Encapsulation and Dynamic Modularity in the $\pi$-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: $\pi$, $D\pi$, Join;
  - component-based programming: Kells (Fractal).
Motivations

- We have good models for
  - concurrency and mobility: $\pi$, $D\pi$, Join;
  - component-based programming: Kells (Fractal).

- What about the combination of these paradigms?
Extending the $\pi$-calculus

$$ P ::= 0 \mid P|P \mid (\nu a)P \mid a\langle P \rangle . P \mid a(X).P \mid X \quad (\text{HO}_{\pi}) $$
$$ \quad \mid a[P] \mid a[X] \triangleright P \quad (\text{Kells}) $$

- 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 $\pi$-calculus

\[
P ::= 0 \mid P \mid P \mid (\nu a)P \mid \overline{a}\langle P \rangle \cdot P \mid a(X).P \mid X \quad \text{(HO$\pi$)}
\]
\[
\mid a[P] \mid a[X] \triangleright P \quad \text{(Kells)}
\]

- 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.

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

- Examples:

\[
a[P] \mid a[X] \triangleright b[X] \quad \rightarrow \quad b[P] \quad \text{(rename)}
\]
\[
a[P] \mid a[X] \triangleright 0 \quad \rightarrow \quad 0 \quad \text{(kill)}
\]
\[
a[P] \mid a[X] \triangleright a[X] \mid a[X] \quad \rightarrow \quad a[P] \mid a[P] \quad \text{(duplicate)}
\]
\[
a[P] \mid a[X] \triangleright \overline{b}\langle X \rangle \cdot 0 \quad \rightarrow \quad \overline{b}\langle P \rangle \cdot 0 \quad \text{(send)}
\]
More informal examples
We chose to forbid name extrusion across module boundaries; this restriction ensures a form of encapsulation by which a module always comes with its own set of private names.
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 an 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 an 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

- The abstract machine is not weakly bisimilar to the calculus.
- We have to use coupled bisimilarity.
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:

  ![Diagram showing examples of non-bisimilar states](image)

- 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!