

Ports, Protocols, and Processes: a Programming Paradigm?

Peter Grogono

Computer Science and Software Engineering

Concordia University

22 November 2007

The Erasmus Project



Desiderius Erasmus of Rotterdam (1466-1536)

Dramatis Personæ

Brian Shearing

Peter Grogono

Rana Issa

Nima Jafroodi

Nurudeen Lameed

Inspiring Thoughts

"The fox has many tricks. The hedgehog has but one. But that is the best of all."

Erasmus, c. 1500

"It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."

Perlis, 1982

"One thing [language designers] should not do is to include untried ideas of their own. Their task is consolidation, not innovation."

Hoare, 1974

Road Map

▶ Programming

what we've done

▶ Principles

why we've done it

▶ Reasoning

how we know it works

▶ Development

how we fit it together

▶ For and against

why we bother

Cell



Main

```
Main = ();
```

```
Main();
```

Cell

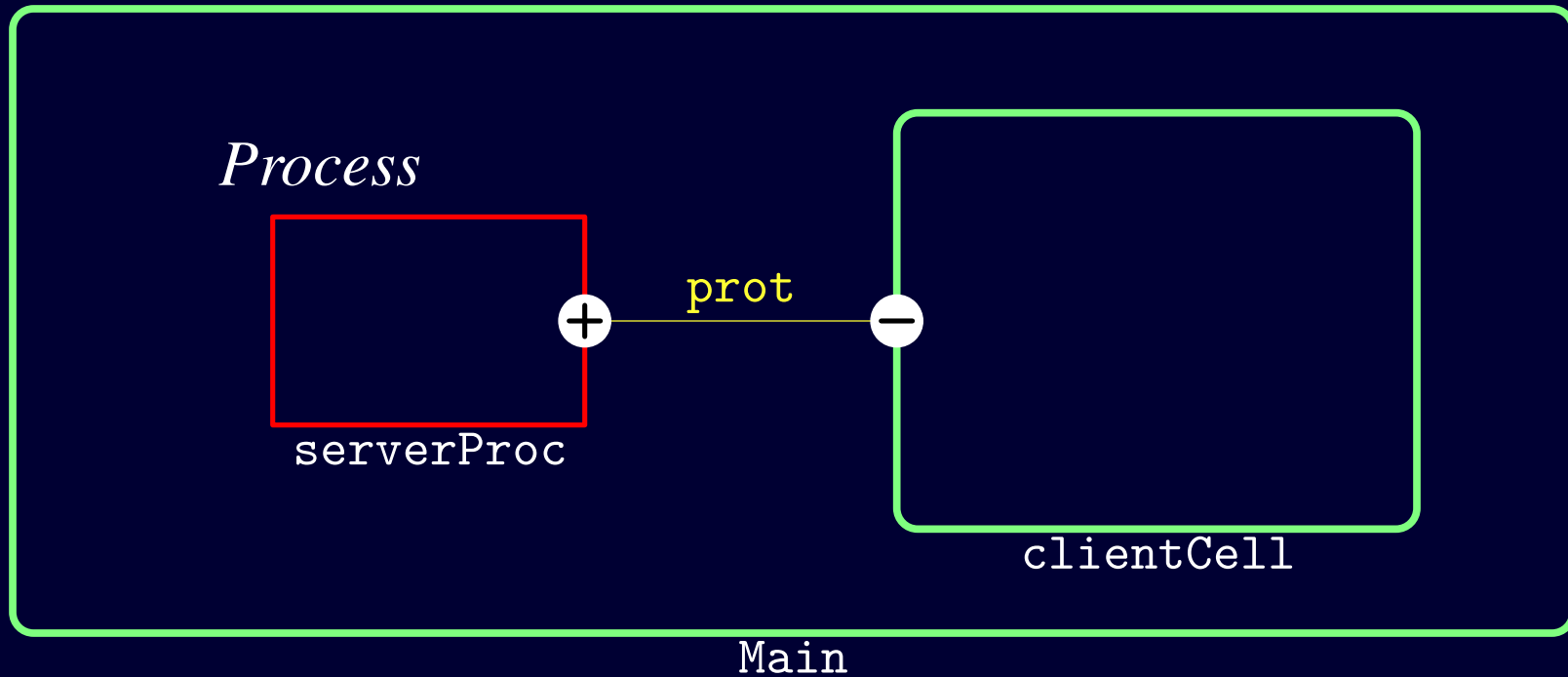


Main

```
prot = [ ];
```

```
Main = ();
```

```
Main();
```



```
prot = [ ];  
serverProc = { p +: prot | };  
clientCell = ( p -: prot | );  
Main = ( p :: prot; serverProc(p); clientCell(p) );  
  
Main();
```

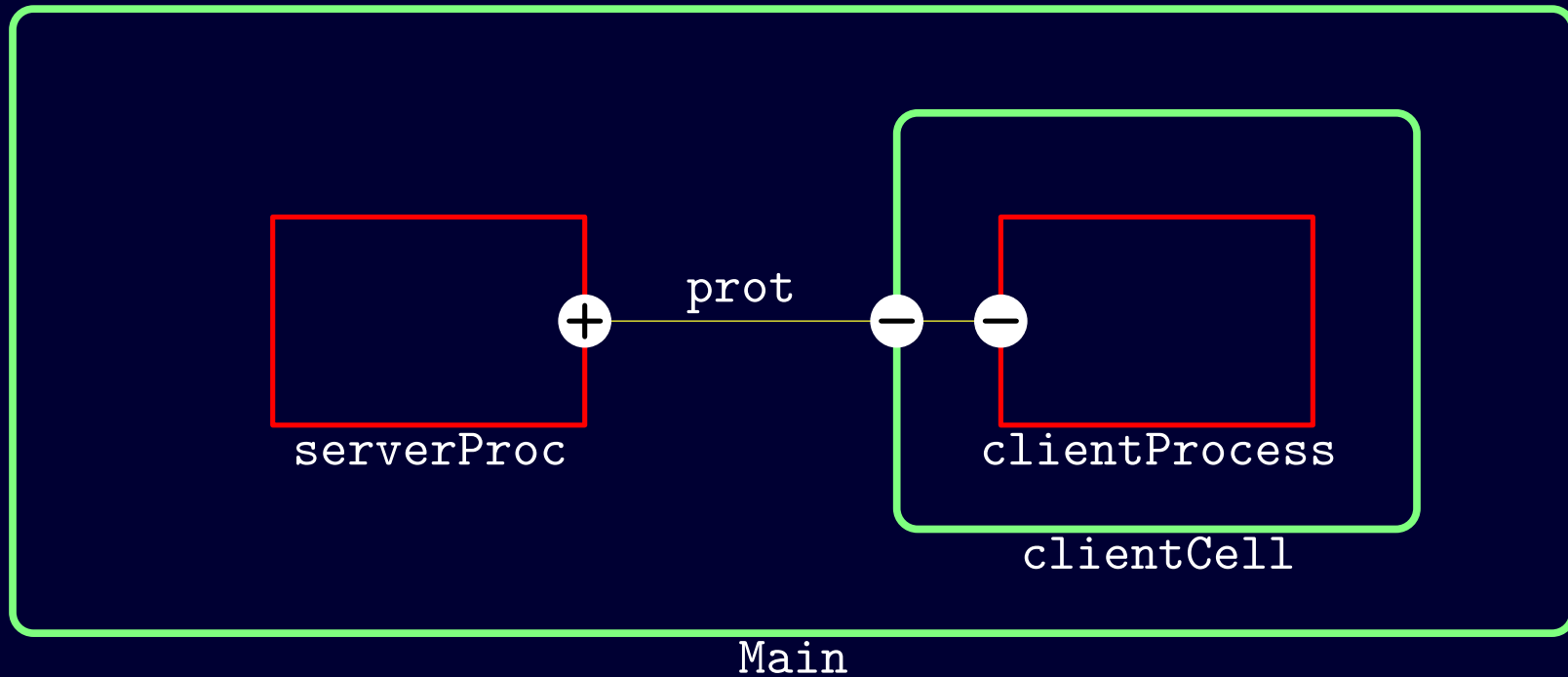
```
prot = [ start; *( query: Text; ^reply: Integer ); stop ];

serverProc = { p +: prot |
  p.start;
  loopselect
    || input: Text := p.query; p.reply := 0
    || p.stop; exit
  end
};

clientCell = ( p -: prot | );

Main = ( p :: prot; serverProc(p); clientCell(p) );

Main();
```



```
clientProc = { p -: prot | };
```

```
clientCell = ( p -: prot | clientProc(p) );
```

Protocols

```
query = [ question; ^answer ]
```

```
sequence = [  
    first: Integer;  
    second: Text;  
    third: Float ]
```

```
method1 = [ *( arg1; arg2; ...; ^result ) ]
```

```
method2 = [ *( arg1; arg2; ...; ^res1; ^res2 ) ]
```

```
class = [ *( M1 | M2 | ... | Mn ) ]
```

Statements

```
select
  || p.red; ...
  || p.yellow; ...
  || p.green; ...
end
```

```
select
  |stored < 10| buff[i] := p.x; ...
  |stored > 0| q.y := buff[j]; ...
end
```

```
select fair ...
select ordered ...
select random ...
```

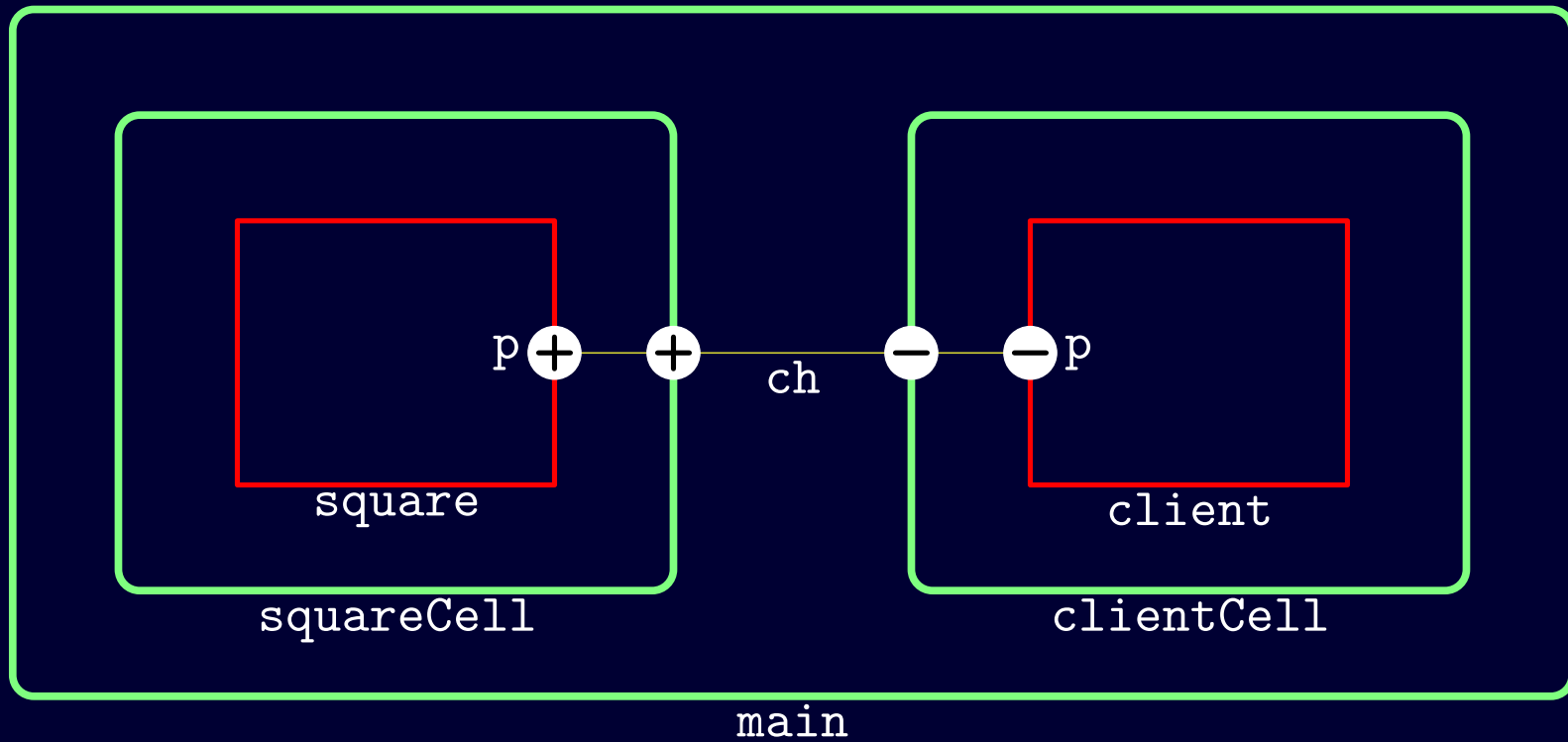
Processes

```
prot = [ *( arg: Integer ) ];

filter = { p +: prot |
  prime: Integer := p.arg;
  sys.out := text prime + ' ';
  q -: prot;
  filter(q);
  loop
    n: Integer := p.arg;
    if n % prime != 0
      then q.arg := n
    end
  end
};
```




Semantics vs. Deployment



Code

```
sqProt = [ *( query: Float; ^reply: Text ) ];
square = { p +: sqProt |
  loop
    q: Float := p.query;
    p.reply := text(q * q);
  end
};
squareCell = ( port +: sqProt | square(port) );
client = { p -: sqProt |
  p.query := 2;
  sys.out := p.reply + "\n";
};
clientCell = ( port -: sqProt | client(port) );
main = ( ch :: sqProt; squareCell(ch); clientCell(ch) );
main();
```

Metacode

```
<Mapping>
  <Processor> alpha.encs.concordia.ca
    <Port> 5555 </Port>
    <Cell> squareCell </Cell>
    <Cell> clientCell1 </Cell>
  </Processor>
  <Processor> beta.encs.concordia.ca
    <Port> 5555 </Port>
    <Cell> squareCell1 </Cell>
    <Cell> clientCell </Cell>
  </Processor>
</Mapping>
```

Road Map

▶ Programming

what we've done

▶ Principles

why we've done it

▶ Reasoning

how we know it works

▶ Development

how we fit it together

▶ For and against

why we bother

Cells

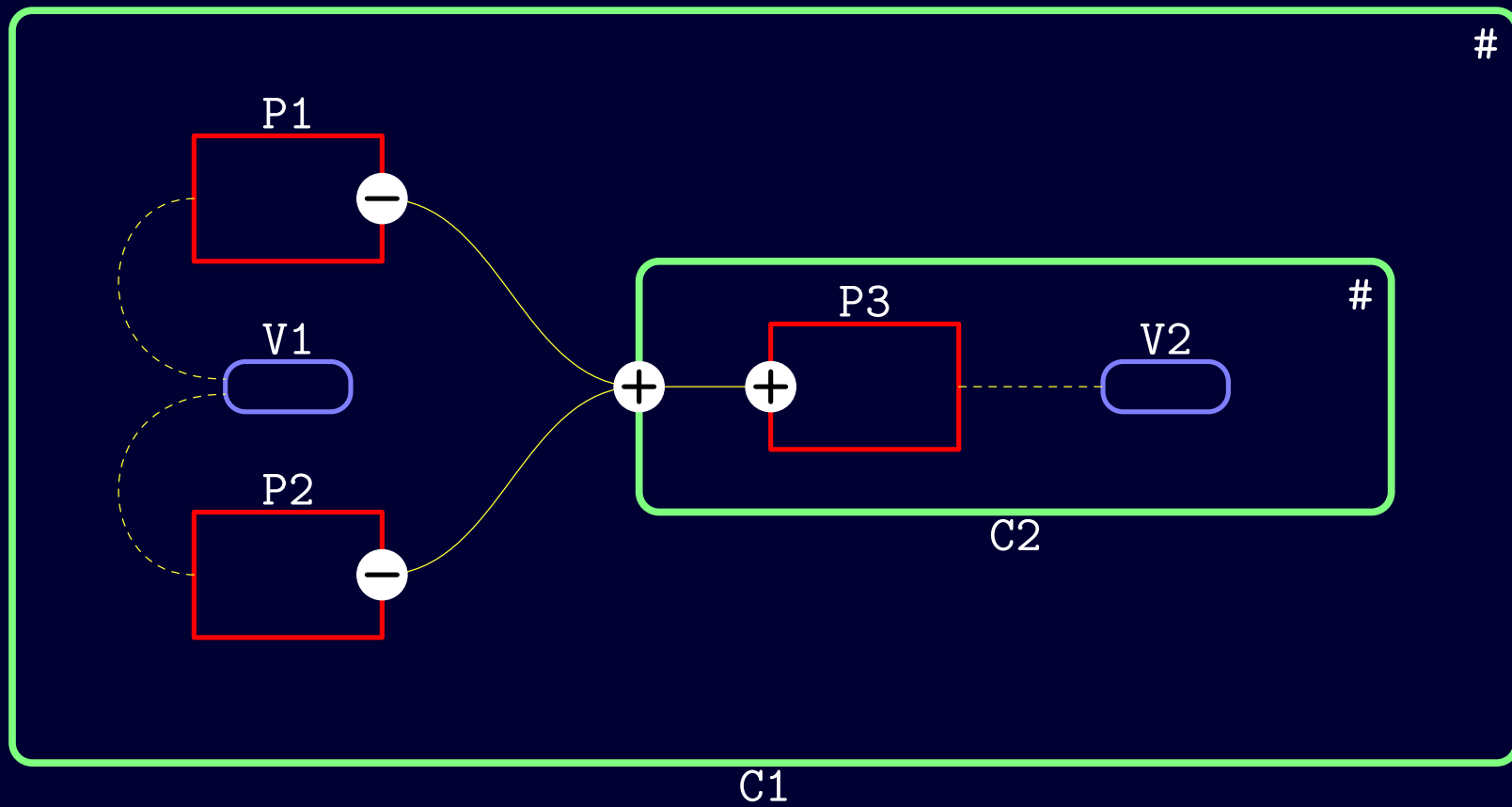
- ▶ Programs consist of cells
- ▶ Cells may contain variables, processes, and cells
- ▶ Cells can be of any size
- ▶ *programs are "fractal"*
- ▶ Cells are "first-class citizens"
- ▶ Control flow never crosses a cell boundary
- ▶ Cells are explicitly provided with all needed resources
- ▶ Cells may exchange messages
- ▶ Processes within a cell behave as co-routines



Processes

- ▶ A process is always inside a cell
- ▶ Processes may contain variables, processes, and cells
- ▶ Processes are "first-class citizens"
- ▶ All actions are performed within processes
- ▶ Control flow never crosses a process boundary
- ▶ A process may access variables within its cell
- ▶ Processes communicate by exchanging messages
- ▶ A process relinquishes control when it communicates

no race conditions



One program counter per cell

Shared Variables

```
proc = { p += prot; sv: Float |  
  ... sv ...  
  sys.out := sv;  
  p.val := ...  
  sys.out := sv;  
}
```

Shared Variables

```
proc = { p +: prot; sv: Float |  
  ... sv ...  
  sys.out := sv;  
  p.val := sv;  
  sys.out := sv;  
}
```

Shared Variables

```
proc =  
{ ... |  
  atomic  
  {  
    ...  
    open  
    {  
      p.val := ...  
    }  
    ...  
  }  
}
```

Protocols

- ▶ Protocols define interfaces
- ▶ Protocols specify communication patterns
- ▶ Protocols consist of typed messages and signals
- ▶ Protocols define sequence, choice, and repetition
- ▶ There is a "satisfaction" relation on protocols

details later

Messages

- ▶ A "sent" message is an **lvalue**:

```
p.result := 42;
```

- ▶ A "received" message is an **rvalue**:

```
sum := p.val + ...;
```

- ▶ Signals **synchronize**:

```
p.stop
```

Messages

- ▶ A "sent" message is an **lvalue**:

```
p.result := 42;
```

- ▶ A "received" message is an **rvalue**:

```
sum := p.val + q.val;
```

- ▶ Signals **synchronize**:

```
p.stop
```

Separation of Concern

Cells define structure ~ Processes define action

Code defines meanings ~ Metacode defines deployment

Protocols specify processes ~ Protocols ensure satisfaction

Road Map

- ▶ Programming

what we've done

- ▶ Principles

why we've done it

- ▶ Reasoning

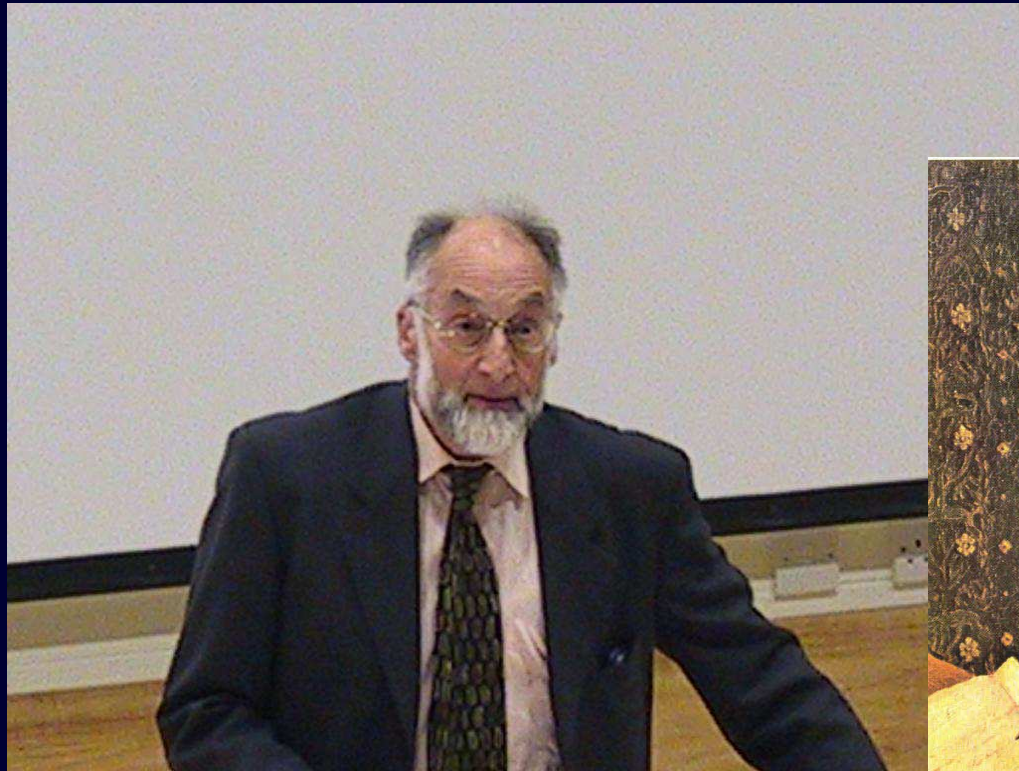
how we know it works

- ▶ Development

how we fit it together

- ▶ For and against

why we bother

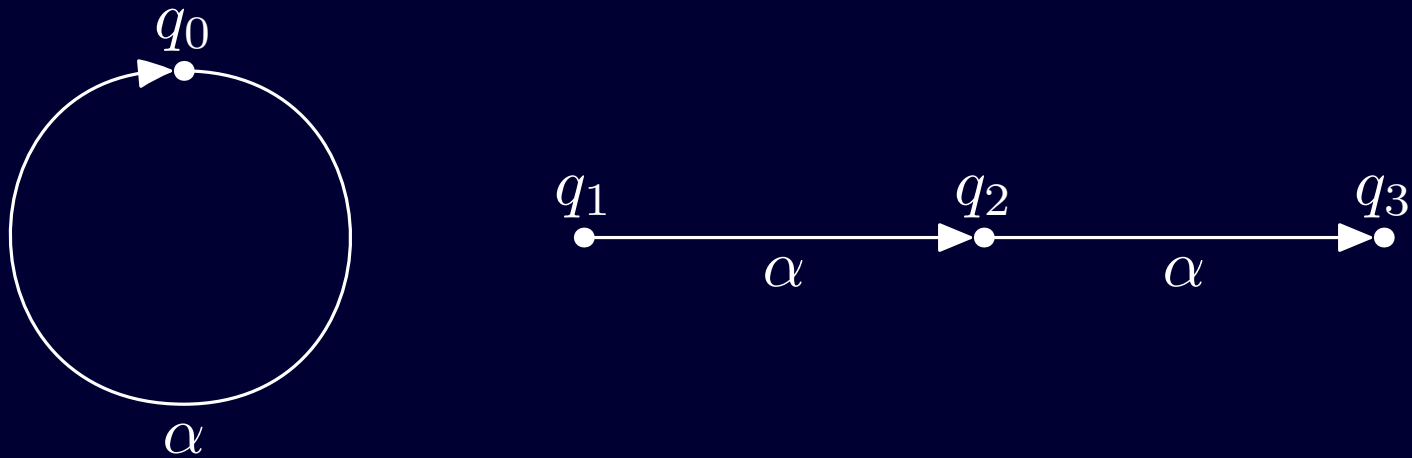


Labelled Transition System (LTS):

$$Q = \{q_0, q_1, q_2, q_3\},$$

$$L = \{\alpha\},$$

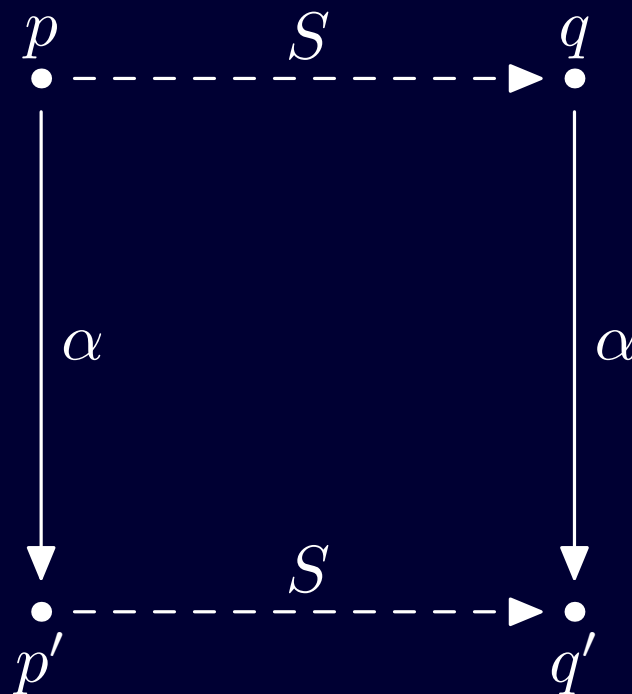
$$T = \{(q_0, \alpha, q_0), (q_1, \alpha, q_2), (q_2, \alpha, q_3)\}.$$

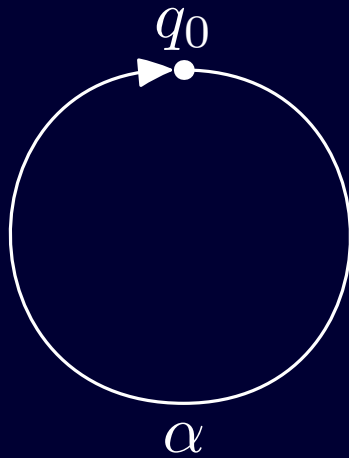


Strong Simulation: $S \subseteq Q \times Q$

If $(p, q) \in S$ and $p \xrightarrow{\alpha} p'$ then

there exists $q' \in Q$ such that $(p', q') \in S$ and $q \xrightarrow{\alpha} q'$.





$$S = \{ (q_1, q_0), (q_2, q_0), (q_3, q_0) \}$$

is a strong simulation.

If

- (Q, L, T) is a LTS
- $A \subseteq Q$
- $B \subseteq Q$
- $S \subseteq A \times B$ is a strong simulation

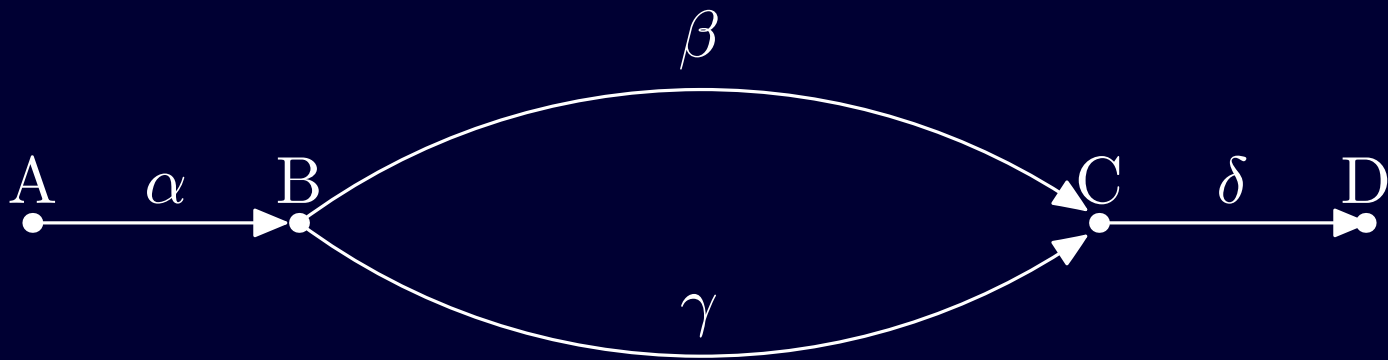
then $\left\{ \begin{array}{l} A \sqsubseteq B \quad (A \text{ is simulated by } B) \\ B \sqsupseteq A \quad (B \text{ simulates } A) \end{array} \right.$

\sqsupseteq is reflexive, transitive, **computable**.

Protocol P :

$\alpha; (\beta | \gamma); \delta$

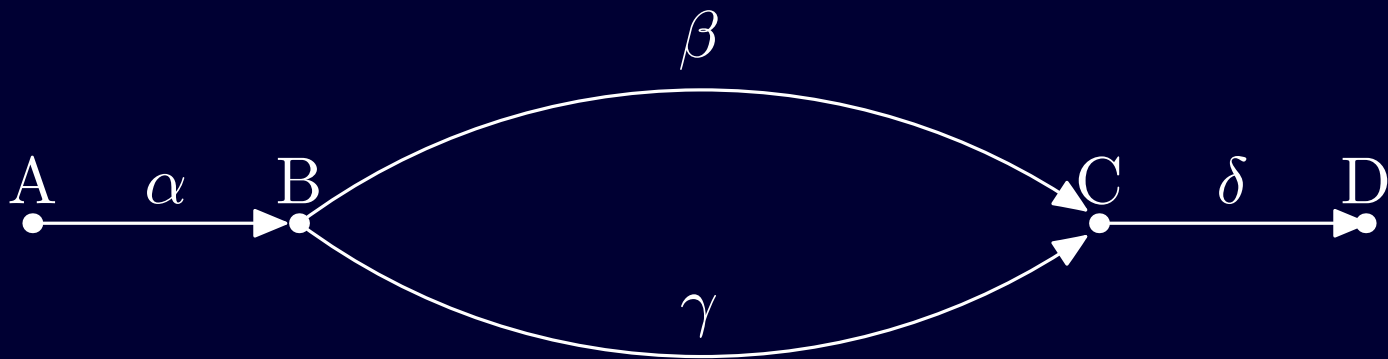
LTS $\mathcal{L}(P)$:



Code C :

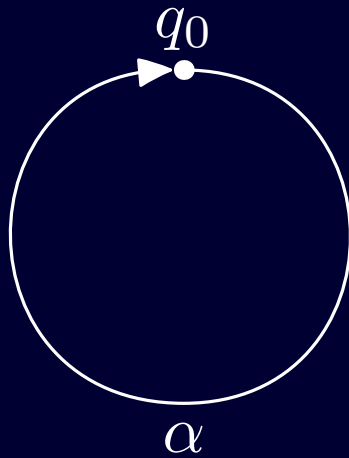
```
p.alpha;  
select  
  || p.beta; ...  
  || p.gamma; ...  
end;  
p.delta
```

LTS $\mathcal{L}(C)$:





$$\mathcal{L}(S) \sqsupseteq \mathcal{L}(P_S) \sqsupseteq \mathcal{L}(P_C) \sqsupseteq \mathcal{L}(C)$$



Server:

```

loop
  p.alpha := ...
end

```

Protocol:

```

[ *( alpha ) ]

```

Client:

```

x := p.alpha;
y := p.alpha

```

Hennessy-Milner Logic

Syntax:

F	$=$	X	(states satisfying X)
		tt	(all states)
		ff	(no states)
		$F_1 \wedge F_2$	(intersection)
		$F_1 \vee F_2$	(union)
		$\langle \alpha \rangle F$	(some α -trans $\longrightarrow F$)
		$[\alpha] F$	(all α -trans $\longrightarrow F$)

Semantics (with respect to an LTS (Q, L, T)):

$\llbracket \cdot \rrbracket : \Sigma \rightarrow \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$

$$\llbracket X \rrbracket(S) = S$$

$$\llbracket tt \rrbracket(S) = Q$$

$$\llbracket ff \rrbracket(S) = \emptyset$$

$$\llbracket F_1 \wedge F_2 \rrbracket(S) = \llbracket F_1 \rrbracket(S) \cap \llbracket F_2 \rrbracket(S)$$

$$\llbracket F_1 \vee F_2 \rrbracket(S) = \llbracket F_1 \rrbracket(S) \cup \llbracket F_2 \rrbracket(S)$$

$$\llbracket \langle \alpha \rangle \rrbracket(S) = \{ p \mid \exists p' \in S . p \xrightarrow{\alpha} p' \}$$

$$\llbracket [\alpha] \rrbracket(S) = \{ p \mid \forall p' . p \xrightarrow{\alpha} p' \Rightarrow p' \in S \}$$

Reasoning - Summary



Also:

▶ FSP / LTSA (Magee & Kramer)

Finite State Processes / Labelled Transition System Analyzer

▶ TLA+ (Lamport)

Temporal Logic of Actions Plus

Road Map

▶ Programming

what we've done

▶ Principles

why we've done it

▶ Reasoning

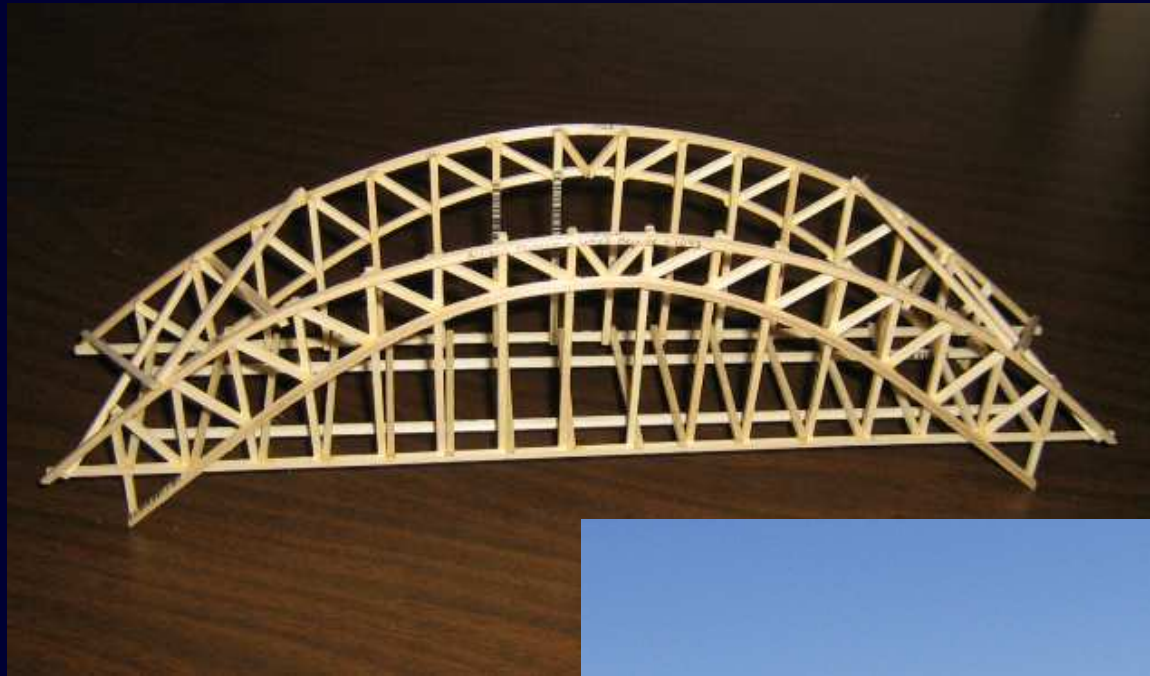
how we know it works

▶ Development

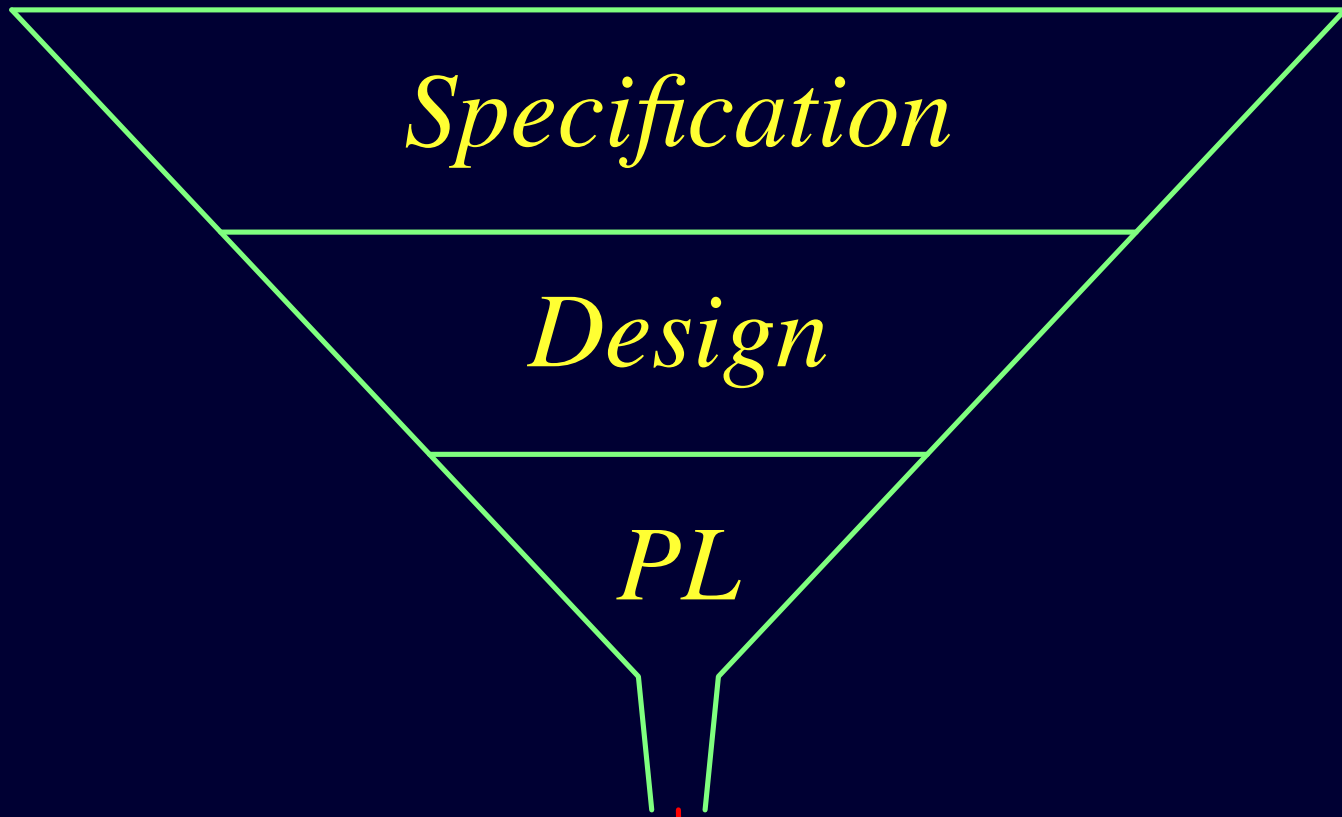
how we fit it together

▶ For and against

why we bother



Requirements



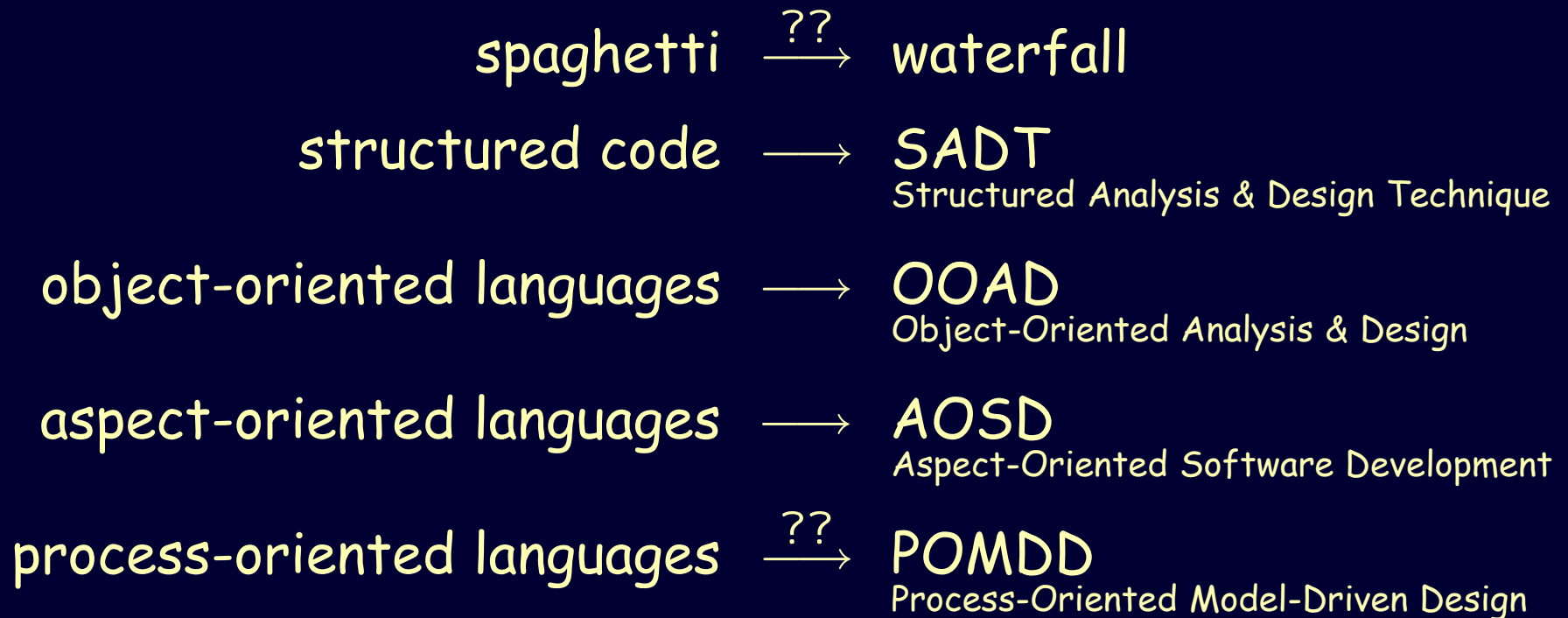
Specification

Design

PL

machine code

Change moves upwards in the funnel:



Therefore:
To effect change in the
software development process,
we must change the
programming paradigm.

Hypothesis

POMDD will succeed because:

- ▶ real world \cong concurrent processes
- ▶ concurrent processes \Rightarrow multiprocessors
- ▶ multiprocessors \Rightarrow concurrent software
- ▶ concurrent software \Rightarrow models real world

- ▶ cells/processes \Rightarrow lower coupling
- ▶ lower coupling \Rightarrow refactoring

Road Map

▶ Programming

what we've done

▶ Principles

why we've done it

▶ Reasoning

how we know it works

▶ Development

how we fit it together

▶ For and against

why we bother

The case against

The case against

- ▶ Programming languages are not the problem

The case against

- ▶ Programming languages are not the problem
- ▶ Object-oriented programming is good enough

The case against

- ▶ Programming languages are not the problem
- ▶ Aspect-oriented programming is good enough

The case against

- ▶ Programming languages are not the problem
- ▶ Aspect-oriented programming is good enough
- ▶ We've hidden the hard bits

CICS, J2EE, CORBA, ...

The case against

- ▶ Programming languages are not the problem
- ▶ Aspect-oriented programming is good enough
- ▶ We've hidden the hard bits
 - CICS, J2EE, CORBA, ...*
- ▶ Introducing a new paradigm is no longer feasible

The case for

The case for

- ▶ Many distributed applications

The case for

- ▶ Many distributed applications
- ▶ Multicore processors

The case for

- ▶ Many distributed applications
- ▶ Multicore processors
- ▶ Process-oriented programming is ... good

The case for

- ▶ Many distributed applications
- ▶ Multicore processors
- ▶ Process-oriented programming is ... good

We need software

development
maintenance
growth
adaptation
evolution
refactoring

Competition

ABCL/1

Joyce

Ada

Mozart/Oz

Cilk

Occam- π

Concurrent Pascal

SALSA

Erlang

SR

Wrap Up

- ▶ Objects ... Aspects ... Threads ... Locks ... ×
- ▶ Hidden guru code ... ×
- ▶ Modular concurrency ... ✓

Erasmus provides ...

- ▶ safe concurrency
- ▶ modularity
- ▶ scalability
- ▶ modelling capability
- ▶ weak coupling
- ▶ testing

Erasmus provides ...

- ▶ safe concurrency
- ▶ modularity
- ▶ scalability
- ▶ modelling capability
- ▶ weak coupling
- ▶ testing

... we hope!

The End