

# Communicating Sequential Processes

# CSP

The Algebra  
(mostly from C.A.R Hoare)

# Communicating Sequential Processes

A mathematical theory for specifying and verifying complex patterns of behavior arising from interactions between concurrent objects.

**CSP** has a formal, and *compositional*, semantics that is in line with our informal intuition about the way things work.

**Claim**

# Why CSP?

- Encapsulates fundamental principles of communication.
- Semantically defined in terms of structured mathematical model.
- Sufficiently expressive to enable reasoning about deadlock and livelock.
- Abstraction and refinement central to underlying theory.
- Robust ***and commercially supported*** software engineering tools exist for formal verification.

# Why CSP?

- **CSP** libraries available for Java (**JCSP**, **CTJ**).
- Ultra-lightweight kernels have been developed yielding ***sub-microsecond*** overheads for context switching, process startup/shutdown, synchronized channel communication and high-level shared-memory locks.
  - not yet available for JVMs (or Core JVMs! )
- Easy to learn and easy to apply ...

# Why CSP?

- After 5 hours teaching:
  - exercises with 20-30 threads of control
  - regular and irregular interactions
  - appreciating and eliminating race hazards, deadlock, etc.
- **CSP** is (parallel) architecture neutral:
  - message-passing
  - shared-memory

# So, what is CSP?

**CSP** deals with *processes*, *networks* of processes and various forms of *synchronization / communication* between processes.

A network of processes is also a process - so **CSP** naturally accommodates layered network structures (*networks of networks*).

We do not need to be mathematically sophisticated to work with **CSP**. *That sophistication is pre-engineered into the model.* We benefit from this simply by using it.

# INTRODUCTION

- CSP is a simple programming language designed for multiprocessor machines
- Its key feature is its reliance on **non-buffered** message passing with **explicit naming** of source and destination processes
- CSP uses **guarded commands** to let processes wait for messages coming from different sources.

# The Model

- Each process runs on its own processor
- All communications between concurrent processes are made through message passing.
- CSP relies on the most primitive message passing mechanism:
  - Non-buffered sends and receives
  - Explicit naming of source and destination processes

# Non buffered sends and receives

- When a process issues a send( ), the system call does not complete until the message is received by another process  
*Also called “blocking send”*
- When a process issues a receive( ), the system call does not complete until a message is received by the process  
*Also called “blocking receive”*

# Explicit naming

- Also known as **direct naming**
- A process issuing a `send( )` specifies the name of the process to which the message is sent
- A process issuing a `receive( )` specifies the name of the process from which it expects a message

# The alternative: indirect naming

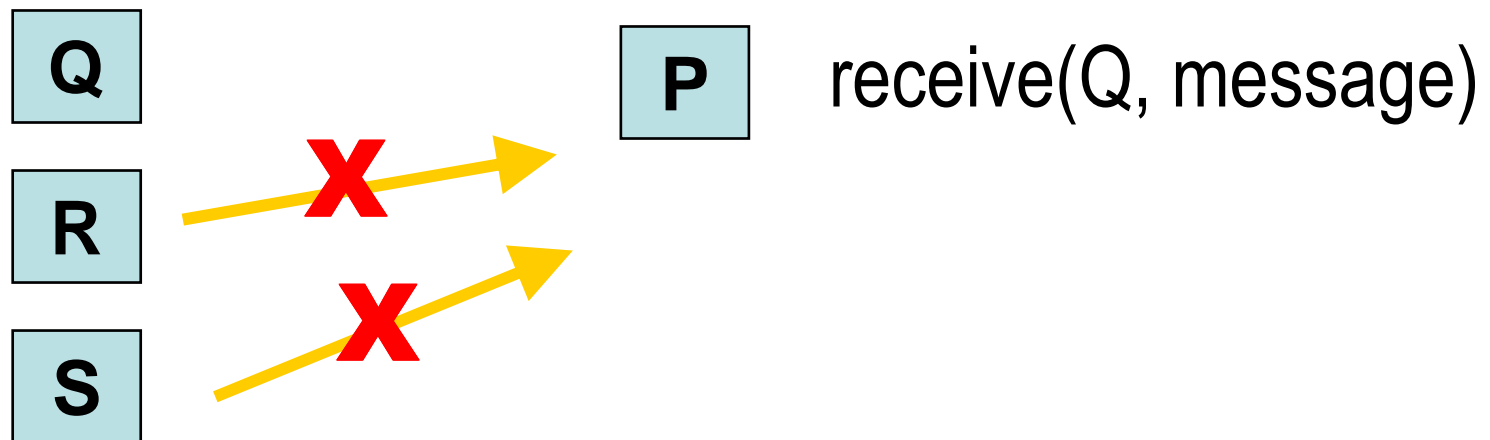
- Most message passing architectures include an intermediary entity often called port but also mailbox, message queue or socket
- A process issuing a `send( )` specifies the port number to which the message is sent
- A process issuing a `receive( )` specifies a port number and will wait for the the first message that arrives at that port

# The problem (I)

- Processes should be able to receive messages from different senders
- Can use **blocking receive** and **indirect naming**
  - Process issuing a receive will wait for first message arriving at that port
- Can also use **non-blocking receive** and **direct naming**
  - Requires receiving process to poll senders

# The problem (II)

- Using blocking receives with direct naming does not allow the receiving process to receive any messages from any process but the one it has specified



# The problem (III)

- CSP paper presents a solution involving **guarded commands**
- Guarded commands are a **very unconventional** programming construct
- So is CSP syntax
  - Do not let yourself be intimidated by it

# CSP

- A CSP program consists of a sequence of commands
- Commands can either succeed or fail
  - $a = b$  will fail if either
    - $b$  is undefined or
    - the types of  $a$  and  $b$  do not match
- Success and failure of a command are **time-dependent**

# Parallel Commands

- $\langle command\_list1 \rangle \parallel \langle command\_list2 \rangle$   
specifies that the two command lists should be executed in parallel
- $\langle CL1 \rangle \parallel \langle CL2 \rangle \parallel \langle \dots \rangle \parallel \langle CLn \rangle$   
can have more than two processes per command lists
- $producer:: \langle CL1 \rangle \parallel consumer:: \langle CL2 \rangle$   
can add process labels

# I/O Commands (I)

- Input command:  
*<source\_process> ? <target value>*  
keyboard ? m
- Output command:  
*<destination\_process> ! <value>*  
screen ! average
- Input and output commands are **blocking**:
  - Messages are **never buffered**

# I/O Commands (II)

- Communication will take place whenever :
  - Process  $P$  executes an input command specifying process  $Q$  as its source and
    - Process  $Q$  executes an output command specifying process  $P$  as its destination
    - The target variable in the input statement matches the value in the output statement.

# I/O Commands (III)

- An **input command** will **fail** when its **source** process **terminates**
- An **output command** will **fail** when either
  - its **destination** process **terminates**
  - the **value** it attempts to send becomes **undefined**

# Guarded Commands (I)

- Command preceded by a **guard** :  
$$\mathbf{v > 0; client?P() \rightarrow v := v - 1}$$
- Previous line reads
  - *When value is positive, wait for P() message from process client*
  - *When message arrives, decrement value*

# Guarded Commands (II)

- A guard consists of
  - a possibly empty list of declaration and Boolean expressions
  - optionally followed by a single input statement

**`in < out + 10; producer?buffer(in mod 10) ->`**

# Guarded Commands (III)

- Execution of a guarded command is **delayed** until either
  - The **guard succeeds** and the command is executed or
  - The **guard fails** and the command aborts without being executed

# Alternative Commands (I)

- An alternative command consists of
  - list of one or more guarded commands
  - separated by "||"
  - surrounded by square brackets

**[  $x \geq y \rightarrow \text{max} := x$  ||  $y \geq x \rightarrow \text{max} := y$  ]**

# Alternative Commands (II)

- An alternative command specifies the execution of exactly one of its components
  - If all of them fail, the command fails
  - Otherwise an arbitrary component with a successfully executable guard is selected and executed
- **Order of components does not matter**  
**[  $x \geq y \rightarrow \text{max} := x \parallel y \geq x \rightarrow \text{max} := y$  ]**

# Alternative commands (III)

- We can now write

[

**Q ? msg -> <process msg from Q> ||**

**R ? msg -> <process msg from R> ||**

**S ? msg -> <process msg from S>**

]

- Process will wait for first message from any of three senders Q, R and S

# Repetitive command

- Alternative command preceded by an asterisk

**\*[ i > 0 -> fact := fact\*i; i:=i - 1 ]**

- Executed repeatedly until they fail:  
in the previous command until i becomes zero

# A bounded buffer

```
buffer: (0..9) portion;  
in, out : integer; in:=0; out:=0;  
*  
  in<out + 10; producer?buffer(in mod 10) ->  
    in:=in + 1  
  ||  
    out < in; consumer?more() ->  
      consumer!buffer(out mod 10) -> out:=out + 1  
]
```

# One shorthand notation

$(i:1..100) X(i)?V() \rightarrow v:=v + 1$

stands for

$X(1)?V() \rightarrow v:=v + 1 \parallel$

$\dots \parallel$

$X(100)?V() \rightarrow v:=v + 1$

# A semaphore

**v : integer; v := 0;**

**\*[**

**v > 0; (i:1..100) X(i)?P() ->**  
**v := v - 1**

**||**

**(i:1..100) X(i)?V() ->**  
**v := v + 1**

**]**

# A binary semaphore

**v : integer; v := 0;**

**\*[**

**v > 0; (i:1..100) X(i)?P() ->**  
**v := 0**

**||**

**(i:1..100) X(i)?V() ->**  
**v := 1**

**]**

# Working with CSP

No algebra required  
(mostly from Peter Welch)

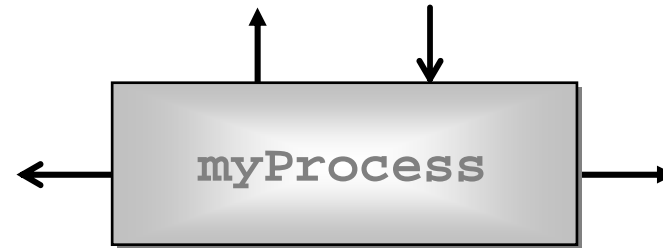
# Processes



myProcess

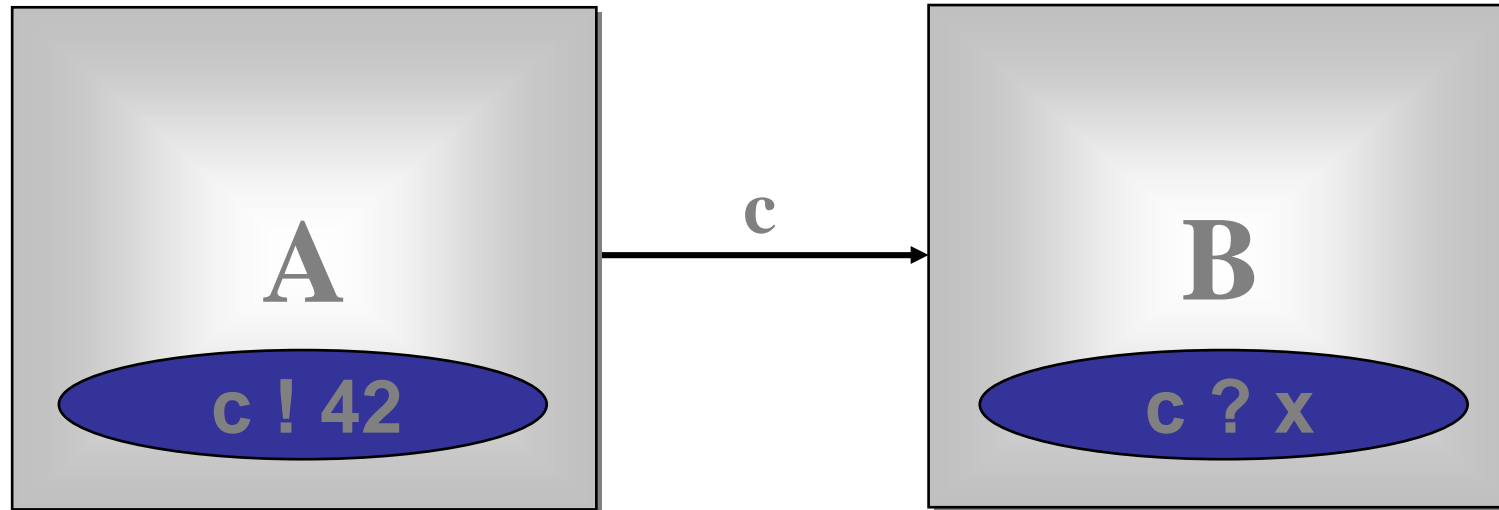
- A **process** is a component that encapsulates some data structures and algorithms for manipulating that data.
- Both its data and algorithms are **private**. The outside world can neither see that data nor execute those algorithms! [They are not *objects*.]
- The algorithms are executed by the process 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

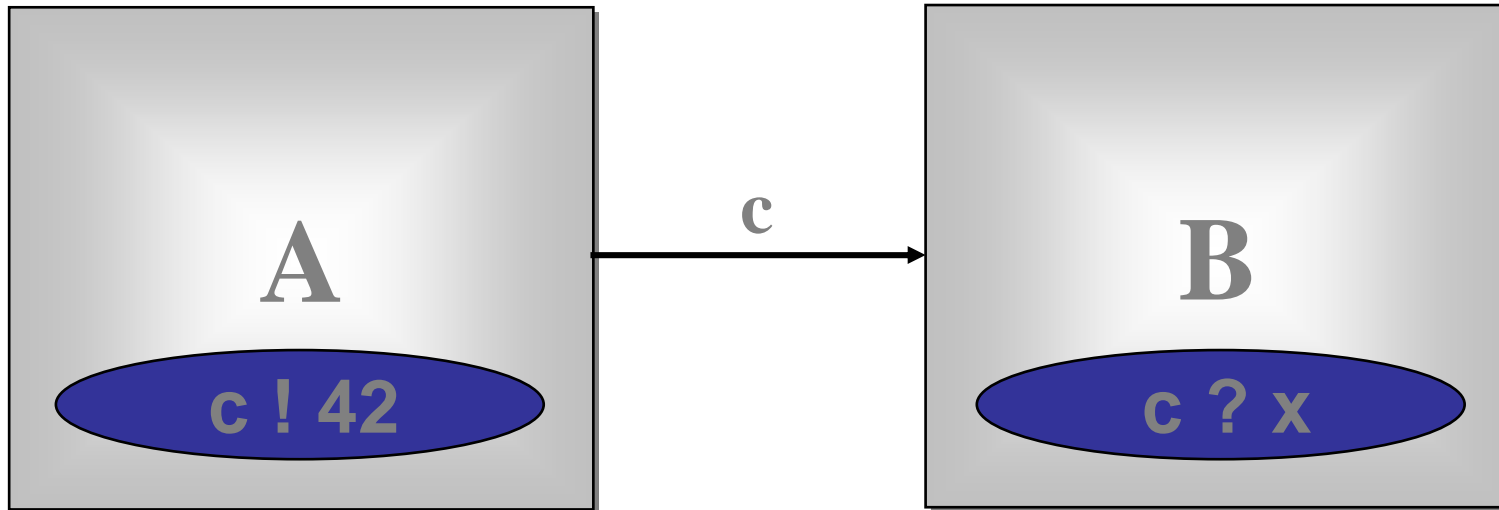


**A** may write on  $c$  at any time, but has to wait for a *read*.

**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  $c$ .

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

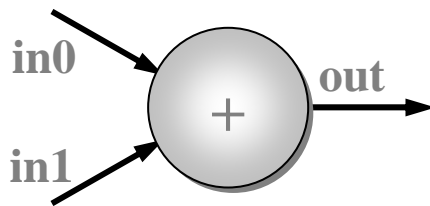
# 'Legoland' Catalog



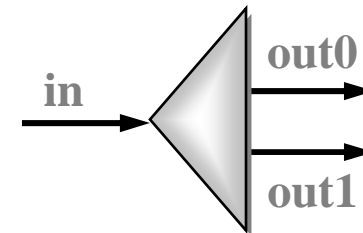
**IdInt (in, out)**



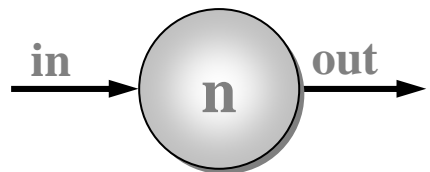
**SuccInt (in, out)**



**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.

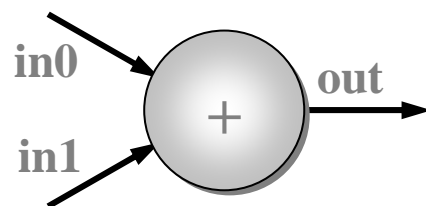
Let's look at some **CSP** *pseudo-code* for these processes ...



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

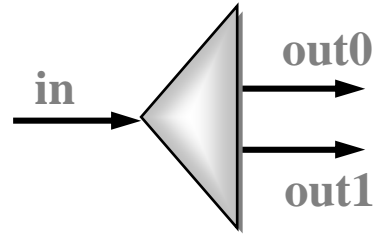


`SuccInt (in, out) = in?x --> out!(x + 1) --> SuccInt (in, out)`



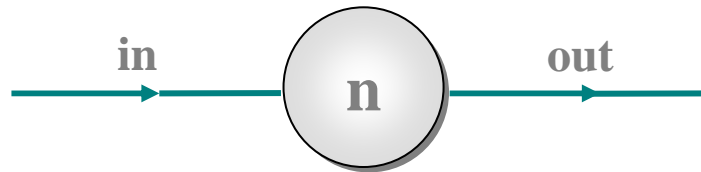
Note the parallel input

`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)
```

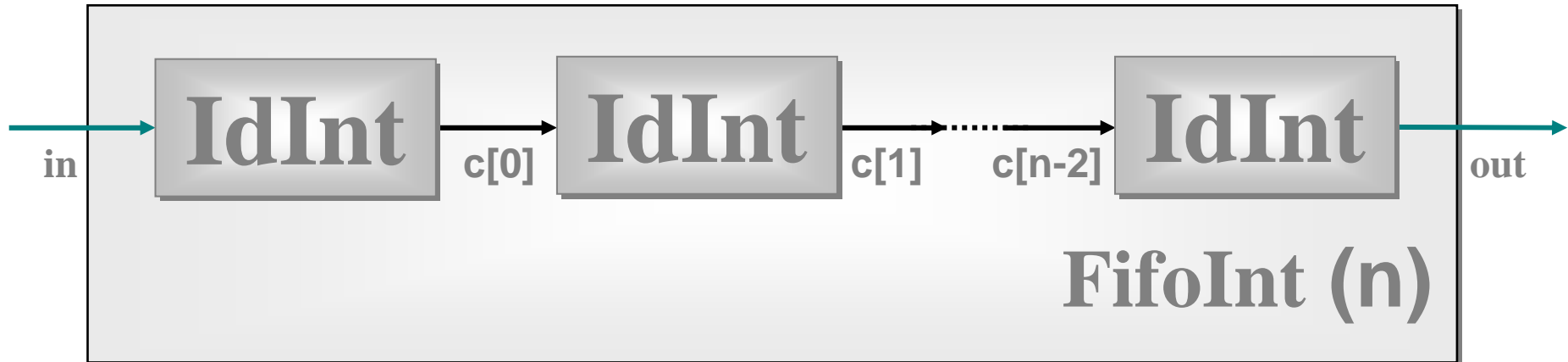


```
PrefixInt (n, in, out) = out!n --> IdInt (in, out)
```



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

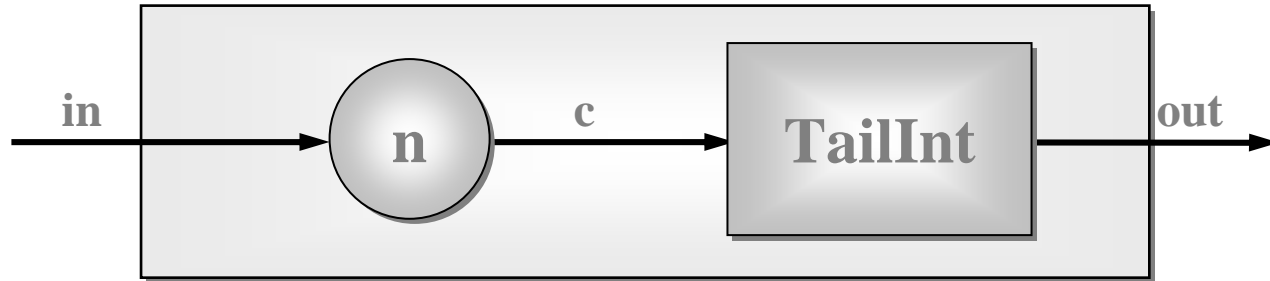
# A Blocking FIFO Buffer



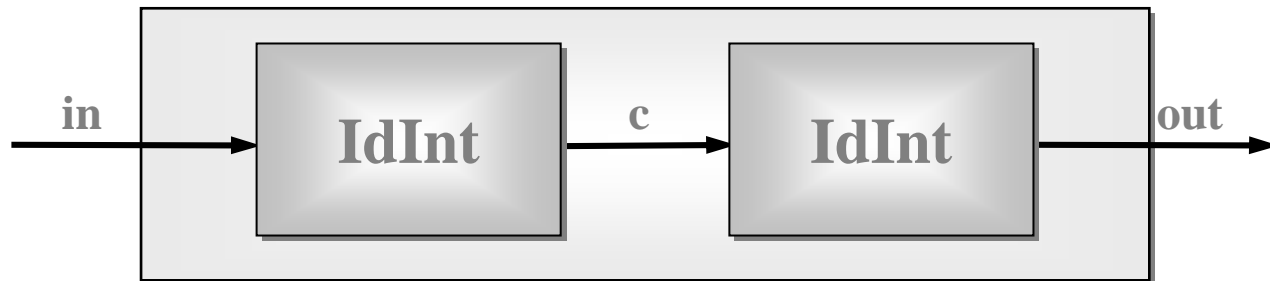
```
FifoInt (n, in, out) =  
  IdInt (in, c[0]) ||  
  ([|| i = 0 FOR n-2] IdInt (c[i], c[i+1])) ||  
  IdInt (c[n-2], out)
```

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!

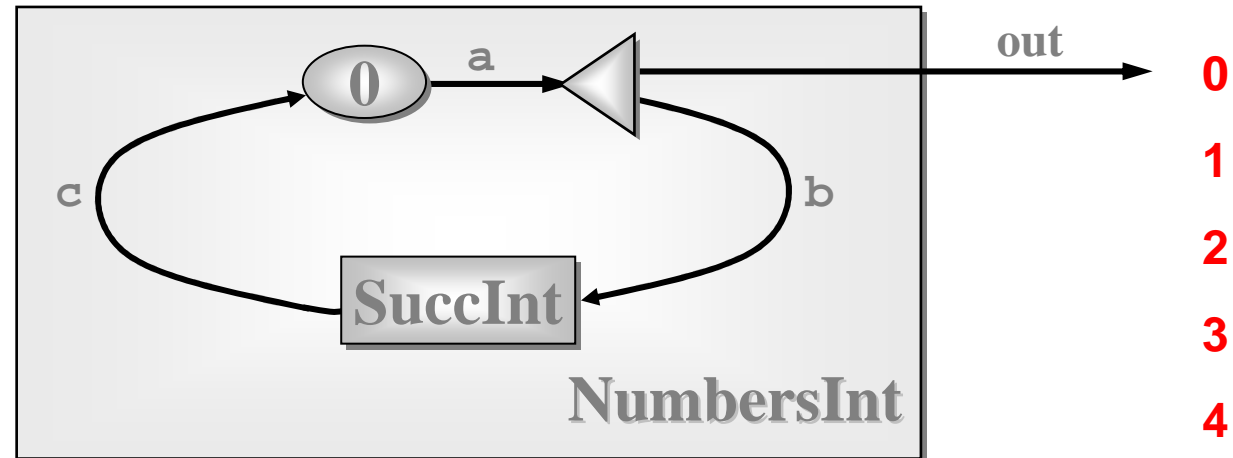
The good news is that we can 'see' this semantic equivalence with just one glance.

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

This quickly builds up confidence ...



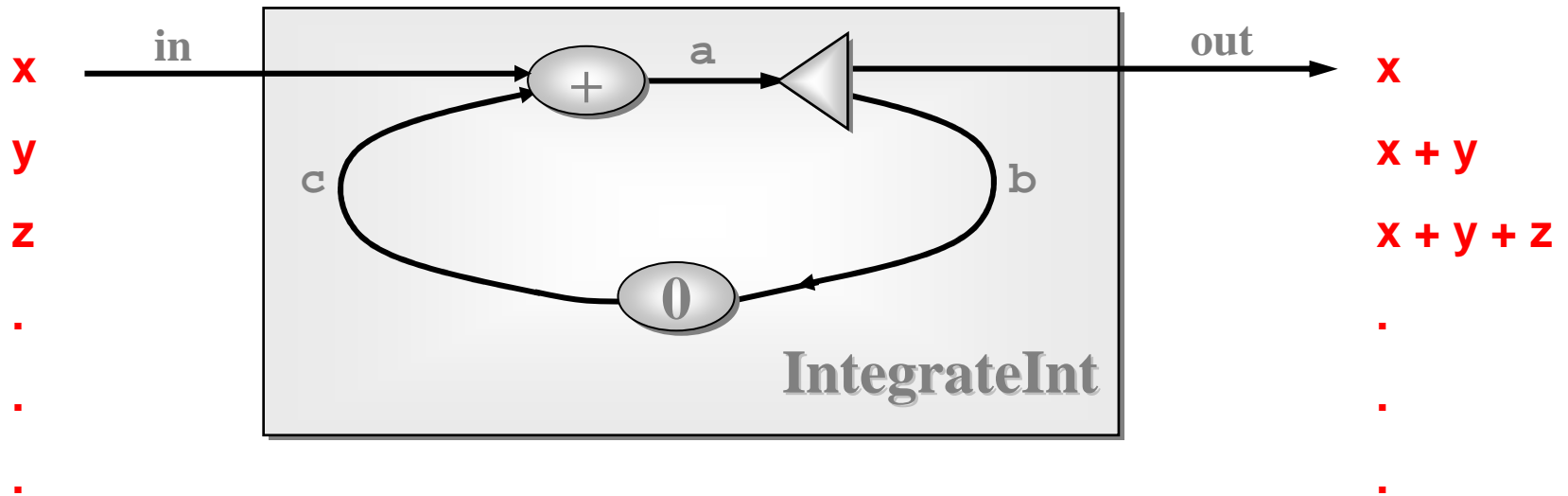
# Some Simple Networks



```
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.

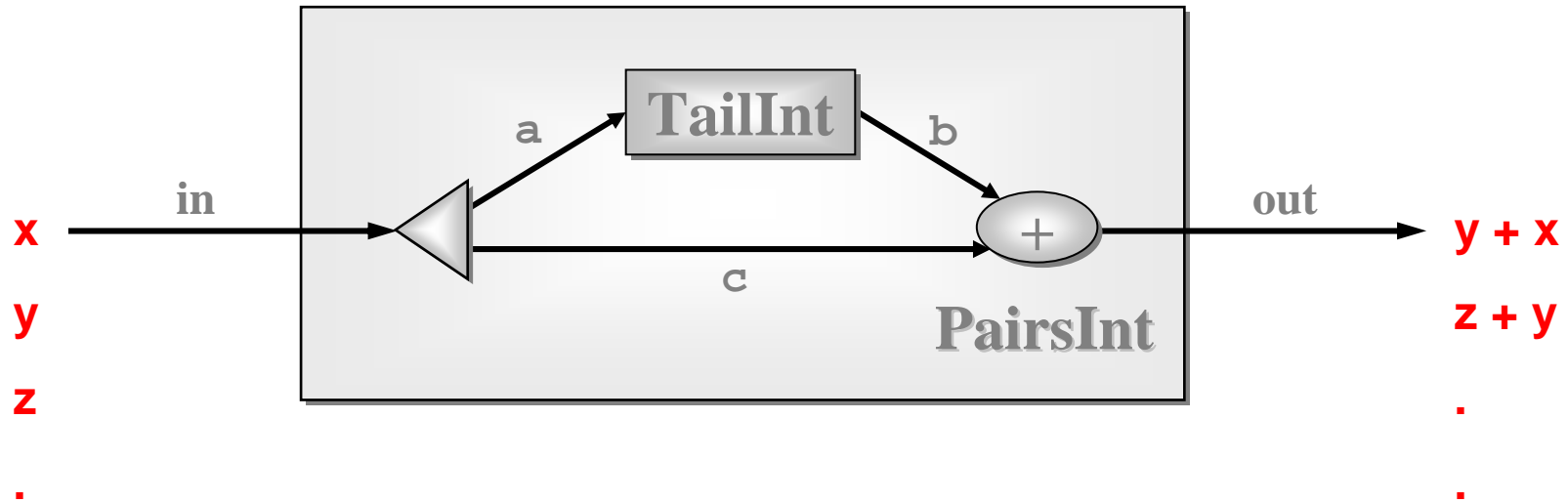
# Some Simple Networks



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

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

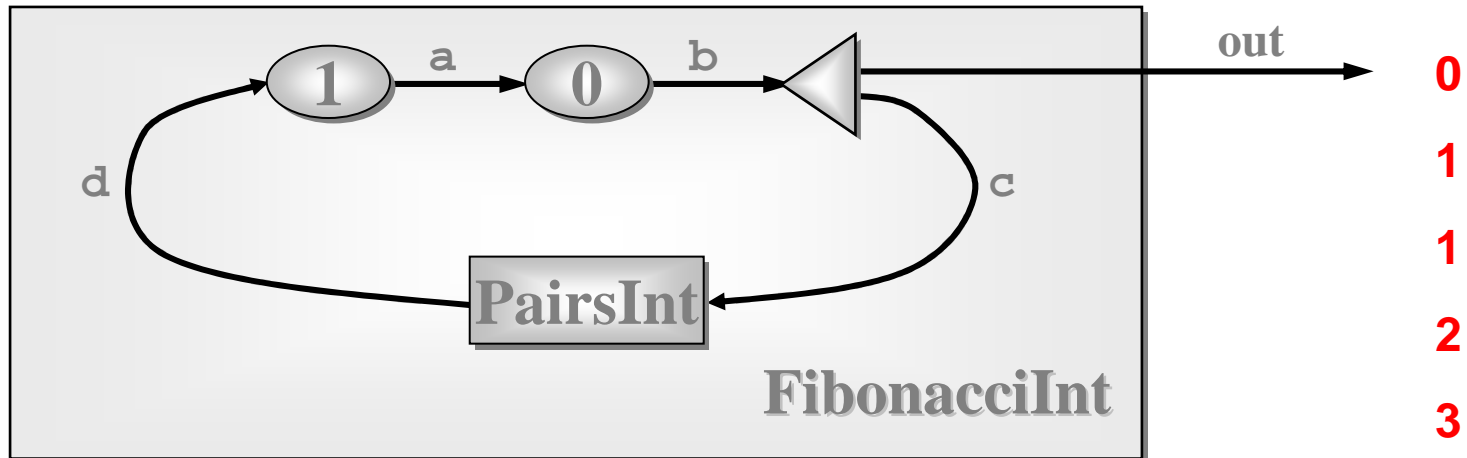
# Some Simple Networks



· `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

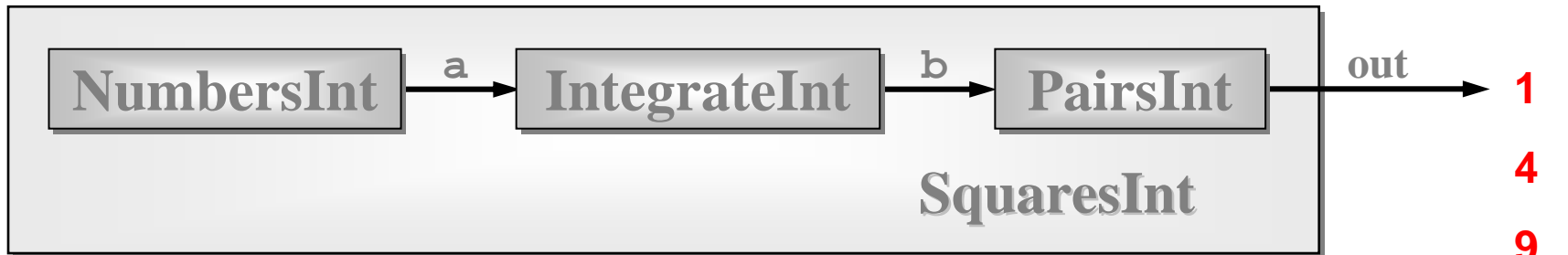


```

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



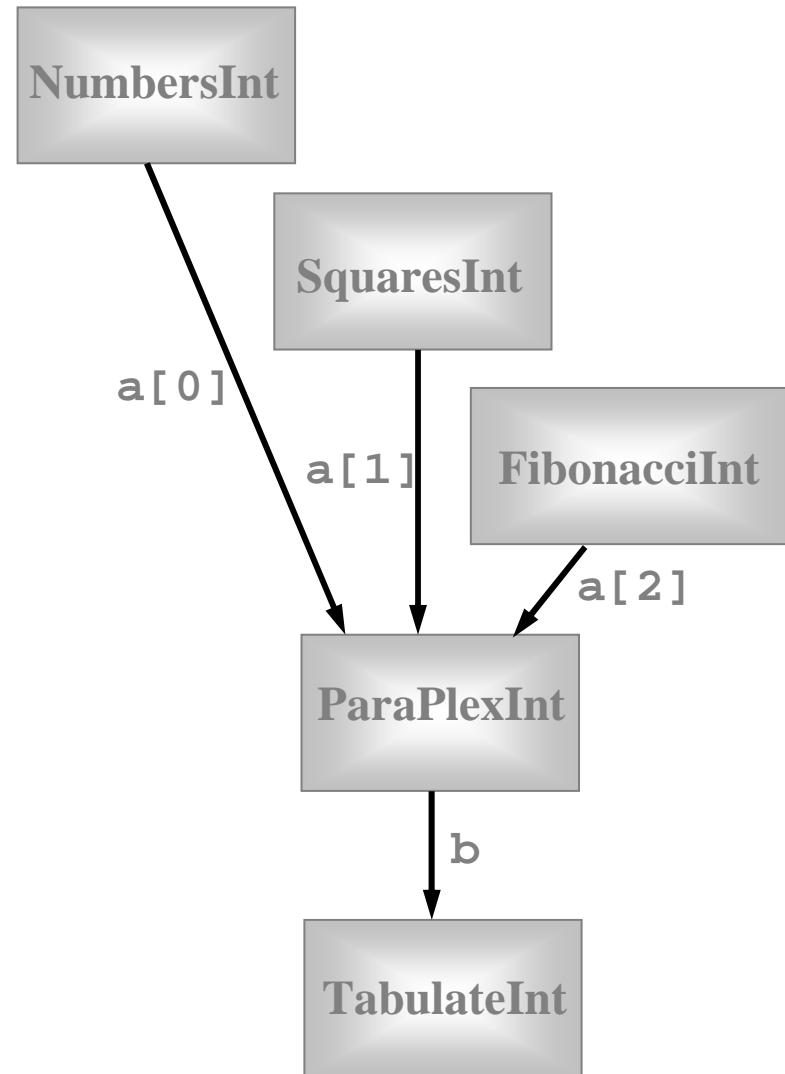
`SquaresInt (out) = NumbersInt (a) ||`  
`IntegrateInt (a, b) ||`  
`PairsInt (b, out)`

**Note: the traffic on individual channels:**

<code>&lt;a&gt;</code>	<code>= [0, 1, 2, 3, 4, 5, 6, 7, 8, ...]</code>	<code>1</code>
<code>&lt;b&gt;</code>	<code>= [0, 1, 3, 6, 10, 15, 21, 28, 36, ...]</code>	<code>4</code>
<code>&lt;out&gt;</code>	<code>= [1, 4, 9, 16, 25, 36, 49, 64, 81, ...]</code>	<code>9</code>
		<code>16</code>
		<code>25</code>
		<code>36</code>
		<code>49</code>
		<code>64</code>
		<code>81</code>
		<code>.</code>
		<code>.</code>

# Quite a Lot of Processes

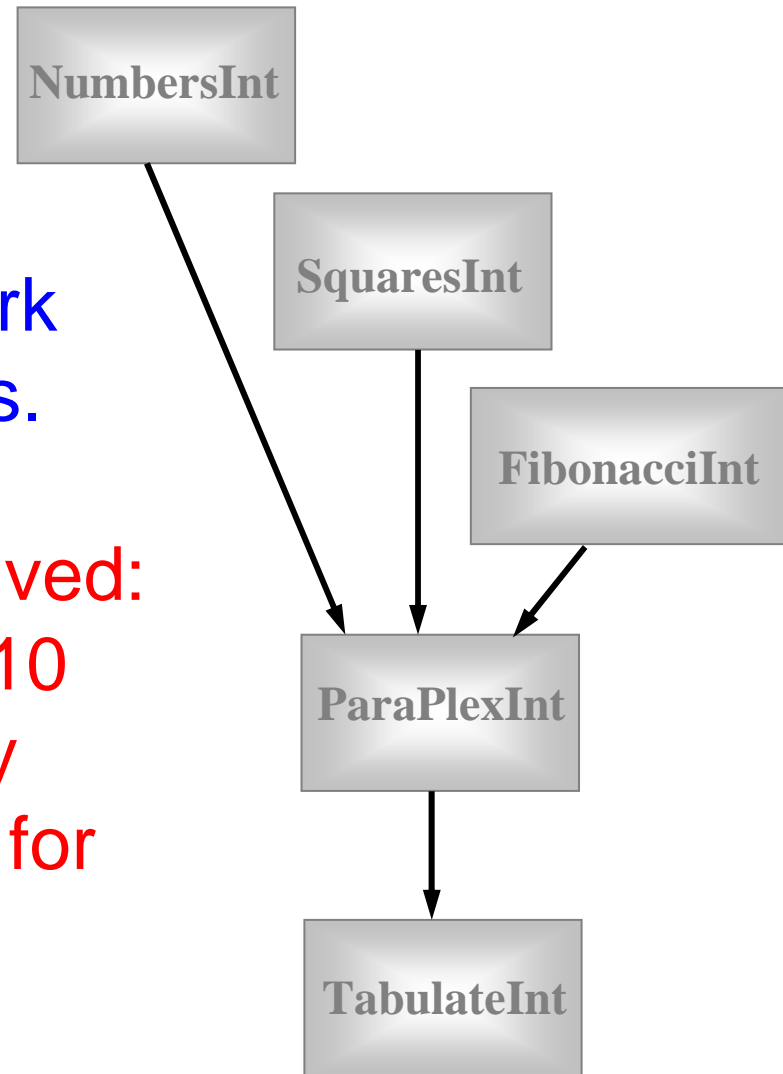
```
NumbersInt (a[0]) ||  
SquaresInt (a[1]) ||  
FibonacciInt (a[2]) ||  
ParaPlexInt (a, b) ||  
TabulateInt (b)
```



# Quite a Lot of Processes

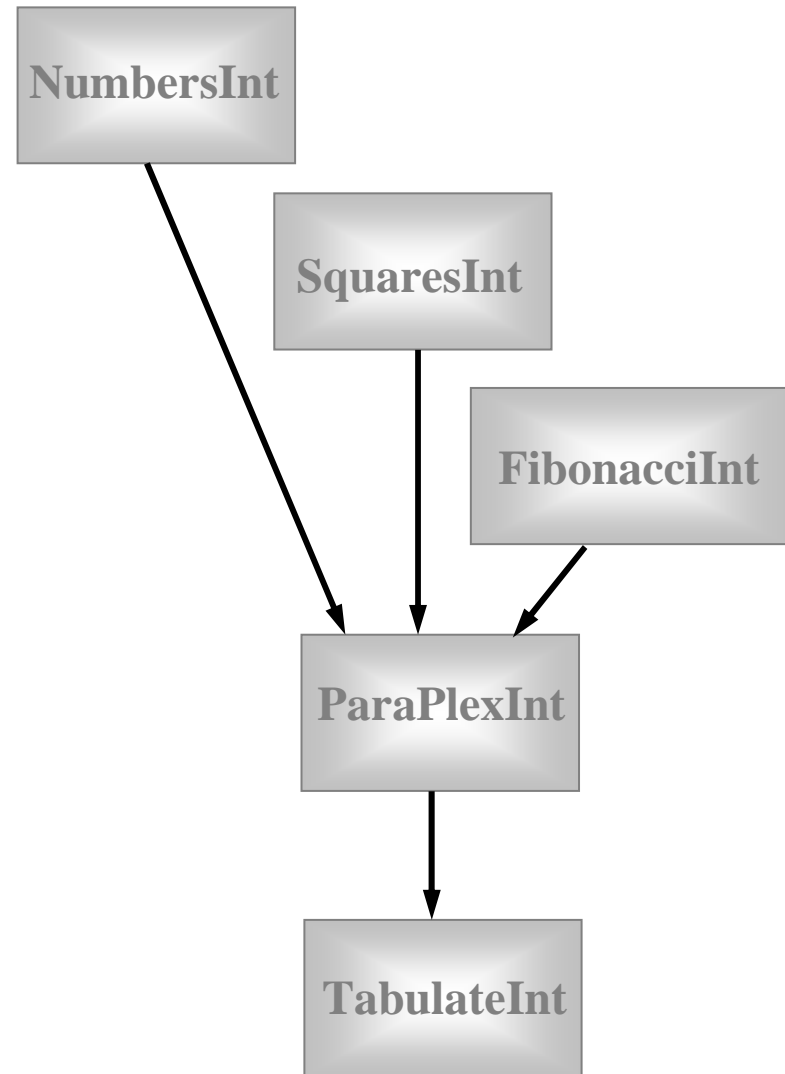
At this level, we have a network of 5 communicating processes.

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:

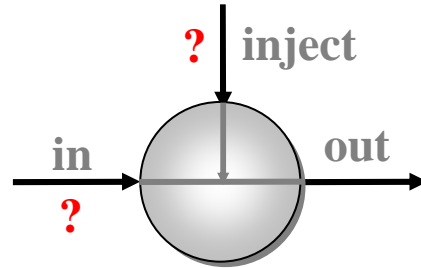
- what happened in the past;
- when (or, at least, in what order) things happened.

In this world, things are scheduling dependent.

**CSP** addresses these issues *explicitly*.

**Non-determinism does not arise by default.**

# A Control Process



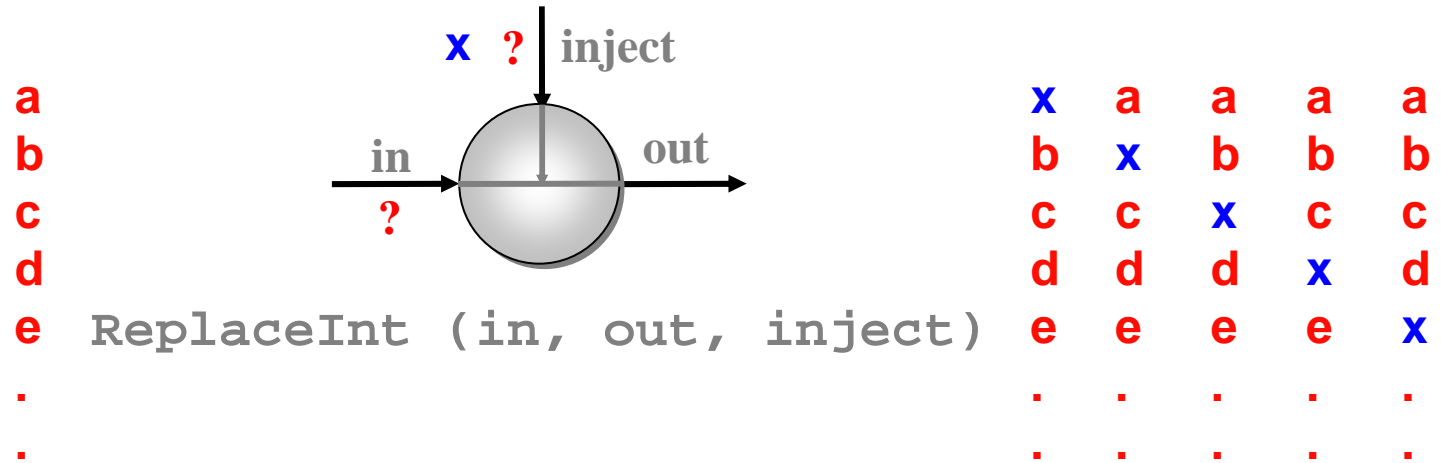
`ReplaceInt (in, out, inject)`

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

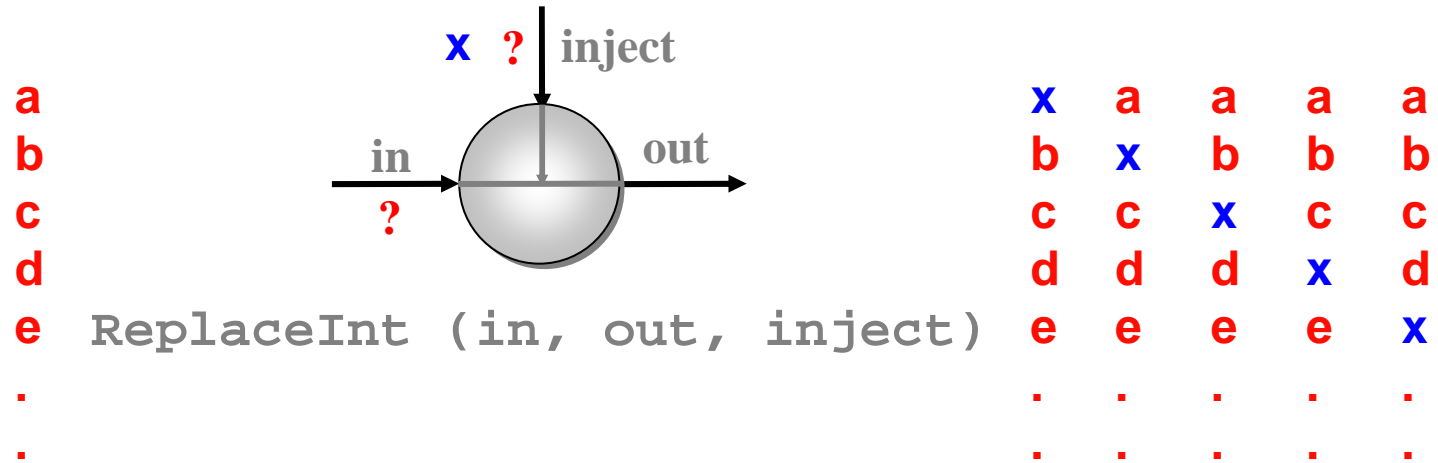


The **out** stream depends upon:

- 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



```

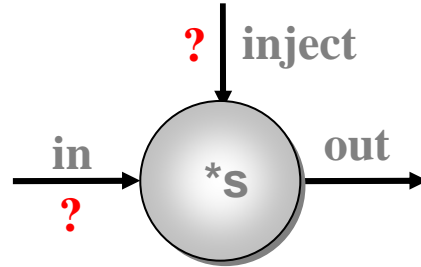
ReplaceInt (in, out, inject) =
  (inject?x --> ((in?a --> SKIP) || (out!x --> SKIP))
  [PRI]
  in?a --> out!a --> SKIP
  );
ReplaceInt (in, out, inject)

```

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

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

# Another Control Process



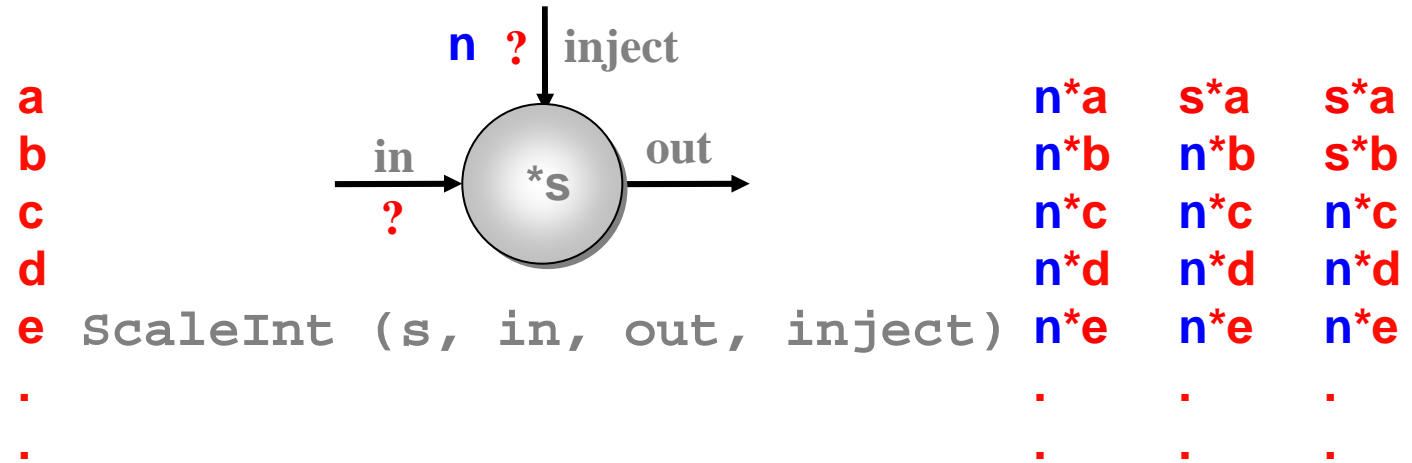
`ScaleInt (s, in, out, inject)`

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

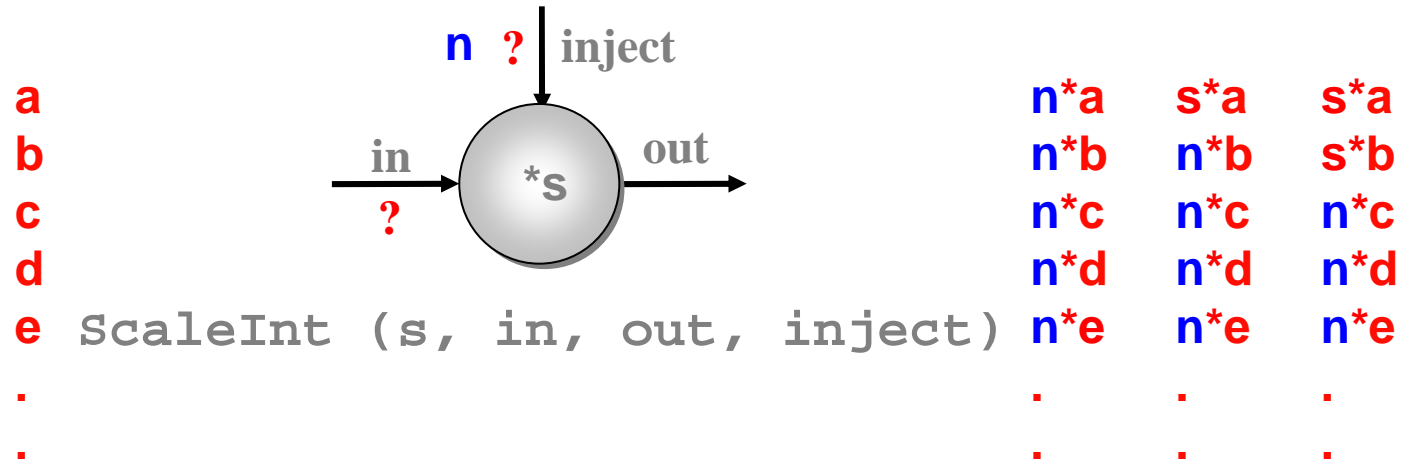


The **out** stream depends upon:

- 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

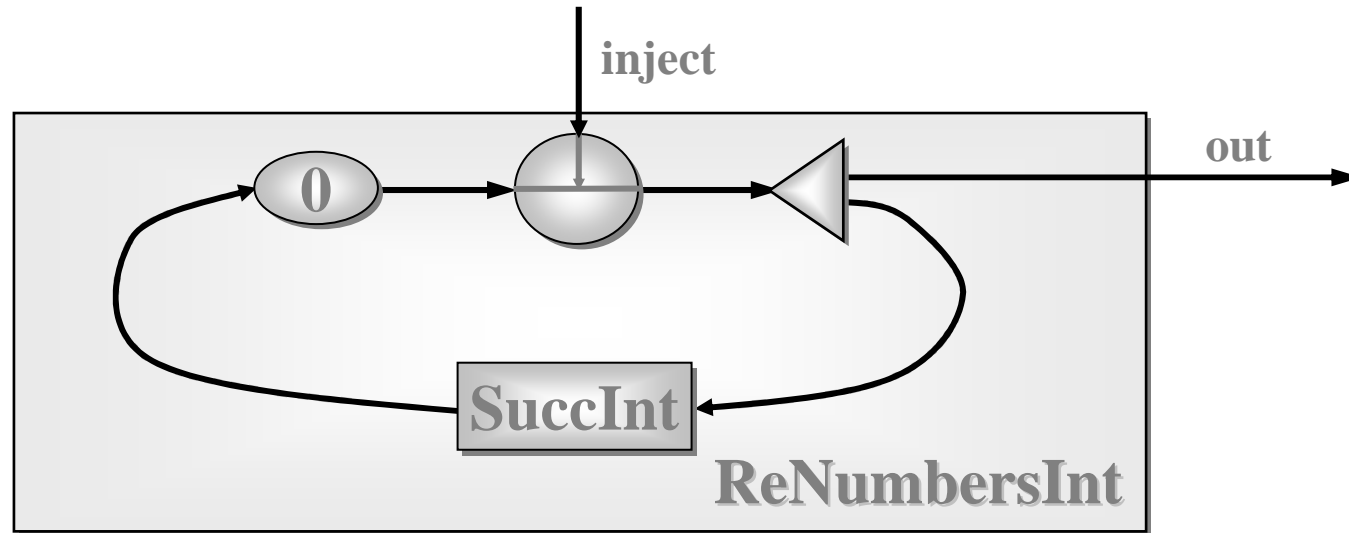


```
ScaleInt (s, in, out, inject) =
  (inject?s --> SKIP
   [PRI]
   in?a --> out!s*a --> SKIP
  );
ScaleInt (s, in, out, inject)
```

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

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

# Some Resettable Networks

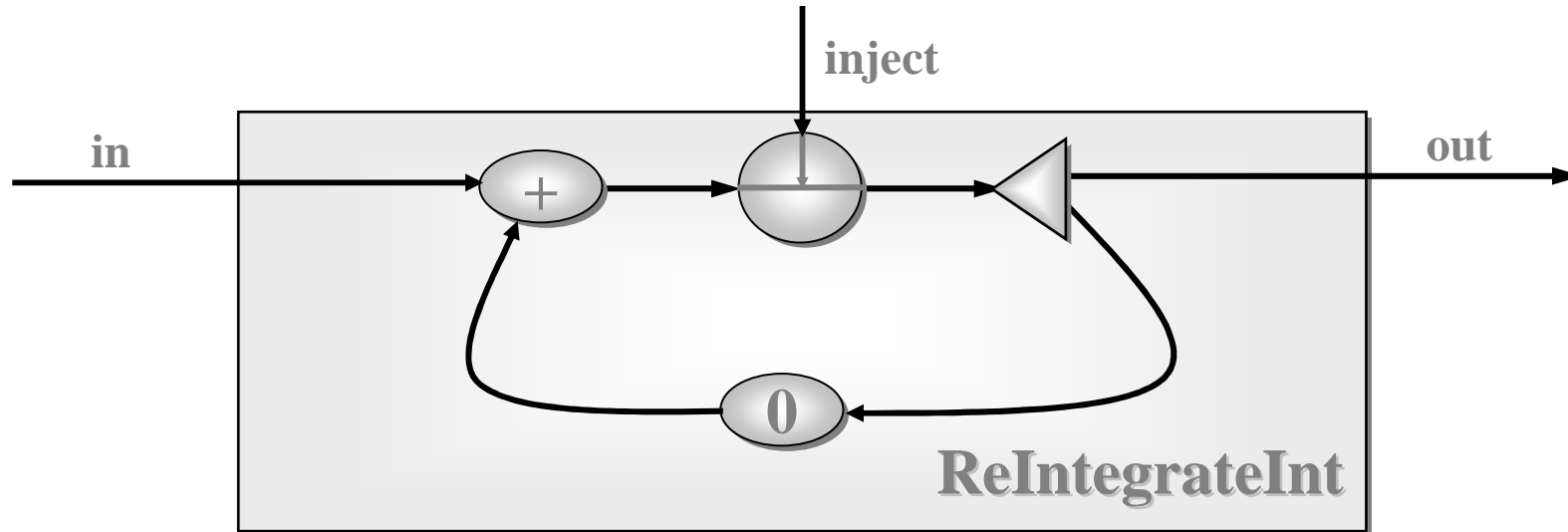


This is a *resettable* version of the **NumbersInt** process.

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

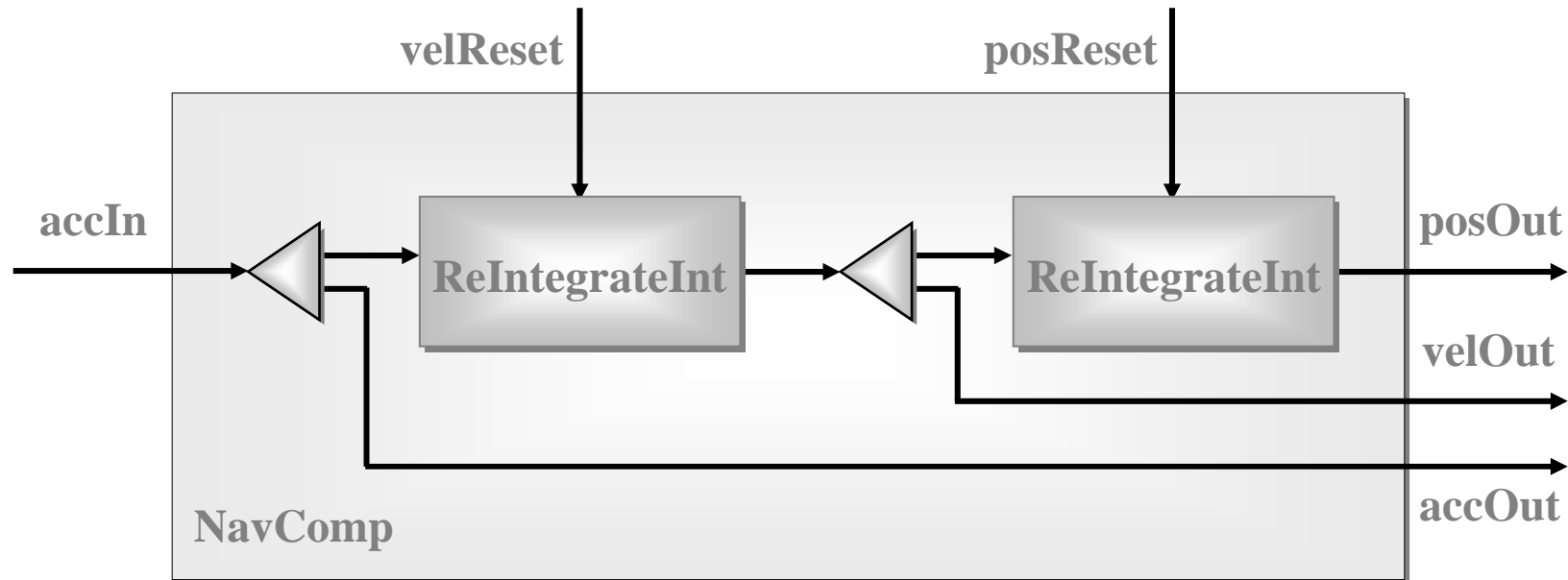


This is a *resettable* version of the **IntegrateInt** process.

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

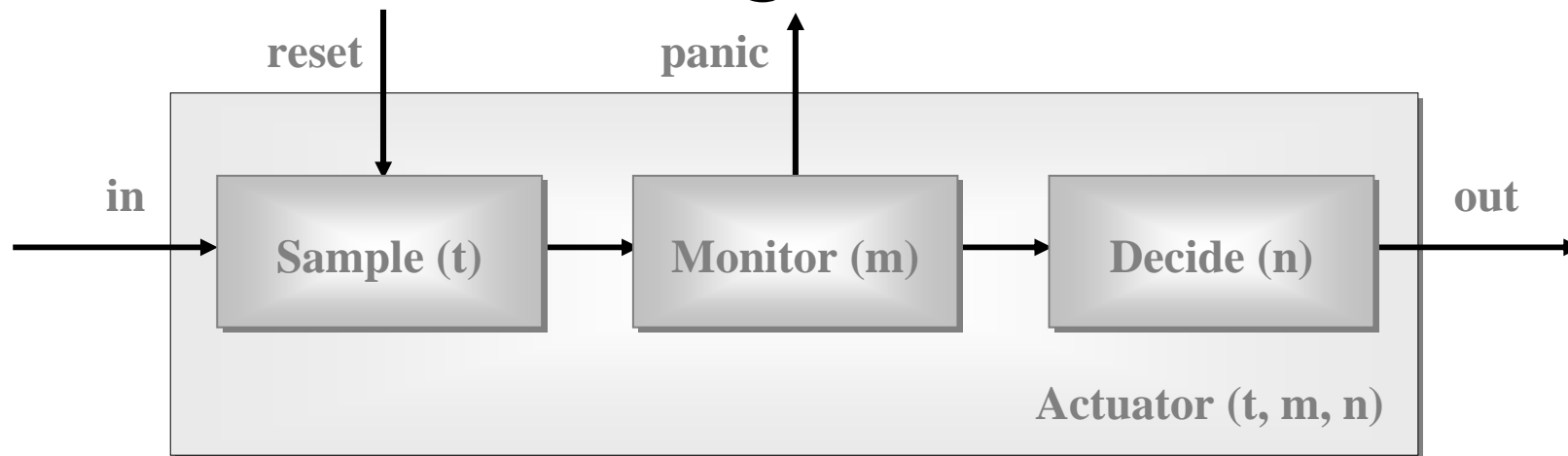
But its running sum may be reset to *any* number at *any* time.

# 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.