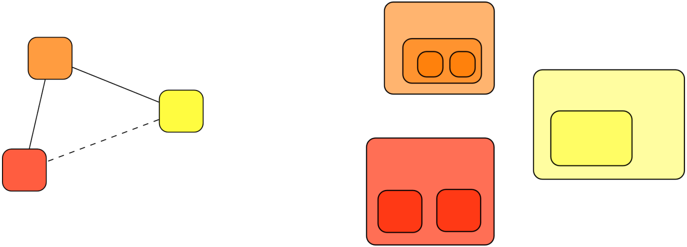
Encapsulation and Dynamic Modularity in the π-calculus

Daniel Hirschkoff, Tom Hirschowitz, Samuel Hym,
Aurélien Pardon and Damien Pous

Oslo, June 7th, 2008

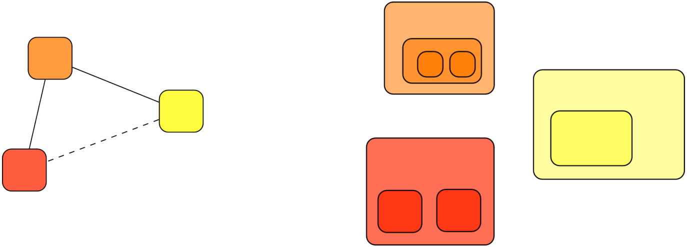
Motivations

- We have good models for
  - concurrency and mobility: π, Dπ, Join;
  - component-based programming: Kells (Fractal).



Motivations

- We have good models for
    - concurrency and mobility: π, Dπ, Join;
    - component-based programming: Kells (Fractal).



- What about the combination of these paradigms?

Extending the π-calculus

$$
\begin{array}{llr}
P ::= 0 \mid P|P \mid (\nu a)P \mid \overline{a}\langle P\rangle.P \mid a(X).P \mid X & & (\mathrm{HO}\pi) \\
\quad \mid a[P] \mid a[X] \triangleright P & & (\mathrm{Kells})
\end{array}
$$

- Global communications, as expected,
- modules ($a[P]$) give a static tree structure,
- passivation, defined by the following rule, makes it possible to manipulate this modular structure at runtime.

$$a[P] \mid a[X] \triangleright Q \quad \rightarrow \quad Q\{P/X\}$$

Extending the π-calculus

$$\begin{array}{lr} P ::= 0 \mid P|P \mid (\nu a)P \mid \overline{a}\langle P\rangle.P \mid a(X).P \mid X & (\mathrm{HO}\pi) \\ \qquad \mid a[P] \mid a[X] \triangleright P & (\mathrm{Kells}) \end{array}$$

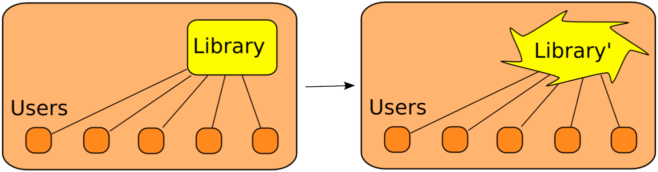
- Global communications, as expected,
- modules ($a[P]$) give a static tree structure,
- passivation, defined by the following rule, makes it possible to manipulate this modular structure at runtime.

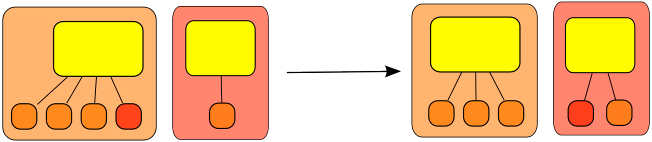
$$a[P] \mid a[X] \triangleright Q \quad \to \quad Q\{P/X\}$$

- Examples:

$$\begin{array}{lclr} a[P] \mid a[X] \triangleright b[X] & \to & b[P] & (\text{rename}) \\ a[P] \mid a[X] \triangleright 0 & \to & 0 & (\text{kill}) \\ a[P] \mid a[X] \triangleright a[X]|a[X] & \to & a[P]|a[P] & (\text{duplicate}) \\ a[P] \mid a[X] \triangleright \overline{b}\langle X\rangle.0 & \to & \overline{b}\langle P\rangle.0 & (\text{send}) \end{array}$$

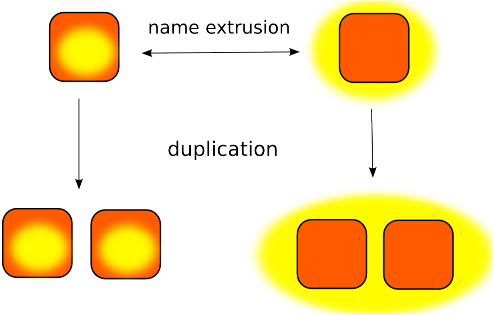
More informal examples





Name extrusion and passivation

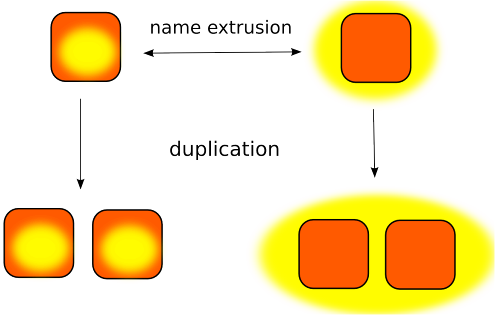
$$m[(\nu a)P] \qquad \equiv \qquad (\nu a)(m[P])$$



$$m[(\nu a)P] \mid m[(\nu a)P] \qquad\qquad (\nu a)(m[P] \mid m[P])$$

Name extrusion and passivation

$$m[(\nu a)P] \qquad \equiv \qquad (\nu a)(m[P])$$



$$m[(\nu a)P] \mid m[(\nu a)P] \qquad\qquad (\nu a)(m[P] \mid m[P])$$

- We chose to forbid name extrusion across module boundaries;
- this restriction ensures a form of encapsulation:
  a module always comes with its own set of private names.

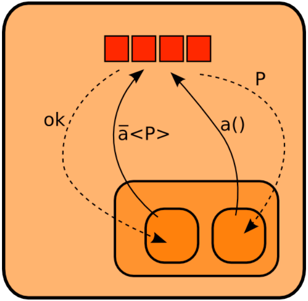
Distributed Implementation

- Implemented in OCaml, specified by a distributed abstract machine.

- Although some modules may run on the same host, we consider each module as an asynchronous entity: we ignore the physical distribution of modules and execute each of them in its own logical location.
- Then, we only use asynchronous messages between locations.

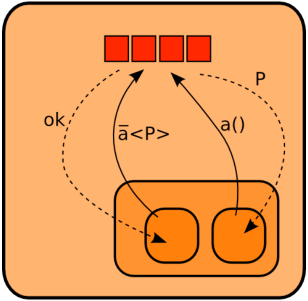
Protocol for communication

- Since a name cannot be extruded out of the module where it was declared, we can install a message queue in the location of that module.
- Then each process wanting to communicate on that name can send requests to the queue, and wait for an answer.



Protocol for communication

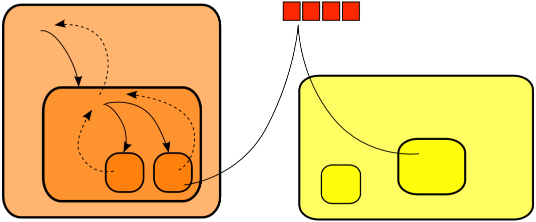
- Since a name cannot be extruded out of the module where it was declared, we can install a message queue in the location of that module.
- Then each process wanting to communicate on that name can send requests to the queue, and wait for an answer.



- There is no problem in the case where the location hosting the queue gets passivated: all clients of that queue will get passivated too!
- We use a type system to prevent illegal scope extrusions.

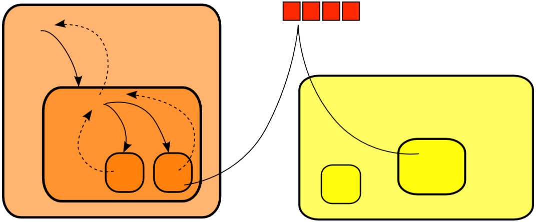
Protocol for passivation

- Passivation is more complex since it may involve a large number of locations.
- We implement it in a incremental way, by walking recursively through the sub-tree to passivate.



Protocol for passivation

- Passivation is more complex since it may involve a large number of locations.
- We implement it in a incremental way, by walking recursively through the sub-tree to passivate.

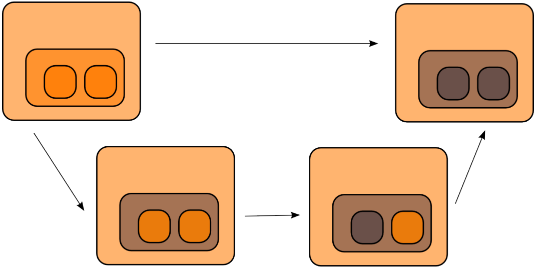


- Since some locations may be waiting for communications to happen, we have to use additional messages in order to cancel the corresponding requests.

Correctness of the abstract machine

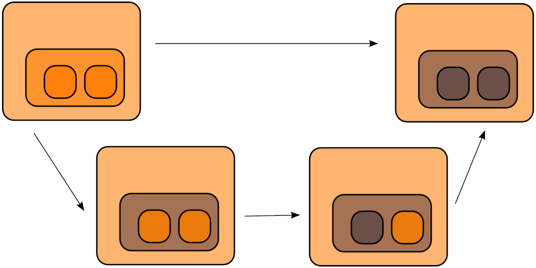
Correctness of the abstract machine

- The abstract machine is not weakly bisimilar to the calculus:



Correctness of the abstract machine

- The abstract machine is not weakly bisimilar to the calculus:



- We have to use coupled bisimilarity.

Future work

- Finish the proof of correctness for the abstract machine.
- Implement the type-checker.
- Introduce optimisations, e.g., for passivation.

- Develop a theory of behavioural equivalences.
- Extend the calculus with primitives for dynamic linking.

Thanks!