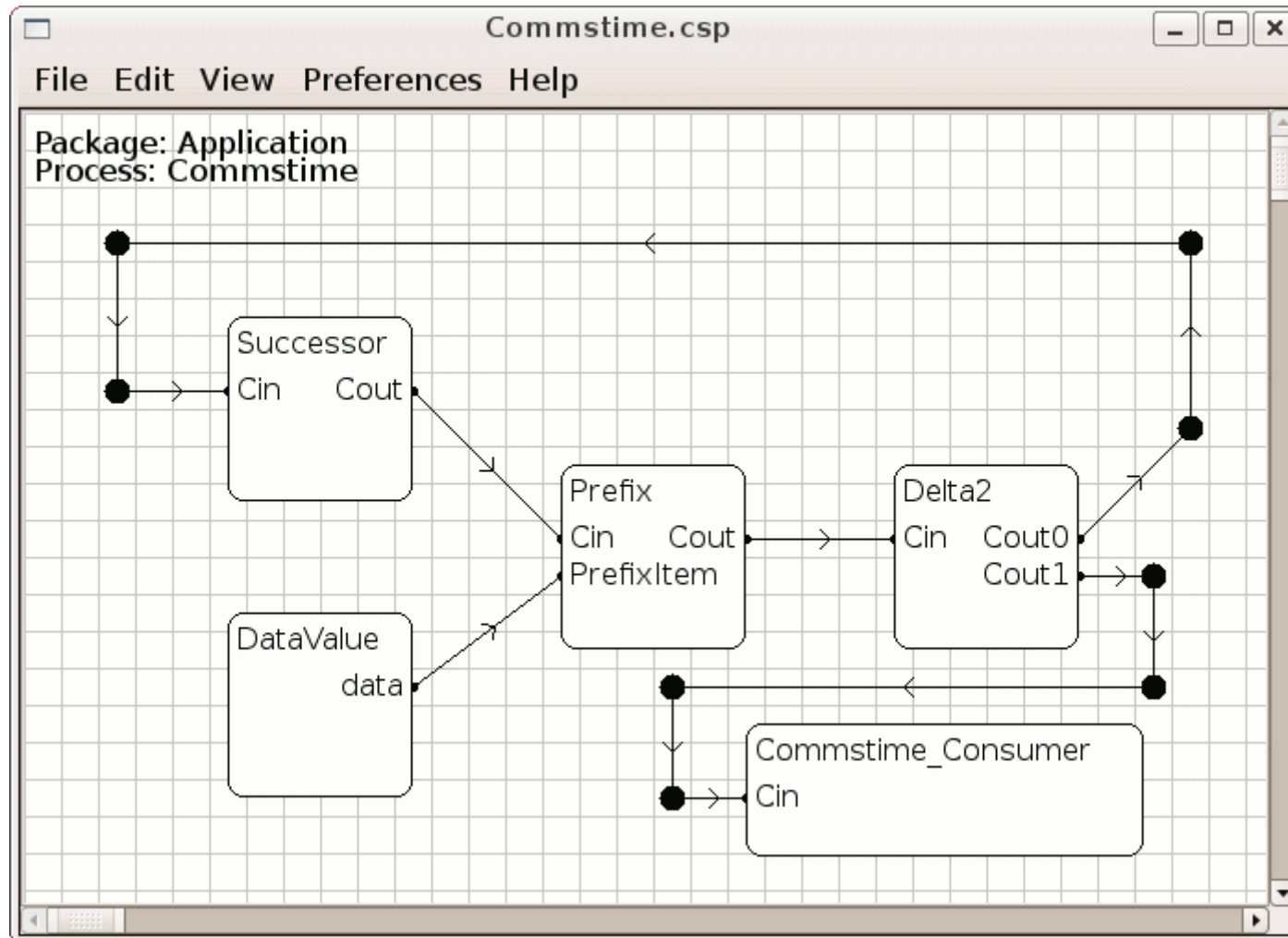
Motivation

- Scientists are capable of using graphical frontends that uses functions and connections
 - The Kepler Project
 - Knime
 - LabVIEW
 - FlowDesigner
 - Taverna

Can these be improved by adding CSP semantics?

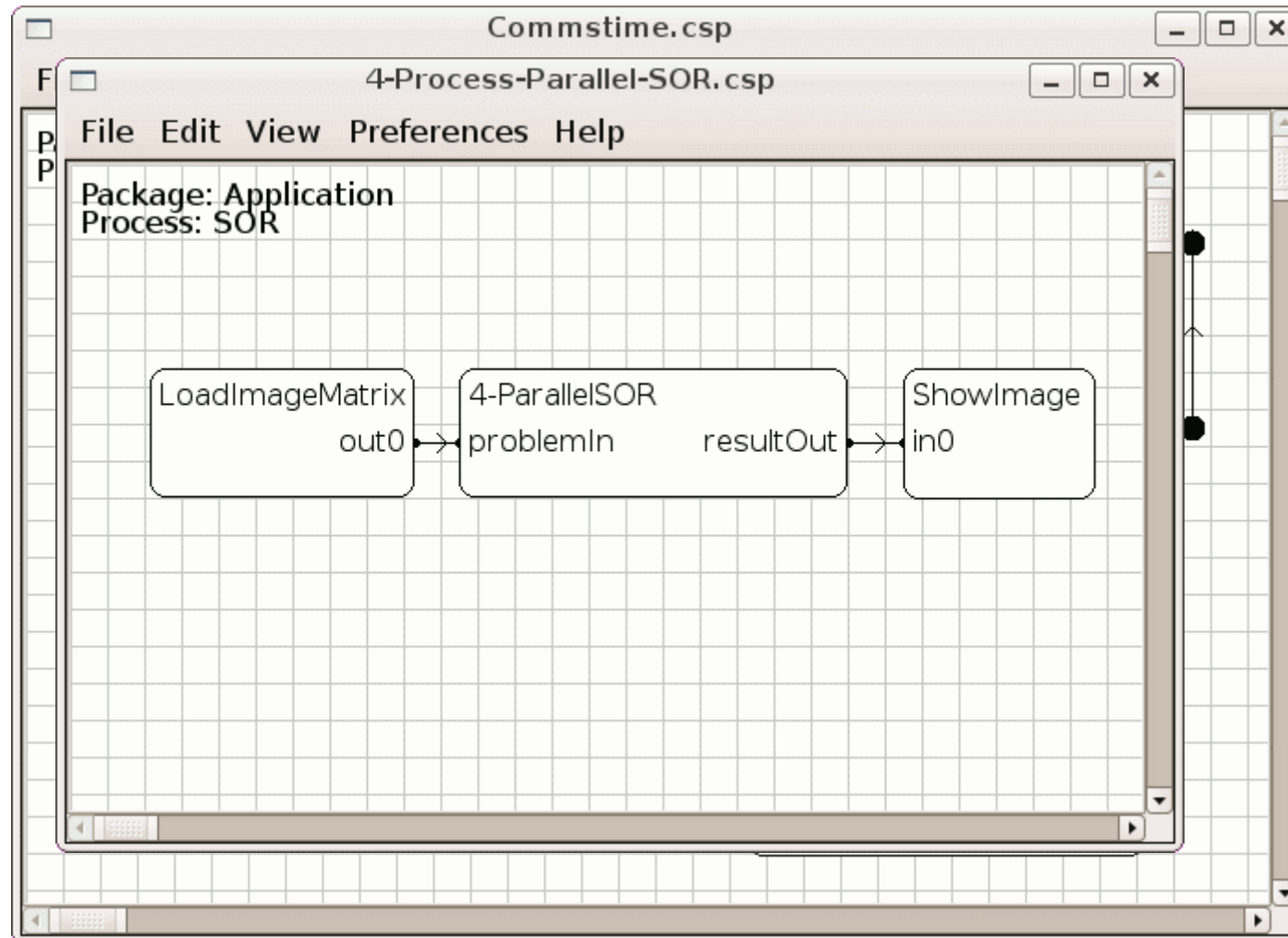
The framework

- A graphical tool that enables users



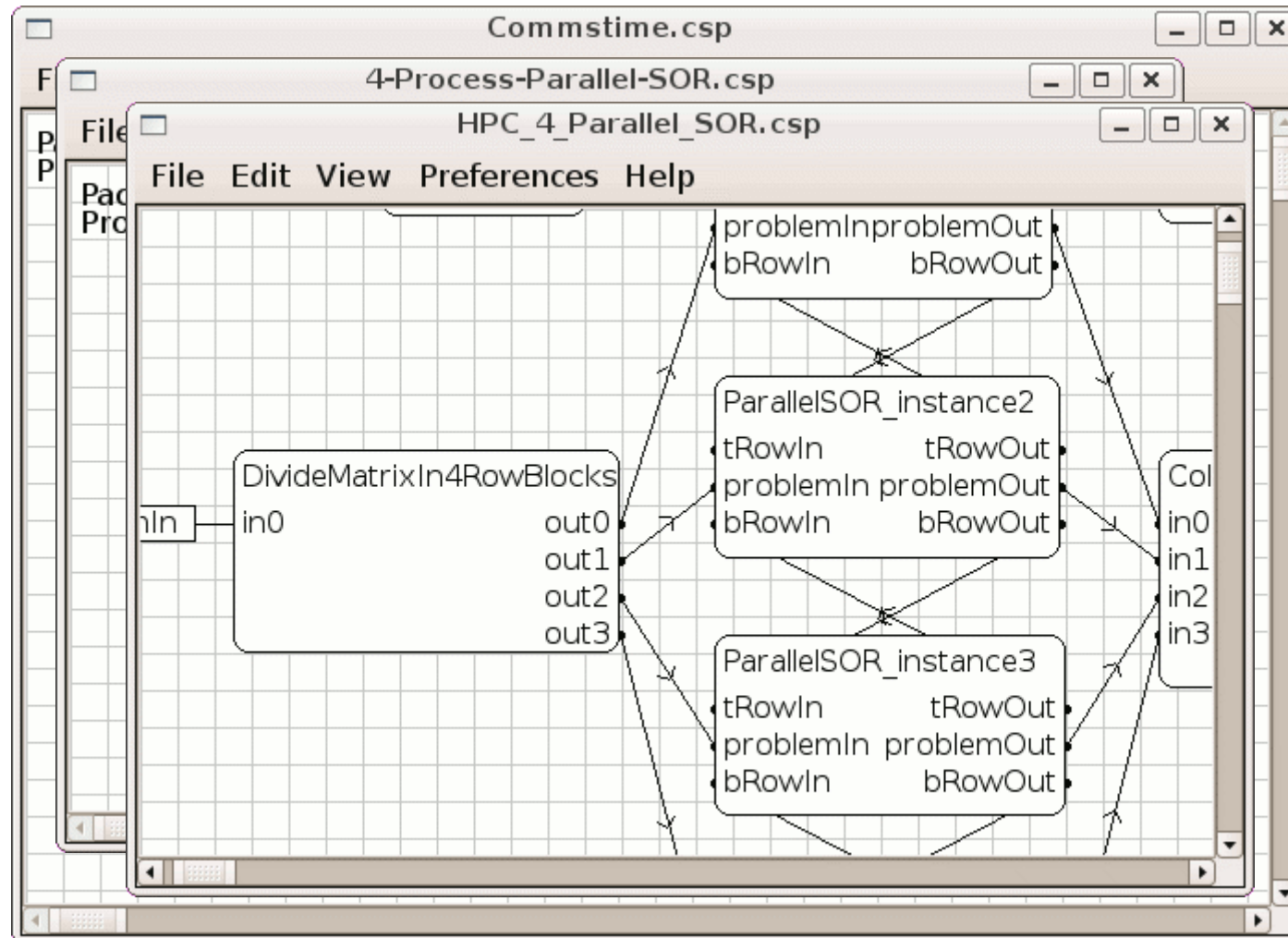
The framework

- A graphical tool that enables users



The framework

- A graphical tool that enables users



The framework

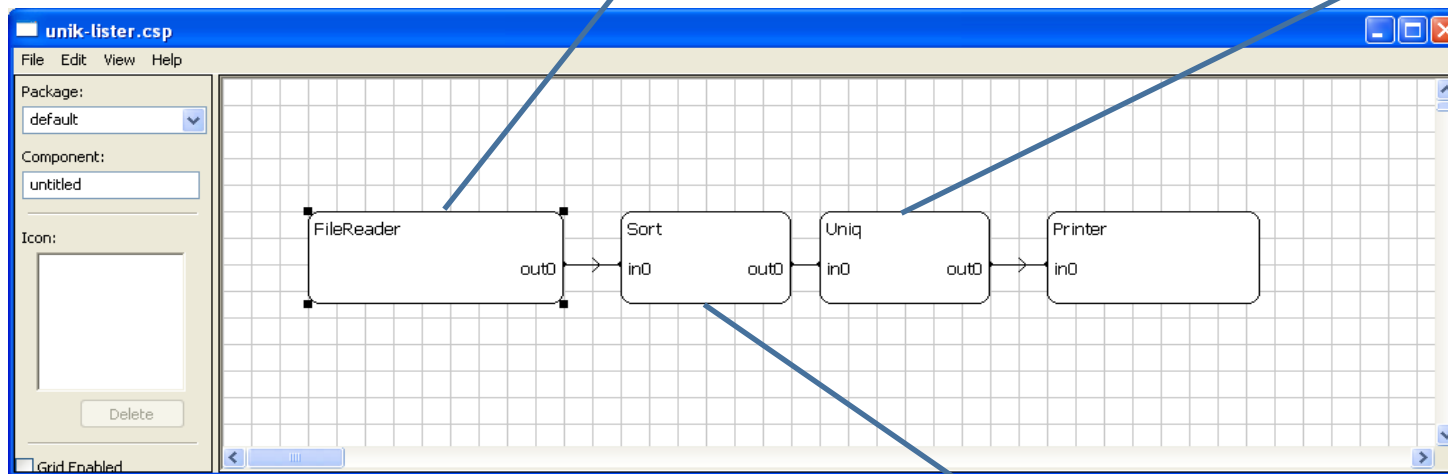
- An execute tool that executes the constructed CSP network using PyCSP

The framework

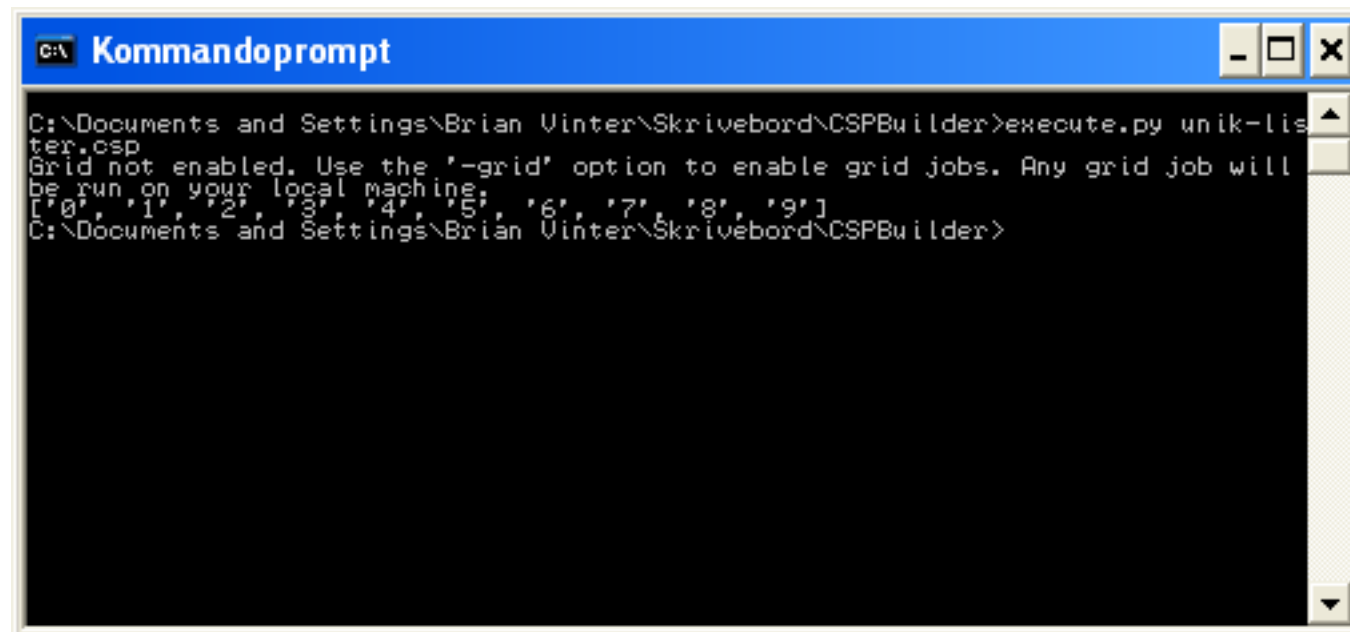
- A graphical component system that encourage code reuse.
- Easy creation of new components using a wizard.
- Compositional components is easily added to a component library.
- Components can be written in Python, C or Fortran.


```
def FileReaderFunc(out0):
    result0 = open("test.txt").readlines()
    out0(result0)
    poisonChannel(out0)
```

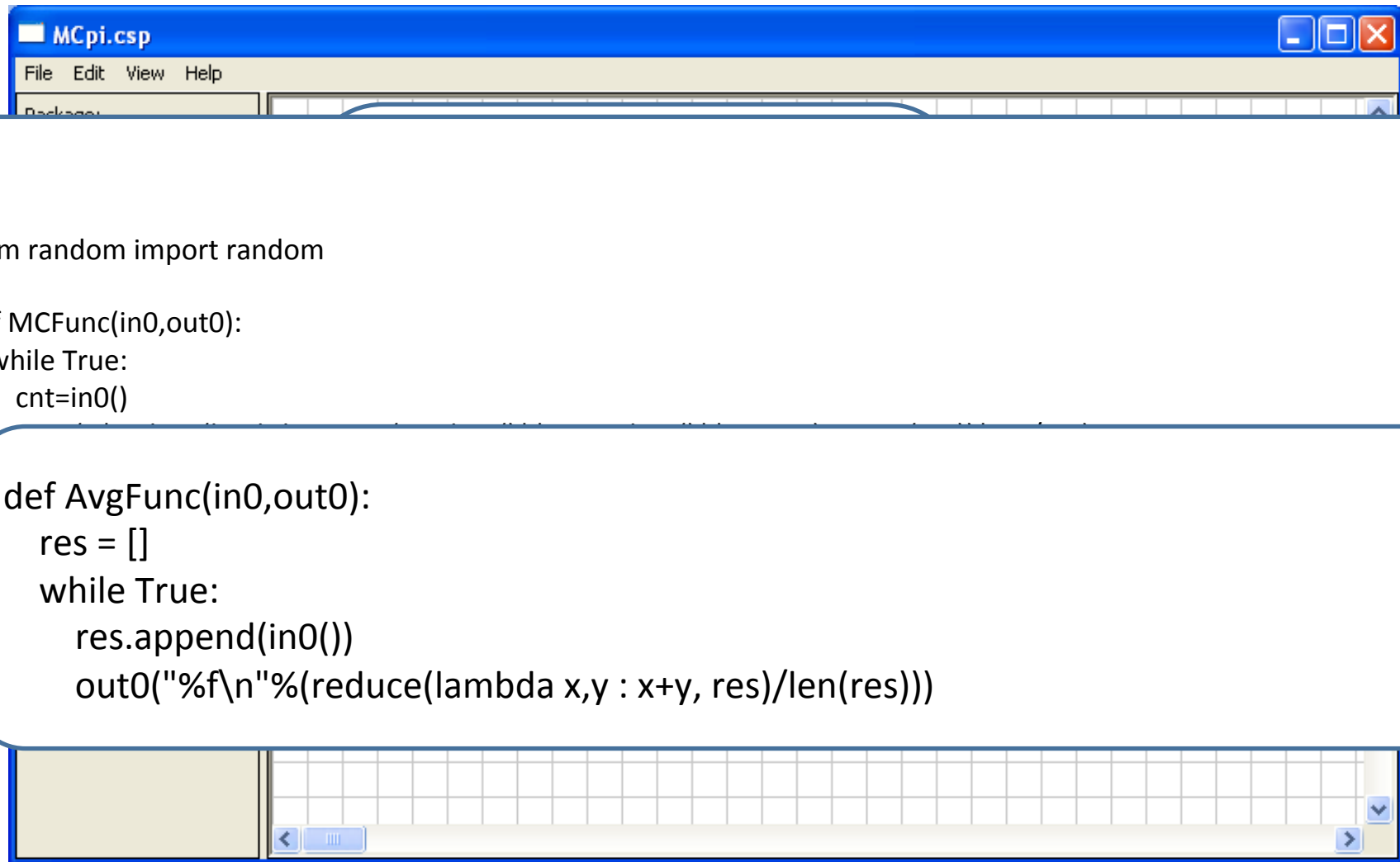
```
def UniqFunc(in0,out0):
    while True:
        val0 = in0()
        result0=[]
        for v in val0:
            if not v in result0:
                result0.append(v.strip())
        out0(result0)
```



```
def SortFunc(in0,out0):
    while True:
        val0 = in0()
        val0.sort()
        out0(val0)
```



```
C:\Documents and Settings\Brian Vinter\Skrivebord\CSPBuilder>execute.py unik-lister.csp
Grid not enabled. Use the '-grid' option to enable grid jobs. Any grid job will be run on your local machine.
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
C:\Documents and Settings\Brian Vinter\Skrivebord\CSPBuilder>
```



An Experiment: Successive Over-Relaxation

- Used in eScience applications for
- Calculating weather predictions
- Solving linear systems of equations

For all points compute

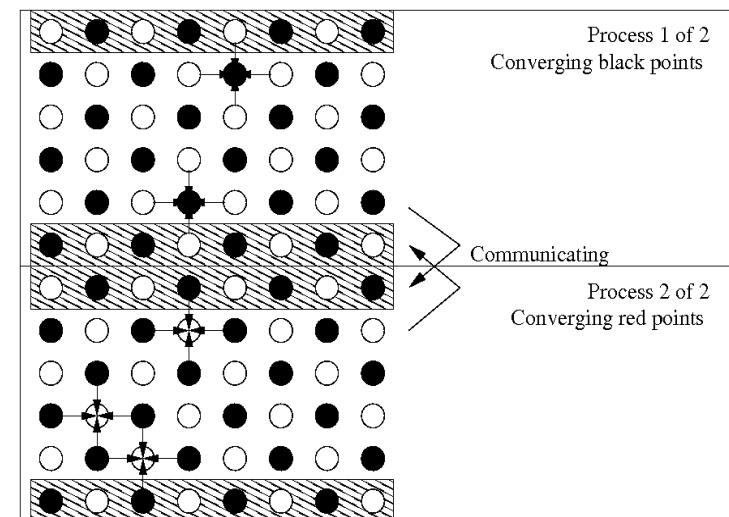
$$(x,y) = 0.2 * ((x-1,y)+(x+1,y)+(x,y-1)+(x,y+1))$$

until a stable state is reached where the overall change is less than epsilon.

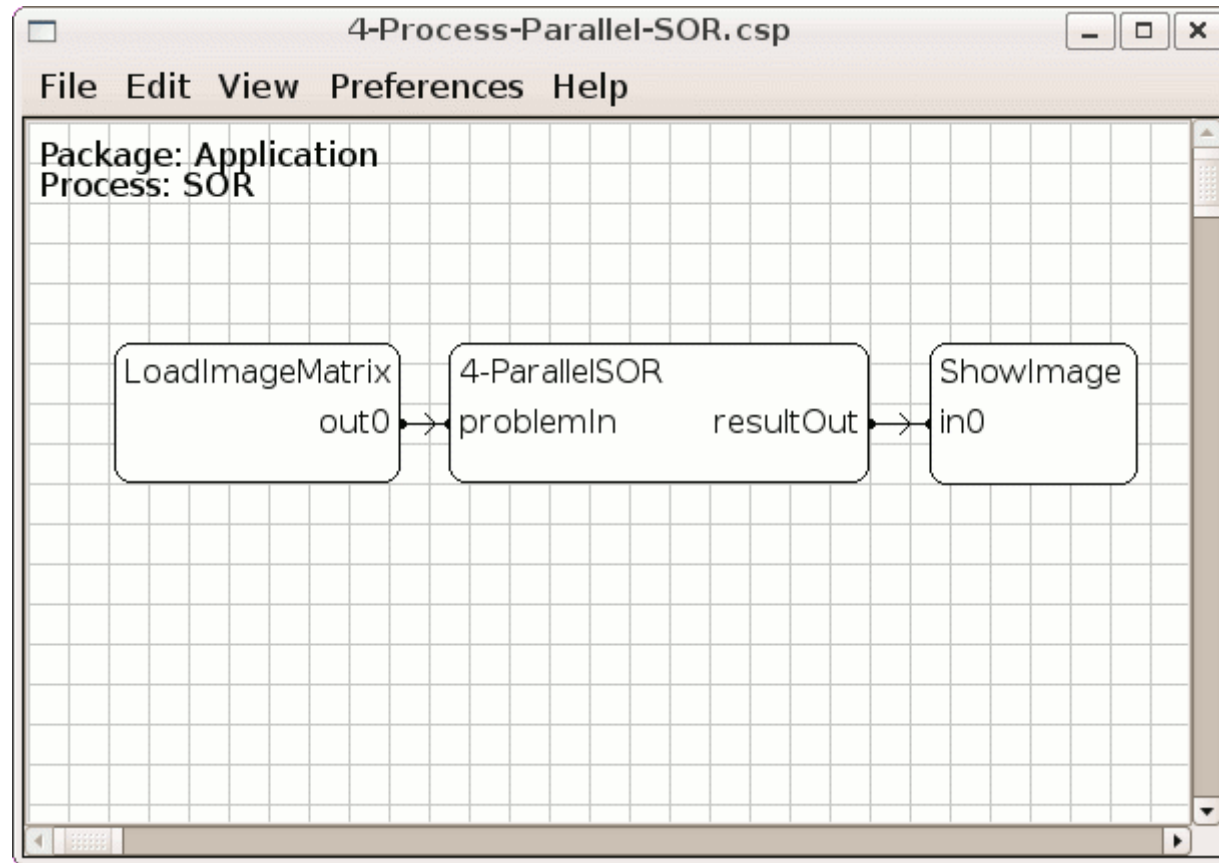
Successive Over-Relaxation in parallel

- A common and effective method to eliminate the dependency on neighboring points is to color them and divide them into 2 sets. This requires the double amount of convergence steps.
- Alternating between red and black points makes it possible to compute the convergence in parallel

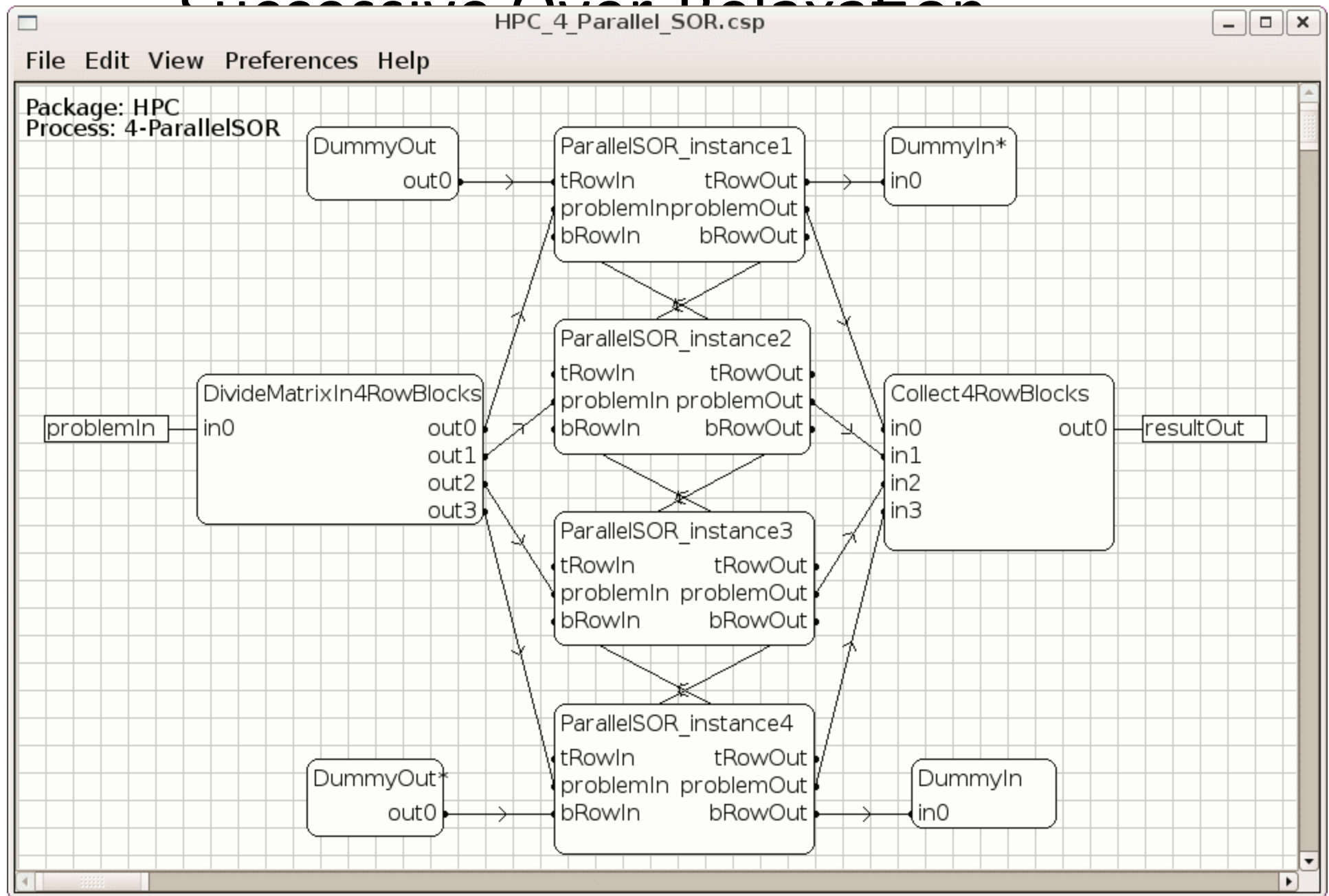
```
while (delta > epsilon) {  
    compute all black or white points  
    communicate border points  
}
```



Successive Over-Relaxation in CSPBuilder

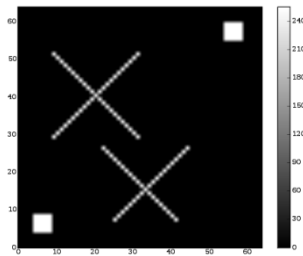


Successive Over-Relaxation



Successive Over-Relaxation results

- Results of distributing the CSP



Result set:

