Common Lisp Actor System

Naveen Sundar G

February 20, 2010

Contents

1 Introduction

This document describes a simple to use Actor extension for Common Lisp. The Actor Model is a concurrent model of computation in which *actors*, which are independent computational processes, are the basic units of computation. Computation proceeds by message passing across actors. For more on the Actor model of computation please see [A+86, Var10]. In the `CLAS` extension for Common Lisp each actor get its own thread of execution which can be reused. My implementation has been inspired by the SALSA Actor-based distributed programming language:

`http://wcl.cs.rpi.edu/salsa/`

`CLAS` has been tested on examples found in [Var10]. The current state of the software is experimental, but I hope to improve it and add distrubtion and mobility. The code for these examples can be found in the source distribution. This manual contains a brief tutorial, an explanation of the syntax of `CLAS` , shows how to change the behavior of actors dynamically and explains the examples to some extent.

1.1 Prerequisites

`CLAS` requires the *Bordeaux-Threads* package to work. The package along with instructions for installation can be found at:

`http://common-lisp.net/project/bordeaux-threads/`.

2 Overview

- Creating an actor class or template is similar to defining a function.

```
(defactor Actor-Class
          (state)
          (message-vars)
          behavior next)
```

- Creating an actor instance is done using the following syntax. This can be thought of as coarsely equivalent to the **new** operator in Java.

```
(setq my-actor (Actor-Class (:state-var_1 value_1 ...
                    :state-var_n value_n)))
```

- Sending a message

```
(send my-actor message_arg1 ... message_argn)
```

3 Tutorial

3.1 Stateless Actors

A simple printer actor class with no state can be created as :

```
(defactor printer () (val) (pr val) next)
```

A new instance is initialized as :

```
(setf print-actor (printer))
```

An asynchronous message can be sent to this actor as :

```
(send print-actor "hello,world")
;"hello,world"
```

3.2 Actors with State

A simple reference cell actor class can be created as follows :

```
(defactor cell (c) (message)
  (cond ((get? message) (send (cust message) c))
        ((set? message) (setf c (contents message))))
next)
```

c holds the state of the actor.

A new instance with value -1 is initialized as :

```
(setf cell-actor (cell :c -1))
```

An asynchronous message can be sent to this actor as :

```
(send cell-actor (make-get print-actor)
;-1
```

The value of the cell is sent to the print actor which then prints it. See section 6.2 for a full implementation.

3.3 Message passing

The code below typed into the repl illustrates use of default state variables and the message sending syntax which is similar to function calls in Lisp. We create an actor class `linear` which takes the linear combination of its message components x , y with its state variables a and b , $ax + by$ and sends it to `cust` actor. Note we do not set the state variable b, and therefore it assumes its default value 1.

```
CL-USER> (defactor linear (a (b 1)) (x y cust)
                    (send cust (+ (* a  x) (* b y))))
;LINEAR
CL-USER> (setf lin-actor (linear :a 2 ))
;(#<MULTIPROCESSING:PROCESS Anonymous thread(10) @ #x10753d8a>
;#<Interpreted Closure (LABELS MAKE-ACTOR ADD) @ #x1075381a> NIL)
CL-USER> (send lin-actor 1 2 print-actor)
;4
```

4 Syntax

The syntax of `CLAS` uses LISP s-expressions. The main macros, functions and keywords of `CLAS` are summarized in this section.

- **behav** : macro
  arguments : [state message &body body]
  Create a behavior that has state `state`, message format `message` and behavior specified by `body`

- **defactor** : macro
  arguments : [name state message &body body]
  Create an actor named `name` that has state `state`, message format `message` and behavior specified by `body`

- **send**: function
  arguments : [actor message]
  Send actor `actor` message `message`.

- **self**: constant keyword
  Refers to own actor.

- **next**: constant keyword
  Used to change behavior dynamically. Set by default to current behavior.

- **me**: constant keyword
  Refers to own behavior. This is meant to be an easy replacement of Y-combinators. Not to be used explicitly by the user.

- **make-actor** : function
  arguments : [behav]
  Creates an actor and returns a reference to the actor with behavior specified by `behav`. Not supposed to be used by the CLAS user.

5 Dynamic Change of Behavior

`CLAS` supports dynamic behavior by changing the behavior λ itself as formulated in the Actor Calculus. For example a cell actor is changed to a print actor as follows.

```
(send cell-actor (make-set -1) :next (behav () (val) (pr val) next))
```

Behavior change is real behavior change and not a change of state variables as can be seen from above.

6 Examples

This section discusses examples from the Chapter 2 of [Var10] implemented in `CLAS`
.

6.1 Join Continuation

Lines $4-7$ in the listing 6.1 below contain the class template for a join continuation actor which becomes sink at the end of computation.

Join Continuation for Tree Product

```
1  (defun left (tree) (first tree))
2  (defun right (tree) (second tree))
3  ;Join
4  (defactor joincont (customer (firstnum nil)) (message)
5    (if (null firstnum)
6        (progn (setf firstnum message) next)
7        (progn (send customer (* firstnum message)) #'sink)))
8
9  (defactor treeprod () (customer tree)
10   (if (atom tree)
11       (send customer tree)
12       (let  ((newcust (joincont :customer customer))
13              (lp (treeprod))
14              (rp (treeprod)))
15         (send lp newcust (left tree))
16         (send rp newcust (right tree))))
17   next)
```

Create a new tree product actor

```
(setf tree-actor (treeprod))
```

Send message to the tree product actor, and send the result back to a printing actor.

```
(send tree-actor print-actor '((2 3) ((4 5) (10 2))))
2400
```

6.2 Reference Cell

A simple reference cell actor class can be built as shown below:

Reference Cell

```
(defactor cell (c) ( message)
  (cond ((get? message) (send (cust message)  c))
        ((set? message) (setf c (contents message))))
next)

(defun make-set (val) `("set" ,val))
(defun make-get (cust) `("get" ,cust))

(defun get? (message) (equal (first message) "get"))
(defun set? (message) (equal (first message) "set"))

(defun cust (message) (second message))
(defun contents (message) (second message))
```

A new instance with value 7 is initialized as :

```
(setf cell-actor (cell :c 0))
```

Messages can be sent as follows:

```
(let ((a (cell :c 0)))
          (send a (make-set 7))
          (send a (make-set 2))
          (send a (make-get print-actor)))
```

6.3 Mutual Exclusion

A semaphore actor class and a customer actor class are shown in listing 6.3

Mutual Exclusion

```
(defactor sem ((h nil)) (m)
  (if (get? m)
      (if (null h)
          (progn (send (cust m) t)
                 (setf h (cust m))
                 next)
          (progn (send (cust m) nil)
                 next))
      (if (release? m)
          (progn (setf h nil) next)
          next)))

(defactor customer (semaphore) (m)
  (if m
      (progn (pr "Critical code")
             (send semaphore (make-release)))
      (send semaphore (make-get self)))
  next)

(defun make-release () '("release" ))
(defun make-get (cust) '("get" ,cust))

(defun get? (message) (equal (first message) "get"))
(defun release? (message) (equal (first message) "release"))

(defun cust (message) (second message))
```

Two actors trying to enter a critical region can be modeled as

```
(let ((s (sem))
                (a1 (customer :semaphore s))
                (a2 (customer :semaphore s)))
           (send a1 nil)
           (send a2 nil))
```

6.4 Dining Philosophers

The philosopher actor class and the chopstick actor class are show in listing 6.4

Mutual Exclusion

```
(defactor phil (l r (sticks 0)) (m)
  (if (zerop sticks)
      (progn
        (pr "Got one")
        (incf sticks) next)
      (progn
        (pr "Got both")
        (send l (make-drop self)) (send r (make-drop self))
        (pr "Dropped")
        (send l (make-pick self)) (send r (make-pick self))
        (setf sticks 0) next)))

(defactor chopstick ((h nil) (w nil)) (m)
  (if (pick? m)
      (if (null h)
          (progn (send (get-phil m) nil)
                 (setf h (get-phil m) w nil)
                 next)
          (progn (setf w (get-phil m))
                 next)))
  (if (drop? m)
      (if (null w)
          (progn (setf w nil h nil)
                 next)
          (progn (send w nil)
                 (setf h w w nil))))
  next)

(defun make-drop (cust) `("drop" ,cust))
(defun make-pick (cust) `("pick" ,cust))
(defun pick? (x) (equal (first x) "pick"))
(defun drop? (x) (equal (first x) "drop"))
(defun get-phil (m) (second m))
```

New philosophers and chopsticks are created as follows

```
(let* ((c1 (chopstick))
               (c2 (chopstick))
               (p1 (phil :l c1 :r c2))
               (p2 (phil :l c2 :r c1)))
          (send c1 (make-pick p1))
```

```
(send c2 (make-pick p1))
(send c1 (make-pick p2))
(send c2 (make-pick p2)))
```

7 More Resources

The CLAS system can be downloaded from GitHub:

`http://github.com/naveensundarg/Common-Lisp-Actors`

Miscellaneous stuff:

`http://www.cs.rpi.edu/~govinn/clas.html`

This manual can be found at :

`http://www.cs.rpi.edu/~govinn/actors.pdf`

8 Future Work

I plan to implement distribution and mobility of actors in CLAS . The only major issue, that I foresee, is serialization of Common Lisp objects, but this can be overcome by explicitly stating which objects *are* serializable.

References

[A+86] G.A. Agha et al. *Actors: a model of concurrent computation in distributed systems*. MIT press Cambridge, MA, 1986.

[Var10] C. Varela. *Distributed Computing: A Foundational Approach*. MIT Press, 2010.