# 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

$(\mathrm{new}\ r@diabolik.unibo.it)($ // create a fresh channel at this location
$\quad !r(a,b,c).\overline{a}\langle b,c\rangle$ // place a relaying server there
$\quad \mid \overline{r}\langle \&mbox, \text{“}hello\text{''}, \text{“}world\text{''}\rangle$ // 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? ⊙

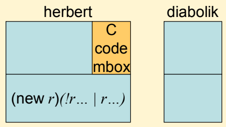
**Q.** How does it fit in with normal languages? ⊙

**Q.** Is there a behavioural type system? ...
And transactions or compensations?

---

## distributed channel machine

$(\mathrm{new}\ r@diabolik.unibo.it)($ // create a fresh channel at this location
$\quad !r(a,b,c).\overline{a}\langle b,c\rangle$ // place a relaying server there
$\quad \mid \overline{r}\langle \&mbox, \text{“}hello\text{''}, \text{“}world\text{''}\rangle$ // get it to relay this message
$)$

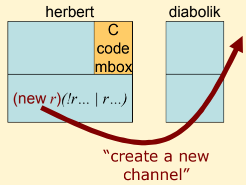



Pi calculus feature: it has a straightforward model for distribution – located channels, with program fragments at them.

---

## distributed channel machine

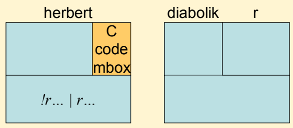
$(\mathrm{new}\ r@diabolik.unibo.it)($ // create a fresh channel at this location
$\quad !r(a,b,c).\overline{a}\langle b,c\rangle$ // place a relaying server there
$\quad \mid \overline{r}\langle \&mbox, \text{“}hello\text{''}, \text{“}world\text{''}\rangle$ // get it to relay this message
$)$




---

## distributed channel machine

$(\mathrm{new}\ r@diabolik.unibo.it)($ // create a fresh channel at this location
$\quad !r(a,b,c).\overline{a}\langle b,c\rangle$ // place a relaying server there
$\quad \mid \overline{r}\langle \&mbox, \text{“}hello\text{''}, \text{“}world\text{''}\rangle$ // get it to relay this message
$)$

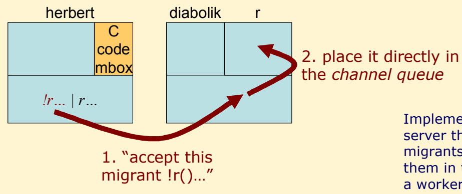



Optimisation: channel creation could be asynchronous (not requiring network messages) if herbert can generate a GUID for diabolik

---

## distributed channel machine

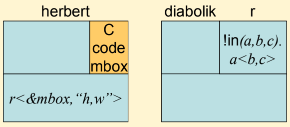
$(\mathrm{new}\ r@diabolik.unibo.it)($ // create a fresh channel at this location
$\quad !r(a,b,c).\overline{a}\langle b,c\rangle$ // place a relaying server there
$\quad \mid \overline{r}\langle \&mbox, \text{“}hello\text{''}, \text{“}world\text{''}\rangle$ // get it to relay this message
$)$




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

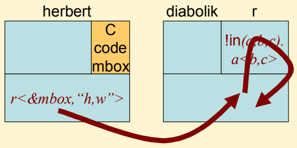
(new $r@diabolik.unibo.it$)( // create a fresh channel at this location
$!r(a,b,c).\bar{a}\langle b,c\rangle$ // place a relaying server there
$|\ \bar{r}\langle \&mbox, “hello”, “world”\rangle$ // get it to relay this message
)




Tech note: the address &mbox 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
$!r(a,b,c).\bar{a}\langle b,c\rangle$ // place a relaying server there
$|\ \bar{r}\langle \&mbox, “hello”, “world”\rangle$ // get it to relay this message
)



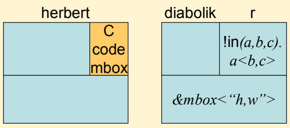

2.
react with the waiting program!

place the resulting continuation in the *deployment* area.

1. “accept this migrant r<>...”

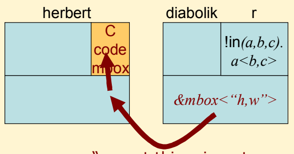
## distributed channel machine

(new $r@diabolik.unibo.it$)( // create a fresh channel at this location
$!r(a,b,c).\bar{a}\langle b,c\rangle$ // place a relaying server there
$|\ \bar{r}\langle \&mbox, “hello”, “world”\rangle$ // get it to relay this message
)




## distributed channel machine

(new $r@diabolik.unibo.it$)( // create a fresh channel at this location
$!r(a,b,c).\bar{a}\langle b,c\rangle$ // place a relaying server there
$|\ \bar{r}\langle \&mbox, “hello”, “world”\rangle$ // get it to relay this message
)



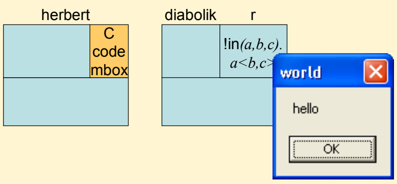

“accept this migrant &mbox<>...”

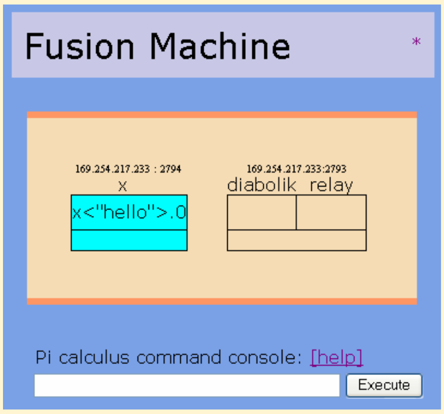
(again, it reacts immediately, this time by invoking C code)

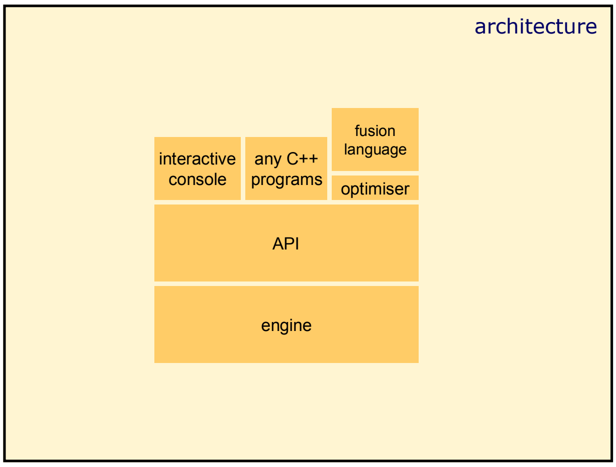
Pi calculus feature: integrates neaty with conventional langs: invoking a function = sending a message.

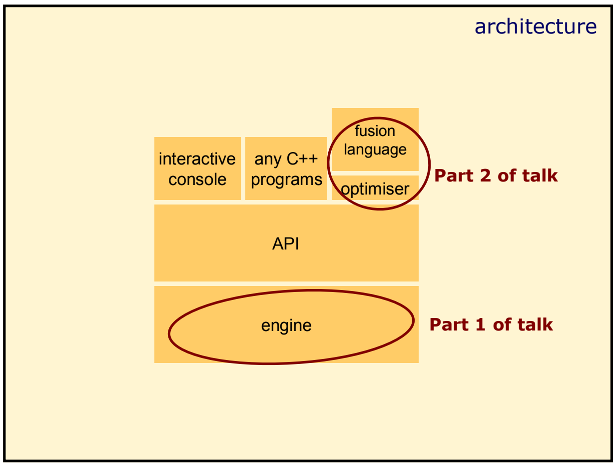
## distributed channel machine

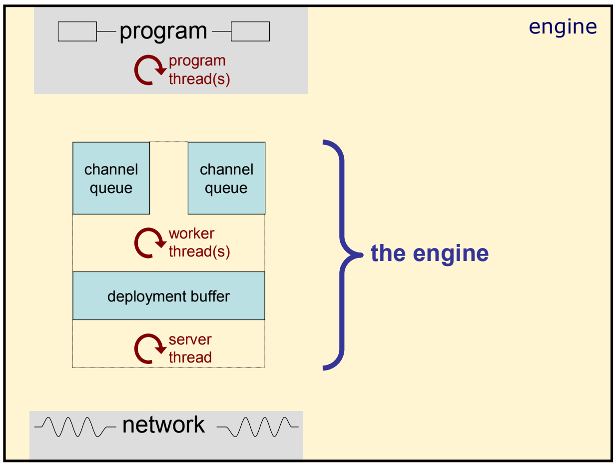
(new $r@diabolik.unibo.it$)( // create a fresh channel at this location
$!r(a,b,c).\bar{a}\langle b,c\rangle$ // place a relaying server there
$|\ \bar{r}\langle \&mbox, “hello”, “world”\rangle$ // get it to relay this message
)

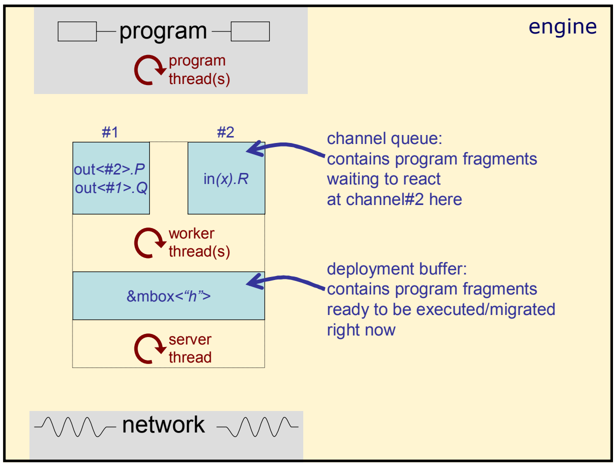


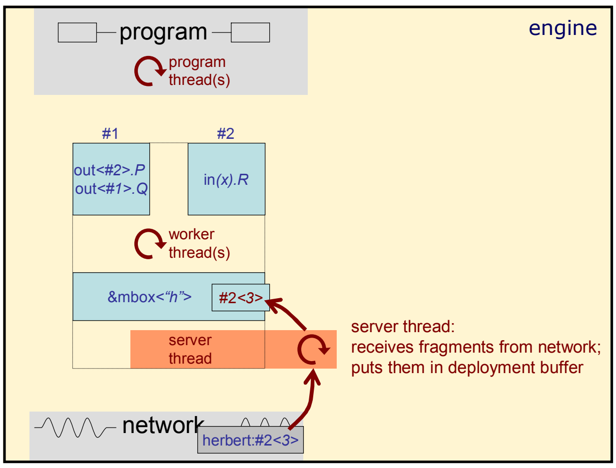



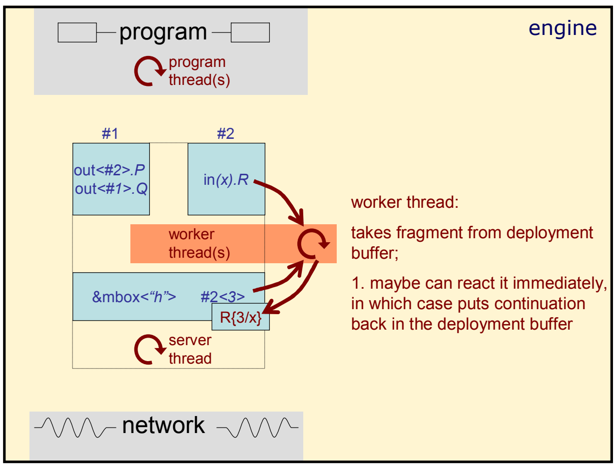

architecture
interactive console
any C++ programs
fusion language
optimiser
API
engine

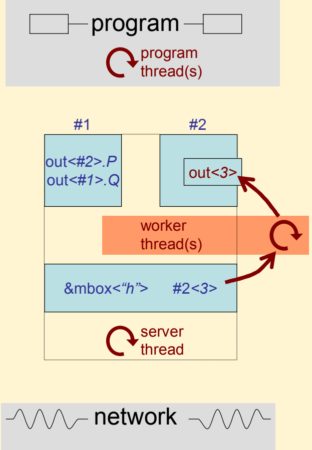

architecture
interactive console
any C++ programs
fusion language
optimiser
Part 2 of talk
API
engine
Part 1 of talk


engine
program
program thread(s)
channel queue
channel queue
worker thread(s)
deployment buffer
server thread
the engine
network


engine
program
program thread(s)
#1
#2
out<#2>.P
out<#1>.Q
in(x).R
channel queue:
contains program fragments
waiting to react
at channel#2 here
worker thread(s)
&mbox<“h”>
deployment buffer:
contains program fragments
ready to be executed/migrated
right now
server thread
network


engine
program
program thread(s)
#1
#2
out<#2>.P
out<#1>.Q
in(x).R
worker thread(s)
&mbox<“h”>
#2<3>
server thread
server thread:
receives fragments from network;
puts them in deployment buffer
network
herbert:#2<3>

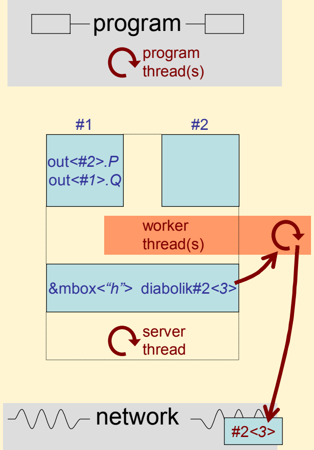

engine
program
program thread(s)
#1
#2
out<#2>.P
out<#1>.Q
in(x).R
worker thread(s)
worker thread:
takes fragment from deployment buffer;
1. maybe can react it immediately, in which case puts continuation back in the deployment buffer
&mbox<“h”>
#2<3>
R{3/x}
server thread
network




worker thread:

takes fragment from deployment buffer;

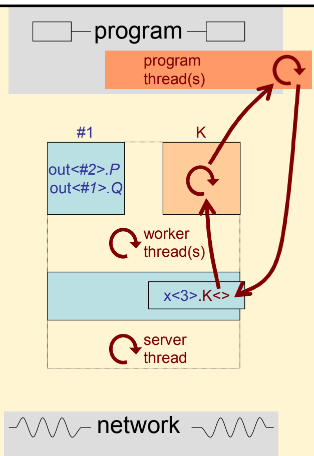
2. maybe it can't react yet, in which case puts fragment in the channel queue




worker thread:

takes fragment from deployment buffer;

3. maybe it's for a different IP, in which case sends it over the network

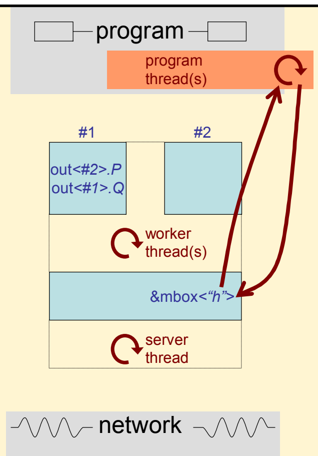



program thread:

can send, receive, create new names, …

for blocking operations, the thread is stored in a dummy *continuation-channel* K:
when the operation completes, this channel is invoked, so waking up the thread

***result:*** a way to integrate C++ code with blocking pi calculus




program thread:

frequently "polls" the engine (currently: a message is sent to the main UI thread)

functions like *mbox* can be executed in that thread's context

***result:*** ease of use for programmers, who do not have to bother making *mbox* thread-safe

## bytecode

$(\mathrm{new}\ r@diabolik.unibo.it)($ // create a fresh channel at this location
$\quad !r(a,b,c).\bar{a}\langle b,c\rangle$ // place a relaying server there
$\quad |\ \bar{r}\langle \&mbox, \text{"hello"}, \text{"world"}\rangle$ // get it to relay this message
$)$

bytecode

```
0  new @1
10 par +40
20 rcv!5(3)
30 snd 6<7,8>
40 nil
50 snd 5<2,3,4>
60 nil
```

stack

```
1  127.9.3.15 : 2793
2  127.9.3.15 : 2793 : 0x400A23
3  "hello"
4  "world"
5
6
7
8
```

IP+port address of the engine running at diabolik

IP+port address of the engine running at diabolik

*r* will go here at line *0*

*a,b,c* will go here at line *20*

## a calculus for the implementation

$$u[\mathsf{out}x.P;\mathsf{in}(y).Q] \xrightarrow{\text{react}} u[P;Q\{x/y\}]$$
$$u[\mathsf{out}x.P;!\mathsf{in}(y).Q] \xrightarrow{\text{react}} u[P;Q\{x/y\};!\mathsf{in}(y).Q]$$
$$u[\overline{v}\,x.P]\ v[\,] \xrightarrow{\text{dep.out}} u[\,]\ v[\mathsf{out}x.P]$$
$$u[(\mathrm{new}\,x)P] \xrightarrow{\text{dep.new}} u[P\{x'/x\}]\ (x')[\,] \quad *$$
$$u[P|Q] \xrightarrow{\text{dep.par}} u[P;Q]$$
$$u[\mathbf{0}] \xrightarrow{\text{dep.nil}} u[\,]$$

$*$ $x'$ fresh, unique

***THEOREM*** $P \sim Q$ iff $u[P] \sim u[Q]$

## the fusion project at bologna

**goal:**

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

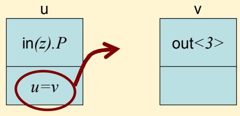
***fusions:***

- 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.

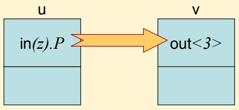





**A fusion is something that allows two names to be used interchangeably**

- 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.

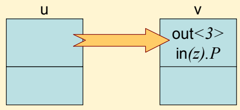





**A fusion is something that allows two names to be used interchangeably**

- 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.






**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.



**The explicit fusion calculus**

$$P \quad ::= \quad x{=}y \quad | \quad \overline{u}\,\widetilde{x}.P \quad | \quad u\,\widetilde{x}.P \quad | \quad P|P \quad | \quad (x)P \quad | \quad \mathbf{0}$$

$$\overline{u}\,\widetilde{x}.P\,|\,u\,\widetilde{y}.Q \quad \longrightarrow \quad \widetilde{x}{=}\widetilde{y}\,|\,P\,|\,Q$$

$$x{=}y\,|\,P \equiv x{=}y\,|\,P\{y/x\} \qquad \textit{substitution}$$

$$(x)(x{=}y) \equiv \mathbf{0} \qquad \textit{local alias}$$

$$x{=}x \equiv \mathbf{0} \qquad \textit{reflexivity}$$

$$x{=}y \equiv y{=}x \qquad \textit{symmetry}$$

$$x{=}y\,|\,y{=}z \equiv x{=}z\,|\,y{=}z \qquad \textit{transitivity}$$



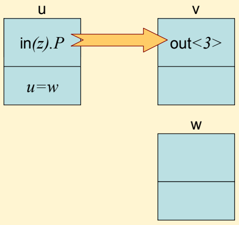
reaction in the pi calculus:

$$\overline{u}\,x.P \mid u(y).Q \quad \to \quad P \mid Q\{x/y\}$$

reaction in the explicit fusion calculus:

$$\begin{array}{lcll}
\overline{u}\,x.P \mid (y)u\,y.Q & \equiv & (y)(\overline{u}\,x.P \mid u\,y.Q) & \textit{assume } y \notin \textit{fnP} \\
 & \to & (y)(x{=}y \mid P \mid Q) & \\
 & \equiv & (y)(P \mid x{=}y \mid Q\{x/y\}) & \textit{substitution due to fusion} \\
 & \equiv & P \mid (y)(x{=}y) \mid Q\{x/y\} & \textit{scope intrusion} \\
 & \equiv & P \mid Q\{x/y\} & \textit{remove local alias}
\end{array}$$

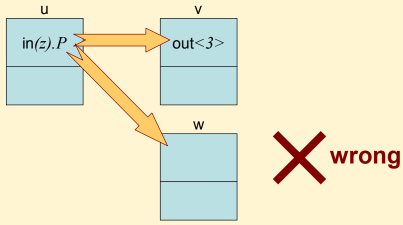
## the fusion clash problem




**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.

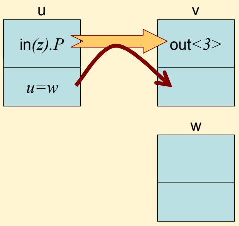
## the fusion clash problem




**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.

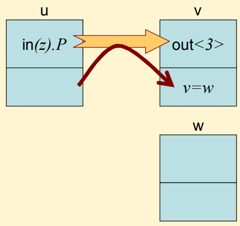
## the fusion clash problem




**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.

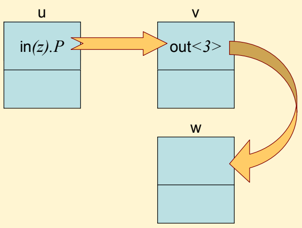
## the fusion clash problem




**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.

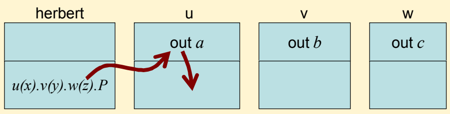
## the fusion clash problem




**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.

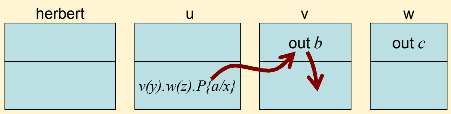
## the continuation problem




**The continuation problem:**

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

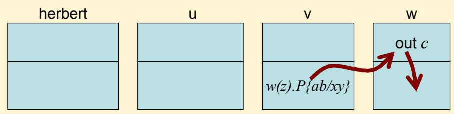
## the continuation problem




**The continuation problem:**

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

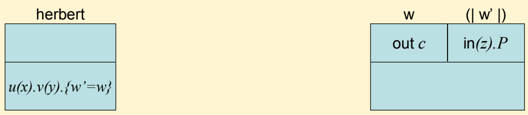
## the continuation problem




**The continuation problem:**

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

## the continuation problem

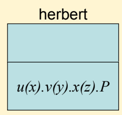



**The continuation problem: (use fusions!)**

- 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

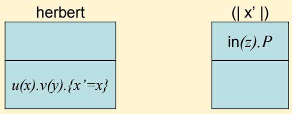



**The input-mobility problem:**

- 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

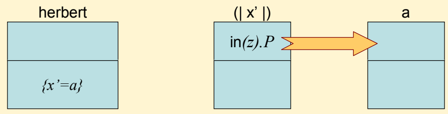



**The input-mobility problem: (use fusions!)**

- 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

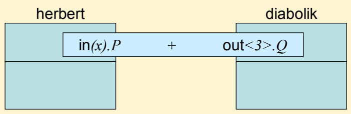



**The input-mobility problem: (use fusions!)**

- 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.

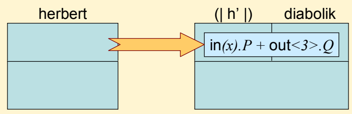
### the choice problem




**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

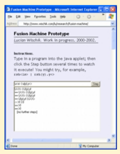
### the fusion question

**Are fusions actually useful?**

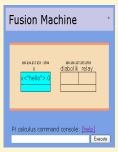
- **yes:** in solving the *continuation problem*
  as a calculus-friendly way of writing pointers.
- **maybe:** as part of an algorithm for *distributed choice*.
- **maybe:** Cosimo thinks to use *false fusions* like 1=2
  as a way to encode failed transactions/compensations.
- **???** Highwire seems 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:

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

$$
\begin{array}{ll}
(\text{new } xyz, v'@v, w'@w)\ ( & \\
\quad ux.\, v'{=}v & //\ \text{after } u \text{ has reacted, it tells} \\
\quad |\ v'y.\, w'{=}w & //\ v' \text{ to fuse to } v, \text{ so allowing} \\
\quad |\ w'z & //\ \text{our } v' \text{ atom to react with } v \text{ atoms} \\
) &
\end{array}
$$

## virtual machine, formally

| | | | |
|---|---|---|---|
| Machines $M$ | $::=$ | $u[B]$ | *channel machine at u* |
| | | $(u)[B]$ | *private channel machine* |
| | | $M, M$ | |
| | | $\mathbf{0}$ | |
| Bodies $B$ | $::=$ | $\mathsf{out}\,\tilde{x}.P$ | *output atom* |
| | | $\mathsf{in}(\tilde{x}).P$ | *input atom* |
| | | $!\mathsf{in}(\tilde{x}).P$ | *repliated input* |
| | | $P$ | *pi process* |
| | | $B; B$ | |

Processes $P$ $::=$ $\overline{u}\tilde{x}.P \;\mid\; [!]u(\tilde{x}).P \;\mid\; (x)P \;\mid\; P|P \;\mid\; \mathbf{0}$

## 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++**

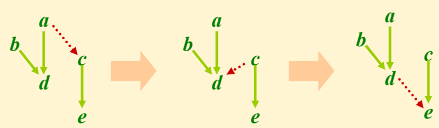
```
{ …        // there's an implicit continuation K after the rcv,
  rcv(x);  // so we stall the thread and put x.K in the work bag.
  …        // When K is invoked, it signals the thread to wake up
}
```

**Calling arbitrary pi code from C++**

```
pi("u!x.v!y | Q");
pi("u!x."+fun_as_chan(&test2)+"|Q");

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

$$u[x{=}y]\ \ x[p{:}\ ]\ \xrightarrow{\text{dep.fu}}\ u[\,]\ \ x[y{:}\ y{=}p]$$

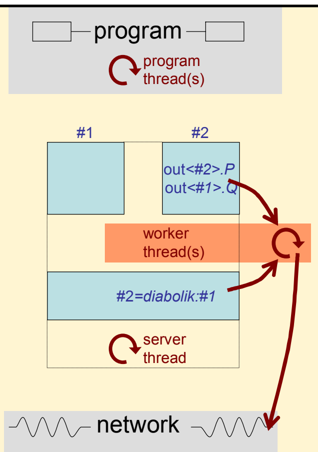
* assuming $x < y$

* 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

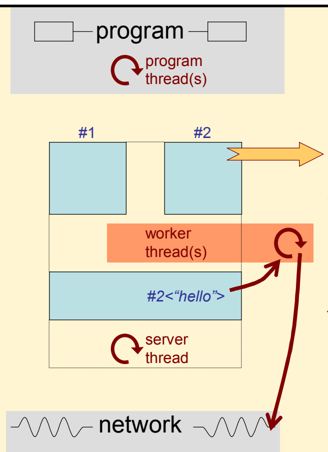



worker thread:

takes fragment from deployment buffer;

if it was a fusion, migrate all the existing contents, and set up a fusion pointer.

## implementing fusions




worker thread:

takes fragment from deployment buffer;

if channel was already fused, then just migrate the fragment immediately