Mobile Ambients in Common Lisp

Joshua Taylor

December 8, 2009
Mobile Ambients

- Cardelli & Gordon specified a process calculus of mobile ambients for reasoning about mobile processes and devices.
- As my term project, I implemented a library with which mobile ambient processes can be written in Common Lisp.
- (This is much more of a simulator than an concurrent and distributed system. Concurrency could be added without too much trouble; distribution would probably require some significant effort.)
The library and examples from the paper are available online: http://www.cs.rpi.edu/~tayloj/mobile-ambients/.

Mobile Ambients in Common Lisp

Overview

This library provides an API for Cardelli & Gordon's Mobile Ambients. I have tried to develop the library to match as closely as possible with Cardelli & Gordon's presentation, and also to be relatively idiomatic Common Lisp. In addition, the library is extensible, allowing programmers to introduce new types of processes, capabilities, and to define transitions for them.

Documentation

The library has complete documentation (generated by CLDOC).

Download

The current version of the library is a single file that should have no problems compiling and running in any Common Lisp implementation. It has been tested under LispWorks, SBCL, and CCL.
**With Documentation!**

```latex
\textbf{input \textit{name} \&\textit{body} \textit{body}}
```

**Syntax:**

\[
\text{input name [[declaration* | documentation]] form* } \Rightarrow \text{ input}
\]

**Arguments and Values:**

- name—a symbol naming a variable
- declaration—a \texttt{declare} expression; not evaluated
- documentation—a documentation string; not evaluated
- forms—forms
- input—an \texttt{input} process

**Description:**

Input produces an input process whose function has the specified body, and within whose body name is bound to the received input value.

\[
(\text{input } x \ldots) \Rightarrow (\text{make-input #'(lambda (x) \ldots))}
\]

**See Also:**

- \texttt{make-input}
Differences from the Ambient Calculus

- The host language supports recursion and iteration; there is no need for replication processes, e.g., $!P$.

- To execute Common Lisp code, we introduce a new kind of process, currently called `fun`. A `fun` process $F$ evolves by executing Common Lisp code to produce some value $v$. If $v$ is a process, then $F \rightarrow v$, else $F \rightarrow 0$.

- The extension of the basic ambient calculus that adds input and output supports communicating *any* values, not just capabilities and process names.
Dining Philosophers

Semaphores

\[
\text{semaphore } s = s[]
\]

\[
\text{acquire } s.P = \text{open } s.P
\]

\[
\text{release } s = s[]
\]
Semaphores

\[
\text{semaphore } s = s[]
\]

\[
\text{acquire } s.P = \text{open } s.P
\]

\[
\text{release } s = s[]
\]

(defun semaphore (s)
  (amb s))

(defun acquire (s P)
  (open s P))

(defun release (s)
  (amb s))
chopstick \( c \) = semaphore \( c \)
chopstick c = semaphore c

(defun chopstick (c)
  (semaphore c))
\begin{quote}
\textbf{philosopher} l r = !\text{acquire} \ l . \text{acquire} \ r . (\text{release} \ l \mid \text{release} \ r)
\end{quote}
philosopher l r = !acquire l.acquire r.(release l | release r)

(defun philosopher (l r)
  (acquire l
   (acquire r
    (par (release l)
      (release r)
      (fun (philosopher l r))))))

Dining Philosophers

Tables

\[ \text{philosopher } c_1 \ c_2 \mid \text{philosopher } c_2 \ c_1 \mid \text{chopstick } c_1 \mid \text{chopstick } c_2 \]
philosopher c₁ c₂ | philosopher c₂ c₁ | chopstick c₁ | chopstick c₂

(par (philosopher 'c₁ 'c₂)
   (philosopher 'c₂ 'c₁)
   (chopstick 'c₁)
   (chopstick 'c₂))
Most of the Lisp code corresponded directly to the ambient calculus expressions.

We used a `fun` process to keep the philosopher going, rather than a replication.

Full dining philosopher code is available online: http://svn.cs.rpi.edu/svn/tayloj/mobile-ambients/philosophers.lisp
Other features & future work

- Implementation is extensible. For instance, the objective \texttt{mv in}, \texttt{mv out}, \texttt{and acid capabilities can be implemented directly (and are, in examples.lisp).}

- True concurrency, using OS or lightweight threads, is desired.

- In a number of places, Cardelli & Gordon simplify parallel process composition by removing provably inert processes. Some clever tracking of names might make this possible in the current library.
Questions?

Thank You!
(defun firewall ()
  (new (w)
    (amb w (amb 'k (out w (in 'k2 (in w))))
      (open 'k2 (open 'k3 (named 'p)))))))

(defun agent ()
  (amb 'k2 (open 'k (amb 'k3 (named 'Q)))))

> (run-process (par (firewall) (agent)))
; 4. #:W[K2[K3[Q]] | open K2.open K3.P]
; 5. #:W[K3[Q] | open K3.P]
; 6. #:W[Q | P]
#:W[Q | P]
Renaming example

```
(defun be (n m P)
  (par (amb m (out n (open n P))) (in m)))

(run-process
  (par (amb 'n
    (be 'n 'm (named 'P))
    (named 'Q)))

; *. N[(M[out N.open N.P] | in M) | Q]
; 0. N[M[out N.open N.P] | in M | Q]
; 1. M[open N.P] | N[in M | Q]
; 2. M[N[Q] | open N.P]
; 3. M[Q | P]
#<M[Q | P]>
```
(defun see (n P)
  (new (r s)
    (par (amb r (in n (out n (be r s P))))
        (open s))))

(run-process
  (par (amb 'n)
    (see 'n (named 'P))))

; *  N[] | (#:R[in N.out N.(#:S[out #:R.open #:R.P] | in #:S)] | open #:S) 
; 0  N[] | #:R[in N.out N.(#:S[out #:R.open #:R.P] | in #:S)] | open #:S 
; 1  N[#:R[out N.(#:S[out #:R.open #:R.P] | in #:S)] | open #:S 
; 2  #:R[#:S[out #:R.open #:R.P] | in #:S] | N[] | open #:S 
; 3  #:S[open #:R.P] | #:R[in #:S] | N[] | open #:S 
; 4  #:S[#:R[] | open #:R.P] | N[] | open #:S 
; 5  #:S[P] | N[] | open #:S 
; 6  P | N[]
#<P | N[]>