

Concurrency control abstractions in Erlang (CPE 5*)

Carlos Varela
Rennselaer Polytechnic Institute

October 13, 2026

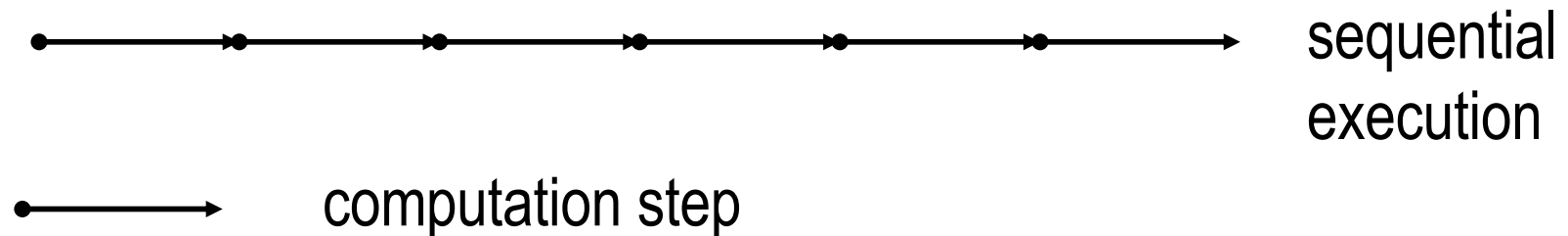
* Concurrent Programming in Erlang, by J. Armstrong, R. Viriding, C. Wikström, M. Williams

Causal order

- In a sequential program all execution states are totally ordered
- In a concurrent program all execution states of a given actor are totally ordered
- The execution state of the concurrent program as a whole is partially ordered

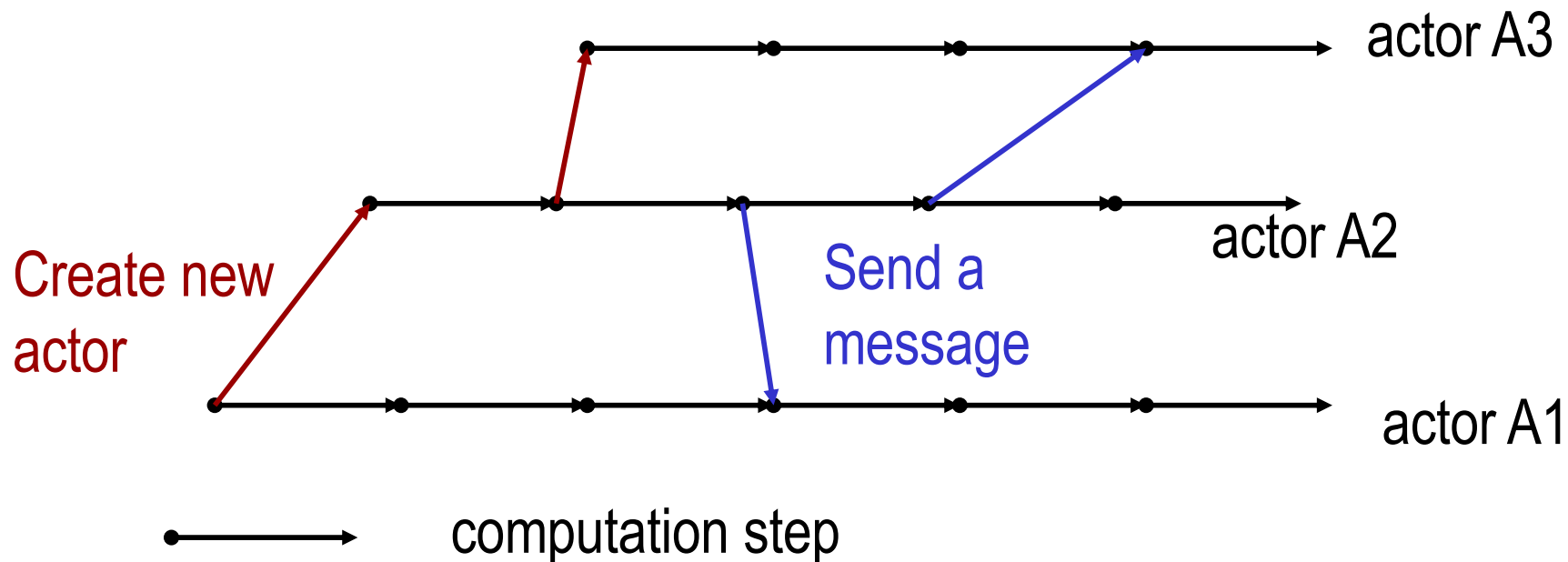
Total order

- In a sequential program all execution states are totally ordered



Causal order in the actor model

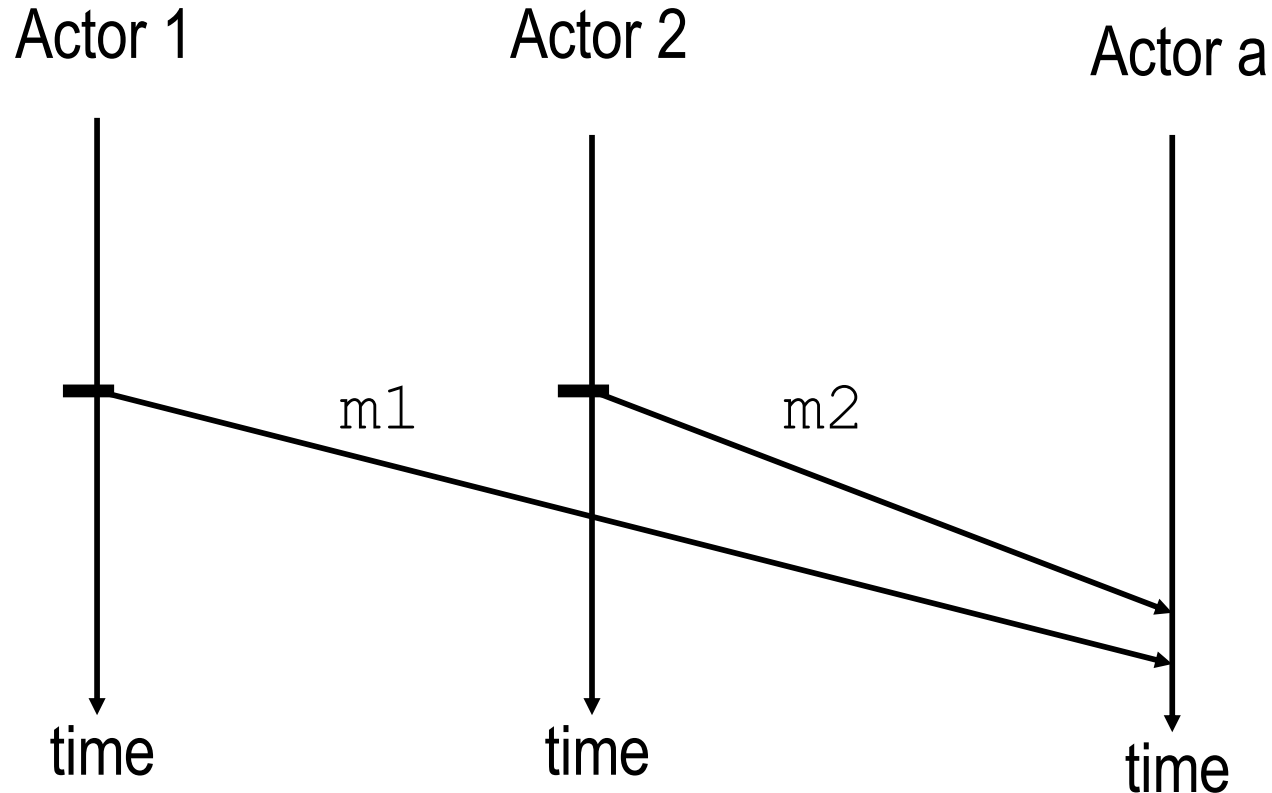
- In a concurrent program all execution states of a given actor are totally ordered
- The execution state of the concurrent program is partially ordered



Nondeterminism

- An execution is nondeterministic if there is a computation step in which there is a choice what to do next
- Nondeterminism appears naturally when there is asynchronous message passing
 - Messages can arrive or be processed in an order different from the sending order.

Example of nondeterminism



Actor a can receive messages $m1$ and $m2$ in any order.

Concurrency control in Erlang

- Erlang uses a *selective receive* mechanism to help coordinate concurrent activities:
 - Message patterns and guards
 - To select the next message (from possibly many) to execute.
 - To receive messages from a specific process (actor).
 - To receive messages of a specific kind (pattern).
 - Timeouts
 - To enable default activities to fire in the absence of messages (following certain patterns).
 - To create timers.
 - Zero timeouts (`after 0`)
 - To implement priority messages, to flush a mailbox.

Selective Receive

`receive`

`MessagePattern1 [when Guard1] ->`

`Actions1 ;`

`MessagePattern2 [when Guard2] ->`

`Actions2 ;`

`...`

`end`

`receive` suspends until a message in the actor's mailbox matches any of the patterns including optional guards.

- Patterns are tried in order. On a match, the message is removed from the mailbox and the corresponding pattern's actions are executed.
- When a message does not match any of the patterns, it is left in the mailbox for future `receive` actions.

Selective Receive Example

Example program and mailbox (head at top):

```
receive
  msg_b -> ...
end
```

msg_a
msg_b
msg_c

`receive` tries to match `msg_a` and fails. `msg_b` can be matched, so it is processed. Suppose execution continues:

```
receive
  msg_c -> ...
  msg_a -> ...
end
```

msg_a
msg_c

The next message to be processed is `msg_a` since it is the next in the mailbox and it matches the 2nd pattern.

Receiving from a specific actor

```
Actor ! {self(), message}
```

`self()` is a Built-In-Function (BIF) that returns the current (executing) process id (actor name). Ids can be part of a message.

```
receive
  {ActorName, Msg} when ActorName == A1 ->
    ...
end
```

`receive` can then select only messages that come from a specific actor, in this example, `A1`. (Or other actors that know `A1`'s actor name.)

Receiving a specific kind of message

```
counter(Val) ->
  receive
    increment -> counter(Val+1);
    {From,get} ->
      From ! {self(), Val},
      counter(Val);
    stop -> true;
    Other -> counter(Val)
  end.
```

**increment is an atom
whereas Other is a
variable (that matches
anything!).**

counter is a behavior that can receive increment messages, get request messages, and stop messages. Other message kinds are ignored.

Order of message patterns matters

receive

```
{Left, Right}, Customer} ->  
    NewCust = spawn(treeprod, join, [Customer]),  
    LP = spawn(treeprod, treeprod, []),  
    RP = spawn(treeprod, treeprod, []),  
    LP!{Left, NewCust},  
    RP!{Right, NewCust};  
{Number, Customer} ->  
    Customer ! Number
```

end

{Left, Right} is a more specific pattern than Number is (which matches anything!). Order of patterns is important.

In this example, a binary tree is represented as a tuple

{Left, Right}, or as a Number, e.g.,

{{{5, 6}, 2}, {3, 4}}

Selective Receive with Timeout

```
receive
  MessagePattern1 [when Guard1] ->
    Actions1 ;
  MessagePattern2 [when Guard2] ->
    Actions2 ;
  ...
  after TimeOutExpr ->
    ActionsT
end
```

`TimeOutExpr` evaluates to an integer interpreted as *milliseconds*.

If no message has been selected within this time, the timeout occurs and `ActionsT` are scheduled for evaluation.

A timeout of `infinity` means to wait indefinitely.

Timer Example

```
sleep(Time) ->  
  receive  
    after Time ->  
      true  
  end.
```

`sleep(Time)` suspends the current actor for `Time` milliseconds.

Timeout Example

```
receive
  click ->
    receive
      click ->
        double_click
      after double_click_interval() ->
        single_click
    end
  ...
end
```

`double_click_interval` evaluates to the number of milliseconds expected between two consecutive mouse clicks, for the `receive` to return a `double_click`. Otherwise, a `single_click` is returned.

Zero Timeout

```
receive
  MessagePattern1 [when Guard1] ->
    Actions1 ;
  MessagePattern2 [when Guard2] ->
    Actions2 ;
  ...
  after 0 ->
    ActionsT
end
```

A timeout of 0 means that the timeout will occur immediately, but Erlang tries all messages currently in the mailbox first.

Zero Timeout Example

```
flush_buffer() ->  
  receive  
    AnyMessage ->  
      flush_buffer()  
    after 0 ->  
      true  
  end.
```

`flush_buffer()` completely empties the mailbox of the current actor.

Priority Messages

```
priority_receive() ->  
  receive  
    interrupt ->  
      interrupt  
    after 0 ->  
      receive  
        AnyMessage ->  
          AnyMessage  
      end  
  end.  
end.
```

`priority_receive()` will return the first message in the actor's mailbox, except if there is an `interrupt` message, in which case, `interrupt` will be given priority.

Explicit join continuations

- Erlang does not have language constructs for join blocks.
- Programmers can explicitly implement that coordination pattern.
- Consider a Sum of Squares example, to take a list of numbers and compute the sum of squares concurrently.
 - we compute the square function in separate actors.
 - we compute the sum in a main (join continuation) actor
 - The join continuation actor receives N messages corresponding to the squares asynchronously.
 - When the N messages are received, it contacts the original customer actor with the sum.

Explicit join continuations (2)

-module(squares).

-export([sumsquares/1,square/0]).

sumsquares(N) ->

Workers = create_actors(N),

send_squares(Workers,N,self()),

Sum = receive_results(N),

io:format("The sum of first ~w squares is: ~w~n",[N,Sum]).

The sumsquares function creates the N actors, sends them messages to compute the squares, and receives the N results adding the values up.

Explicit join continuations (3)

```
create_actors(0) -> [];  
create_actors(N) ->  
  [spawn(squares,square,[])|create_actors(N-1)].
```

create_actors
creates a list
with the N
actors, using
recursion.

```
send_squares([],_,_) -> true;  
send_squares([Worker|Workers], N, Farmer) ->  
  Worker!{Farmer,N},  
  send_squares(Workers,N-1, Farmer).
```

send_squares
sends a
message
containing
the pair
{Farmer, N}
to each of
the N actors.

Explicit join continuations (4)

```
receive_results(N) ->  
  receive_results_aux(N,0).  
  
receive_results_aux(0,Acc) -> Acc;  
receive_results_aux(N,Acc) ->  
  receive  
    S -> receive_results_aux(N-1,Acc+S)  
  end.
```

The
receive_results
function
receives N
messages
containing the
squared values,
and
accumulates
the sum in the
Acc
accumulator
variable.

Explicit join continuations (5)

```
square() ->  
  receive  
    {Farmer,N} -> Farmer!N*N  
  end.
```

The square function, which is the behavior of the square actors, receives a pair {Farmer,N}, and sends the square of N to the Farmer actor.

Exercises

50. Download and execute the tree product and sum of squares examples in Erlang.
51. Create a concurrent `fibonacci` behavior in Erlang using join continuations.
52. Write a solution to the Flavius Josephus problem in Erlang. A description of the problem is at CTM Section 7.8.3 (page 558).
53. How would you implement token-passing continuations, join blocks, and first-class continuations in Erlang?
54. How would you implement selective receive in SALSA?