Concurrency control abstractions
(PDCS 9, CPE 5*)

Carlos Varela
Rensselaer Polytechnic Institute

October 15, 2021

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

Actor Languages Summary

- Actors are concurrent entities that react to messages.
  - State is completely encapsulated. There is no shared memory!
  - Message passing is asynchronous.
  - Actors can create new actors. Run-time has to ensure fairness.
- AMST extends the call-by-value lambda calculus with actor primitives. State is modeled as function arguments. Actors use \texttt{ready} to receive new messages.
- Erlang extends a functional programming language core with processes that run arbitrary functions. State is implicit in the function’s arguments. Control loop is explicit: actors use \texttt{receive} to get a message, and tail-form recursive call to continue. Ending a function denotes process (actor) termination.
- SALSA extends an object-oriented programming language (Java) with universal actors. State is explicit, encapsulated in instance variables. Control loop is implicit: ending a message handler, signals readiness to receive a new message. Actors are garbage-collected.

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 1 can receive messages m1 and m2 in any order.

Concurrency Control in SALSA

- SALSA provides three main coordination constructs:
  - Token-passing continuations
    - To synchronize concurrent activities
    - To notify completion of message processing
    - Named tokens enable arbitrary synchronization (data-flow)
  - Join blocks
    - Used for barrier synchronization for multiple concurrent activities
    - To obtain results from otherwise independent concurrent processes
  - First-class continuations
    - To delegate producing a result to another message, or actor

Token Passing Continuations

- Ensure that each message in the continuation expression is sent after the previous message has been processed. It also enables the use of a message handler return value as an argument for a later message (through the token keyword).

  - Example:
    ```
    a1 <- m1() @
    a2 <- m2(token);
    ```

    Send m1 to a1 asking a1 to forward the result of processing m1 to a2 (as the argument of message m2).

Named Tokens

- Tokens can be named to enable more loosely-coupled synchronization

  - Example:
    ```
    token t1 = a1 <- m1();
    token t2 = a2 <- m2(token);
    token t3 = a3 <- m3(t1);
    token t4 = a4 <- m4(t2);
    a <- a(t1, t2, t3, t4);
    ```

    Sending m(...) to a will be delayed until messages m1(), m2(), m3(t1), and m4(t2) have been processed. m(...) can proceed concurrently with m1().
Causal order in the actor model

receive a message with a token
bind ("return") a token
create new actor
computation step

Deterministic Cell Tester Example

```java
module cell;
behavior TokenCellTester {
    void act(String[] args) {
        Cell c = new Cell(0);
        standardOutput << print("Initial Value:") @
        c << get() @
        standardOutput << println(token) @
        c << set(2) @
        standardOutput << print("New Value:") @
        c << get() @
        standardOutput << println(token);
    }
}
```

token can be optionally used to get the return value (completion proof) of the previous message.
syntax enforces a sequential order of message execution.

Cell Tester Example with Named Tokens

```java
module cell;
behavior NamedTokenCellTester {
    void act(String args[]) {
        Cell c = new Cell(0);
        token p0 = standardOutput << print("Initial Value:");
        token t0 = c << get();
        token p1 = standardOutput << println(t0):waitFor(p0);
        token t1 = c << set(2):waitFor(p1);
        token p2 = standardOutput << print("New Value:") :waitFor(p1);
        token t2 = c << get():waitFor(t1);
        standardOutput << println(t2):waitFor(p2);
    }
}
```

We use t0, t1, t2 tokens to ensure cell messages are processed in order.

Join Blocks

- Provide a mechanism for synchronizing the processing of a set of messages.
- Set of results is sent along as a token containing an array of results.
  
  Example:
  ```java
  UniversalActor[] actors = { searcher0, searcher1, searcher2, searcher3 }; 
  join {
      for (int i=0; i < actors.length; i++)
      actors[i] << find( phrase );
  }
  } @
  resultActor << output( token );
  ```

  Send the find( phrase ) message to each actor in actors[] then after all have completed send the result to resultActor as the argument of an output( … ) message.

Example: Acknowledged Multicast

```java
join{ a1 <= n1(); a2 <= m2(); ... an <= mn(); } @
cust <= n(token);
```

Lines of Code Comparison

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Foundry</th>
<th>SALSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledged Multicast</td>
<td>168</td>
<td>100</td>
<td>31</td>
</tr>
</tbody>
</table>
First Class Continuations

- Enable actors to delegate computation to a third party independently of the processing context.

- For example:

```java
int m (...) {
  b ← n (...) @ currentContinuation;
}
```

Ask (delegate) actor b to respond to this message on behalf of current actor (self) by processing b's message.

Delegate Example

```java
module fibonacci;
behavior Calculator {
  int fib(int n) {
    Fibonacci f = new Fibonacci(n);
    f ← compute() @ currentContinuation;
  }

  int add(int a, int b) {
    return a + b;
  }
}
```

```java
module fibonacci;
behavior Calculator {
  int fib(int n) {
    Fibonacci f = new Fibonacci(n);
    return f.← compute();
  }

  int add(int a, int b) {
    return a + b;
  }
}
```

Fibonacci Example

```java
module fibonacci;
behavior Fibonacci {
  int n;
  Fibonacci(int n) {
    this.n = n;
  }

  int compute() {
    if (n == 0)
      return 0;
    else if (n <= 2)
      return 1;
    else {
      Fibonacci fib1 = new Fibonacci(n - 1); // Create new actor
      Fibonacci fib2 = new Fibonacci(n - 2);
      token x = fib1.← compute(); // Synchronize on result
      token y = fib2.← compute();
      add(x, y) @ currentContinuation; // Non-blocked actor
    }
  }

  void act(String args[ ]) {
    n = Integer.parseInt(args[0]);
    compute() @ standardOutput ← println(token);
  }
}
```

Concurrence 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).
  - 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.

Receiving from a specific actor

Actor ! {self(), message}
send
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.)

Order of message patterns matters

receive
  (Left, Right), Customer ->
end
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 Example

Example program and mailbox (head at top):
receive
msg_a ->
msg_b ->
end
receive tries to match msg_a and fails. msg_b can be matched, so it is processed. Suppose execution continues:
receive
msg_a ->
msg_a ->
end
The next message to be processed is msg_a since it is the next in the mailbox and it matches the 2nd pattern.

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

Selective Receive with Timeout

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

\[
sleep(Time) \rightarrow
\]
\[
\text{receive after Time} \rightarrow true\]
\[
\text{end.}
\]

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

C. Varela

31

Timeout Example

\[
\text{receive}
\]
\[
\text{click} \rightarrow
\]
\[
\text{double_click}
\]
\[
\text{after double_click_interval()} \rightarrow
\]
\[
\text{single_click}
\]
\[
\text{end}
\]
\[
double_click_interval\text{ evaluates to the number of milliseconds expected between two consecutive mouse clicks, for the receive to return a double_click. Otherwise, a single_click is returned.}
\]

C. Varela

32

Zero Timeout

\[
\text{receive}
\]
\[
\text{MessagePattern1} \text{ when Guard1} \rightarrow
\]
\[
\text{Actions1 ;}
\]
\[
\text{MessagePattern2} \text{ when Guard2} \rightarrow
\]
\[
\text{Actions2 ;}
\]
\[
\ldots
\]
\[
\text{after 0} \rightarrow
\]
\[
\text{ActionsT}
\]
\[
\text{end}
\]

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

C. Varela

33

Zero Timeout Example

\[
\text{flush_buffer()} \rightarrow
\]
\[
\text{receive}
\]
\[
\text{AnyMessage} \rightarrow
\]
\[
\text{flush_buffer()}\text{ after 0} \rightarrow true\]
\[
\text{end.}
\]

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

C. Varela

34

Priority Messages

\[
\text{priority_receive()} \rightarrow
\]
\[
\text{receive}
\]
\[
\text{interrupt} \rightarrow
\]
\[
\text{interrupt}
\]
\[
\text{after 0} \rightarrow
\]
\[
\text{receive}
\]
\[
\text{AnyMessage} \rightarrow
\]
\[
\text{AnyMessage}
\]
\[
\text{end}
\]

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.

C. Varela

35

Exercises

46. Download and execute the reference cell and tree product examples in SALSA and Erlang.
47. Write a solution to the Flavius Josephus problem in SALSA and Erlang. A description of the problem is at CTM Section 7.8.3 (page 558).
48. PDCS Exercise 9.6.6 (page 204).
49. How would you implement token-passing continuations, join blocks, and first-class continuations in Erlang?
50. How would you implement selective receive in SALSA?