

## **What is an actor**

Actors are computational agents that can be used in concurrent programs. Other examples of computational agents for concurrent programming are :

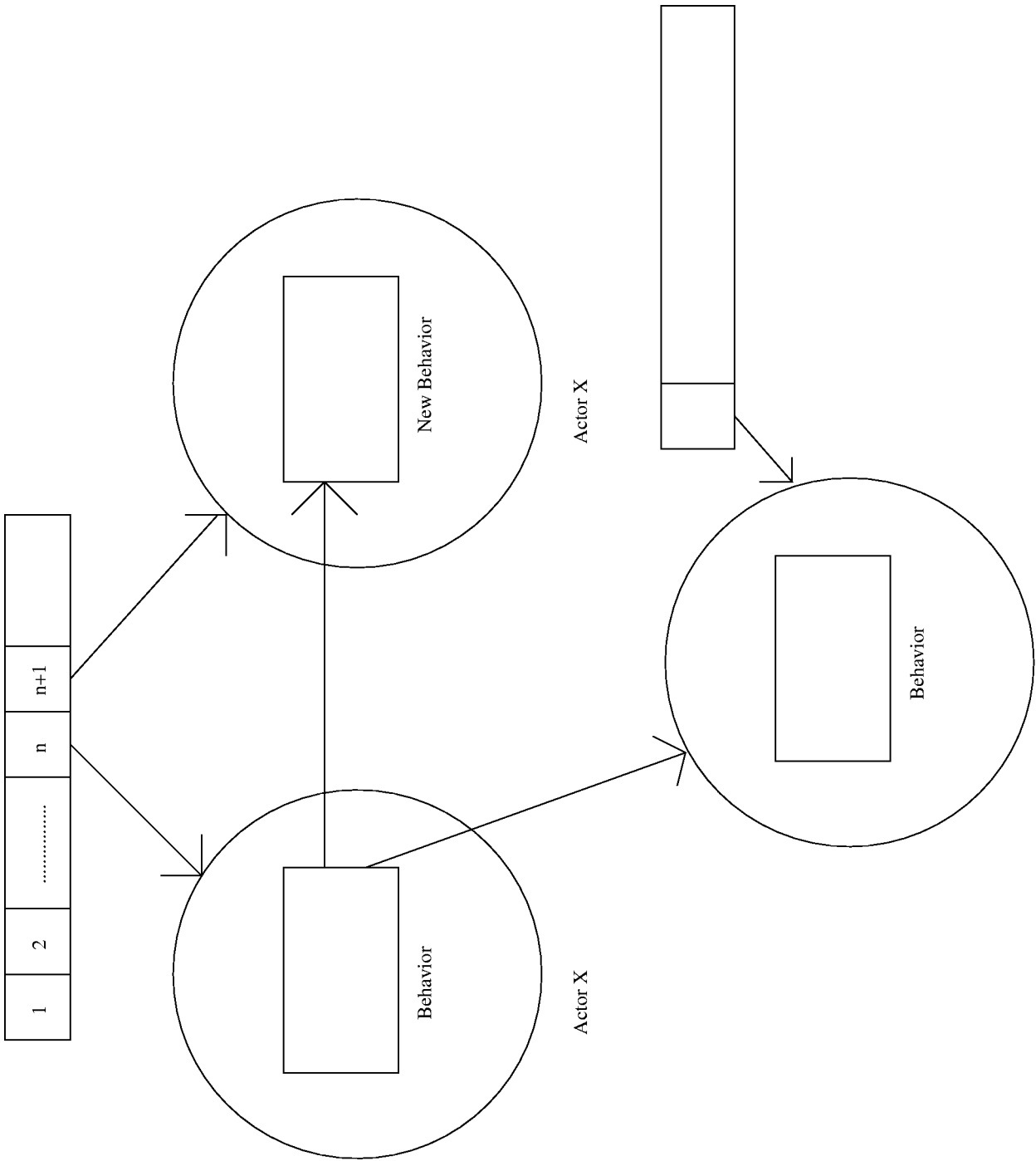
- Sequential processes
- Value-transforming functional systems

Actors can be used to define these two other computational agents.

## How does an actor work

Actors work by communication. Each time an actor receives a message (a communication) it carries out its behavior. A behavior can do three things.

- send communications to other actors
- define a new behavior
- create new actors



# Tasks

Communication are contained in tasks. Tasks are represented as a 3-tuple consisting off:

- a tag which is unique for the system
- a target which is the mail address of the receiver
- a communication which is information to the receiver

As computation proceeds, an actor system evolves to include new tasks and new actors. The set of actors that an actor can communicate with can expand.

# Behavior

Each time an actor accepts a communication, it computes a replacement behavior.

A behavior definition is expressed as a function of the incoming communication.

An actor has a list of values specified at creation called acquaintance list.

The behavior definition uses this list and a second list called communication list which gets its bindings from incoming communication.

## **Minimal actor language**

A program in an actor language consists of

- behavior definitions
- new expressions which create actors
- send command which create tasks
- a receptionist declaration
- an external declaration

## **Declaring Receptionists**

The receptionists are the only actors that are free to receive communications from the outside of the system. The set of receptionists is dynamic. When a communication to the outside contains a mail address of an actor in the system which are not a receptionist, this actor becomes a receptionist.

Declaration of receptionists is optional.

## **Declaring External Actors**

External actors can be specified in a system before the system is connected to the system that holds these actors.

The system can associate identifiers for the external actors with actors that buffers communication for the external actors until a connection to the external actor is created.

This way actor systems can be developed in modules.

The set of external actors is dynamic.

Declaration of external actors is optional.

Systems that have no external actors or receptionists have no meaning.

# Commands

The purpose of commands is to specify actions that are to be carried out.

There are several types of commands. some examples are :

- create new actors
- create new tasks
- specify become behavior
- conditional command

# SAL

SAL uses an Algol-like syntax. The acquaintance list is enclosed in (..) while the communication list is enclosed in [..].

Behavior definitions do not create new actors but simply bind an identifier to a behavior template.

```
<behavior definition> ::=  
  def < beh name>  
    (<acquaintance list>  
     [<communication list>]  
     <command>  
  end def
```

## **Forwarder**

An actor can become a forwarder. A forwarder has one acquaintance the actor to whom the communication must be forwarded.

## **Default behavior**

All actors must specify replacement behavior.

When none is specified the actor responds to communication by becoming an actor with identical behavior.

## Creating Actors

A new expression creates a new actor and returns the mail address of the new actor.

An example of syntax :

```
<new expression> ::=  
    new <beh name> ( {expr { , expr } * } )
```

This new mail address can be bound to an identifier called an actor name by using a let command.

Actors created concurrently by an actor may know each others mail addresses.

## Creating Tasks

A task is created by specifying a target and a communication.

An example of syntax :

```
<send command> ::=  
    send <communication> to <target>
```

Where communication is a sequence of expressions.

## Parameter list

Both acquaintance list and communication list are parameter list.

```
<parameter list> ::= {id | <var list> | {, id | <var list>}* | {}
```

```
<var list> ::= case <tag-field> of <variant>+ end case
```

```
<variant> ::= <case label> : (<parameter list>)
```

Where id is an identifier, the tag field is an identifier and the case label is a constant.

## An Example

A communication list in the behavior of a bank account :

```
case request of
  deposit      : (customer, amount)
  withdrawal   : (customer, amount)
  balance      : (customer)
end case
```

# Command

The syntax of commands :

```
<command> ::=  
  if <logical exp>  
    then <command>  
    {else <command>} fi |  
  become <exp> |  
  <send command> |  
  <let bindings> {<command>} <behavior definition> |  
  <command>*
```

## Let Bindings

```
<let bindings> ::= let id = <exp> { and id = <exp> }*
```

## An Example of Behavior Definition

```
def stack-node (content, link)
  [case operation of
    pop: (customer)
    push:(new-content)
  end case]
  if operation = pop and content not NIL then
    become forwarder(link)
  send [content] customer fi
  if operation = push then
    let P=new stack-node(content, link)
    {become stack-node(new-content, P)} fi
  end def
```

## Recursive Factorial Example

```
def Rec-Factorial (self) [n,u]
  become Rec-Factorial
  if n = 0
    then send [1] to customer
  else
    let c = new Rec-Customer(n,u)
      {send [n-1,c] to self}
    fi
  end def

def Rec-Customer(n,u) [k]
  send [n*k] to u
end def
```



## External Actors Example

```
def buffer (content, link)
  [case operation of
    release : customer
    hold : new-content
  en case]
  if operation = release and content not NIL
  then send [content] to customer
      send [release] to link
      become forwarder(customer)
  if operation = hold
  then let B = new buffer(content, link)
      become buffer(new-content, B)
  end def
```