Modeling Erlang in the π–Calculus

Thomas Noll[1] Chanchal Kumar Roy[2]

[1]Lehrstuhl für Informatik 2
RWTH Aachen University
noll@cs.rwth-aachen.de

[2]School of Computing
Dublin City University
chanchalkroy@yahoo.com

Fourth ACM SIGPLAN Erlang Workshop, 2005

Motivation

- 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.

Motivation

- 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.

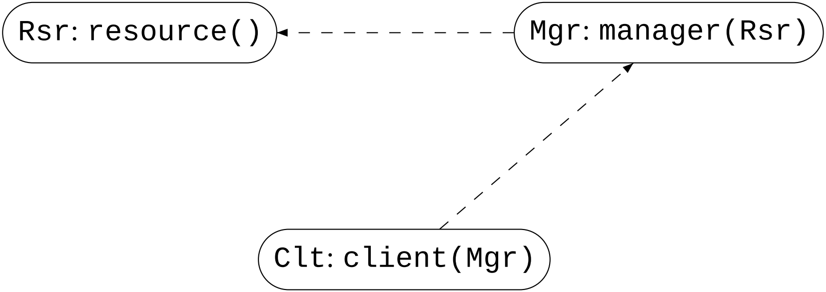
Here:

- Concentrate on first step: model construction
- Put emphasis on mobility

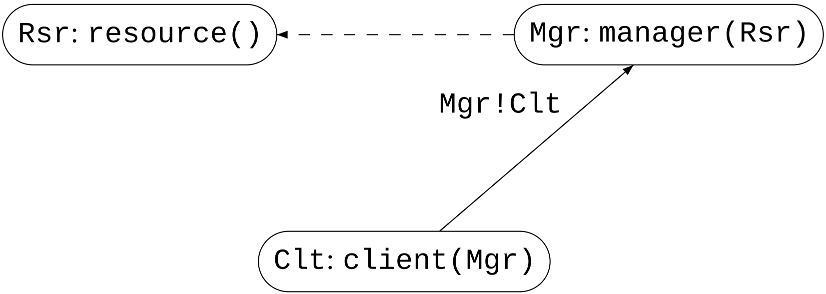
Mobility in Erlang I

A simplistic resource manager

```
-module(resmgr).
-export([start/0]).

start() ->
  Rsr = spawn(resource, []),
  Mgr = spawn(manager, [Rsr]),
  client(Mgr).

resource() ->
  receive
    Req ->
      action
  end.

manager(Rsr) ->
  receive
    C ->
      C!Rsr
  end.

client(Mgr) ->
  Mgr!self(),
  receive
    R ->
      R!request
  end.
```



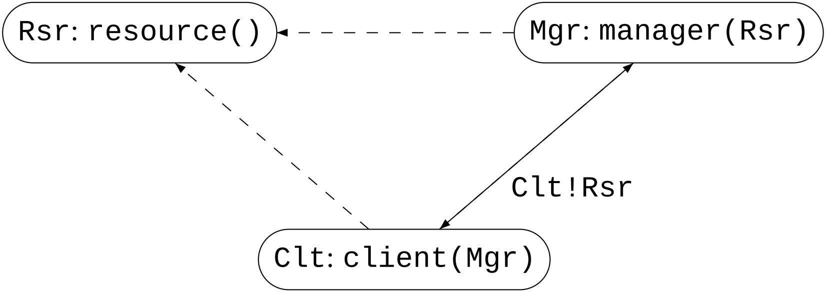
Rsr: resource()
Mgr: manager(Rsr)
Clt: client(Mgr)



Rsr: resource()
Mgr: manager(Rsr)
Mgr!Clt
Clt: client(Mgr)

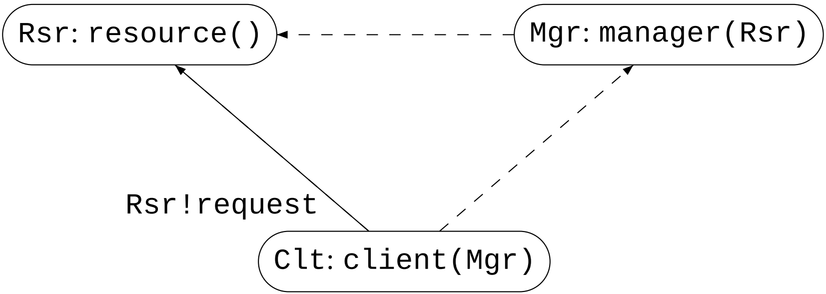
Mobility in Erlang II

Behaviour of resource manager



Mobility in Erlang II

Behaviour of resource manager



The (Asynchronous) π–Calculus

$Sys ::= Pdef_1 \ldots Pdef_n$	% system
$Pdef ::= i(x_1, \ldots, x_n) = Proc$	% process definition
$Proc ::= \texttt{nil}$	% inactive process
$\mid\ x_0(x_1, \ldots, x_n).Proc$	% input
$\mid\ \overline{x_0}\texttt{<}x_1, \ldots, x_n\texttt{>}.\texttt{nil}$	% asynchronous output
$\mid\ Proc_1 \parallel Proc_2$	% parallel composition
$\mid\ Proc_1 + Proc_2$	% non–deterministic choice
$\mid\ (\nu\, x)Proc$	% new name
$\mid\ [x_1 = x_2]Proc$	% match
$\mid\ [x_1 \texttt{<>} x_2]Proc$	% mismatch
$\mid\ i\texttt{<}x_1, \ldots, x_n\texttt{>}$	% process instantiation

Communication in the π–Calculus

Reaction rule:

$$\begin{array}{rl} & \overline{x_0}\texttt{<}y_1\texttt{,}\ldots\texttt{,}y_n\texttt{>.nil} \parallel x_0\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{).}P \\ \longrightarrow & \texttt{nil} \parallel P[x_1 \mapsto y_1,\ldots,x_n \mapsto y_n] \end{array}$$

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

The Resource Manager in π–Calculus

$$
\begin{aligned}
\texttt{resource(Rsr)} &= \texttt{Rsr(Req)}.\overline{\texttt{action}}\texttt{<>.nil} \\
\texttt{manager(Rsr,Mgr)} &= \texttt{Mgr(C)}.\overline{\texttt{C}}\texttt{<Rsr>.nil} \\
\texttt{client(Mgr,Clt)} &= \overline{\texttt{Mgr}}\texttt{<Clt>.nil} \parallel \texttt{Clt(R)}.\overline{\texttt{R}}\texttt{<request>.nil}
\end{aligned}
$$

The Resource Manager in π–Calculus

$$\texttt{resource(Rsr)} = \texttt{Rsr(Req)}.\overline{\texttt{action}}\texttt{<>.nil}$$
$$\texttt{manager(Rsr,Mgr)} = \texttt{Mgr(C)}.\overline{\texttt{C}}\texttt{<Rsr>.nil}$$
$$\texttt{client(Mgr,Clt)} = \overline{\texttt{Mgr}}\texttt{<Clt>.nil} \parallel \texttt{Clt(R)}.\overline{\texttt{R}}\texttt{<request>.nil}$$

Operational behaviour:

$$(\nu\,\texttt{Rsr})(\nu\,\texttt{Mgr})(\nu\,\texttt{Clt})\begin{pmatrix}\texttt{resource<Rsr>} \parallel \\ \texttt{manager<Rsr,Mgr>} \parallel \\ \texttt{client<Mgr,Clt>}\end{pmatrix}$$

The Resource Manager in π–Calculus

$$\begin{aligned}
\texttt{resource(Rsr)} &= \texttt{Rsr(Req)}.\overline{\texttt{action}}\texttt{<>.nil} \\
\texttt{manager(Rsr,Mgr)} &= \texttt{Mgr(C)}.\overline{\texttt{C}}\texttt{<Rsr>.nil} \\
\texttt{client(Mgr,Clt)} &= \overline{\texttt{Mgr}}\texttt{<Clt>.nil} \parallel \texttt{Clt(R)}.\overline{\texttt{R}}\texttt{<request>.nil}
\end{aligned}$$

Operational behaviour:

$$\begin{array}{l}(\nu\,\texttt{Rsr})(\nu\,\texttt{Mgr})(\nu\,\texttt{Clt}) \\ \left(\begin{array}{l}\texttt{resource<Rsr>} \parallel \\ \texttt{manager<Rsr,Mgr>} \parallel \\ \texttt{client<Mgr,Clt>}\end{array}\right)\end{array} \longrightarrow \begin{array}{l}(\nu\,\texttt{Rsr})(\nu\,\texttt{Mgr})(\nu\,\texttt{Clt}) \\ \left(\begin{array}{l}\texttt{resource<Rsr>} \parallel \\ \overline{\texttt{Clt}}\texttt{<Rsr>.nil} \parallel \\ \texttt{nil} \parallel \texttt{Clt(R)}.\overline{\texttt{R}}\texttt{<request>.nil}\end{array}\right)\end{array}$$

The Resource Manager in π–Calculus

$$
\begin{aligned}
\texttt{resource(Rsr)} &= \texttt{Rsr(Req).}\overline{\texttt{action}}\texttt{<>.nil}\\
\texttt{manager(Rsr,Mgr)} &= \texttt{Mgr(C).}\overline{\texttt{C}}\texttt{<Rsr>.nil}\\
\texttt{client(Mgr,Clt)} &= \overline{\texttt{Mgr}}\texttt{<Clt>.nil} \parallel \texttt{Clt(R).}\overline{\texttt{R}}\texttt{<request>.nil}
\end{aligned}
$$

Operational behaviour:

$$
(\nu\,\texttt{Rsr})(\nu\,\texttt{Mgr})(\nu\,\texttt{Clt})
\begin{pmatrix}
\texttt{resource<Rsr>} \parallel \\
\texttt{nil} \parallel \\
\texttt{nil} \parallel \overline{\texttt{Rsr}}\texttt{<request>.nil}
\end{pmatrix}
\longleftarrow
(\nu\,\texttt{Rsr})(\nu\,\texttt{Mgr})(\nu\,\texttt{Clt})
\begin{pmatrix}
\texttt{resource<Rsr>} \parallel \\
\overline{\texttt{Clt}}\texttt{<Rsr>.nil} \parallel \\
\texttt{nil} \parallel \texttt{Clt(R).}\overline{\texttt{R}}\texttt{<request>.nil}
\end{pmatrix}
$$

$$
\begin{array}{rcl}
\texttt{resource(Rsr)} & = & \texttt{Rsr(Req)}.\overline{\texttt{action}}\texttt{<>.nil} \\
\texttt{manager(Rsr,Mgr)} & = & \texttt{Mgr(C)}.\overline{\texttt{C}}\texttt{<Rsr>.nil} \\
\texttt{client(Mgr,Clt)} & = & \overline{\texttt{Mgr}}\texttt{<Clt>.nil} \parallel \texttt{Clt(R)}.\overline{\texttt{R}}\texttt{<request>.nil}
\end{array}
$$

Operational behaviour:

$$
(\nu\,\texttt{Rsr})(\nu\,\texttt{Mgr})(\nu\,\texttt{Clt})
\begin{pmatrix}
\texttt{resource<Rsr>} \parallel \\
\texttt{nil} \parallel \\
\texttt{nil} \parallel \overline{\texttt{Rsr}}\texttt{<request>.nil}
\end{pmatrix}
\longrightarrow
(\nu\,\texttt{Rsr})(\nu\,\texttt{Mgr})(\nu\,\texttt{Client})
\begin{pmatrix}
\overline{\texttt{action}}\texttt{<>.nil} \parallel \\
\texttt{nil} \parallel \\
\texttt{nil} \parallel \texttt{nil}
\end{pmatrix}
$$

The Translation Mapping

Goal: define

$$\mathit{trans}[\![\cdot]\!] : \text{Core Erlang} \rightarrow \pi\text{–Calculus}$$

such that the “essential behaviour” of programs is represented

The Translation Mapping

Goal: define

$$trans[\![\cdot]\!] : \text{Core Erlang} \to \pi\text{–Calculus}$$

such that the “essential behaviour” of programs is represented

Important issues:

- Data structures
- Process creation
- Asynchronous communication via mailboxes

Translation of Modules and Function Definitions

$$trans_M[\![\cdot]\!] : Module \longrightarrow Sys$$

$$trans_M[\![\texttt{module}\ a\ [\,\ldots\,]\ \texttt{attributes}\ [\,\ldots\,]\ fd_1 \ldots fd_n]\!] := trans_F[\![fd_1]\!] \ldots trans_F[\![fd_n]\!]$$

$$trans_F[\![\cdot]\!] : Fdef \longrightarrow Pdef$$

$$trans_F[\![f\ \texttt{=}\ \texttt{fun(}v_1\texttt{,}\ \ldots\texttt{,}v_n\texttt{)->}\ e]\!] := f\texttt{(}v_1\texttt{,}\ \ldots\texttt{,}v_n\texttt{,self,res)=}\ trans_E[\![e]\!]$$

Parameters:

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

`res`: return channel for result

Translation of Expressions I

$$trans_E[\![\cdot]\!] : Expr \longrightarrow Proc$$

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

$$
\begin{aligned}
trans_E[\![v]\!] &:= \overline{\texttt{res}}\texttt{<}v\texttt{>.nil}\\
trans_E[\![a]\!] &:= \overline{\texttt{res}}\texttt{<}a\texttt{>.nil}\\
trans_E[\![f]\!] &:= \overline{\texttt{res}}\texttt{<unknown>.nil}\\
trans_E[\![z]\!] &:= \overline{\texttt{res}}\texttt{<unknown>.nil}\\
trans_E[\![c]\!] &:= \overline{\texttt{res}}\texttt{<unknown>.nil}\\
trans_E[\![s]\!] &:= \overline{\texttt{res}}\texttt{<unknown>.nil}\\
trans_E[\![\texttt{[]}]\!] &:= \overline{\texttt{res}}\texttt{<unknown>.nil}\\
trans_E[\![\texttt{[}e_1\,|\,e_2\texttt{]}]\!] &:= \overline{\texttt{res}}\texttt{<unknown>.nil}\\
trans_E[\![\{e_1, \ldots, e_n\}]\!] &:= \overline{\texttt{res}}\texttt{<unknown>.nil}
\end{aligned}
$$

Translation of Expressions III

Sequencing expressions

$$trans_E[\![\texttt{let}\ v = e_1\ \texttt{in}\ e_2]\!]$$
$$:= \quad (\nu\, \texttt{res}')(trans_E[\![e_1]\!] \parallel \texttt{res}'(v).trans_E[\![e_2]\!])$$
$$trans_E[\![\texttt{do}\ e_1\ e_2]\!]$$
$$:= \quad (\nu\, \texttt{res}')(trans_E[\![e_1]\!] \parallel \texttt{res}'(\texttt{dummy}).trans_E[\![e_2]\!])$$

Translation of Expressions III

Sequencing expressions

$$trans_E[\![\texttt{let}\ v = e_1\ \texttt{in}\ e_2]\!]$$
$$:= \ (\nu\,\texttt{res}')(trans_E[\![e_1]\!] \parallel \texttt{res}'(v).trans_E[\![e_2]\!])$$
$$trans_E[\![\texttt{do}\ e_1\ e_2]\!]$$
$$:= \ (\nu\,\texttt{res}')(trans_E[\![e_1]\!] \parallel \texttt{res}'(\texttt{dummy}).trans_E[\![e_2]\!])$$

Example:

$$trans_E[\![\texttt{let X = a in}\ \{\texttt{X,b}\}]\!]$$
$$= \ (\nu\,\texttt{res}')(\overline{\texttt{res}'}\texttt{<a>.nil} \parallel \texttt{res}'(\texttt{X}).\overline{\texttt{res}}\texttt{<unknown>.nil})$$

Translation of Expressions IV

Function calls

$$trans_E[\![\texttt{apply}\; f\texttt{(}e_1\texttt{,}\ \ldots\texttt{,}e_n\texttt{)}]\!]$$

$$:= \quad f\texttt{<}trans_P[\![e_1]\!]\texttt{,}\ \ldots\texttt{,}trans_P[\![e_n]\!]\texttt{,self,res>}$$

$$trans_E[\![\texttt{primop}\; a\texttt{(}e_1\texttt{,}\ \ldots\texttt{,}e_n\texttt{)}]\!]$$

$$:= \quad \overline{\texttt{res}}\texttt{<unknown>.nil}$$

Translation of Expressions V

Process creation

$$trans_E[\![\texttt{call 'erlang':'spawn'(}f\texttt{, [}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{])}]\!]$$

$$:= (\nu\,\texttt{self}')(\nu\,\texttt{res}')\left(\begin{array}{l} f\texttt{<}trans_P[\![e_1]\!]\texttt{,}\ldots\texttt{,}trans_P[\![e_n]\!]\texttt{,self}'\texttt{,res}'\texttt{>} \parallel \\ \texttt{res}'\texttt{(dummy).nil} \parallel \\ \overline{\texttt{res}}\texttt{<self}'\texttt{>.nil} \end{array}\right)$$

$$trans_E[\![\texttt{call 'erlang':'self'()}]\!]$$

$$:= \overline{\texttt{res}}\texttt{<self>.nil}$$

Translation of Expressions VI

Message passing

$$trans_E[\![\texttt{call 'erlang':'!'}(e_1,e_2)]\!]$$

$$:= \quad \overline{trans_P[\![e_1]\!]}\texttt{<}trans_P[\![e_2]\!]\texttt{>.nil} \parallel \overline{\texttt{res}}\texttt{<}trans_P[\![e_2]\!]\texttt{>.nil}$$

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

Translation of Expressions VI

Message passing

$$trans_E[\![\texttt{call 'erlang':'!'(}e_1\texttt{,}e_2\texttt{)}]\!]$$

$$:= \overline{trans_P[\![e_1]\!]}\texttt{<}trans_P[\![e_2]\!]\texttt{>.nil} \parallel \overline{\texttt{res}}\texttt{<}trans_P[\![e_2]\!]\texttt{>.nil}$$

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

Example:

```
P!a,P!b
```

Translation of Expressions VI

Message passing

$$\mathit{trans}_E[\![\texttt{call 'erlang':'!'}(e_1\texttt{,}e_2\texttt{)}]\!]$$

$$:= \quad \mathit{trans}_P[\![e_1]\!]\texttt{<}\mathit{trans}_P[\![e_2]\!]\texttt{>.nil} \parallel \overline{\texttt{res}}\texttt{<}\mathit{trans}_P[\![e_2]\!]\texttt{>.nil}$$

- 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 VI

Message passing

$$
\begin{aligned}
&\mathit{trans}_E[\![\texttt{call 'erlang':'!'}(e_1,e_2)]\!] \\
&\quad := \overline{\mathit{trans}_P[\![e_1]\!]}\texttt{<}\mathit{trans}_P[\![e_2]\!]\texttt{>.nil} \parallel \overline{\texttt{res}}\texttt{<}\mathit{trans}_P[\![e_2]\!]\texttt{>.nil}
\end{aligned}
$$

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

Example:

$$
\texttt{P!a,P!b} \xrightarrow{\mathit{trans}_E[\![\cdot]\!]} (\nu\, \mathit{res}')\left(\begin{array}{l}\overline{\texttt{P}}\texttt{<a>.nil} \parallel \overline{\texttt{res}'}\texttt{<a>.nil} \parallel \\ \texttt{res}'\texttt{(dummy).}(\overline{\texttt{P}}\texttt{<b>.nil} \parallel \overline{\texttt{res}}\texttt{<b>.nil})\end{array}\right)
$$

$\downarrow *$

$\text{Mailbox}_{\texttt{P}}$: | ... | a | b |

$$\mathit{trans}_E[\![\mathtt{call\ 'erlang':'!'}(e_1,e_2)]\!]$$
$$:=\ \overline{\mathit{trans}_P[\![e_1]\!]}\mathtt{<}\mathit{trans}_P[\![e_2]\!]\mathtt{>.nil} \parallel \overline{\mathtt{res}}\mathtt{<}\mathit{trans}_P[\![e_2]\!]\mathtt{>.nil}$$

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

Example:

`P!a,P!b` $\xrightarrow{\mathit{trans}_E[\![\cdot]\!]}$ $(\nu\, \mathit{res}')\Big(\overline{\mathtt{P}}\mathtt{<a>.nil} \parallel \overline{\mathtt{res}'}\mathtt{<a>.nil} \parallel \mathtt{res}'\mathtt{(dummy).}(\overline{\mathtt{P}}\mathtt{<b>.nil} \parallel \overline{\mathtt{res}}\mathtt{<b>.nil})\Big)$

$\downarrow *$ (left) and $\downarrow *$ (right)

Mailbox$_{\mathtt{P}}$: | … | a | b |

$\overline{\mathtt{P}}\mathtt{<a>.nil} \parallel \overline{\mathtt{P}}\mathtt{<b>.nil} \parallel \overline{\mathtt{res}}\mathtt{<b>.nil}$

Translation of Expressions VII

Branching expressions

$$trans_E[\![\texttt{case}\ e\ \texttt{of}\ c_1 \ldots c_n]\!]$$

$$:= \quad (\nu\,\texttt{res}')(trans_E[\![e]\!] \parallel (trans_C[\![c_1]\!](\texttt{res}') + \ldots + trans_C[\![c_n]\!](\texttt{res}')))$$

$$trans_E[\![\texttt{receive}\ c_1 \ldots c_n\ \texttt{after}\ e_1\ \texttt{->}\ e_2]\!]$$

$$:= \quad (\nu\,\texttt{res}')\left(trans_E[\![e_1]\!] \parallel \texttt{res}'(\texttt{dummy}).\left(\begin{array}{l} [\texttt{dummy=infinity}](trans_C[\![c_1]\!](\texttt{self}) + \ldots + trans_C[\![c_n]\!](\texttt{self})) + \\ [\texttt{dummy<>infinity}](trans_C[\![c_1]\!](\texttt{self}) + \ldots + trans_C[\![c_n]\!](\texttt{self}) + trans_E[\![e_2]\!]) \end{array} \right) \right)$$

- Abstracts from order of clauses (overlapping patterns)

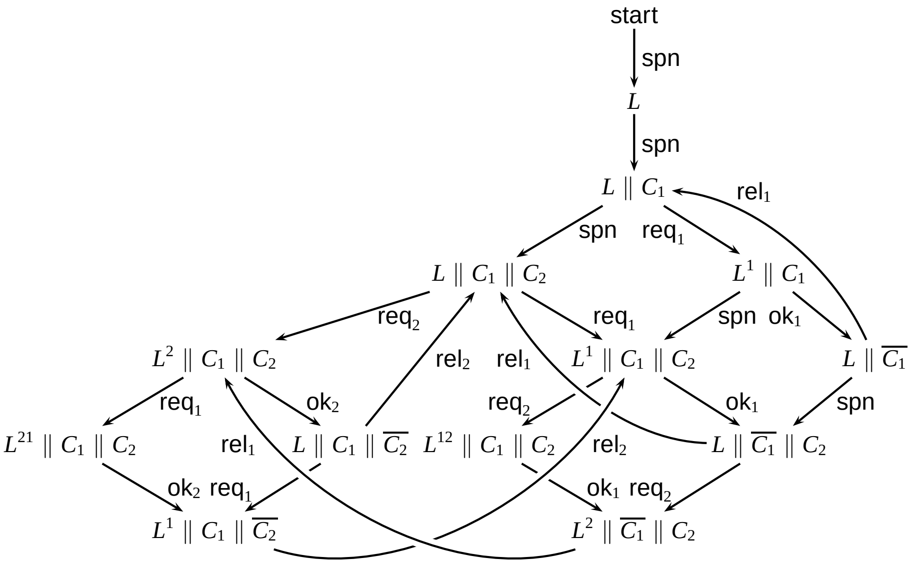
Another Example: a Locker System

```
start() ->
  Locker = spawn(locker, []),
  spawn(client, [Locker]),
  spawn(client, [Locker]).
```

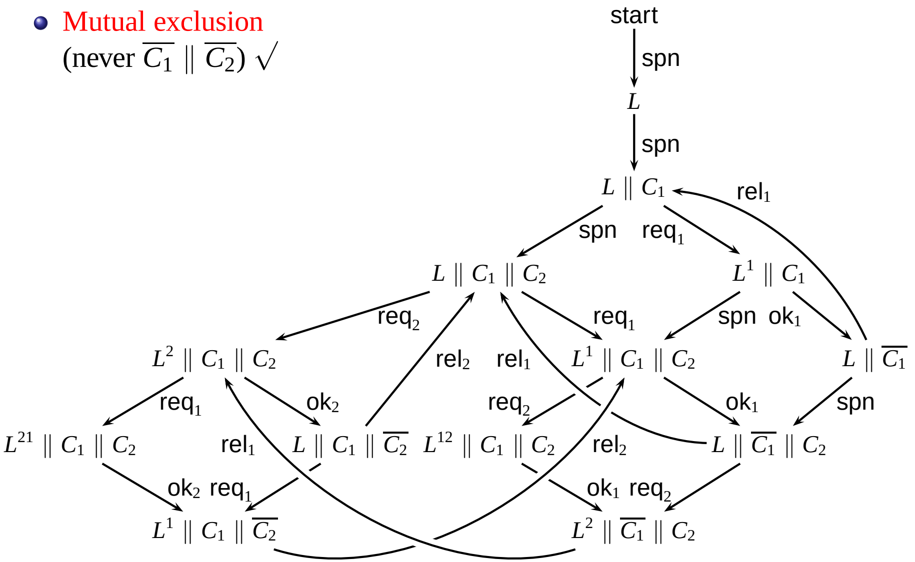
```
locker() ->
  receive
    {req, Client} ->
      Client!ok,
      receive
        {rel, Client} ->
          locker()
      end
  end.
```

```
client(Locker) ->
  Locker!{req, self()},
 receive
   ok ->
     % critical section
    Locker!{rel, self()},
     client(Locker)
  end.
```

The Locker Transition System

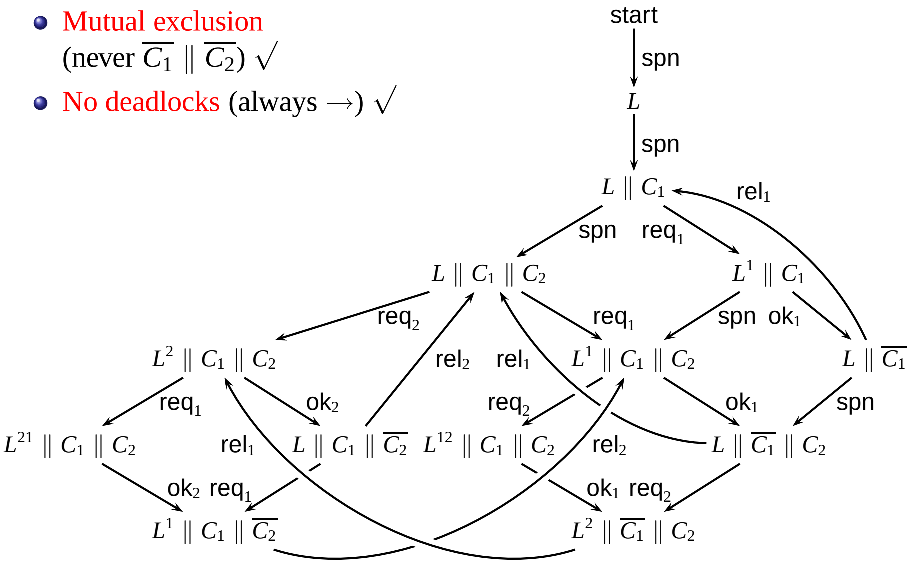


- Mutual exclusion
  (never $\overline{C_1} \parallel \overline{C_2}$) $\surd$



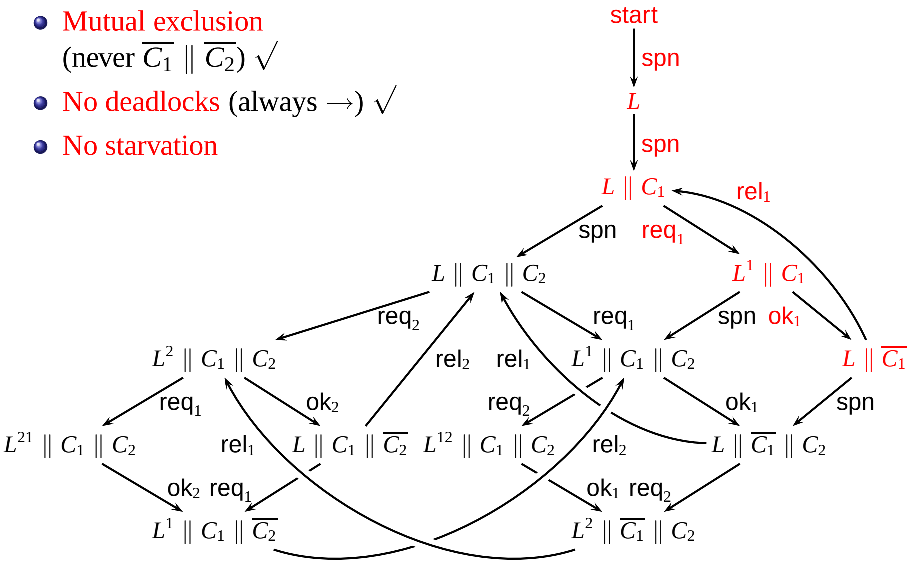
The Locker Transition System

- Mutual exclusion
  (never $\overline{C_1} \parallel \overline{C_2}$) $\surd$
- No deadlocks (always $\longrightarrow$) $\surd$



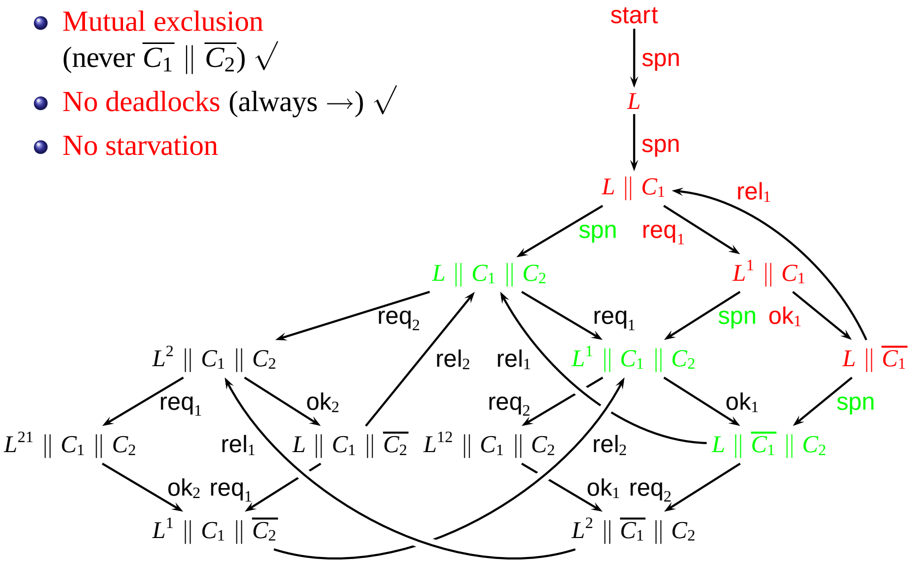
The Locker Transition System

- Mutual exclusion (never $\overline{C_1} \parallel \overline{C_2}$) $\surd$
- No deadlocks (always $\longrightarrow$) $\surd$
- No starvation



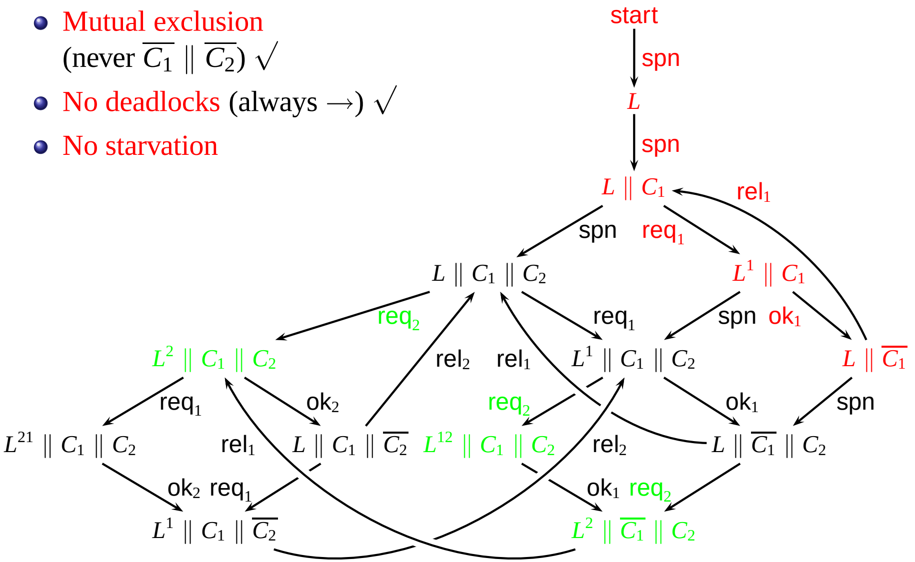
The Locker Transition System

- Mutual exclusion (never $\overline{C_1} \parallel \overline{C_2}$) $\surd$
- No deadlocks (always $\longrightarrow$) $\surd$
- No starvation



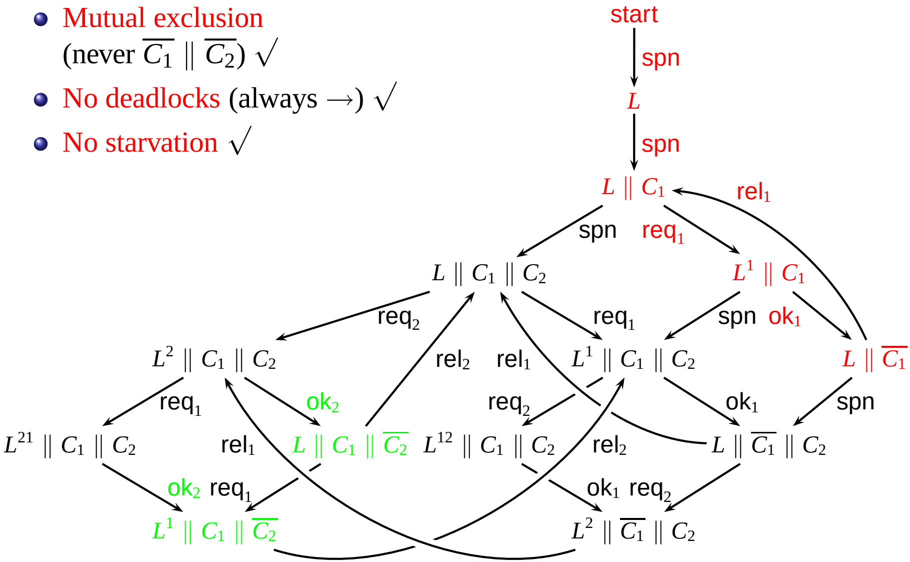
The Locker Transition System

- Mutual exclusion (never $\overline{C_1} \parallel \overline{C_2}$) $\surd$
- No deadlocks (always $\longrightarrow$) $\surd$
- No starvation



The Locker Transition System

- Mutual exclusion (never $\overline{C_1} \parallel \overline{C_2}$) $\surd$
- No deadlocks (always $\rightarrow$) $\surd$
- No starvation $\surd$



Conclusion

Done: developed a π–calculus model which reflects “essential” behaviour of an Erlang program

Conclusion

Done: developed a π–calculus model which reflects "essential" behaviour of an Erlang program

To do:

- Larger case studies

Conclusion

Done: developed a π–calculus model which reflects "essential" behaviour of an Erlang program

To do:

- Larger case studies
- More detailed representation of data structures (e.g., `P!{request,self()}`)

Conclusion

Done: developed a π–calculus model which reflects “essential” behaviour of an Erlang program

To do:

- Larger case studies
- More detailed representation of data structures (e.g., `P!{request,self()}`)
- Respect order of messages

Conclusion

Done: developed a π–calculus model which reflects "essential" behaviour of an Erlang program

To do:

- Larger case studies
- More detailed representation of data structures (e.g., `P!{request,self()}`)
- Respect order of messages
- Respect order of (overlapping) patterns
  example: `receive a -> b; X -> c`
  currently: self(dummy).[dummy=a]$\overline{\texttt{res}}$<b>.nil $+$
  self(X).$\overline{\texttt{res}}$<c>.nil

Conclusion

Done: developed a π–calculus model which reflects “essential” behaviour of an Erlang program

To do:

- Larger case studies
- More detailed representation of data structures
  (e.g., `P!{request,self()}`)
- Respect order of messages
- Respect order of (overlapping) patterns
  example: `receive a -> b; X -> c`
  currently: `self(dummy).[dummy=a]`$\overline{\texttt{res}}$`<b>.nil` $+$
  `self(X).`$\overline{\texttt{res}}$`<c>.nil`
  better: `self(dummy).`$($`[dummy=a]`$\overline{\texttt{res}}$`<b>.nil` $+$
  `[dummy<>a]`$\overline{\texttt{res}}$`<c>.nil`$)$

Translation of Clauses

$$trans_C[\![\cdot]\!] : Clause \times X \longrightarrow Proc$$

$$trans_C[\![v \; \texttt{when 'true' ->} \; e]\!](x)$$
$$:= \; x\texttt{(}v\texttt{)}.trans_E[\![e]\!]$$
$$trans_C[\![p \; \texttt{when 'true' ->} \; e]\!](x)$$
$$:= \; x\texttt{(dummy)}.\texttt{[dummy=}trans_P[\![p]\!]\texttt{]}trans_E[\![e]\!]$$
$$trans_C[\![v_1 \; \texttt{when call 'erlang':'=:='(}v_1\texttt{,}v_2\texttt{) ->} \; e]\!](x)$$
$$:= \; x\texttt{(}v_1\texttt{)}.\texttt{[}v_1\texttt{=}v_2\texttt{]}trans_E[\![e]\!]$$

Translation of Patterns

$$trans_P[\![\cdot]\!] : Pat \longrightarrow Proc$$

$$
\begin{aligned}
trans_P[\![v]\!] &:= v \\
trans_P[\![a]\!] &:= a \\
trans_P[\![z]\!] &:= \texttt{unknown} \\
trans_P[\![c]\!] &:= \texttt{unknown} \\
trans_P[\![s]\!] &:= \texttt{unknown} \\
trans_P[\![\texttt{[ ]}]\!] &:= \texttt{unknown} \\
trans_P[\![\texttt{[}p_1\,\texttt{|}\,p_2\texttt{]}]\!] &:= \texttt{unknown} \\
trans_P[\![\{p_1\texttt{,}\ \ldots\ \texttt{,}p_n\}]\!] &:= \texttt{unknown}
\end{aligned}
$$