Software Tools for Concurrent Programming

Peter Grogono

Department of Computer Science and Software Engineering

Concordia University

10 May 2012

The Erasmus Project

Goals:

- a new programming language
- associated infrastructure:
  - compiler
  - run-time system
  - development environment
  - libraries
  - ...

Why introduce a new language?

Cons:

- there are too many languages already
- why will a new language be an improvement over its predecessors?
- *There are only two kinds of programming language: those people always complain about and those nobody uses* — Bjarne Stroustrup

Pros:

- we need a context/environment in which to explore new ideas and validate hypotheses
- programming languages grow: young and clean $\rightarrow$ old and ugly
- a few languages become popular — at least for a while

The Problem

We need to build applications that are:

- complex
- distributed
- correct
- efficient
- understandable
- adaptable
- maintainable
- ...

Nothing new here 

The Problem

Why is it so hard to do this?

- complexity
- high coupling
- implicit coupling
- Object Oriented Programming — a nice idea, but:
  - powerful mechanisms can be overused (e.g., inheritance)
  - interfaces do not tell the whole story
  - objects + threads = disaster ?

Hypothesis

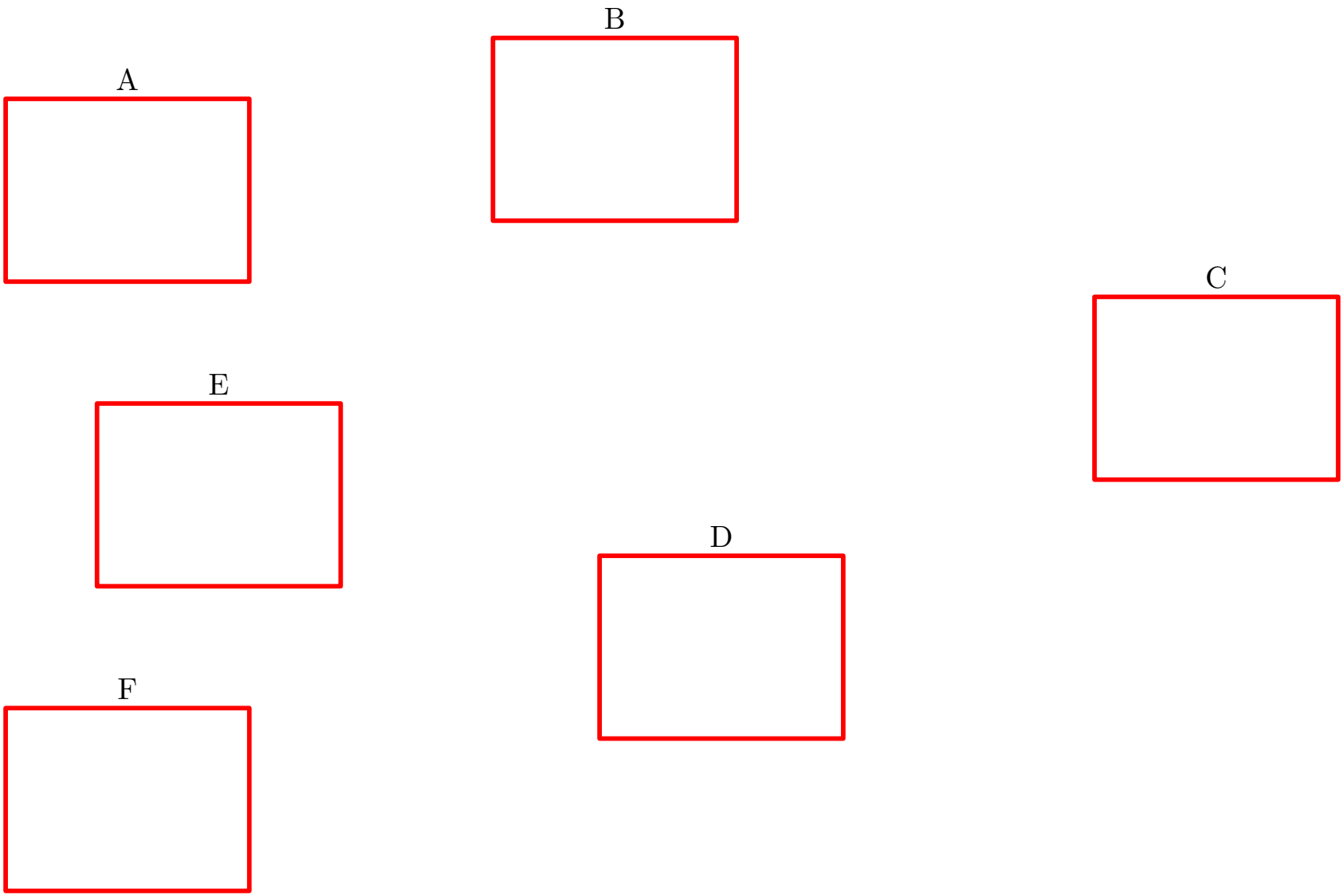
Process are a better abstraction than objects

Why?

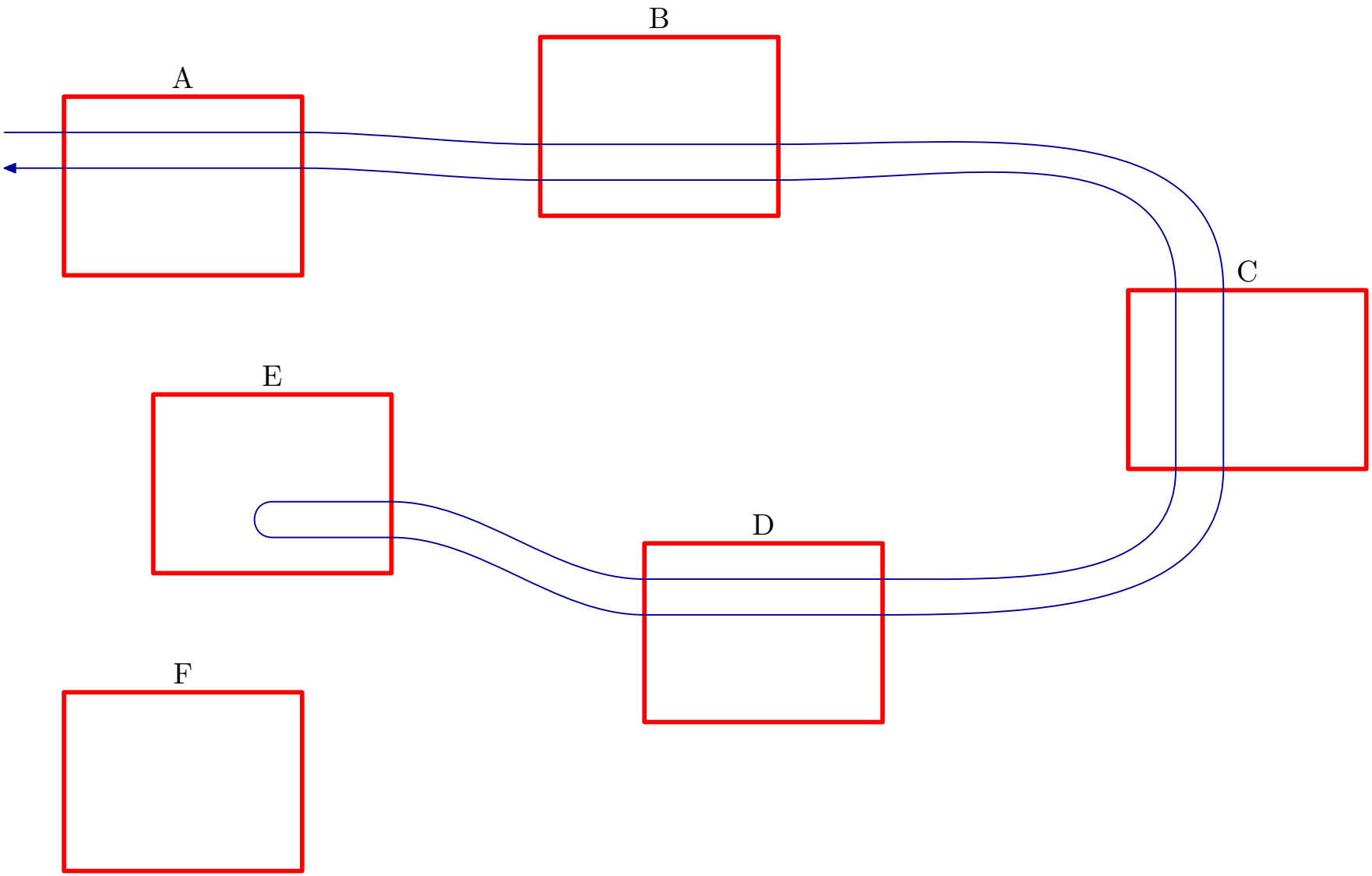
- control flow is local
- synchronization problems occur only at rendezvous
- interfaces are complete
- bonus: we can exploit multicore architecture 

The Erasmus Project is an *experiment*
designed to confirm (or refute) this hypothesis.

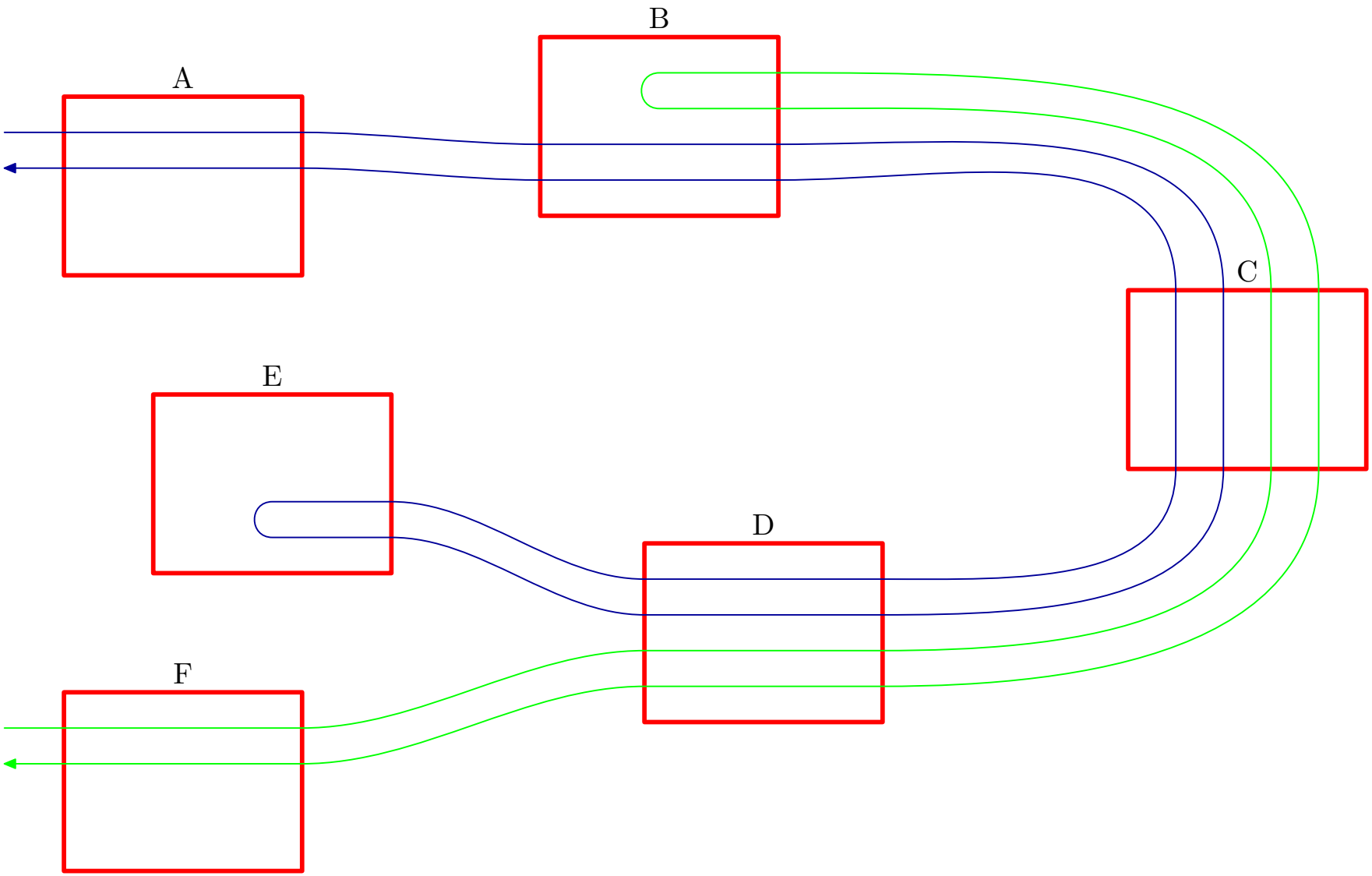
Objects



Single-threaded objects



Multi-threaded objects



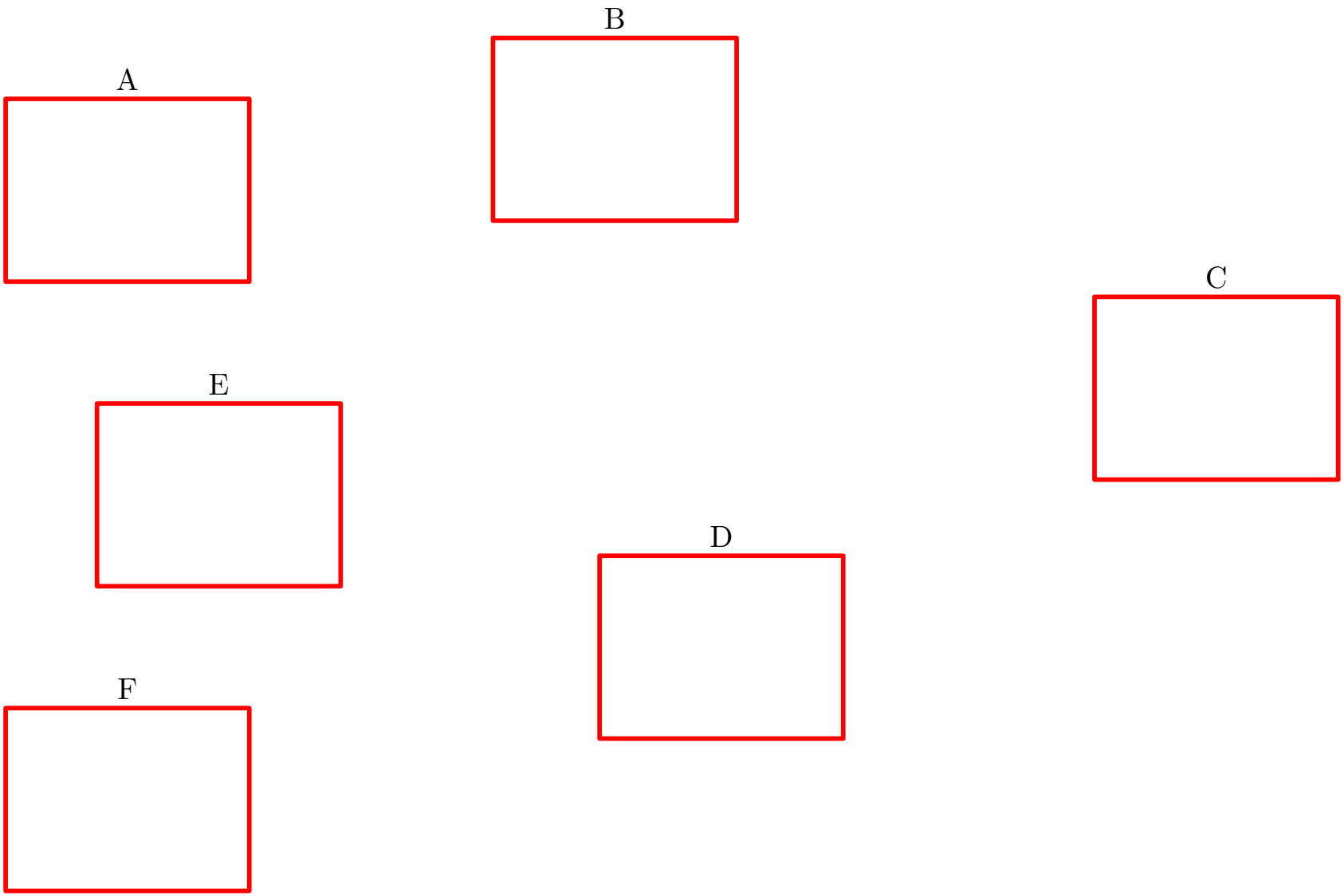
Joe Armstrong: creator of Erlang

Armstrong on Objects

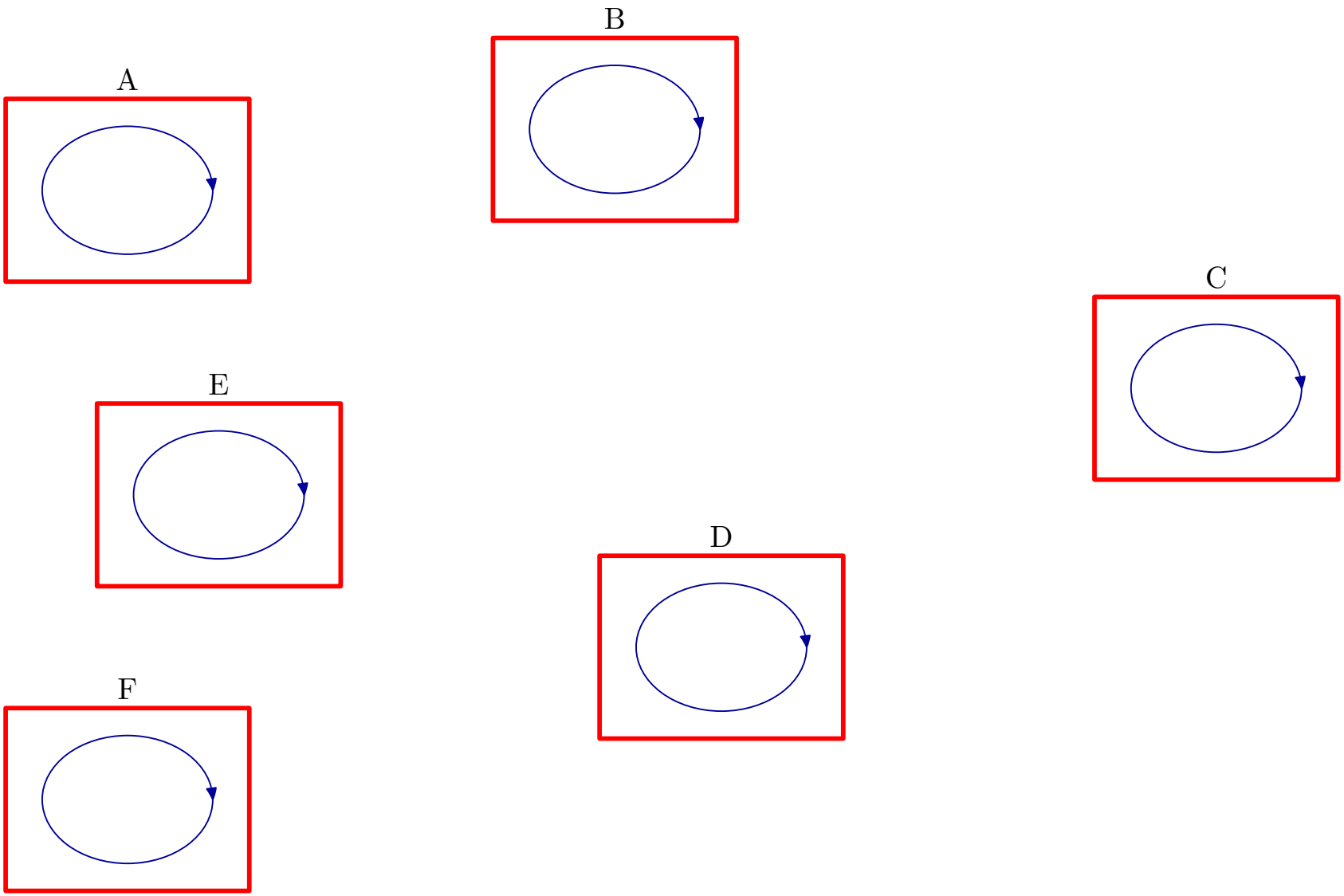
. . . the problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

Joe Armstrong, *Coders At Work*

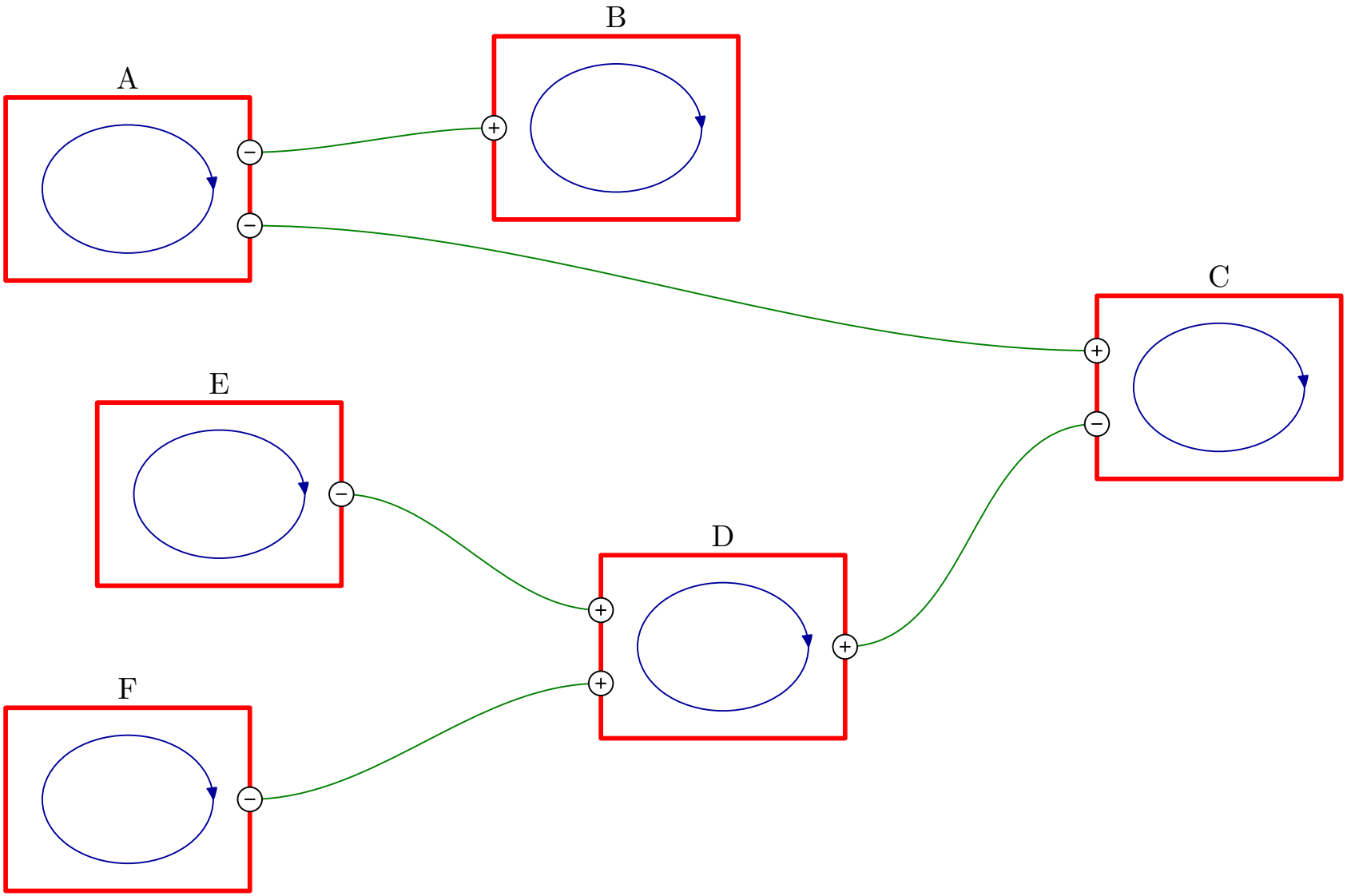
Processes



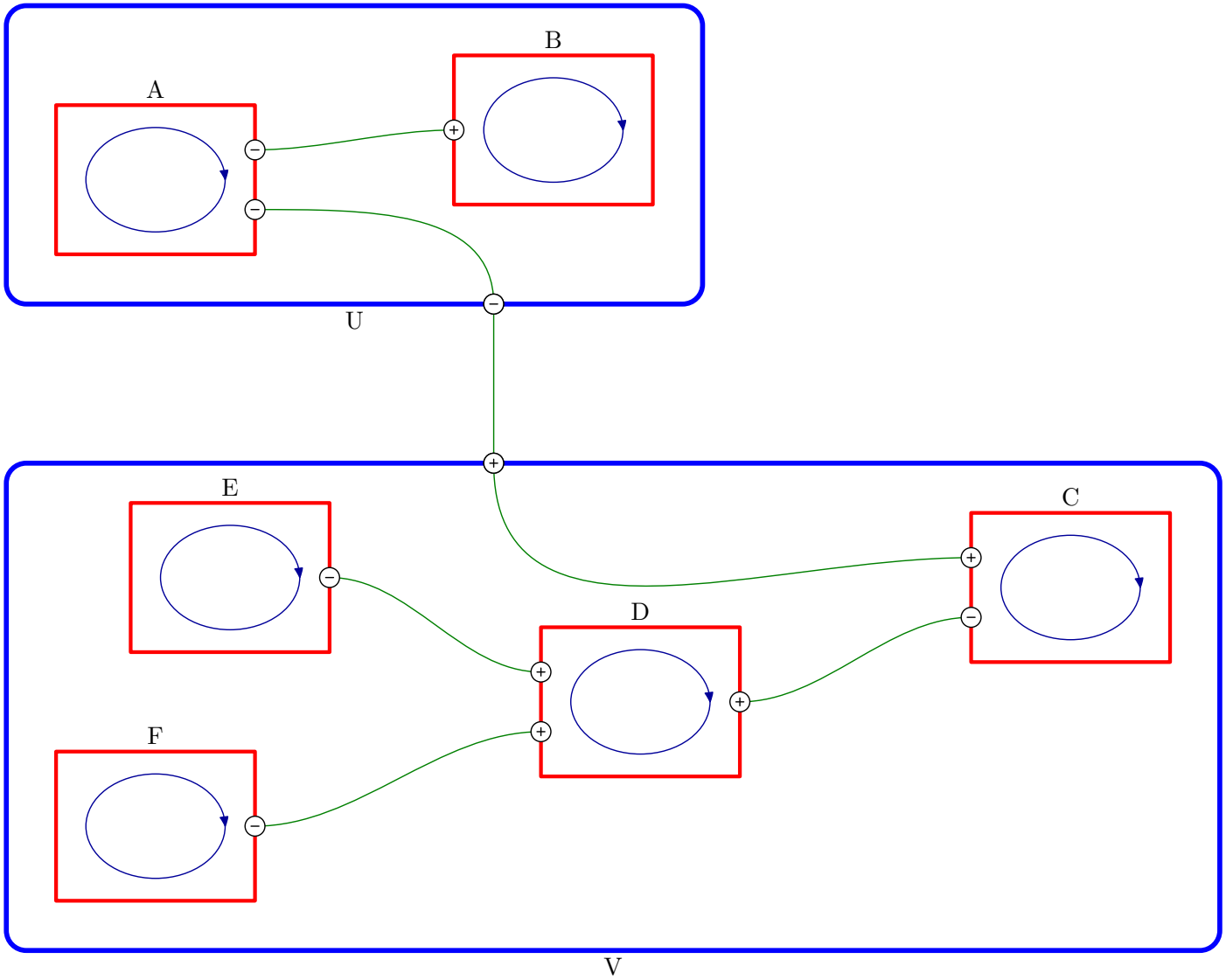
Processes with flow of control



Communicating processes



Modularization using cells



Processes

An old idea:

- Dijkstra: *Cooperating Sequential Processes* (EWD 123, 1965)
- Brinch Hansen: *The Nucleus of a Multiprogramming System*, (CACM, 1970)

  Concurrent Pascal, Solo, Edison, Joyce, ...
- Hoare: *Communicating Sequential Processes* (CACM, 1978)

  occam, Ada, Erlang, JCSP, Go, ...
- Milner: *Calculus of Communicating Systems* (Springer, 1982)
- Milner: *The Polyadic π-Calculus*, (Edinburgh, 1991)

  occam-π, Pict, JoCaml, CubeVM, ...

Andrew Binstock, Editor, *Dr. Dobbs Journal*

Processes

The real problem going forward is not program decomposition, but composition. Why are we not currently designing programs as a series of small asynchronous tasks? After all, we have already crossed into a world in which we break programs into objects. Why not then into tasks? Properly done, this would move today's OOP more closely to its original intent, which was to focus on the messages passed between objects, rather than the objects themselves (according to the widely quoted observation from Alan Kay, who coined the term "object orientation").

The problems facing such an approach rest on its profound unfamiliarity. There are few languages that provide all the needs of this model, few frameworks that facilitate its design, and few developers conversant with the problems and limitations of this approach. . . . In the meantime, it's worth considering how an existing program broken down into smaller tasks might function. What exactly would it look like?

Andrew Binstock (Editorial, Dr Dobbs, 12/03/2012)

Processes

The big idea is "messaging" — that is what the kernel of Smalltalk/Squeak is all about (and it's something that was never quite completed in our Xerox PARC phase). The Japanese have a small word — ma *— for "that which is in between" — perhaps the nearest English equivalent is "interstitial".*
The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

Alan Kay, Squeak mailing list, 1998.

Where are we now?

What we have:

- Desi: a simple PL; probably a subset of Erasmus
- UDC: a compiler that generates Desi Intermediate Language (DIL)
- JIT: a just-in-time compiler that generates machine code
- A very primitive IDE

Where are we now?

What we need:

- Libraries
- Testing tools — difficult for concurrent programs
- Debugger — but debuggers are not used much
- Methods and tools for analyzing programs

Language Features

- **Cells** provide abstraction, organization, and encapsulation
- **Process** perform independent tasks and pass messages to one another
- **Protocols** describe the contents and ordering of messages
- **Routines** perform small tasks without communication

A Differential Equation

- $A\frac{d^2x}{dt^2} + B\frac{dx}{dt} + Cx = 0$

A Differential Equation

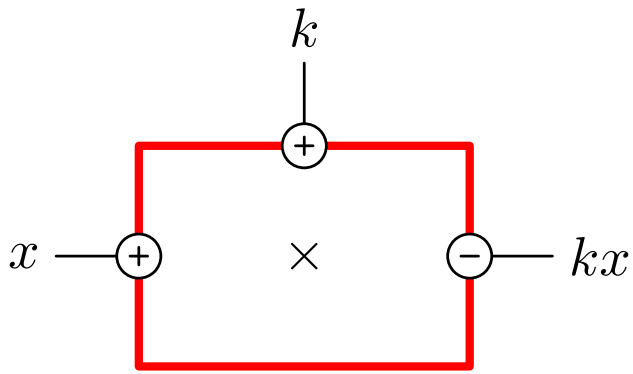
- $A\frac{d^2x}{dt^2} + B\frac{dx}{dt} + Cx = 0$
- $\ddot{x} = -(B/A)\dot{x} - (C/A)x$

Protocol

$$Nums = \textbf{protocol}\ \{\ *val\text{: } Real\ \}$$

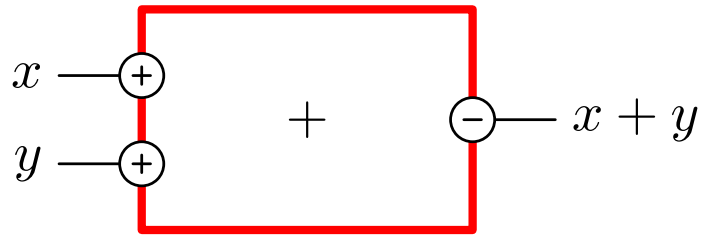
Constant Multiplier

```
mulcon = process kin: +Nums; x: +Nums; kx: -Nums {
  k: Real := kin.val;
  loop {
     kx.val := k * x.val
   }
}
```



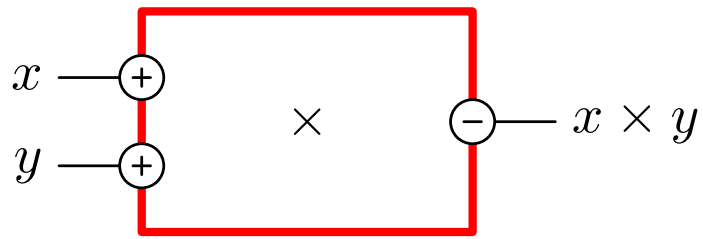
Adder

```
add = process x: +Nums; y: +Nums; sum: -Nums {
  loop select {
    ||  t: Real := x.val; sum.val := y.val + t
    ||  t: Real := y.val; sum.val := x.val + t
  }
}
```



Multiplier

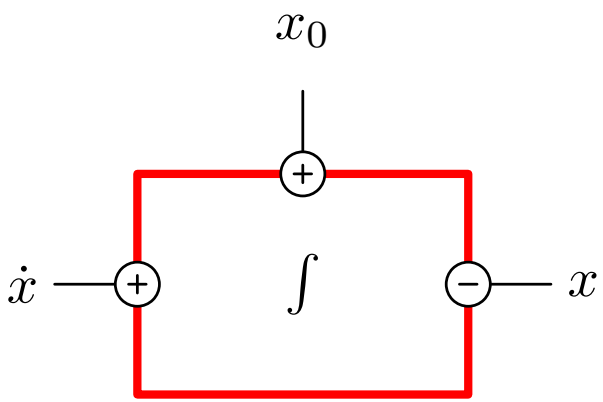
```
mul = process x: +Nums; y: +Nums; prod: -Nums {
  loop select {
    ||  t: Real := x.val; prod.val := y.val * t
    ||  t: Real := y.val; prod.val := x.val * t
  }
}
```



Integrator

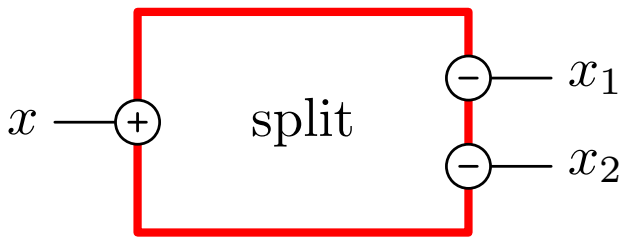
```
DT = constant 0.001;

integrate = process x0: +Nums; xdot: +Nums; x: -Nums {
  x: Real := x0.val;
  loop {
    x += xdot.val * DT;
    x.val := x
  }
}
```

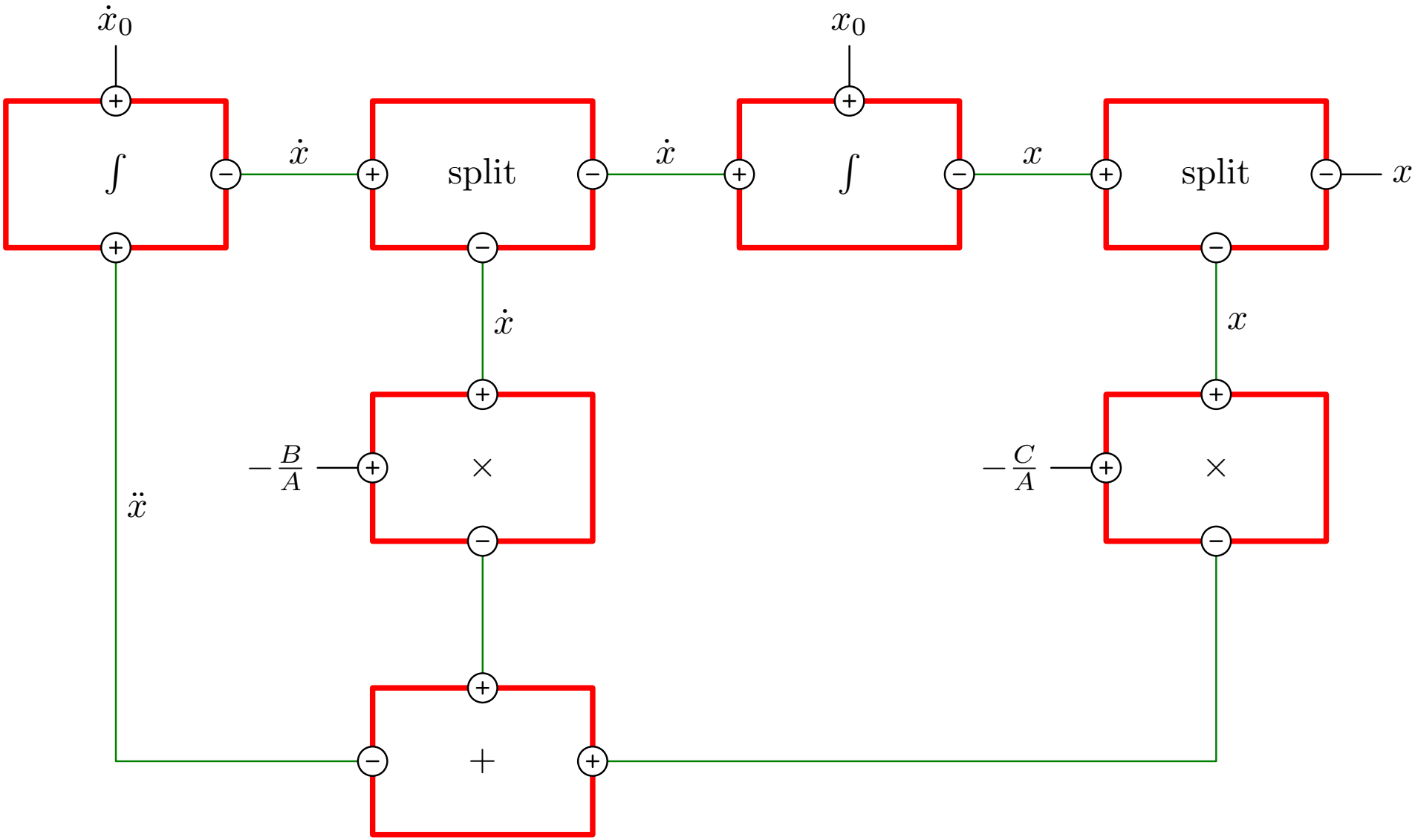


Splitter

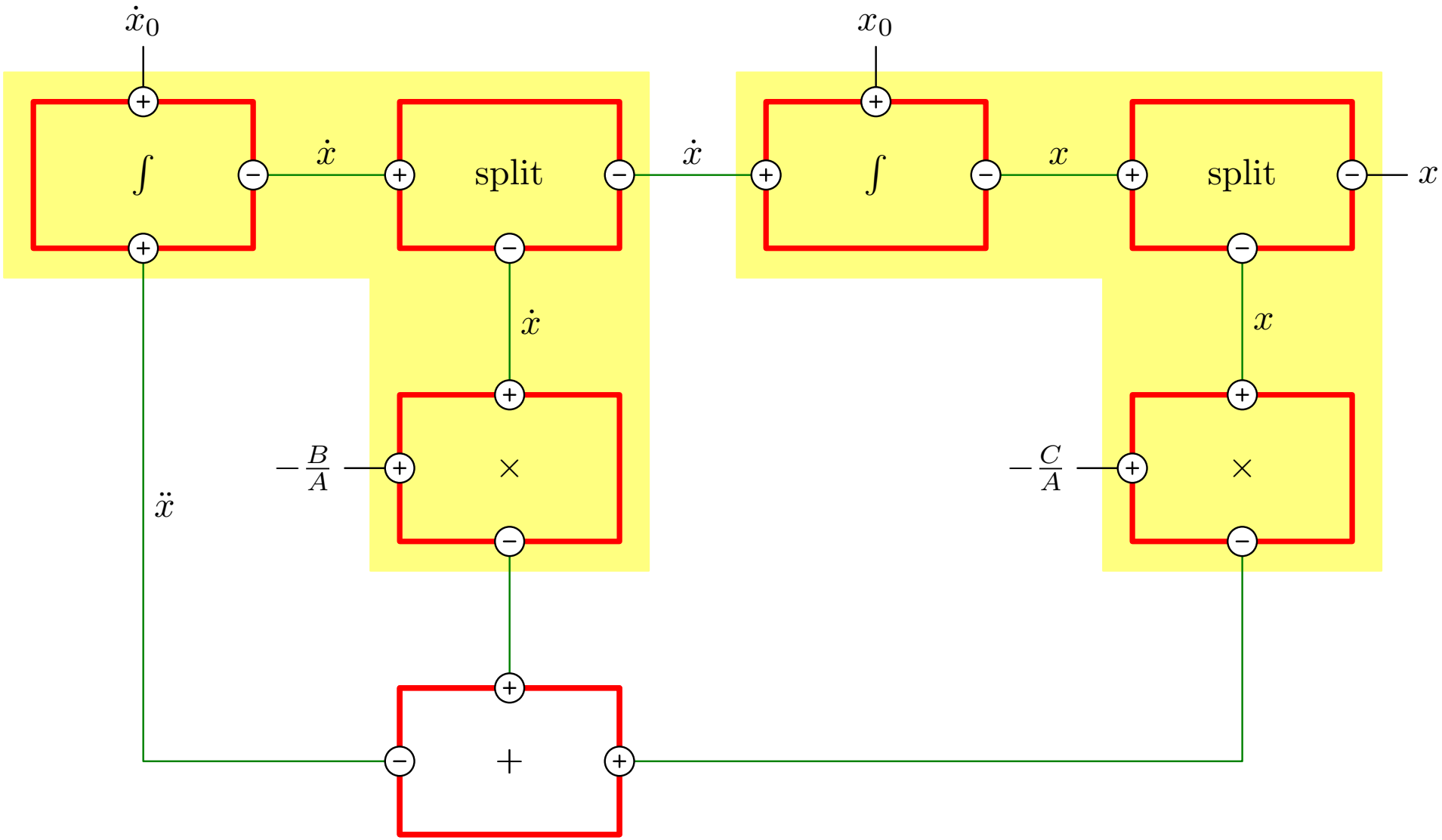
```
split = process x: +Nums; x1: -Nums; x2: -Nums {
    loop {
        x: Real := x.val;
        select {
        ||  x1.val := x; x2.val := x
        ||  x2.val := x; x1.val := x
        }
    }
}
```



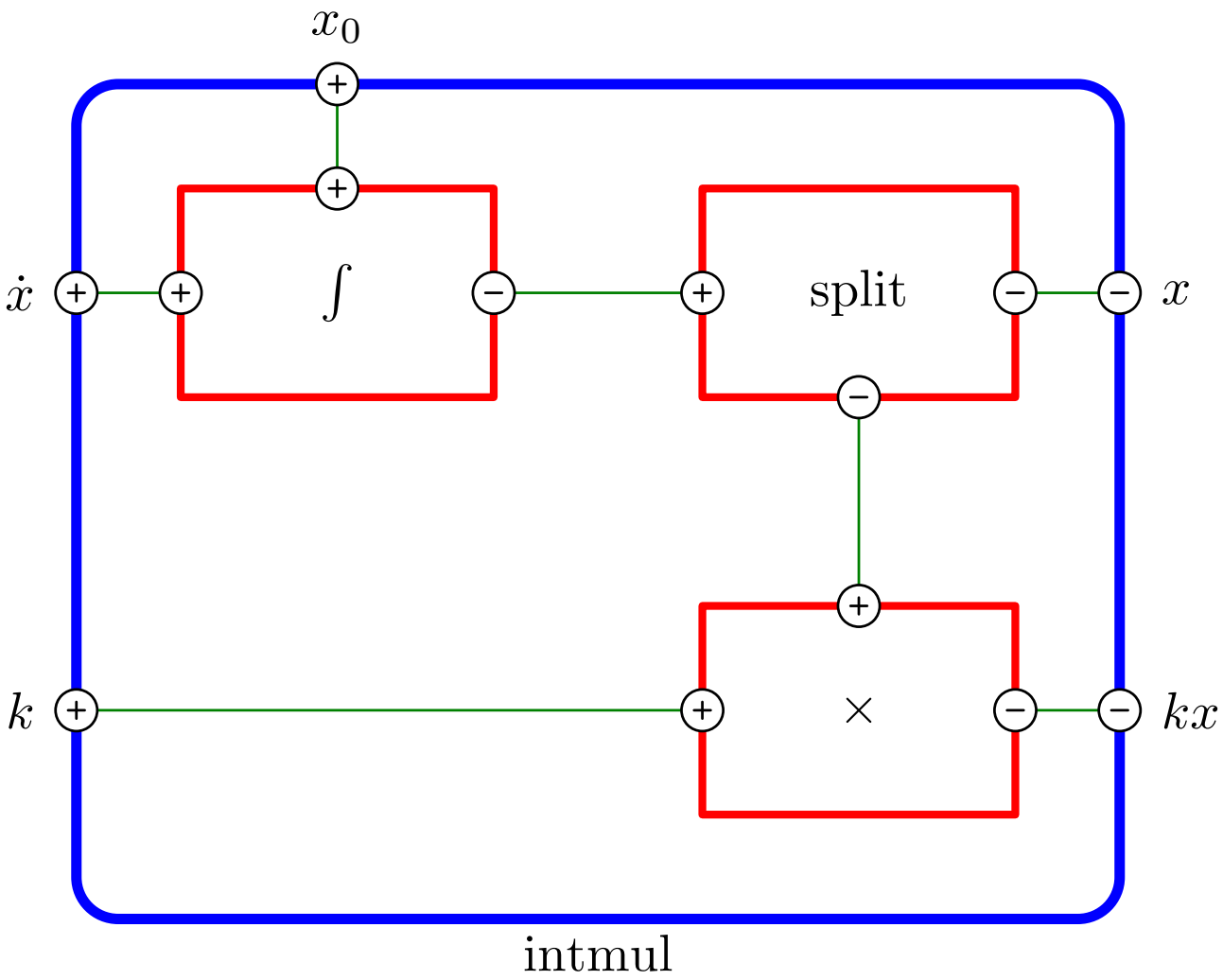
$\ddot{x} = -(B/A)\dot{x} - (C/A)x$



$\ddot{x} = -(B/A)\dot{x} - (C/A)x$



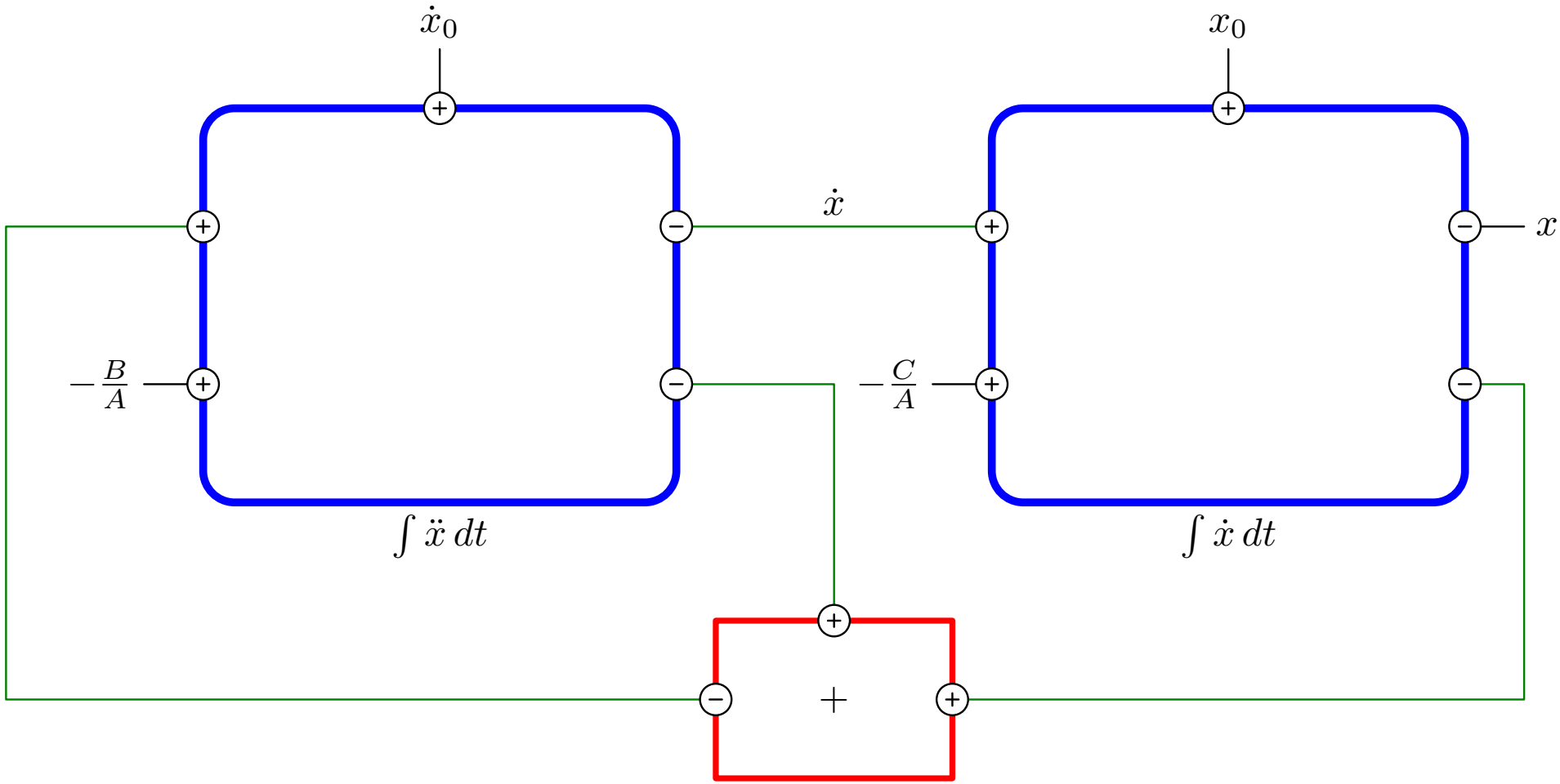
A cell for integration



A cell for integration

```
intmul = cell
    con1: +Nums; con2: +Nums; inp: +Nums;
    out1: -Nums; out2: -Nums
{
    c1, c2: Nums;
    integrate(con1, inp, c1);
    split(c1, c2, out1);
    mulcon(con2, c2, out2);
}
```

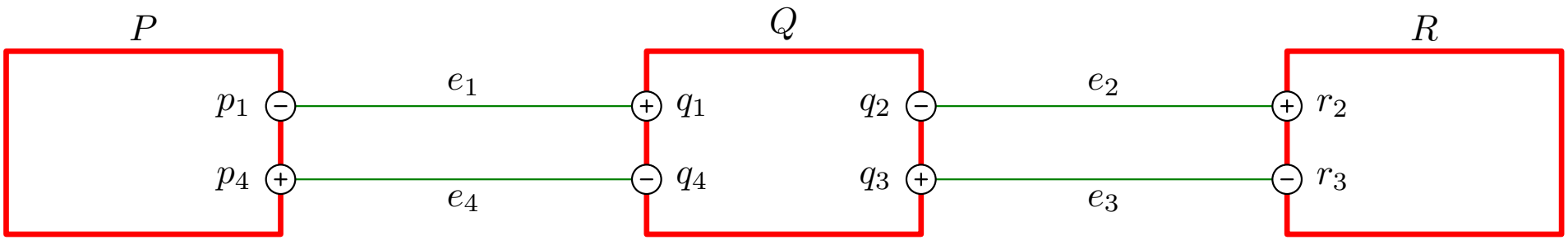
$\ddot{x} = -(B/A)\dot{x} - (C/A)x$



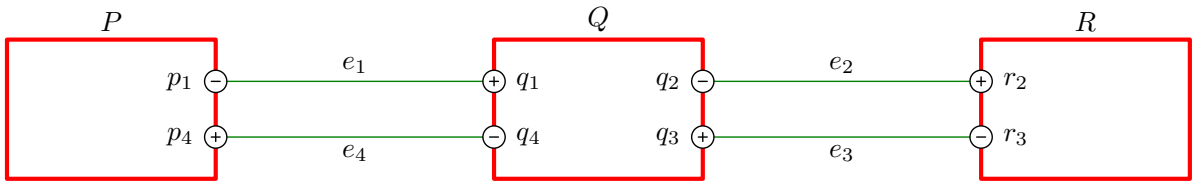
Static Analysis

- *Static analysis* derives properties of programs from their source code
- E.g., type checking
- Other static techniques:
  - process algebra
  - abstract interpretation
- We need static analysis because testing is inadequate for large, concurrent, distributed systems

A chain of responsibility



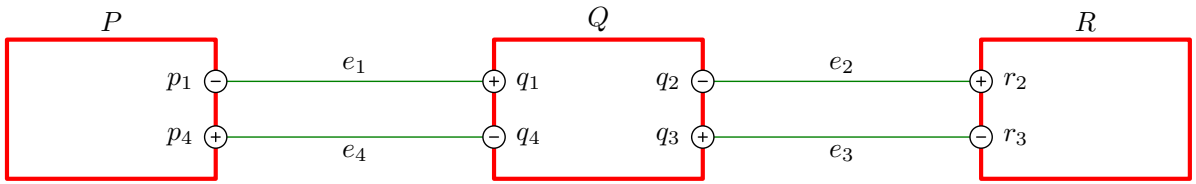
The requesting process



$P =$ **process** *-p1, +p4* {
  loop {
    (0) p1.snd;
    (1) p4.rcv
  }
}

$$P_0 \xrightarrow{e_1} P_1$$
$$P_1 \xrightarrow{e_4} P_0$$

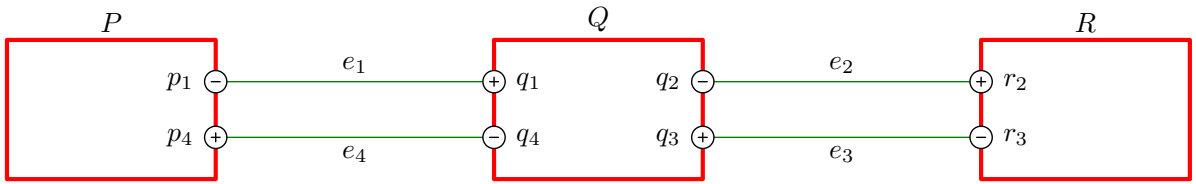
The last responding process



$R =$ **process** *+r2, -r3* {
  loop {
    (0) r2.rcv;
    (1) r3.snd
  }
}

$$R_0 \xrightarrow{e_2} R_1$$
$$R_1 \xrightarrow{e_3} R_0$$

The intermediate responding process



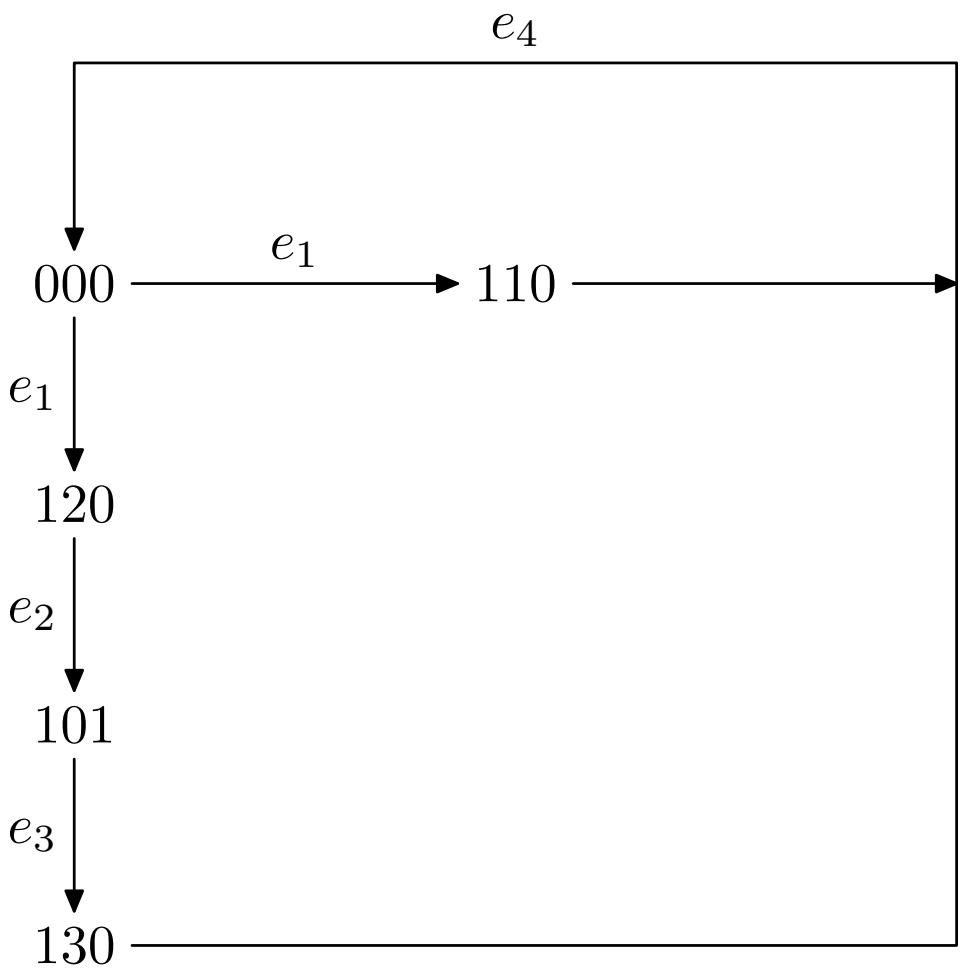
$Q =$ **process** *+q1, -q2, +q3, -q4* {
  loop select { *(0)*
    || *q1.rcv;*
      if *canAnswer*
        then *(1) q4.snd*
        else *(2) q2.snd*
    || *q3.rcv; (3) q4.snd*
  }
}

$$Q_0 \xrightarrow{e_1} Q_1$$
$$Q_0 \xrightarrow{e_1} Q_2$$
$$Q_0 \xrightarrow{e_3} Q_3$$
$$Q_1 \xrightarrow{e_4} Q_0$$
$$Q_2 \xrightarrow{e_2} Q_0$$
$$Q_3 \xrightarrow{e_4} Q_0$$

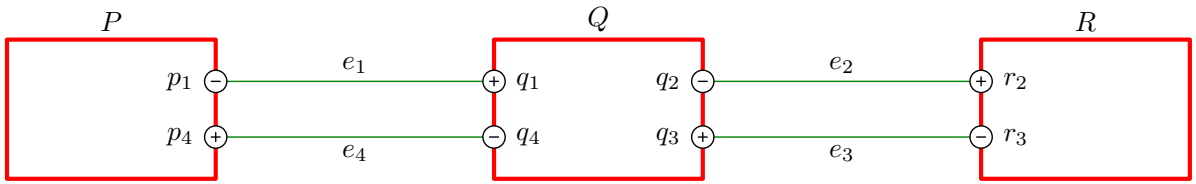
Process transitions

P	Q	R
$P_0 \xrightarrow{e_1} P_1$	$Q_0 \xrightarrow{e_1} Q_1$	$R_0 \xrightarrow{e_2} R_1$
$P_1 \xrightarrow{e_4} P_0$	$Q_0 \xrightarrow{e_1} Q_2$	$R_1 \xrightarrow{e_3} R_0$
	$Q_0 \xrightarrow{e_3} Q_3$	
	$Q_1 \xrightarrow{e_4} Q_0$	
	$Q_2 \xrightarrow{e_2} Q_0$	
	$Q_3 \xrightarrow{e_4} Q_0$	

System transitions



The requesting process



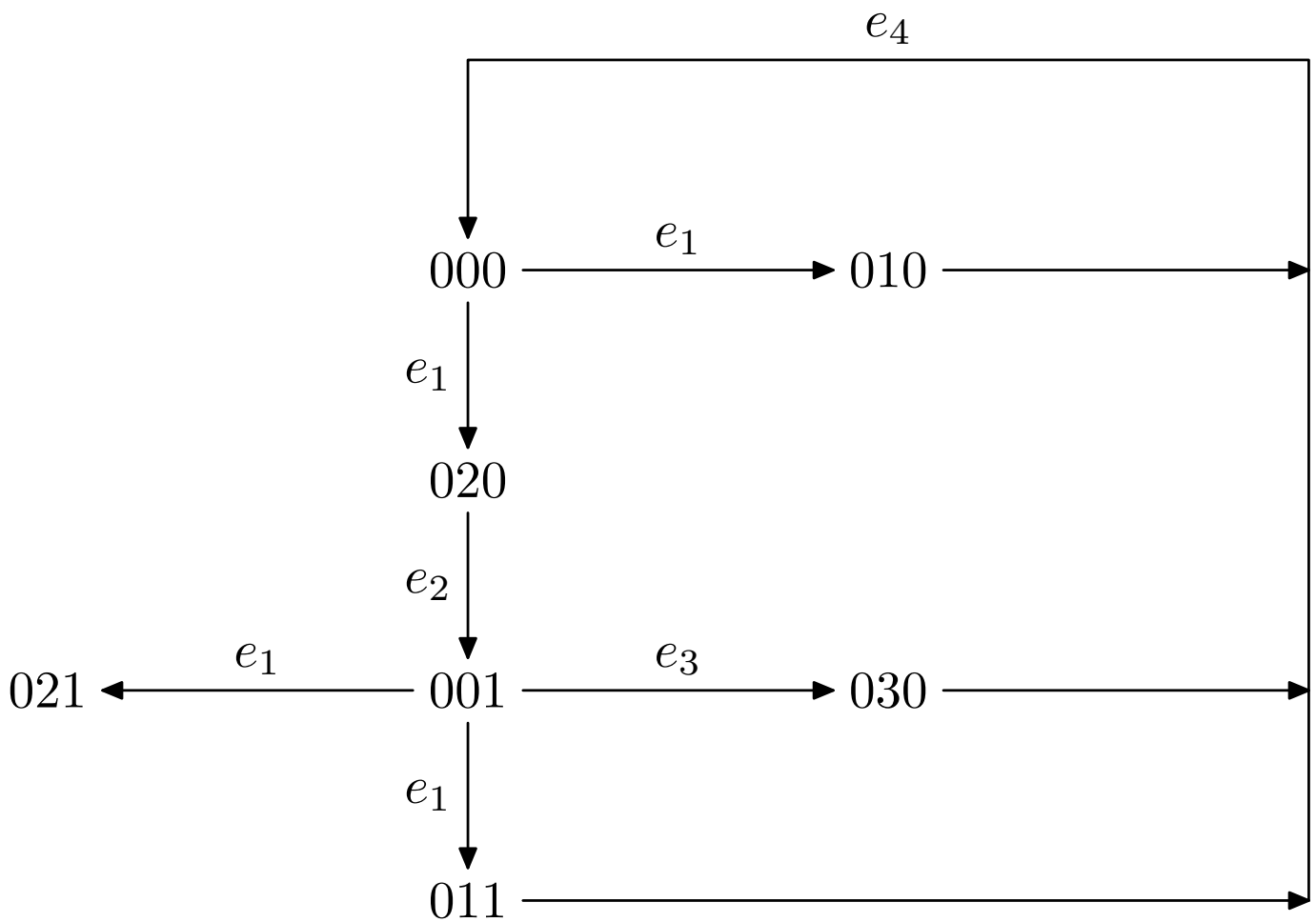
$P =$ **process** *-p1, +p4* {
  loop select { *(0)*
    || *p1.snd*
    || *p4.rcv*
  }
}

$$P_0 \xrightarrow{e_1} P_0$$
$$P_0 \xrightarrow{e_4} P_0$$

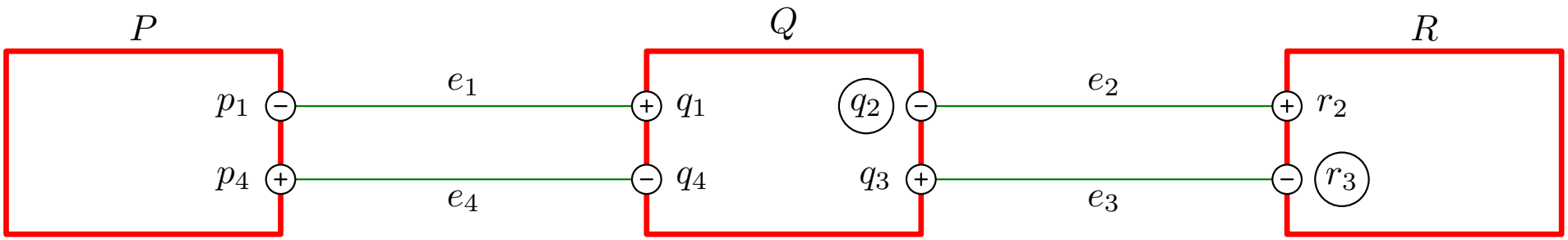
Process transitions

P	Q	R
$P_0 \xrightarrow{e_1} P_0$	$Q_0 \xrightarrow{e_1} Q_1$	$R_0 \xrightarrow{e_2} R_1$
$P_0 \xrightarrow{e_4} P_0$	$Q_0 \xrightarrow{e_1} Q_2$	$R_1 \xrightarrow{e_3} R_0$
	$Q_0 \xrightarrow{e_3} Q_3$	
	$Q_1 \xrightarrow{e_4} Q_0$	
	$Q_2 \xrightarrow{e_2} Q_0$	
	$Q_3 \xrightarrow{e_4} Q_0$	

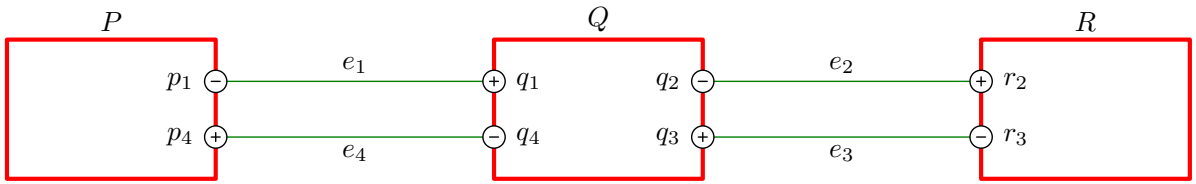
System transitions



A chain of responsibility



The last responding process



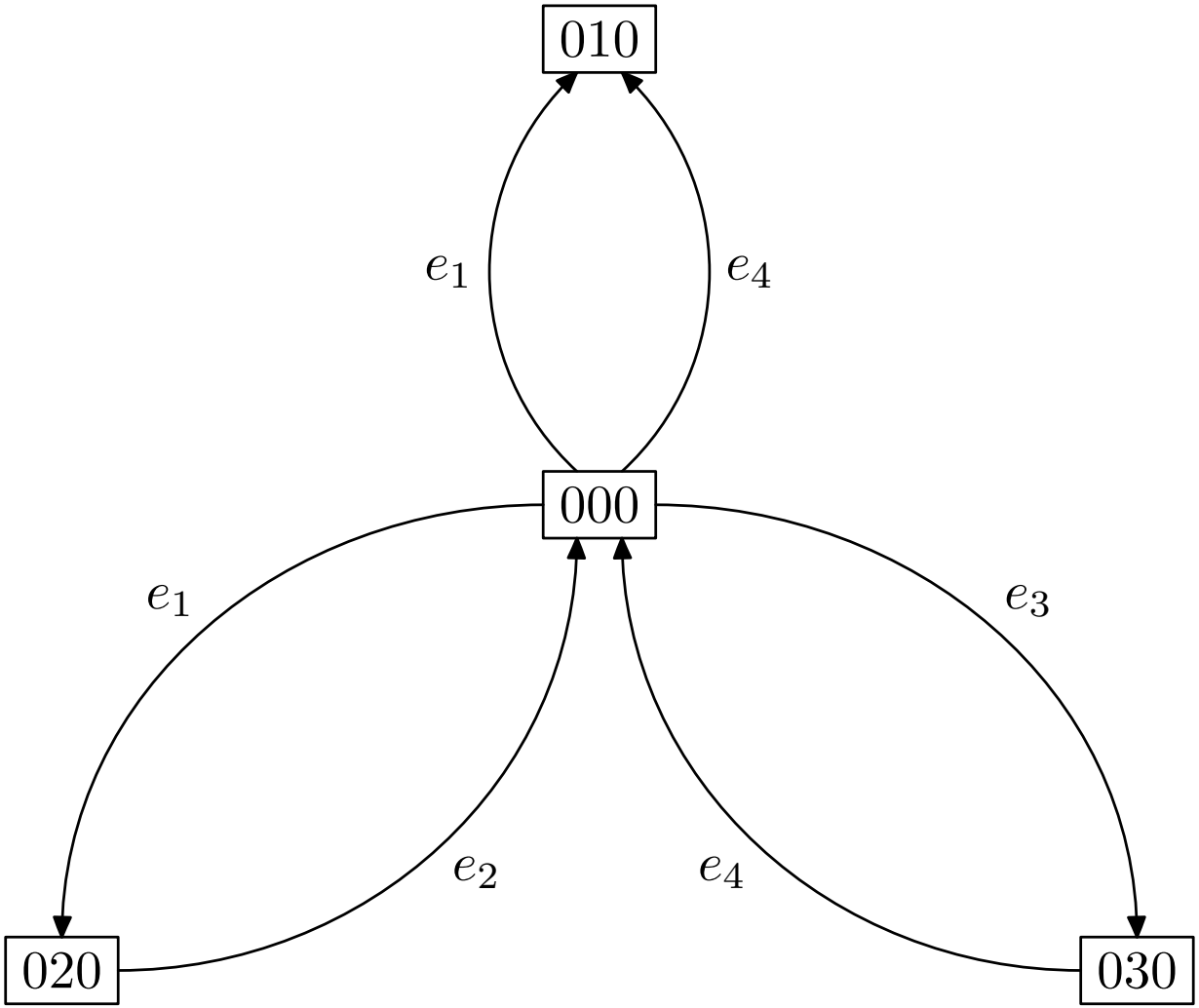
$$
\begin{aligned}
&R = \textbf{process } \textit{+r2, -r3} \{ \\
&\quad \textbf{loop select } \{ \ \textit{(0)} \\
&\qquad \| \quad \textit{r2.rcv} \\
&\qquad \| \quad \textit{r3.snd} \\
&\quad \} \\
&\}
\end{aligned}
$$

$$R_0 \xrightarrow{e_2} R_1$$
$$R_1 \xrightarrow{e_3} R_0$$

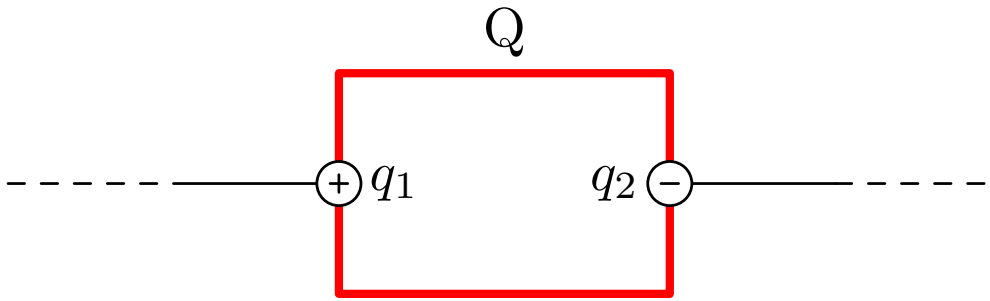
Process transitions

P	Q	R
$P_0 \xrightarrow{e_1} P_0$	$Q_0 \xrightarrow{e_1} Q_1$	$R_0 \xrightarrow{e_2} R_0$
$P_0 \xrightarrow{e_4} P_0$	$Q_0 \xrightarrow{e_1} Q_2$	$R_0 \xrightarrow{e_3} R_0$
	$Q_0 \xrightarrow{e_3} Q_3$	
	$Q_1 \xrightarrow{e_4} Q_0$	
	$Q_2 \xrightarrow{e_2} Q_0$	
	$Q_3 \xrightarrow{e_4} Q_0$	

System transitions



A better pattern



$X =$ **protocol** { *q: Text* | *r: Float* }

$Q =$ **process** *+q1: X, -q2: X* {
  loop select {
    || *q1.q;* **if** *canAnswer* **then** *q2.r* **else** *q2.q*
    || *q1.r; q2.r*
  }
}

Model Checking

This approach is *model checking* — or close to it.

- Well-understood method
- Known to suffer from *state-explosion problem*
- Various techniques to avoid state explosion are also known

Model Checking

Ways of avoiding the state explosion

- *Abstraction* reduces the state space
- Erasmus *cells* allow modular analysis
- *Abstract Interpretation* avoids state space search

Abstract Interpretation

- Idea: approximate semantics using monotonic functions over lattices
- Accumulate properties by simulating execution
- Pioneers: Patrick and Radhia Cousot
- Examples: type checking, interval analysis, ...

Patrick Cousot

Abstract Interpretation of CSP

- Cousot applied his techniques to CSP:
  Semantic Analysis of Communicating Sequential Processes
  Patrick Cousot and Radhia Cousot
  Automata, Languages and Programming
  Seventh Colloquium, Noordwijkerhout, the Netherlands
  14–18 July 1980, pages 119–133.
- Difficult to read and understand
- Limited results (e.g., only two processes)
- Generally discouraging
- Not much work has been done since 1980

Cousot's semantics for CSP

- $\iota \in [[S \times S] \to B]$

  $\iota = \lambda((xa,ca),(xb,cb)).[\forall i \in [1,\underline{\pi}],\ (ca(i)=cb(i)=\underline{\lambda}(i,1) \wedge xa(i)=xb(i)) \vee (ca(i)=\underline{\lambda}(i,1) \wedge \tau[i]^*[(xa(i),ca(i)),(xb(i),cb(i))] \wedge cb(i) \in \underline{\lambda}(i,\underline{C\ell}(i)))]$

  (If E, E1, E2 are sets, $f \in [E1 \to E2]$ and $E \subseteq E1$ then f(E) is defined as $\{f(x): x \in (dom(f) \cap E)\}$. The transition relation ι defines the "ready to communicate" or "stop" states which are possible successors of the "entry" states. As far as cooperation between processes is concerned, a process which is never willing to communicate and never terminates does not progress).

- $\underline{Ch} = \{\langle i,j,k \to \ell,m,n\rangle : i,\ell \in [1,\underline{\pi}] \wedge j \in \underline{C}(i) \wedge k \in \underline{O}(i,j) \wedge m \in \underline{C}(\ell) \wedge n \in \underline{I}(\ell,m) \wedge i=\underline{\theta}(\ell,m,n) \wedge \ell=\underline{\theta}(i,j,k) \wedge \{\underline{e}(i,j,k)(x): x \in dom(\underline{e}(i,j,k))\} \cap \underline{t}(\ell)(\underline{\alpha}(\ell,m,n)) \neq \emptyset\}$.

  (The set $\underline{Ch}$ of communication channels is isomorphic with the set of statically matching pairs of input-output guards).

- $\mu \in [[S \times S] \to B]$

  $\mu = \lambda((xa,ca),(xb,cb)).[\exists \langle i,j,k \to \ell,m,n\rangle \in \underline{Ch}$:
  $[\forall q \in ([1,\underline{\pi}]-\{i,\ell\}),\ (ca(q)=cb(q))] \wedge (xa(q)=xb(q))]$
  $\wedge [Rsao(i,j,k)((xa(i),ca(i)),(xb(i),cb(i))) \wedge cb(i) \in \underline{\lambda}(i,\underline{C\ell}(i))]$
  $\wedge [\underline{e}(i,j,k)(xa(i)) \in \underline{t}(\ell)(\underline{\alpha}(\ell,m,n))]$
  $\wedge [\overline{Rsai}(\ell,m,n)((xa(\ell),ca(\ell)),\underline{e}(i,j,k)(xa(i)),(xb(\ell),cb(\ell))) \wedge cb(\ell) \in \underline{\lambda}(\ell,\underline{C\ell}(\ell))]]$

  (The transition relation μ defines the "ready to communicate" or "stop" states which are the possible successors of "ready to communicate" states. The dynamic discrimination of input messages is modeled by dynamic type checking. When several rendez-vous are possible the selection is free. Hence μ specifies all possible orderings of the communications between processes).

Abstract Interpretation for Erasmus

Why should we try Abstraction Interpretation for Erasmus?

- Erasmus is not the same as CSP
- We do not aim for a full semantics
- We need only an *approximation* of actual behaviour
- "Fail safe": if a program has a bad property, we must detect it.
- Accept false positives: if we detect a bad property, the program may not actually possess it.

Abstract Interpretation for Erasmus

- State: $\langle P_0, Q_0, R_0 \rangle$
- State transitions (events omitted):

$$\begin{aligned} \langle P_0, Q_0, R_0 \rangle &\Rightarrow \langle P_0, Q_1, R_0 \rangle \\ \langle P_0, Q_0, R_0 \rangle &\Rightarrow \langle P_0, Q_2, R_0 \rangle \end{aligned}$$

- Set of states:

$$\Sigma = \{ \langle P_0, Q_0, R_0 \rangle, \langle P_0, Q_1, R_0 \rangle, \langle P_0, Q_2, R_0 \rangle, \ldots \}$$

- Semantic function:

$$F(\Sigma) = \Sigma \cup \left\{ s' \mid s \in \Sigma \text{ and } s \Rightarrow s' \right\}$$

Abstract Interpretation for Erasmus

- F is *monotonic*. From the definition

$$F(\Sigma) \quad = \quad \Sigma \cup \left\{ s' \mid s \in \Sigma \text{ and } s \Rightarrow s' \right\}$$

  and so $F(\Sigma) \supseteq \Sigma$.
- Σ is *bounded above* by the set of all possible states.
- Consequently, F has a *fixed point*.
  (That is, an X such that $F(X) = X$.)

Abstract Interpretation for Erasmus

To find the fixed point of F:

- Start with $\Sigma_0 = \{\sigma_0\}$.
  σ_0 is an initial state (e.g., $\langle P_0, Q_0, R_0 \rangle$).
- Compute $\Sigma_0, \Sigma_1, \Sigma_2, \ldots, \Sigma_n, \ldots$
  where $\Sigma_{n+1} = F(\Sigma_n)$
  until $\Sigma_{n+1} = \Sigma_n$.
- Then Σ_n is the fixed point.
- Σ_n is the set of *all reachable states*.

State set for the example

- Case 1 (no concurrency):

$$\Sigma = \{000, 001, 101, 110, 120, 130\}$$

- Case 2 (deadlock):

$$\Sigma = \{000, 001, 010, 011, 020, 021, 030\}$$

- Case 3 (concurrency, no deadlock):

$$\Sigma = \{000, 010, 020, 030\}$$

Detecting deadlock

- Practical procedure for computing F:

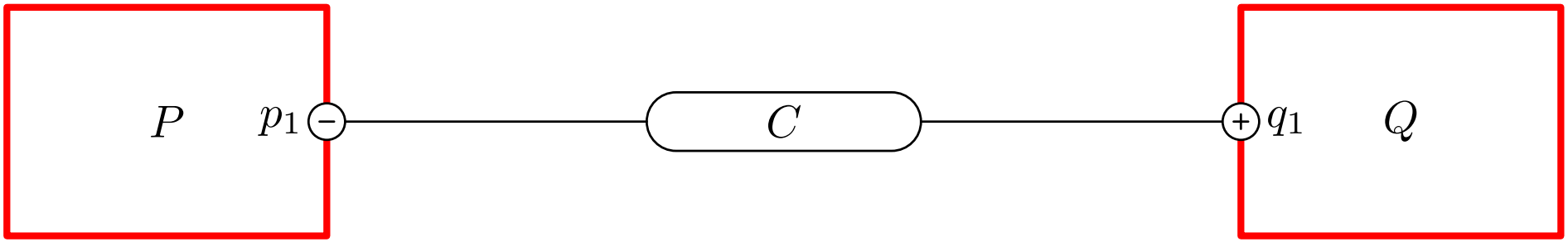
$S := \{\ s0\ \}$
while *choose unmarked s from S:*
  mark s;
  for each *successor s' of s:*
    insert s' into S

- Computation detects states with no successors
- These are deadlocked states
- Computation checks reachable states *only*
- Processes with only one state can be ignored
- Computation can be performed per cell

Abstract Interpretation: Alternatives

- We can check properties other than deadlock:
  All we have to do is define a suitable abstract semantics.
- We can use partial orders other than reachable states:
  E.g., the lattice of failures in conventional CSP semantics.

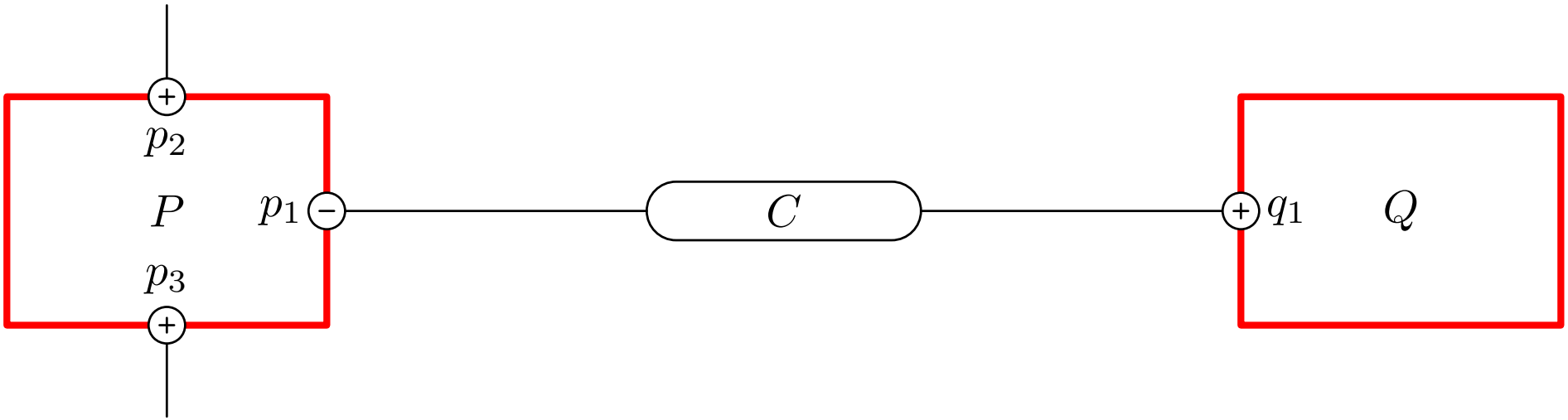
Communication without selection



$P =$ **process** *p1: -K* {
  loop {
    p1.snd
  }
}

$Q =$ **process** *q1: +K* {
  loop {
    q1.rcv
  }
}

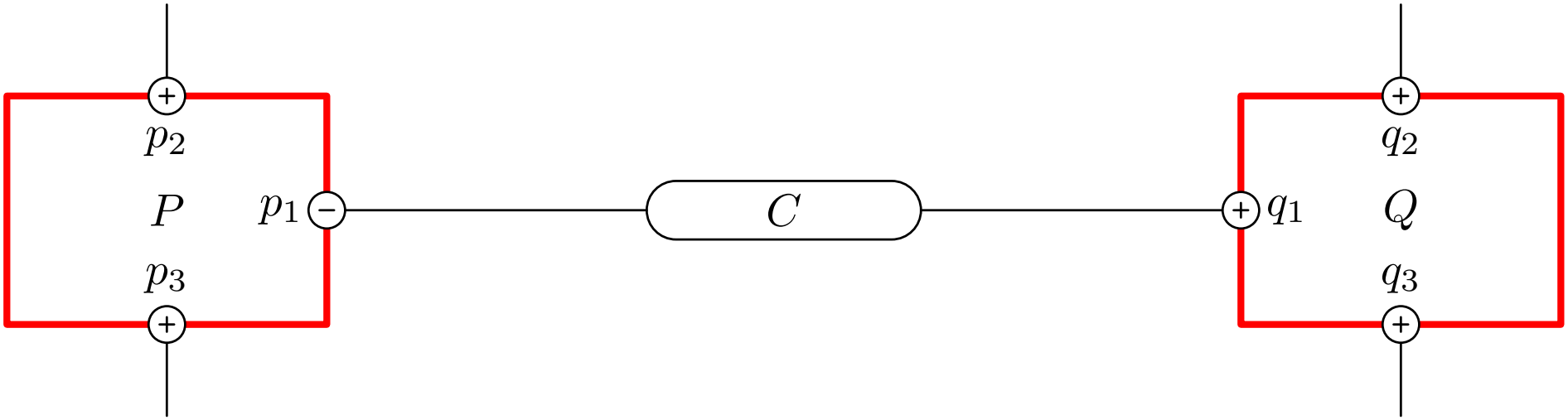
Communication with selection by one process



$P =$ **process** *p1: -K* {
  loop select {
  || *p1.snd*
  || *p2.rcv*
  || *p3.rcv*
  }
}

$Q =$ **process** *q1: +K* {
  loop {
    q1.rcv
  }
}

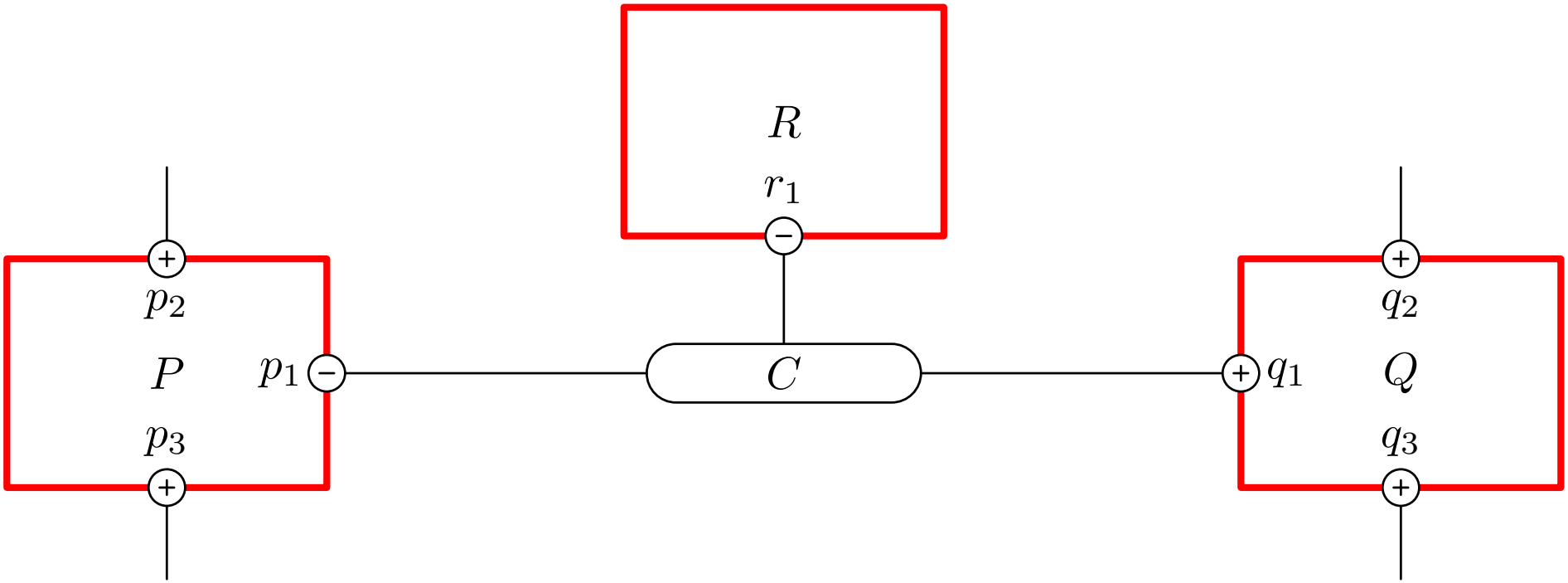
Communication with selection by two processes



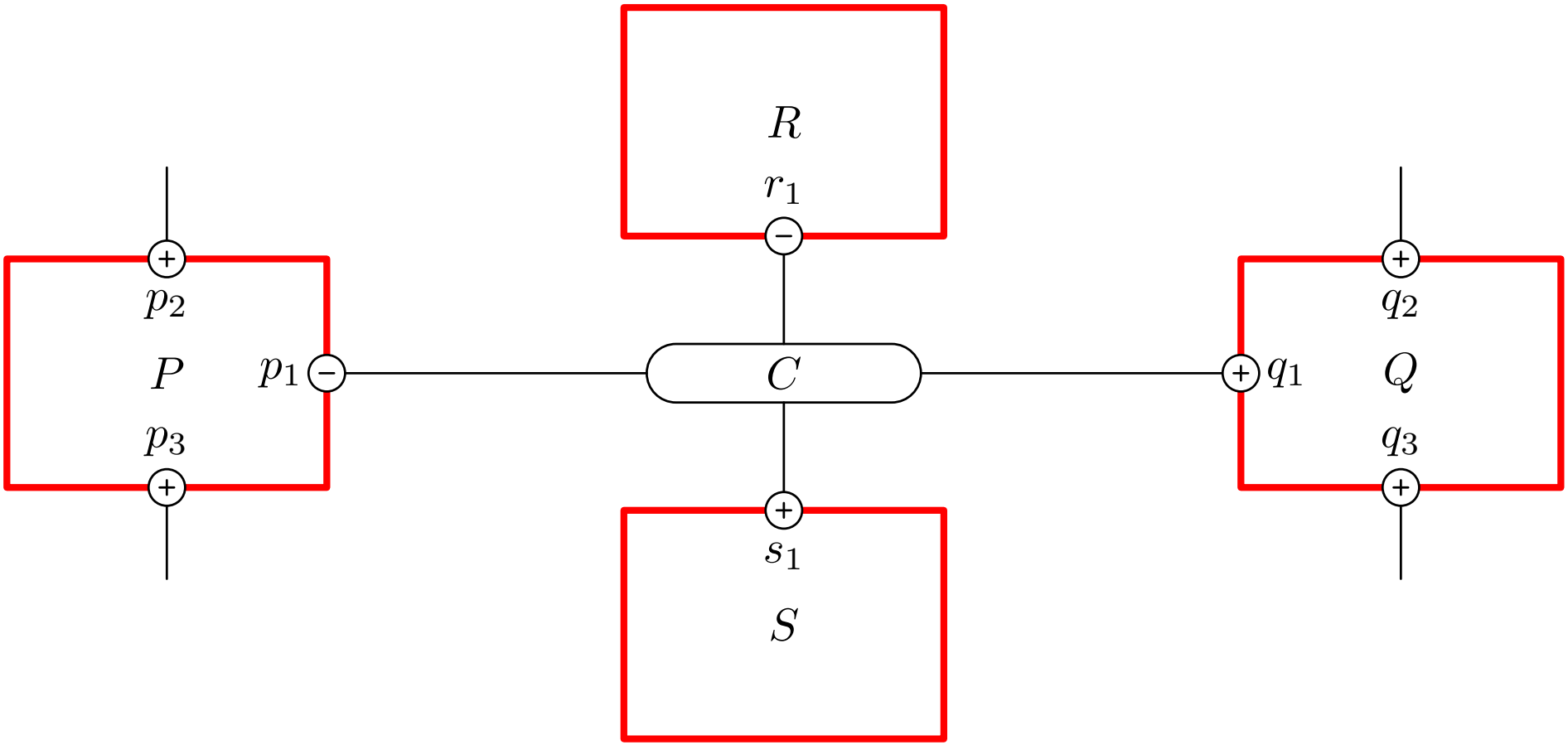

```
P = process p1: -K {
  loop select {
  ||  p1.snd
  ||  p2.rcv
  ||  p3.rcv
  }
}
```

```
Q = process q1: +K {
  loop select {
  ||  q1.rcv
  ||  q2.rcv
  ||  q3.rcv
  }
}
```

One server, many clients on a channel



Multiple servers and clients on a channel



Verifying Communication Algorithms

- Describe the communication algorithm in pseudocode
- Translate the pseudocode into a specification written in micro Common Representation Language 2 (mCRL2)
- Process the specification with the Linearizer and LTS Generator from the mCRL2 toolkit
- Verify that the Labelled Transition System (LTS) has the desired properties

Conclusion

If we can achieve these goals:

- High-level analysis (abstract interpretation),
  performed once for each program,
  will deetect potential communication problems
- Low-level analysis (verification of communication algorithms by process algebra),
  performed once only,
  will prevent synchronization problems

The Erasmus Team

- **Brian Shearing** and **Peter Grogono**, principal investigators
- **Nima Jafroodi**, Ph.D., process algebras
- **Maryam Zakeryfar**, Ph.D., abstract interpretation
- **Duo Peng**, M.C.Sc., web applications
- **Shruti Rathee**, M.C.Sc., not decided yet