

Towards Automatic Verification of Erlang Programs by π -Calculus Translation

Chanchal Roy¹ Thomas Noll² Banani Roy¹ James R Cordy¹

¹School of Computing
Queen's University, Canada
{croy, broy, cordy}@cs.queensu.ca

²Lehrstuhl für Informatik 2
RWTH Aachen University, Germany
noll@cs.rwth-aachen.de

Fifth ACM SIGPLAN Erlang Workshop, 2006

- **High quality demands** for telecommunication software (availability, robustness, correctness, ...)
- **Testing** not sufficient to guarantee properties
- Solution: **formal verification**

Use of formal methods to **prove** that (a model of) a **system** has certain **properties** specified in a suitable **logic**.

- **High quality demands** for telecommunication software (availability, robustness, correctness, ...)
- **Testing** not sufficient to guarantee properties
- Solution: **formal verification**

Use of formal methods to **prove** that (a model of) a **system** has certain **properties** specified in a suitable **logic**.

Here:

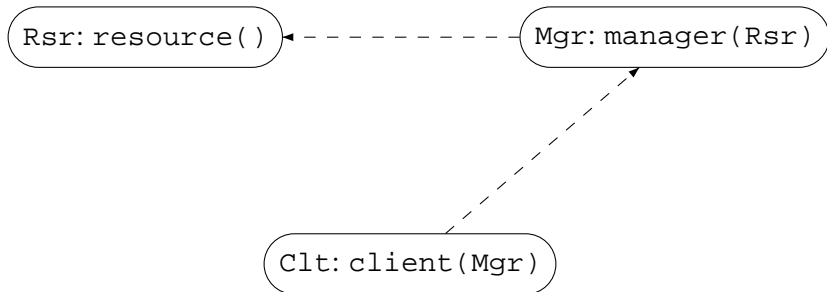
- Concentrate on first step: **model construction**
- Put emphasis on **mobility**
- Refinement of translation presented at last year's Erlang WS

PIErlang Syntax: Subset of Erlang

$$\begin{aligned} \text{Program} & ::= Fdef_1 \dots Fdef_n ; n > 0 \\ Fdef & ::= f(X_1, \dots, X_n) \rightarrow E ; n \geq 0 \\ E & ::= n \mid a \mid X \\ & \quad \mid X = E \mid E_1, E_2 \\ & \quad \mid self() \mid f(A_1, \dots, A_n) ; n \geq 0 \\ & \quad \mid spawn(f, [A_1, \dots, A_n]) ; n \geq 0 \\ & \quad \mid \{A_1, \dots, A_n\} ; n > 0 \\ & \quad \mid A_1!A_2 \mid A!\{A_1, \dots, A_n\} ; n > 0 \\ & \quad \mid receive M_1 ; \dots ; M_n end ; n > 0 \\ & \quad \mid case E of M_1 ; \dots ; M_n end ; n > 0 \\ M & ::= P \rightarrow E \mid \{P_1, \dots, P_n\} \rightarrow E ; n > 0 \\ P & ::= n \mid a \mid X \\ A & ::= n \mid a \mid X \mid self() \end{aligned}$$

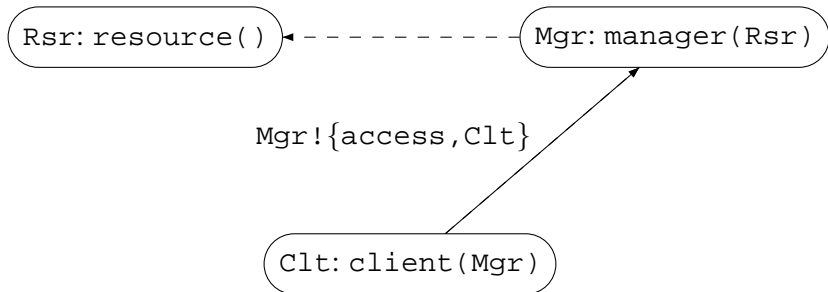
Mobility in Erlang II

Behaviour of resource manager



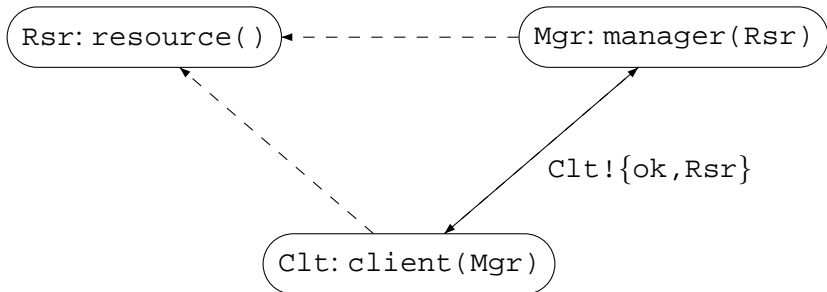
Mobility in Erlang II

Behaviour of resource manager



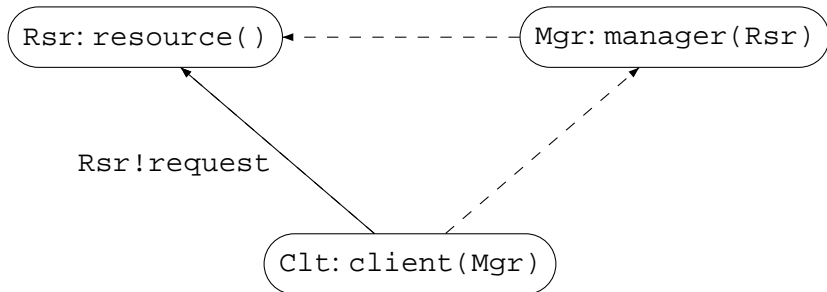
Mobility in Erlang II

Behaviour of resource manager



Mobility in Erlang II

Behaviour of resource manager



Mobility in Erlang I

A simplistic resource manager

```
start() ->
```

```
  Rsr = spawn(resource, []),  
  Mgr = spawn(manager, [Rsr]),  
  client(Mgr).
```

```
resource() ->
```

```
  receive  
    Req->  
      action  
  end.
```

```
manager(Rsr) ->
```

```
  receive  
    {access, C} ->  
      C!{ok, Rsr}  
  end
```

```
client(Mgr) ->
```

```
  Mgr!{access, self()},  
  Receive  
    {ok, R} ->  
      R!request  
  end.
```

The (Polyadic and Asynchronous) π -Calculus

```
Sys ::= Pdef1 ... Pdefn           % system

Pdef ::= i(x1, ..., xn) = Proc    % process definition

Proc ::= nil                          % inactive process
      | x0(x1, ..., xn).Proc      % input
      |  $\overline{x_0}$ <x1, ..., xn>.nil    % asynchronous output
      | Proc1 || Proc2             % parallel composition
      | Proc1 + Proc2              % non-deterministic choice
      | ( $\nu$  x) Proc                 % new name
      | [x1=x2] Proc                % match
      | [x1<>x2] Proc              % mismatch
      | i(x1, ..., xn)           % process instantiation
```

Reaction rule:

$$\begin{aligned} & \bar{x}_0 \langle y_1, \dots, y_n \rangle . \text{nil} \parallel x_0(x_1, \dots, x_n) . P \\ \rightarrow & \text{nil} \parallel P[x_1 \mapsto y_1, \dots, x_n \mapsto y_n] \end{aligned}$$

- actually synchronous
- however, special form of output is “non-blocking”

The Resource Manager in π -Calculus

```

Main = ( $\nu$  self)(start(self))
start(self) = ( $\nu$  rPID, mPID, cPID, p, q)
              ( $\bar{p}$ <rPID>.nil || resource(rPID) ||
               p(Rsr).( $\bar{q}$ <mPID>.nil ||
                    manager(mPID,Rsr) ||
                    q(Mgr).client(cPID,Mgr)))
resource(self) = self(Req). $\bar{res}$ <action>.nil
manager(self,Rsr) = self(input,C).
                  [input=access] $\bar{C}$ <ok,Rsr>.nil
client(self,Mgr) =  $\bar{Mgr}$ <access,self>.nil ||
                  self(input,R).
                  [input=ok] $\bar{R}$ <request>.nil
```

Observing Behavior of Resource Manager in π -Calculus

Instantiation of `start process` \implies react on `p` and `q` \implies omit `nil` process

$$\left(\begin{array}{l} (\nu \text{ rPID}, \text{ mPID}, \text{ cPID}) \\ \text{resource}(\text{ rPID}) \\ \parallel \\ \text{manager}(\text{ mPID}, \text{ rPID}) \\ \parallel \\ \text{client}(\text{ cPID}, \text{ mPID}) \end{array} \right)$$

Observing Behavior of Resource Manager in π -Calculus

Instantiation of `start process` \implies react on `p` and `q` \implies omit `nil` process

$$\left(\begin{array}{c} (\nu \text{ rPID}, \text{ mPID}, \text{ cPID}) \\ \left(\begin{array}{c} \text{resource}(\text{rPID}) \\ \parallel \\ \text{manager}(\text{mPID}, \text{ rPID}) \\ \parallel \\ \text{client}(\text{cPID}, \text{ mPID}) \end{array} \right) \end{array} \right)$$

Upon instantiation of `manager` and `client` process, we get

$$\left(\begin{array}{c} (\nu \text{ rPID}, \text{ mPID}, \text{ cPID}) \\ \left(\begin{array}{c} \text{resource}(\text{rPID}) \\ \parallel \\ \text{mPID}(\text{input}, \text{C}) . [\text{input}=\text{access}] \bar{\text{C}}\langle \text{ok}, \text{rPID} \rangle . \text{nil} \\ \parallel \\ \overline{\text{mPID}}\langle \text{access}, \text{cPID} \rangle . \text{nil} \parallel \\ \text{cPID}(\text{input}, \text{R}) . [\text{input}=\text{ok}] \bar{\text{R}}\langle \text{request} \rangle . \text{nil} \end{array} \right) \end{array} \right)$$

Observing Behavior of Resource Manager in π -Calculus

client asks manager for handle to resource: react on mPID

$$\left(\begin{array}{l} (\nu \text{ rPID}, \text{ mPID}, \text{ cPID}) \\ \left(\begin{array}{l} \text{resource}(\text{rPID}) \\ \parallel \\ [\text{access}=\text{access}] \overline{\text{cPID}}\langle \text{ok}, \text{rPID} \rangle . \text{nil} \\ \parallel \\ \text{nil} \parallel \text{cPID}(\text{input}, \text{R}) . [\text{input}=\text{ok}] \overline{\text{R}}\langle \text{request} \rangle . \text{nil} \end{array} \right) \end{array} \right)$$

Observing Behavior of Resource Manager in π -Calculus

client asks manager for handle to resource: react on mPID

$$\left(\begin{array}{c} (\nu \text{ rPID}, \text{ mPID}, \text{ cPID}) \\ \left(\begin{array}{c} \text{resource}(\text{rPID}) \\ \parallel \\ [\text{access}=\text{access}] \overline{\text{cPID}}\langle \text{ok}, \text{rPID} \rangle . \text{nil} \\ \parallel \\ \text{nil} \parallel \text{cPID}(\text{input}, R) . [\text{input}=\text{ok}] \overline{R}\langle \text{request} \rangle . \text{nil} \end{array} \right) \end{array} \right)$$

Matching `access=access`, react on cPID

$$\left(\begin{array}{c} (\nu \text{ rPID}, \text{ mPID}, \text{ cPID}) \\ \left(\begin{array}{c} \text{resource}(\text{rPID}) \\ \parallel \\ \text{nil} \\ \parallel \\ \text{nil} \parallel [\text{ok}=\text{ok}] \overline{\text{rPID}}\langle \text{request} \rangle . \text{nil} \end{array} \right) \end{array} \right)$$

Observing Behavior of Resource Manager in π -Calculus

Invoking the resource process, we get

$$\left((\nu \text{rPID}, \text{mPID}, \text{cPID}) \left(\begin{array}{l} \text{rPID}(\text{Req}) . \overline{\text{res}}\langle \text{action} \rangle . \text{nil} \\ \parallel \\ \text{nil} \\ \parallel \\ \text{nil} \parallel [\text{ok}=\text{ok}] \overline{\text{rPID}}\langle \text{request} \rangle . \text{nil} \end{array} \right) \right)$$

Observing Behavior of Resource Manager in π -Calculus

Invoking the resource process, we get

$$\left(\begin{array}{c} (\nu \text{ rPID}, \text{ mPID}, \text{ cPID}) \\ \left(\begin{array}{c} \text{rPID}(\text{Req}) . \overline{\text{res}}\langle \text{action} \rangle . \text{nil} \\ \parallel \\ \text{nil} \\ \parallel \\ \text{nil} \parallel [\text{ok}=\text{ok}] \overline{\text{rPID}}\langle \text{request} \rangle . \text{nil} \end{array} \right) \end{array} \right)$$

client can send actual request to resource

$$\left(\begin{array}{c} (\nu \text{ rPID}, \text{ mPID}, \text{ cPID}) \\ \left(\begin{array}{c} \overline{\text{res}}\langle \text{action} \rangle . \text{nil} \\ \parallel \\ \text{nil} \\ \parallel \\ \text{nil} \parallel \text{nil} \end{array} \right) \end{array} \right)$$

The Translation Mapping

Goal: define

$$\text{TrPI} : \text{Erlang} \rightarrow \pi\text{-Calculus}$$

such that the “essential behaviour” of programs is represented

Goal: define

$$\text{TrPI} : \text{Erlang} \rightarrow \pi\text{-Calculus}$$

such that the “essential behaviour” of programs is represented **Important**

issues:

- Data structures
- Process creation
- Asynchronous communication via mailboxes
- **Polyadic (i.e., tuple) communication**
- **Deterministic matching (case/receive)**

Translation of Programs and Function Definitions

Translation of programs:

$$TrPI_{prog} : Program \rightarrow Sys$$

$$TrPI_{prog}(Fdef_1 \dots Fdef_n) \\ := \left(\begin{array}{l} Main = (\nu \text{ self}, \dots) TrPI_{exp}(\text{self}, f_0) \\ TrPI_{fundef}(Fdef_1) \dots TrPI_{fundef}(Fdef_n) \end{array} \right)$$

Translation of Programs and Function Definitions

Translation of programs:

$$\text{TrPI}_{prog} : \text{Program} \rightarrow \text{Sys}$$

$$\begin{aligned} & \text{TrPI}_{prog}(Fdef_1 \dots Fdef_n) \\ & := \left(\begin{array}{l} \text{Main} = (\nu \text{ self}, \dots) \text{TrPI}_{exp}(\text{self}, f_0) \\ \text{TrPI}_{fundef}(Fdef_1) \dots \text{TrPI}_{fundef}(Fdef_n) \end{array} \right) \end{aligned}$$

Translation of function definitions:

$$\text{TrPI}_{fundef} : Fdef \rightarrow Pdef$$

$$\begin{aligned} & \text{TrPI}_{fundef}(f(X_1, \dots, X_n) \rightarrow E) \\ & := f(\text{self}, X_1, \dots, X_n) = \text{TrPI}_{exp}(\text{self}, E) \end{aligned}$$

`self`: pid of evaluating process (= `self()`)

$$\text{TrPI}_{exp} : \text{Name} \times \text{Expression} \rightarrow \text{Proc}$$

- yields a process which evaluates the given expression...
- ... and returns the value along the `res` channel
- abstracts from (most) data structures (numbers, lists, ...)
- atoms and pids are faithfully represented

Translation of Expressions II

Simple expressions

Using the auxiliary function

$$\mathit{TrPI}_{arg} : \mathit{Argument} \rightarrow \mathit{Name}$$

with

$$\mathit{TrPI}_{arg}(n) := \mathit{unknown}$$

$$\mathit{TrPI}_{arg}(a) := a$$

$$\mathit{TrPI}_{arg}(X) := X$$

$$\mathit{TrPI}_{arg}(\mathit{self}()) := \mathit{self}$$

simple expressions can be translated as follows:

$$\mathit{TrPI}_{exp}(\mathit{self}, A) := \overline{\mathit{res}} \langle \mathit{TrPI}_{arg}(A) \rangle . \mathit{nil}$$

$$\mathit{TrPI}_{exp}(\mathit{self}, \{A_1, \dots, A_n\}) := \overline{\mathit{res}} \langle \mathit{TrPI}_{arg}(A_1), \dots, \mathit{TrPI}_{arg}(A_n) \rangle . \mathit{nil}$$

Translation of Expressions III

Sequencing expressions

$$\begin{aligned} & TrPI_{exp}(\mathbf{self}, X = E_1, E_2) \\ & := (\nu \mathit{res}') (TrPI_{exp}(\mathbf{self}, E_1) \parallel \mathit{res}'(X). TrPI_{exp}(\mathbf{self}, E_2)) \end{aligned}$$

$$\begin{aligned} & TrPI_{exp}(\mathbf{self}, E_1, E_2) \\ & := (\nu \mathit{res}') (TrPI_{exp}(\mathbf{self}, E_1) \parallel \mathit{res}'(\mathit{dummy}). TrPI_{exp}(\mathbf{self}, E_2)) \end{aligned}$$

Translation of Expressions III

Sequencing expressions

$$\begin{aligned} TrPI_{exp}(\text{self}, X = E_1, E_2) \\ := (\nu \text{res}') (TrPI_{exp}(\text{self}, E_1) \parallel \text{res}'(X).TrPI_{exp}(\text{self}, E_2)) \end{aligned}$$

$$\begin{aligned} TrPI_{exp}(\text{self}, E_1, E_2) \\ := (\nu \text{res}') (TrPI_{exp}(\text{self}, E_1) \parallel \text{res}'(\text{dummy}).TrPI_{exp}(\text{self}, E_2)) \end{aligned}$$

Example:

$$\begin{aligned} TrPI_{exp}(\text{self}, X = a, \{X, b\}) \\ = (\nu \text{res}') (\overline{\text{res}'\langle a \rangle}.\text{nil} \parallel \text{res}'(X).\overline{\text{res}\langle X, b \rangle}.\text{nil}) \end{aligned}$$

Translation of Expressions IV

Function calls and process creation

$$\begin{aligned} TrPI_{exp}(\text{self}, f(A_1, \dots, A_n)) \\ := f(\text{self}, TrPI_{arg}(A_1), \dots, TrPI_{arg}(A_n)) \end{aligned}$$

$$\begin{aligned} TrPI_{exp}(\text{self}, X = \text{spawn}(f, [A_1, \dots, A_n]), E) \\ := (\nu \text{self}') (\nu \text{res}') \\ \left(\begin{array}{l} f(\text{self}', TrPI_{arg}(A_1), \dots, TrPI_{arg}(A_n)) \parallel \\ \text{res}'(\text{dummy}).\text{nil} \parallel \\ \overline{\text{res}}\langle \text{self}' \rangle.\text{nil} \parallel \\ \text{res}(X).TrPI_{exp}(\text{self}, E) \end{array} \right) \end{aligned}$$

Translation of Expressions V

Message passing (polyadic)

$$\text{TrPI}_{exp}(\text{self}, A_1 ! A_2)$$
$$:= \overline{\text{TrPI}_{arg}(A_1)} \langle \text{TrPI}_{arg}(A_2) \rangle . \text{nil} \parallel \overline{\text{res}} \langle \text{TrPI}_{arg}(A_2) \rangle . \text{nil}$$
$$\text{TrPI}_{exp}(\text{self}, A ! \{A_1, \dots, A_n\})$$
$$:= \overline{\text{TrPI}_{arg}(A)} \langle \text{TrPI}_{arg}(A_1), \dots, \text{TrPI}_{arg}(A_n) \rangle . \text{nil} \parallel \\ \overline{\text{res}} \langle \text{TrPI}_{arg}(A_1), \dots, \text{TrPI}_{arg}(A_n) \rangle . \text{nil}$$

- Implicit representation of Erlang mailbox using concurrent “output particles”
- Abstracts from arrival order of messages

Translation of Expressions V

Message passing (polyadic)

$$TrPI_{exp}(self, A_1 ! A_2)$$
$$:= \overline{TrPI_{arg}(A_1)} < TrPI_{arg}(A_2) > . nil \parallel \overline{res} < TrPI_{arg}(A_2) > . nil$$
$$TrPI_{exp}(self, A ! \{A_1, \dots, A_n\})$$
$$:= \overline{TrPI_{arg}(A)} < TrPI_{arg}(A_1), \dots, TrPI_{arg}(A_n) > . nil \parallel \\ \overline{res} < TrPI_{arg}(A_1), \dots, TrPI_{arg}(A_n) > . nil$$

- Implicit representation of Erlang mailbox using concurrent “output particles”
- Abstracts from arrival order of messages

Example:

$$P ! a, P ! b$$

Translation of Expressions V

Message passing (polyadic)

$$TrPI_{exp}(self, A_1 ! A_2)$$
$$:= \overline{TrPI_{arg}(A_1)} < TrPI_{arg}(A_2) > . nil \parallel \overline{res} < TrPI_{arg}(A_2) > . nil$$
$$TrPI_{exp}(self, A ! \{A_1, \dots, A_n\})$$
$$:= \overline{TrPI_{arg}(A)} < TrPI_{arg}(A_1), \dots, TrPI_{arg}(A_n) > . nil \parallel \\ \overline{res} < TrPI_{arg}(A_1), \dots, TrPI_{arg}(A_n) > . nil$$

- Implicit representation of Erlang mailbox using concurrent “output particles”
- Abstracts from arrival order of messages

Example:

$$P ! a, P ! b$$

\downarrow^*
Mailbox_P :

...	a	b
-----	---	---

Translation of Expressions V

Message passing (polyadic)

$$TrPI_{exp}(\text{self}, A_1 ! A_2)$$
$$:= \overline{TrPI_{arg}(A_1)} < TrPI_{arg}(A_2) > . nil \parallel \overline{res} < TrPI_{arg}(A_2) > . nil$$
$$TrPI_{exp}(\text{self}, A ! \{A_1, \dots, A_n\})$$
$$:= \overline{TrPI_{arg}(A)} < TrPI_{arg}(A_1), \dots, TrPI_{arg}(A_n) > . nil \parallel \\ \overline{res} < TrPI_{arg}(A_1), \dots, TrPI_{arg}(A_n) > . nil$$

- Implicit representation of Erlang mailbox using concurrent “output particles”
- Abstracts from arrival order of messages

Example:

$$P ! a, P ! b \xrightarrow{TrPI_{exp}} (\nu res') \left(\overline{P} < a > . nil \parallel \overline{res'} < a > . nil \parallel \right. \\ \left. res' (\text{dummy}) . (\overline{P} < b > . nil \parallel \overline{res} < b > . nil) \right)$$

↓*

Mailbox_P :

...	a	b
-----	---	---

Translation of Expressions V

Message passing (polyadic)

$$\text{TrPI}_{exp}(\text{self}, A_1 ! A_2)$$
$$:= \overline{\text{TrPI}_{arg}(A_1)} \langle \text{TrPI}_{arg}(A_2) \rangle . \text{nil} \parallel \overline{\text{res}} \langle \text{TrPI}_{arg}(A_2) \rangle . \text{nil}$$
$$\text{TrPI}_{exp}(\text{self}, A ! \{A_1, \dots, A_n\})$$
$$:= \overline{\text{TrPI}_{arg}(A)} \langle \text{TrPI}_{arg}(A_1), \dots, \text{TrPI}_{arg}(A_n) \rangle . \text{nil} \parallel \\ \overline{\text{res}} \langle \text{TrPI}_{arg}(A_1), \dots, \text{TrPI}_{arg}(A_n) \rangle . \text{nil}$$

- Implicit representation of Erlang mailbox using concurrent “output particles”
- Abstracts from arrival order of messages

Example:

$$P ! a, P ! b$$
$$\xrightarrow{\text{TrPI}_{exp}} (\nu \text{res}') \left(\overline{P} \langle a \rangle . \text{nil} \parallel \overline{\text{res}'} \langle a \rangle . \text{nil} \parallel \right. \\ \left. \text{res}'(\text{dummy}). (\overline{P} \langle b \rangle . \text{nil} \parallel \overline{\text{res}} \langle b \rangle . \text{nil}) \right)$$

Mailbox_P : \downarrow^*

...	a	b
-----	---	---

\downarrow^*
 $\overline{P} \langle a \rangle . \text{nil} \parallel \overline{P} \langle b \rangle . \text{nil} \parallel \overline{\text{res}} \langle b \rangle . \text{nil}$

Translation of Expressions VI

Branching expressions

Previous simple translation: nondeterministic and monadic

```
receive
```

```
  ok -> a;
```

```
  {req, P} -> b;
```

```
  X -> c
```

```
end
```

```
self(in).[in=ok] $\overline{\text{res}}$ <a>.nil +  
self(in).[in=unknown] $\overline{\text{res}}$ <b>.nil +  
self(X). $\overline{\text{res}}$ <c>.nil
```

Translation of Expressions VI

Branching expressions

Previous simple translation: nondeterministic and monadic

```
receive
  ok -> a;
  {req, P} -> b;
  X -> c
end
```

$\xrightarrow{TrPI_{exp}}$

```
self(in).[in=ok] $\overline{res}$ <a>.nil +
self(in).[in=unknown] $\overline{res}$ <b>.nil +
self(X). $\overline{res}$ <c>.nil
```

Improved translation: deterministic and polyadic

```
self(in).[in=ok] $\overline{res}$ <a>.nil +
self(in,P).[in=req] $\overline{res}$ <b>.nil +
self(X).[X<>ok] $\overline{res}$ <c>.nil
```

(*case*: analogous)

Done:

- Developed a π -calculus model which reflects “essential” behaviour of an Erlang program
- Improvement of previous approach:
 - respects order of overlapping patterns (deterministic branching)
 - supports tuple communication

Done:

- Developed a π -calculus model which reflects “essential” behaviour of an Erlang program
- Improvement of previous approach:
 - respects order of overlapping patterns (deterministic branching)
 - supports tuple communication

To do:

- Larger case studies
- Representation of list data structures
- Respect order of messages

Questions please?