A Tutorial Introduction to Message Passing (Examples)

K. V. S. Prasad
(for the course TDA384/DIT391)
Department of Computer Science
Chalmers University

September 25, 2017

1 Examples

The program `SynchSem.pml` shows how to use a semaphore as a process communicating by synchronous messages.

Erlang does not have channels; nor does it have synchronous communication. The program `AsynchSem.pml` shows how to use a semaphore as a process communicating by asynchronous messages. The user requests a token, saying where it is to be sent (their address). Otherwise, another user who has made a request may take the token. The return of the token can happen on a common return channel.

The program `DiningPhil.pml` has 10 channels, each connecting exactly one fork to one philosopher. So the forks and philosophers can be Erlang processes, addressing each other in the usual way (the channels aren't shared between more than 2 processes). But the program won't work if we make the channels asynchronous. For that, try remaking each fork to look like the asynchronous semaphore—it is requested by the philosopher to its left or right and is granted to the first requester.

Sum-series is a simpleprogram to do 1+2+3+...N. The main point being illustrated here is that channels can be used to store shared integers and arrays. The barrier synchronisation, sorting, and the sieve of Eratosthenes programs all work for both synchronous and asynchronous channels. Try them in Erlang.

```
#define USERS 10

chan s = [0] of {bool};
chan w = [0] of {bool};
int CS = 0;

proctype Sem(){
    bool dum;
    do
    :: s ? dum;
       w ? dum
    od
}

proctype U(int i){
    do
    :: s ! true;
       printf("User %d enters CS\n", i);
       CS++;
       assert (CS <= 1);
       CS--;
       printf("User %d leaves CS\n", i);
       w ! true
    od
}

init{
    int i=0;
    do
    :: i > USERS-1 -> break;
    :: else -> run U(i); i++
    od;
    run Sem()
}
```

Figure 1: SynchSem.pml

```
#define USERS 10

chan serv = [1] of {int};  //request token
chan tok[USERS] = [1] of {bool}; //give token
chan ret = [1] of {bool}; //return token
int CS = 0;

proctype Sem(){
    int i; bool dum;
    do
    :: serv ? i;
       tok[i]! dum;
       ret? dum;
    od
}

proctype U(int i){
    bool dum;
    do
    :: serv ! i;
       tok[i]? dum;
       printf("User %d enters CS\n", i);
       CS++;
       assert (CS <= 1);
       CS--;
       printf("User %d leaves CS\n", i);
       ret ! true
    od
}

init{
    int i=0;
    do
    :: i > USERS-1 -> break;
    :: else -> run U(i); i++
    od;
    run Sem()
}
```

Figure 2: `AsynchSem.pml`

```
/* Dining philosophers - one pair of processes per channel */
/* Initialize to asymmetric configuration */

chan forkl[5] = [0] of { bool };  chan forkr[5] = [0] of { bool };
byte numEating = 0;

proctype Phil(byte n; chan left; chan right ) {
do
::  left ! true;
    right ! true;
    numEating++;
    printf("MSC: %d eating, total = %d\n", n, numEating);
    assert (numEating <= 2);
    numEating--;
    right ? _;
    left ? _;
od
}

proctype Fork(chan left; chan right) {
   do
   :: left  ? _; left  ! true
   :: right ? _; right ! true
   od
}

init {
atomic {
  run Fork(forkl[0], forkr[0]);
  run Fork(forkl[1], forkr[1]);
  run Fork(forkl[2], forkr[2]);
  run Fork(forkl[3], forkr[3]);
  run Fork(forkl[4], forkr[4]);
  run Phil(0, forkr[0], forkl[1]);
  run Phil(1, forkr[1], forkl[2]);
  run Phil(2, forkr[2], forkl[3]);
  run Phil(3, forkr[3], forkl[4]);
  run Phil(4, forkl[0], forkr[4]);
}
}
```

Figure 3: DiningPhil.pml

```
#define COUNT 10
#define TEAM 65
chan space = [COUNT] of {int} ;
chan pop = [1] of {int} ;

proctype Worker() {
    int n1, n2, c, curwin;
    do
    ::  pop? c;
        if
        :: c > 1 -> space ? n1;
                    space ? n2;
                    atomic{ pop ! c-1;
                            space ! (n1 + n2)
                    }
        :: else ->  space ? n1; printf ("\n Sum = %d\n", n1)
        fi
    od
}

init {
     pop!COUNT;
     int i = 1;
     do
     :: i>COUNT -> break;
     :: else -> space ! i;
                 i++
     od;
     i = 1;
     do
     :: i>TEAM -> break;
     :: else -> run Worker();
                i++
     od
    }
}
```

Figure 4: sum-series.pml

```
/*  Barrier synchronisation expressed in Promela.
The boss has COUNT workers, and 4 tasks to deliver.  Each worker
completes their part of task i before the system does task (i+1).  */

#define COUNT 10
chan start[COUNT] = [0] of {bool}; //COUNT start chans of capacity 0.
chan done = [COUNT] of {bool}; // One done chan. The bool is a dummy.

proctype Worker(int i) {
    bool temp=true;
    do
      ::  start[i]? temp;
          k=0;
          do             //some dummy computation to allow interleaving
          :: k>1000 -> break;
          :: else -> k++
          od;
          done ! temp
          printf ("\n worker %d here\n", i);
    od
}

proctype Boss() {
    bool temp=true;  int task = 1, i;
    do
    :: task > 4 -> break
    :: else ->     i=0;
                   do
                   :: i>COUNT-1 -> break;
                   :: else -> start[i]! temp;   i++
                   od;
                   printf ("\n Task %d started\n", task);
                   i=0;
                   do
                   :: i>COUNT-1 -> break;
                   :: else -> done? temp;  i++
                   od;
                   printf ("\n Task %d done\n", task); task++
    od
}

init { int i=0;
        atomic{
        do
        :: i>COUNT-1 -> break;
        :: else -> run Worker(i); i++
        od;
        run Boss()
        };
}
```

Figure 5: barrier-synch.pml

```
#define CAP 1

chan qinit = [CAP] of { int };

proctype Ints(chan qin) {
    qin ! 34;
    qin ! 76;
    qin ! 23;
    qin ! 52;
    qin ! 3;
    qin ! 7;
    qin ! 3;
    qin ! 34;
    qin ! 35;
    qin ! 0;
}

proctype End(chan qend) {
    int n;
    chan q = [CAP] of { int };

    qend ? n;
    if
    :: n==0 ->  skip
    :: else ->  run Cell (n, qend, q);
                run End (q)
     fi
}

proctype Cell(int c; chan qin, qout) {
    int n;
    chan q = [CAP] of { int };
    qin ? n;
    if
    :: n==0  -> printf("%d\n", c);  qout ! 0
    :: n > c -> qout ! n;
                run Cell (c, qin, qout)
    :: else ->  run Cell (n, qin, q);
                run Cell (c, q, qout)
    fi
}

init{
    run Ints (qinit);
    run End  (qinit)
}
```

Figure 6: chan-sort.pml

```
/* Sieve, copyright K. V. S. Prasad 2015 */
#define MAX 100

proctype Generator (chan qout) {
    int n = 2;
    do
     :: n < MAX -> qout!n; n++;
     :: else -> break
    od;
    printf("Gen done\n")
}

proctype Filter (chan qin, qout; int p) {
    int N;
    do
    :: qin?N;
       if
       :: N % p != 0 -> qout!N
       :: else -> skip
       fi
    od
}

proctype Sift(chan qin) {
    int p;
    chan qout = [0] of {int };

    qin?p;
    printf("%d is a prime\n", p);
    run Filter(qin,qout,p);
    run Sift(qout)
}

init{
    chan Q = [0] of {int};
    run Generator(Q);
    run Sift(Q)
}
```

Figure 7: `sieve-eratosthenes.pml`