

Implementing the Fusion Machine

the fusion project at bologna

Lucian Wischik
and Cosimo Laneve, Manuel Mazzarra, also Philippa Gardner
Highwire Bologna September 2002

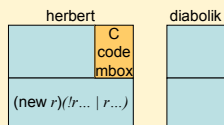
the pi calculus

```
(new r@diabolik.unibo.it)( // create a fresh channel at this location
  !r(a,b,c).ā(b,c) // place a relaying server there
  | f̄(&mbbox, "hello", "world") // get it to relay this message
)
void mbox(const char *text, const char *caption)
{ MessageBox(hwnd, text, caption, MB_OK); }
```

- Q. Where is the stuff located on the network? SOLVED
- Q. How is it implemented? ☉
- Q. How does it fit in with normal languages? ☉
- Q. Is there a behavioural type system? ...
And transactions or compensations?

distributed channel machine

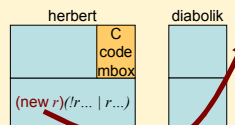
```
(new r@diabolik.unibo.it)( // create a fresh channel at this location
  !r(a,b,c).ā(b,c) // place a relaying server there
  | f̄(&mbbox, "hello", "world") // get it to relay this message
)
```



Pi calculus feature: it has a straightforward model for distribution - located channels, with program fragments at them.

distributed channel machine

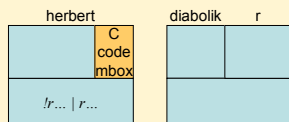
```
(new r@diabolik.unibo.it)( // create a fresh channel at this location
  !r(a,b,c).ā(b,c) // place a relaying server there
  | f̄(&mbbox, "hello", "world") // get it to relay this message
)
```



"create a new channel"

distributed channel machine

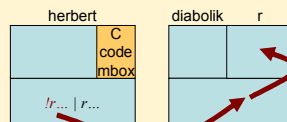
```
(new r@diabolik.unibo.it)( // create a fresh channel at this location
  !r(a,b,c).ā(b,c) // place a relaying server there
  | f̄(&mbbox, "hello", "world") // get it to relay this message
)
```



Optimisation: channel creation could be asynchronous (not requiring network messages) if herbert can generate a GUID for diabolik

distributed channel machine

```
(new r@diabolik.unibo.it)( // create a fresh channel at this location
  !r(a,b,c).ā(b,c) // place a relaying server there
  | f̄(&mbbox, "hello", "world") // get it to relay this message
)
```



2. "place it directly in the channel queue"

1. "accept this migrant !r()..."

Implementation note: a server thread accepts migrants and places them in the lower half; a worker thread picks up jobs from there, and executes them

distributed channel machine

```

(new r@diabolik.unibo.it)( // create a fresh channel at this location
  !r(a,b,c).ā(b,c) // place a relaying server there
  | f̄(&mbbox, "hello", "world") // get it to relay this message
)
  
```

herbert

	C code mbox
r<&mbbox, "h,w">	

diabolik r

	lin(a,b,c). a<b,c>

Tech note: the address &mbbox is a 'network address' comprising herbert's IP number, port number, and the function's location in memory.

distributed channel machine

```

(new r@diabolik.unibo.it)( // create a fresh channel at this location
  !r(a,b,c).ā(b,c) // place a relaying server there
  | f̄(&mbbox, "hello", "world") // get it to relay this message
)
  
```

herbert

	C code mbox
r<&mbbox, "h,w">	

diabolik r

	lin(a,b,c). a<b,c>

1. "accept this migrant r<>..."
2. react with the waiting program!
place the resulting continuation in the *deployment area*.

distributed channel machine

```

(new r@diabolik.unibo.it)( // create a fresh channel at this location
  !r(a,b,c).ā(b,c) // place a relaying server there
  | f̄(&mbbox, "hello", "world") // get it to relay this message
)
  
```

herbert

	C code mbox

diabolik r

	lin(a,b,c). a<b,c>
&mbbox<"h,w">	

distributed channel machine

```

(new r@diabolik.unibo.it)( // create a fresh channel at this location
  !r(a,b,c).ā(b,c) // place a relaying server there
  | f̄(&mbbox, "hello", "world") // get it to relay this message
)
  
```

herbert

	C code mbox

diabolik r

	lin(a,b,c). a<b,c>
&mbbox<"h,w">	

Pi calculus feature:
integrates neatly with
conventional langs:
invoking a function
= sending a message.

(again, it reacts immediately,
this time by invoking C code)

distributed channel machine

```

(new r@diabolik.unibo.it)( // create a fresh channel at this location
  !r(a,b,c).ā(b,c) // place a relaying server there
  | f̄(&mbbox, "hello", "world") // get it to relay this message
)
  
```

herbert

	C code mbox

diabolik r

	lin(a,b,c). a<b,c>

world

hello

OK

Fusion Machine *

169.254.217.233 : 2794

x

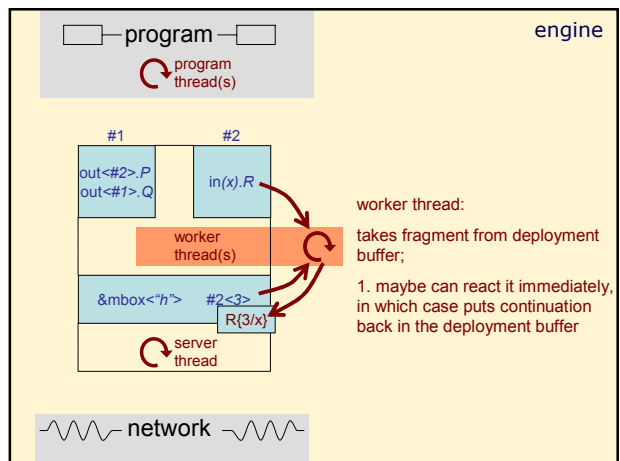
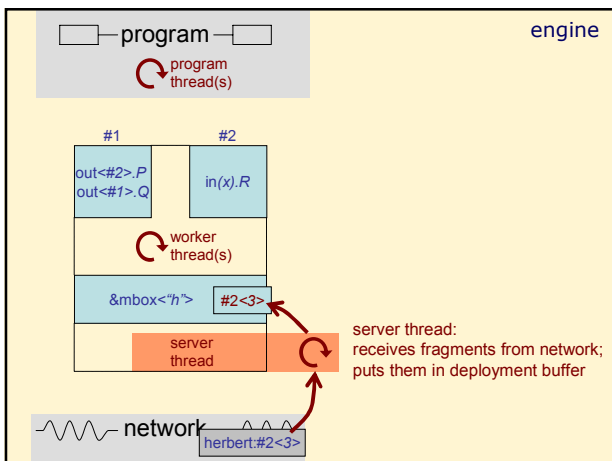
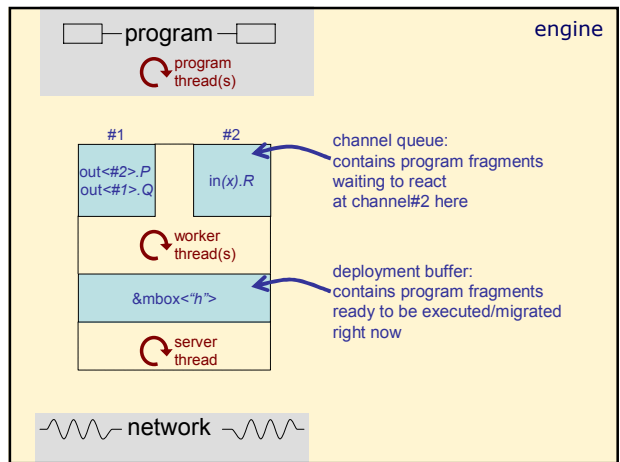
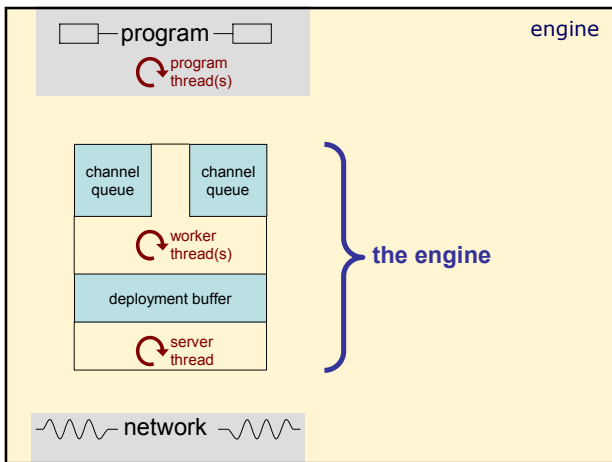
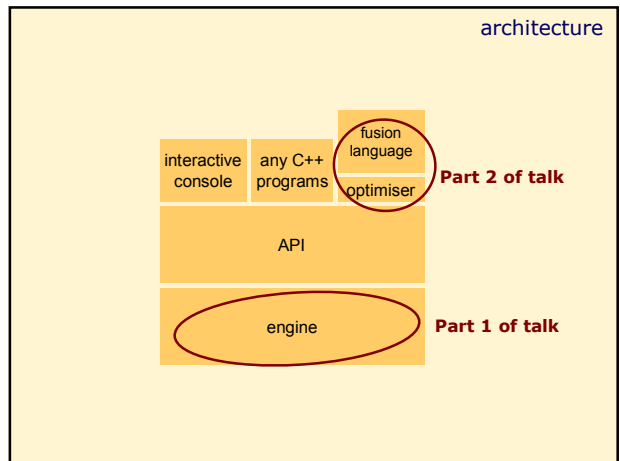
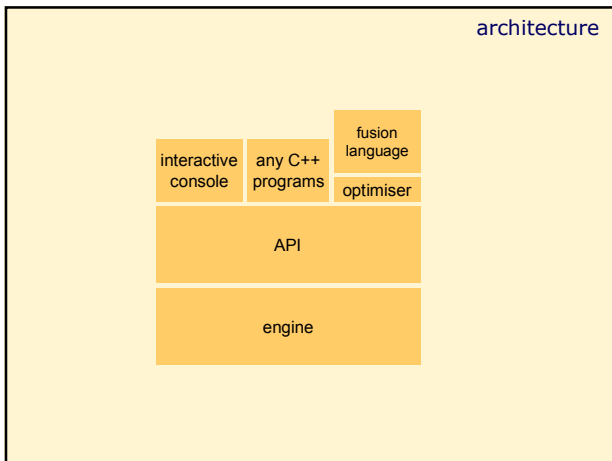
x<"hello">.0

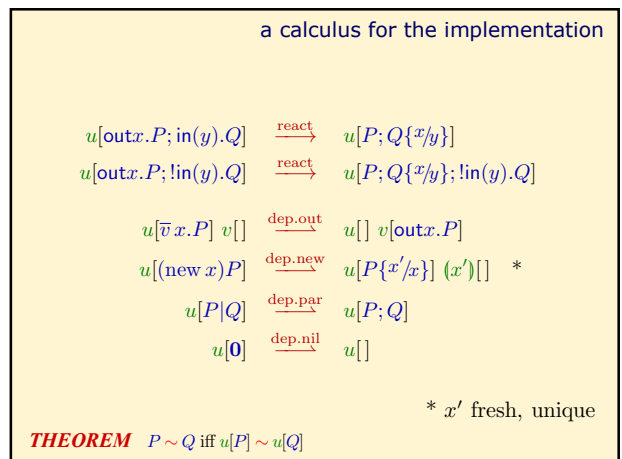
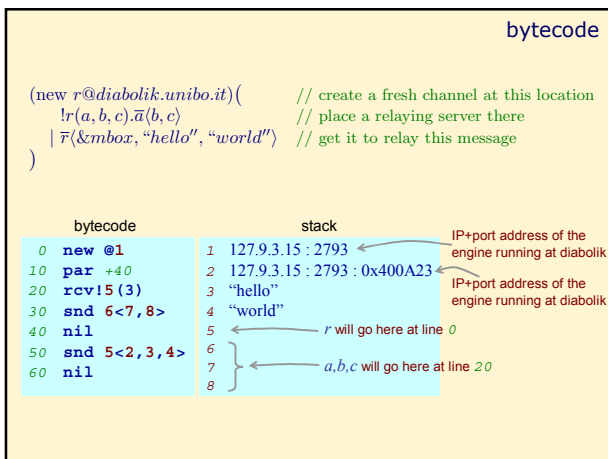
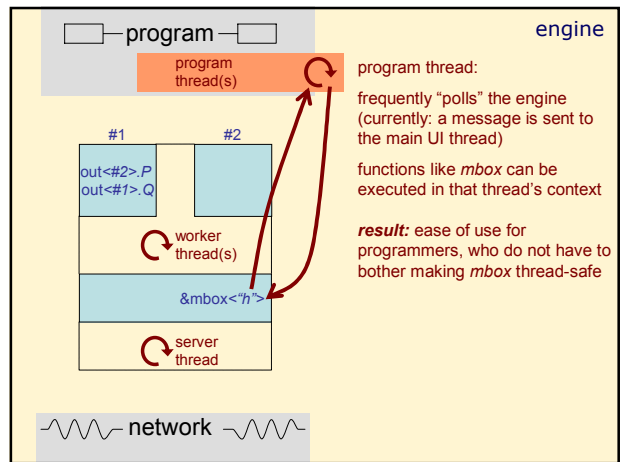
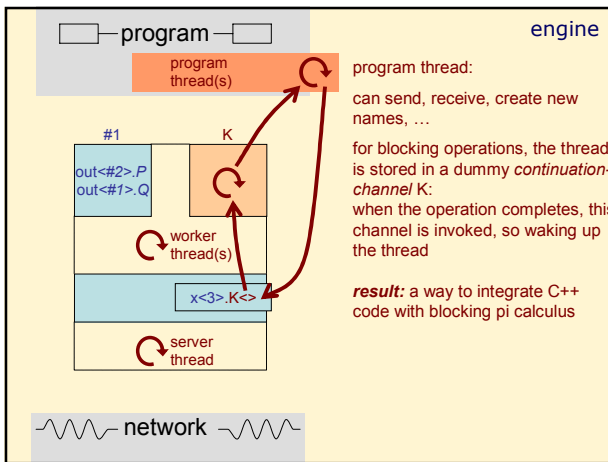
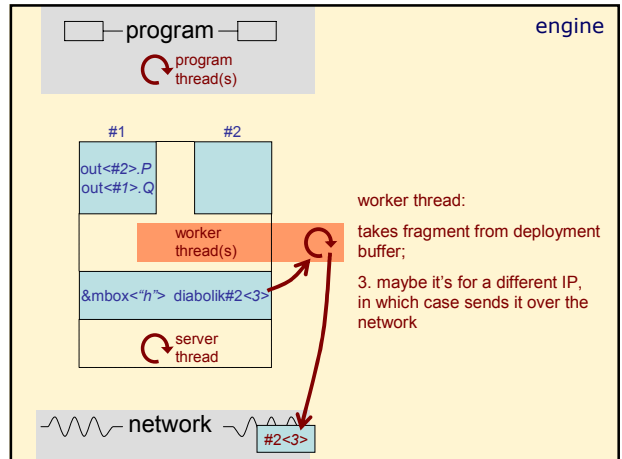
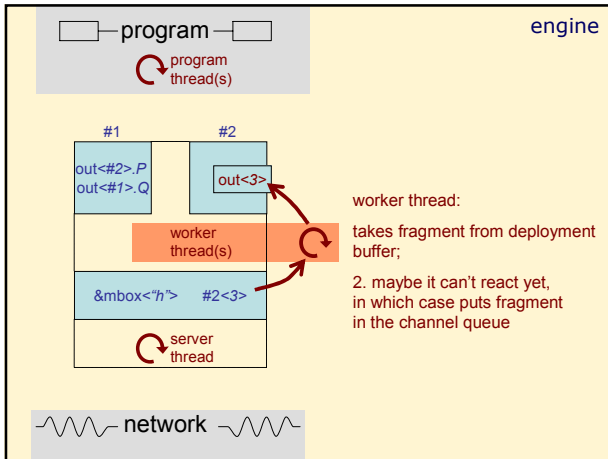
169.254.217.233:2793

diabolik relay

Pi calculus command console: [\[help\]](#)

Execute





the fusion project at bologna

goal:

- a distributed language and implementation based on the pi calculus.

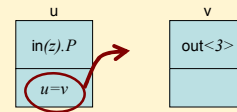
fusions:

- a fusion is "something which allows two channel-names to be used interchangeably" – we implement with forwarders.
- we have used them for **compiler optimisation**, and are looking at **enriching the language** with them.

theory:

- we have proved our engine correct with respect to standard pi calculus theory – i.e. full abstraction.
- we keep a tight link between theory and practice.

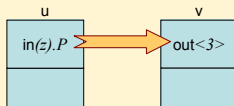
fusions: implementation



A fusion is something that allows two names to be used interchangeably

- We implement with *forwarders*:
- when the fusion $u=v$ is executed, it sets up a forwarder from u to v
- hence: no matter which of u or v you send a message to, the end result is the same.

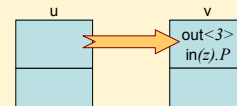
fusions: implementation



A fusion is something that allows two names to be used interchangeably

- We implement with *forwarders*:
- when the fusion $u=v$ is executed, it sets up a forwarder from u to v
- hence: no matter which of u or v you send a message to, the end result is the same.

fusions: implementation



A fusion is something that allows two names to be used interchangeably

- We implement with *forwarders*:
- when the fusion $u=v$ is executed, it sets up a forwarder from u to v
- hence: no matter which of u or v you send a message to, the end result is the same.

fusions: theory

The explicit fusion calculus

$$P ::= x=y \mid \bar{u}\tilde{x}.P \mid u\tilde{x}.P \mid P|P \mid (x)P \mid \mathbf{0}$$

$$\bar{u}\tilde{x}.P \mid u\tilde{y}.Q \longrightarrow \tilde{x}\tilde{y}.P \mid Q$$

$$x=y \mid P \equiv x=y \mid P\{y/x\} \quad \text{substitution}$$

$$(x)(x=y) \equiv \mathbf{0} \quad \text{local alias}$$

$$x=x \equiv \mathbf{0} \quad \text{reflexivity}$$

$$x=y \equiv y=x \quad \text{symmetry}$$

$$x=y \mid y=z \equiv x=z \mid y=z \quad \text{transitivity}$$

fusion vs pi

reaction in the pi calculus:

$$\bar{u}x.P \mid u(y).Q \rightarrow P \mid Q\{x/y\}$$

reaction in the explicit fusion calculus:

$$\begin{aligned} \bar{u}x.P \mid (y)u.y.Q &\equiv (y)(\bar{u}x.P \mid u.y.Q) && \text{assume } y \notin \text{fn}P \\ &\rightarrow (y)(x=y \mid P \mid Q) \\ &\equiv (y)(P \mid x=y \mid Q\{x/y\}) && \text{substitution due to fusion} \\ &\equiv P \mid (y)(x=y) \mid Q\{x/y\} && \text{scope intrusion} \\ &\equiv P \mid Q\{x/y\} && \text{remove local alias} \end{aligned}$$

the fusion clash problem

The fusion clash problem:

- If u becomes fused to two different names, how does it know whether to send its code to v or w?
- answer: migrate the fusion down the forwarders until it can be fulfilled; forwarders respect a total order on names.

the fusion clash problem

The fusion clash problem:

- If u becomes fused to two different names, how does it know whether to send its code to v or w?
- answer: migrate the fusion down the forwarders until it can be fulfilled; forwarders respect a total order on names.

the fusion clash problem

The fusion clash problem:

- If u becomes fused to two different names, how does it know whether to send its code to v or w?
- answer: migrate the fusion down the forwarders until it can be fulfilled; forwarders respect a total order on names.

the fusion clash problem

The fusion clash problem:

- If u becomes fused to two different names, how does it know whether to send its code to v or w?
- answer: migrate the fusion down the forwarders until it can be fulfilled; forwarders respect a total order on names.

the fusion clash problem

The fusion clash problem:

- If u becomes fused to two different names, how does it know whether to send its code to v or w?
- answer: migrate the fusion down the forwarders until it can be fulfilled; forwarders respect a total order on names.

the continuation problem

The continuation problem:

- P gets sent across the network three times
- this is a problem if P is large

the continuation problem

The continuation problem:

- P gets sent across the network three times
- this is a problem if P is large

the continuation problem

The continuation problem:

- P gets sent across the network three times
- this is a problem if P is large

the continuation problem

The continuation problem: (use fusions!)

- answer: pre-deploy $w(z).P$ to a private channel w' , physically adjacent to w , but unable to react
- eventually the fusion $w'=w$ will be executed
- this will allow w and w' to be used interchangeably (implemented with a local forwarder from w' to w)
- thus the continuation is liberated!
- fusions used as a *calculus-friendly way of writing pointers*

the input-mobility problem

The input-mobility problem:

- how can we pre-deploy $x(z).P$? we won't know where it goes until runtime!
- answer 1: well, although $x(z)$ cannot be pre-deployed, at least $v(y)$ and P can be!
- so we can still avoid most of the cost of moving
- Still to do: a mathematical treatment of this.

the input-mobility problem

The input-mobility problem: (use fusions!)

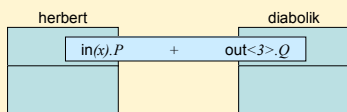
- how can we pre-deploy $x(z).P$? we won't know where it goes until runtime!
- answer 2: we pre-deploy $x(z).P$ to some private channel x'
- when finally x becomes known to us, we set up a forwarder
- a forwarder allows two channels to be used interchangeably.
- **Theorem:** efficiency is no worse than Join/Facile.

the input-mobility problem

The input-mobility problem: (use fusions!)

- how can we pre-deploy $x(z).P$? we won't know where it goes until runtime!
- answer 2: we pre-deploy $x(z).P$ to some private channel x'
- when finally x becomes known to us, we set up a forwarder
- a forwarder allows two channels to be used interchangeably.
- **Theorem:** efficiency is no worse than Join/Facile.

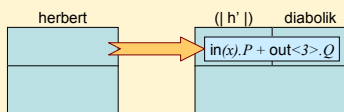
the choice problem



The choice problem:

- how to implement a distributed choice?
- any proposed reaction at *herbert* would have to ask permission from *diabolik* before proceeding. Awkward.

the choice problem



The choice problem: (use fusions!)

- how to implement a distributed choice?
- suggested answer: a fusion $\{herbert=h'\}$ implemented as a forwarder so rest of the program can refer equally *herbert* or *h'*
- the local summation is easy to implement.
- But... must fix forwarders if diabolik gets another fusion

the fusion question

Are fusions actually useful?

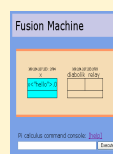
- **yes:** in solving the *continuation problem* as a calculus-friendly way of writing pointers.
- **maybe:** as part of an algorithm for *distributed choice*.
- **maybe:** Cosimo thinks to use *false fusions* like $1=2$ as a way to encode failed transactions/compensations.
- **???** Highwire seems to like them. Why?
- **no:** they seem too dangerous and costly, and hard for programmers to grasp intuitively. Seems difficult to mix normal data-types with fusions.

the fusion project at bologna

www.cs.unibo.it/fusion



prototype implementation in Java



distributed implementation in C++/Win32



conference papers on explicit fusions, fusion machine

Supplemental slides

efficiency result

THEOREM

- Using explicit fusions, we can compile a program with continuations into one without.
- This is a source-code optimisation, prior to execution.
- Every message becomes small (fixed-size).
- This might double the total number of messages but no worse than that. It also reduces latency.
- Our optimisation is a bisimulation congruence:

$$C[P] \sim C[\text{optimise } P]$$

```
(new xyz, v'@v, w'@w) {
  ux. v'-v // after u has reacted, it tells
  | v'y. w'-w // v' to fuse to v, so allowing
  | w'z // our v' atom to react with v atoms
}
```

virtual machine, formally

Machines $M ::= u[B]$ *channel machine at u*
 $(u)[B]$ *private channel machine*
 M, M
 0

Bodies $B ::= \text{out}\bar{c}.P$ *output atom*
 $\text{in}(\bar{x}).P$ *input atom*
 $\text{lin}(\bar{x}).P$ *replicated input*
 P *pi process*
 B, B

Processes $P ::= \bar{u}\bar{x}.P \mid [!u(\bar{x}).P \mid (x)P \mid P \mid P \mid 0$

the API

Treat functions as addresses

- a name $n = 2.3.1.7 : 9 : 0x04367110$
- so that `snd(n)` will invoke the function at that address

Calling `snd/rcv` directly from C++

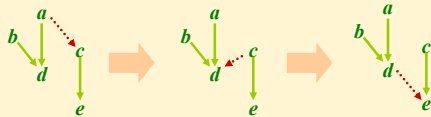
```
{ ... // there's an implicit continuation  $K$  after the rcv,
  rcv(x); // so we stall the thread and put  $x.K$  in the work bag.
  ... // When  $K$  is invoked, it signals the thread to wake up
}
```

Calling arbitrary pi code from C++

```
pi (*u!X.v!y | Qn);
pi (*u!X."+fun_as_chan(&test2)+"|Qn);

void test2()
{ ... }
```

clash avoidance algorithm



Effect: a distributed, asynchronous algorithm for merging trees.

- Correctness: it preserves the total-order on channels names;
- the equivalence relation on channels is preserved, before and after;
- it terminates, since each step moves closer to the root.
- (similar to Tarjan's Union Find algorithm, 1975)

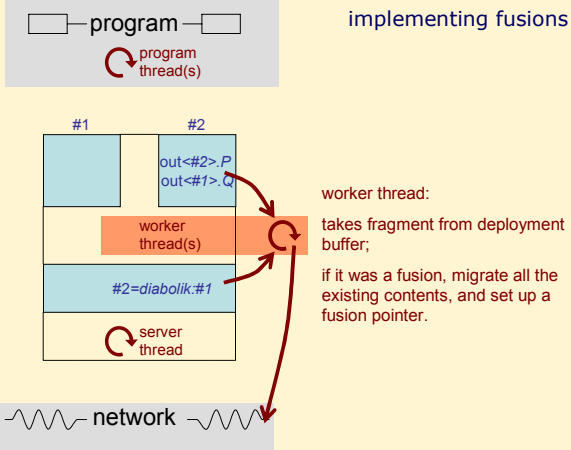
clash avoidance, calculus

$u[x=y] x[p:] \xrightarrow{\text{dep.fu}} u[] x[y:p]$ * assuming $x < y$
 * if p was nil, then discard $y=p$ in the result

The explicit fusion $x=y$ is an *obligation to set up a fusion pointer*.

A channel will either fulfil this obligation (if p was nil), or will pass it on.

implementing fusions



implementing fusions

