Open Distributed Systems

- Addition of new components.
- Replacement of existing components.
- Changes in interconnections.

Actor Configurations

Model open system components:

- set of individually named actors.
- messages "en-route".
- interface to environment:
  - receptionists
  - external actors
Synchronous vs Asynchronous Communication

- Ti-Calculus (and other process algebras such as CCS, CSP) take synchronous communication as a primitive.

- Actors assume asynchronous communication is more primitive.
Communication Medium

- In π-calculus, channels are explicitly modelled. Multiple processes can share a channel, potentially causing interference.

- In the actor model, the communication medium is not explicit. Actors (active objects) are first-class, history-sensitive entities with an explicit identity used for communication.
FAIRNESS

The actor model theory assumes fair computations:

1. message delivery is guaranteed.
2. individual actor computations are guaranteed to progress.

Fairness is very useful for reasoning about equivalences of actor programs but can be hard/expensive to guarantee; in particular when distribution and failures are considered.
Programming Languages Influenced by IT-Calculus and Actors.

- Scheme '75
- Act1 '87
- Acore '87
- Rosette '89
- Oblig '94
- Erlang '93
- ABCL '90
- SALSA '99
- Amber '86
- Facile '89
- CML '91
- Pict '94
- Nonadic Pict '99
- TOCA ML '99
Actor (Agent) Model

Actor

Thread

Message

Internal variables

Methods

State

Mailbox
1. Extend a functional language (λ-Calculus + tuples + pairs) with actor primitives.

2. Define an operational semantics for actor configurations.

3. Study various notions of equivalence of actor expressions and configurations.

4. Assume fairness:
   - guaranteed message delivery.
   - individual actor progress.
\textbf{\LaTeX-calculus}

\textbf{Syntax}

\[ e ::= \nu \mid \lambda v. e \mid (e e) \]

\textbf{Example}

\[(\lambda x. x) 5 \]

\[ \pi \]

\[ x \{5/x\} \]

\[ [5/x] x \]

\[ \theta \]
pr (x, y) returns a pair containing x & y.

ispr (x) returns true if x is a pair; false otherwise.

1st (pr (x, y)) = x 1st returns
The first value x.

2nd (pr (x, y)) = y 2nd returns
The second value y.
actor primitives

send(a, v)

 sends value v to actor a.

new(b)

creates a new actor with behavior b, and returns the identity/name of the newly created actor.

ready(b)

becomes ready to receive a new message with behavior b.
**Actor Language Example**

\[ b_5 = \text{rec}(\forall y. \exists x. \text{seg}(\text{send}(x, 5), \text{ready}(y))) \]

receives an actor name \(x\) and sends the number 5 to that actor, then it becomes ready to process new messages with the same behavior \(y\).

**Sample Usage**

\[ \text{send}(\text{new}(b_5), a) \]

**A Sink**

\[ \text{sink} = \text{rec}(\forall b. \forall m. \text{ready}(b)) \]

an actor that disregards all messages.
\texttt{cell = rec \{a\ b\ c\ \lambda m.}
\par
\hspace{1cm} \texttt{if (get\?(m),}
\hspace{1cm} \texttt{seg \{send \{cust \(m\), \(c\),}
\hspace{2cm} \texttt{ready \(b\(c\))\}},}
\hspace{1cm} \texttt{if (set\?(m),}
\hspace{2cm} \texttt{ready \(b\\text{\(contents\(m\))},}
\hspace{3cm} \texttt{ready \(b\(c\)))\})\}}
\par
\textbf{Using the cell:}
\texttt{let \(a\) = new\(\text{cell\(b\)})\) in}
\par
\hspace{1cm} \texttt{seg \{send \(a, mkset\(3\))\},}
\hspace{1cm} \texttt{send \(a, mkset\(2\))\},}
\hspace{1cm} \texttt{send \(a, mkset\(c\))\})}
JOIN CONTINUATIONS

Consider:

\[
tree\prod = \text{rec}(\lambda f. A\text{tree}.
    \text{if} (\text{isnat}(\text{tree}),
    \text{tree},
    f(\text{left}(\text{tree})) \ast f(\text{right}(\text{tree})))
)
\]

which multiplies all leaves of a tree, which are numbers.

You can do the "left" and "right" computations concurrently.
TREC PRODUCT BEHAVIOR

\[
B_{\text{treeprod}} = \text{rec}(\lambda \text{b}. \lambda \text{self}. \lambda \text{m}.
\begin{align*}
&\text{seg} (\text{become} (\text{b} (\text{self})),
\text{if} (\text{isnat} (\text{tree} (\text{m})),
\text{send} (\text{cust} (\text{m}), \text{tree} (\text{m})),
\text{refactor} \{ \text{newcust} := B_{\text{joincust}} \}
\text{seg} (\text{send} (\text{self},
\text{pr} (\text{left} (\text{tree} (\text{m})), \text{newcust})),
\text{send} (\text{self},
\text{pr} (\text{right} (\text{tree} (\text{m})), \text{newcust}))))
\end{align*}
)\]
)
\[ B_{\text{joincont}} = \]
\[ \text{rec}(\lambda a.v, \lambda n.a, \lambda \text{firstnum}, \lambda n.v \leftarrow \lambda v) \]
\[ \text{if (eq (nargs, 0),} \]
\[ \text{become (b (v, 1, n.v)),} \]
\[ \text{seg (become (sink),} \]
\[ \text{send (v, firstnum + n.v))}) \]
SAMPLE EXECUTION

(a) 
\( f(\text{tree, cust}) \)

(b) 
\( f(\text{left(tree), JC}) \)
\( f(\text{right(tree), JC}) \)
Dining Philosophers in Actor Language

\[ \text{phil} = \text{rec} \{ \lambda b. \lambda l. \lambda r. \lambda \text{self}. \lambda \text{sticks}. \lambda \text{m}. \]

\[ \text{if } (\text{eq?} \ (\text{sticks}, \ 0), \]

\[ \text{ready } (\text{b}(l, r, \text{self}, 1)), \]

\[ \text{seg } (\text{send } (l, \text{mkrelease } (\text{self})), \]

\[ \text{send } (r, \text{mkrelease } (\text{self})), \]

\[ \text{send } (l, \text{mkpickup } (\text{self})), \]

\[ \text{send } (r, \text{mkpickup } (\text{self})), \]

\[ \text{ready } (\text{b}(l, r, \text{self}, 0))))))) \]
chopstick = rec ( ηb. ηh. ηw. ηm.
    if (pickup?(m),
        if (eq?(h, nil),
            seg (send (getphil (m), nil),
                 ready (b(getphil (m), nil))),
            ready (b(h, getphil(m)) ))),
        if (release?(m),
            if (eq?(w, nil),
                ready (b(nil, nil)),
                seg (send (w, nil),
                     ready (b(w, nil))),
                ready (b(h, w))))))
Dining Philosophers in Actor Lang. (2)

\[ \text{letrec } c_1 = \text{new}(\text{chopstick}(\text{nil}, \text{nil})), \]
\[ c_2 = \text{new}(\text{chopstick}(\text{nil}, \text{nil})), \]
\[ p_1 = \text{new}(\text{phil}(c_1, c_2, p_1, 0)), \]
\[ p_2 = \text{new}(\text{phil}(c_2, c_1, p_2, 0)) \text{ in } e \]

where \( e \) is defined as:

\[ e = \text{seq} (\text{send}(c_1, \text{mkpickup}(p_1)), \]
\[ \text{send}(c_2, \text{mkpickup}(p_1)), \]
\[ \text{send}(c_1, \text{mkpickup}(p_2)), \]
\[ \text{send}(c_2, \text{mkpickup}(p_2))) \]
Auxiliary definitions:

\[ mk\text{pickup} = \lambda p. \neg p \]
\[ mk\text{release} = \text{nil} \]
\[ \text{pickup?} = \lambda m. \neg \text{eg?(m, \text{nil})} \]
\[ \text{release?} = \lambda m. \text{eg?(m, \text{nil})} \]
\[ \text{getphid} = \lambda m. m \]
\( x = \ll \alpha \mid \mu \rr \sigma \)

\( \alpha \) is a function mapping actor names (represented as variables) to actor states.

\( \mu \) is a multi-set of messages "en-route".

\( \sigma \) is a set of receptionists.

\( x \) is a set of external actors.

**Given** \( A = \text{Dom}(\alpha) \):

- \( \emptyset \subseteq A \), \( A \cap x = \emptyset \)
- if \( \alpha(a) = (?a') \), then \( a' \in A \)
- if \( a \in A \), then \( \text{FV}(\alpha(a)) \subseteq \text{AUX} \)
- if \( \langle v_0 := v \rangle \in \mu \), \( \text{FV}(v_0, v) \subseteq \text{AUX} \).
\( \alpha \in \mathcal{A} \rightarrow \mathcal{A} \)

\( \mathcal{A}s = (\exists \alpha) \cup (\forall) \cup [E] \)

\( m \in M[w] \)

\( m = <v \leq v> \)

\( p, x \in P[w] \)

\( v = \text{At} \cup \times \cup \lambda x. \mathcal{E} \cup \text{pr}(v, v) \)

\( \mathcal{E} = v \cup \text{app}(E, E) \cup F_n(E^o) \)
**LABELLED TRANSITION RELATION** (\(\rightarrow\))

\[
\langle \text{fun: } a \rangle
\]
\[
e \xrightarrow{\lambda \text{Dom}(a) \cup \{a\}} e' \quad \Rightarrow
\]
\[
\langle \alpha, [e]_a | \mu \rangle_x \xrightarrow{p} \langle \alpha, [e']_a | \mu \rangle_x
\]

\[
\langle \text{newactor: } a, a' \rangle
\]
\[
\langle \alpha, [R[\text{newactor}(e)]]_a | \mu \rangle_x \xrightarrow{p} \langle \alpha, [R[a']]_a, [e]_a' | \mu \rangle_x \quad a' \text{ fresh}
\]

\[
\langle \text{send: } a, v_0, v_i \rangle
\]
\[
\langle \alpha, [R[\text{send}(v_0, v_i)]]_a | \mu \rangle_x \xrightarrow{p} \langle \alpha, [R[nil]]_a | \mu, <v_0 \leftarrow v_i> \rangle_x
\]
{}labeled transition relation \( (\rightarrow) \) continued

\[
\langle \text{receive: } v_0, v_1 \rangle
\]

\[
\langle \alpha, [\text{ready}(v)]_{v_0} \mid \langle v_0 \leq v_1 \rangle, \mu \rangle^p_x \rightarrow
\]

\[
\langle \alpha, [\text{app}(v, v_i)]_{v_0} \mid \mu \rangle^p_x
\]

\[
\langle \text{out: } v_0, v_1 \rangle
\]

\[
\langle \alpha | \mu, \langle v_0 \leq v_1 \rangle \rangle^p_x \rightarrow \langle \alpha | \mu \rangle^p_x
\]

if \( v_0 \in X \) and \( p' = p \cup (\text{FV}(v_i) \cap \text{Dom}(\alpha)) \)

\[
\langle \text{in: } v_0, v_1 \rangle
\]

\[
\langle \alpha | \mu \rangle^p_x \rightarrow \langle \alpha | \mu, \langle v_0 \leq v_1 \rangle \rangle^p_{x'}
\]

if \( v_0 \in p \) and \( \text{FV}(v_i) \cap \text{Dom}(\alpha) \subseteq p \)

and \( x' = x \cup (\text{FV}(v_i) - \text{Dom}(\alpha)) \)
Actor Garbage Collection

Diagram with nodes and arrows representing the structure of the system.
Exercises

0 Write get?
cost
set?
contents
mkset
mkget
to complete the reference cell example in the AMST actor language.

2 Modify Bcell to notify a customer when the cell value is updated (such as in the IT-calculus cell example).