Modeling Erlang in the π–Calculus

Chanchal K. Roy and James R. Cordy

Abstract

This poster presents a mapping of Erlang programs to the π–calculus, a process algebra whose name–passing feature allows representation of the mobile aspects of software written in Erlang in a natural way.

1 Motivation

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

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

2 PIErlang Syntax

A subset of the Erlang programming language called PIErlang is used in this study. Ignors higher–order functions, list comprehensions, interoperation etc.

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Fdef_1} \ldots \mathit{Fdef_n}\ ;\ n>0 \\
\mathit{Fdef} & ::= & \texttt{f}\,(X_1,\ldots,X_n)\texttt{->}\,E\ ;\ n>=0 \\
E & ::= & n \mid a \mid X \\
 & \mid & X{=}E \mid E_1, E_2 \\
 & \mid & \mathit{self}\,() \mid \texttt{f}\,(A_1,\ldots,A_n)\ ;\ n>=0 \\
 & \mid & \texttt{spawn}\,(f,\ [A_1,\ldots,A_n])\ ;\ n>=0 \\
 & \mid & \{A_1,\ldots,A_n\}\ ;\ n>0 \\
 & \mid & A_1\,!\,A_2 \mid A!\{A_1,\ldots,A_n\}\ ;\ n>0 \\
 & \mid & \texttt{receive}\ M_1;\ldots;M_n\ \texttt{end}\ ;\ n>0 \\
 & \mid & \texttt{case}\ E\ \texttt{of}\ M_1;\ldots;M_n\ \texttt{end}\ ;\ n>0 \\
M & ::= & P\texttt{->}E \mid \{P_1,\ldots,P_n\}\texttt{->}E\ ;\ n>0 \\
P & ::= & n \mid a \mid X \\
A & ::= & n \mid a \mid X \mid \mathit{self}\,()
\end{array}
$$

3 A Simplistic Resource Manager

The `start` function first spawns a `resource` and a `manager` process and then invokes the `client` function. The PID of `resource` is initially not known to `client`, and it therefore first needs to retrieve this information from the `manager`. Having received the PID it sends a simple request to `resource`.

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

resource() ->            manager(Rsr) ->
  receive                  receive
    Req->                   {access, C} ->
       action                        C!{ok, Rsr}
  end.                     end

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

4 The Polyadic π–Calculus

Here we introduce the syntax of the Milner et.al.'s asynchronous π–calculus, which is parameterized with respect to a set I of agent (represented by $i \in I$) and to a set X of names (x, y, z etc.). The names serve as both communication channels and data to be transmitted along them.

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

Reaction Rule: communication in the π–calculus

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

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

5 Resource Manager in the π–Calculus:

Having applied the mappings, a π–model of the resource manager is obtained as follows:

$$
\begin{array}{rcl}
\texttt{Main} & = & (\nu\ \texttt{self})\,(\texttt{start(self)}) \\
\texttt{start(self)} & = & (\nu\ \texttt{rPID, mPID, cPID, p, q}) \\
 & & (\overline{\texttt{p}}\texttt{<rPID>.nil} \parallel \texttt{resource(rPID)} \parallel \\
 & & \ \texttt{p(Rsr).}(\overline{\texttt{q}}\texttt{<mPID>.nil} \parallel \\
 & & \ \texttt{manager(mPID,Rsr)} \parallel \\
 & & \ \texttt{q(Mgr).client(cPID,Mgr)))} \\
\texttt{resource(self)} & = & \texttt{self(Req).}\overline{\texttt{res}}\texttt{<action>.nil} \\
\texttt{manager(self,Rsr)} & = & \texttt{self(input,C).} \\
 & & \texttt{[input=access]}\overline{\texttt{C}}\texttt{<ok,Rsr>.nil} \\
\texttt{client(self, Mgr)} & = & \overline{\texttt{Mgr}}\texttt{<access,self>.nil} \parallel \\
 & & \texttt{self(input,R).} \\
 & & \texttt{[input=ok]}\overline{\texttt{R}}\texttt{<request>.nil}
\end{array}
$$

6 Observing Behavior in the π–Calculus

To examine the behavior of obtained π–model, we start from the `Main` process. Instantiation of `start` process $\Longrightarrow$ react on `p` and `q` $\Longrightarrow$ omit `nil` process

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

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

$$
\left(\begin{array}{l}
(\nu\ \texttt{rPID, mPID, cPID}) \\
\left(\begin{array}{l}
\texttt{resource(rPID)} \\
\quad\parallel \\
\textbf{mPID}\texttt{(input,C).[input=access]}\overline{\texttt{C}}\texttt{<ok,rPID>.nil} \\
\quad\parallel \\
\overline{\textbf{mPID}}\texttt{<access,cPID>.nil} \parallel \\
\texttt{cPID(input,R).[input=ok]}\overline{\texttt{R}}\texttt{<request>.nil}
\end{array}\right)
\end{array}\right)
$$

`client` asks `manager` for handle to `resource`: react on `mPID`

$$
\left(\begin{array}{l}
(\nu\ \texttt{rPID, mPID, cPID}) \\
\left(\begin{array}{l}
\texttt{resource(rPID)} \\
\quad\parallel \\
\texttt{[}\textbf{access=access}\texttt{]}\overline{\textbf{cPID}}\texttt{<ok,rPID>.nil} \\
\quad\parallel \\
\texttt{nil} \parallel \textbf{cPID}\texttt{(input,R).[input=ok]}\overline{\texttt{R}}\texttt{<request>.nil}
\end{array}\right)
\end{array}\right)
$$

Matching `access=access`, react on `cPID`

$$
\left(\begin{array}{l}
(\nu\ \texttt{rPID, mPID, cPID}) \\
\left(\begin{array}{l}
\texttt{resource(rPID)} \\
\quad\parallel \\
\quad\texttt{nil} \\
\quad\parallel \\
\texttt{nil} \parallel \texttt{[ok=ok]}\overline{\texttt{rPID}}\texttt{<request>.nil}
\end{array}\right)
\end{array}\right)
$$

Invoking the `resource` process, we get

$$
\left(\begin{array}{l}
(\nu\ \texttt{rPID, mPID, cPID}) \\
\left(\begin{array}{l}
\textbf{rPID}\texttt{(Req).}\overline{\texttt{res}}\texttt{<action>.nil} \\
\quad\parallel \\
\quad\texttt{nil} \\
\quad\parallel \\
\texttt{nil} \parallel \texttt{[ok=ok]}\overline{\textbf{rPID}}\texttt{<request>.nil}
\end{array}\right)
\end{array}\right)
$$

`client` can send actual request to `resource`

$$
\left(\begin{array}{l}
(\nu\ \texttt{rPID, mPID, cPID}) \\
\left(\begin{array}{c}
\overline{\texttt{res}}\texttt{<action>.nil} \\
\parallel \\
\texttt{nil} \\
\parallel \\
\texttt{nil} \parallel \texttt{nil}
\end{array}\right)
\end{array}\right)
$$

7 The Translation Mapping

Goal: define

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

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:

$$TrPI_{prog}\texttt{:}\ \mathit{Name}\ \texttt{X}\ \mathit{Program} \rightarrow \mathit{System}$$

$$
\begin{array}{l}
TrPI_{prog}(\texttt{self},\ F_1,\ldots,F_n) \\
:= \left(\begin{array}{l}
\texttt{Main=}(\nu\ \texttt{self)}\ \texttt{OtherNames)}\ TrPI_{exp}\texttt{(self, }f\_0\texttt{)}, \\
TrPI_{fundef}(\texttt{self},\ F_1),\ldots,TrPI_{fundef}(\texttt{self},\ F_n)
\end{array}\right)
\end{array}
$$

where f_0 is the left hand side of F_1 and `OtherNames` is the set of names/atoms used in the system.

Translation of Function Definitions:

$$TrPI_{fundef}\texttt{:}\ \mathit{Name}\ \texttt{X}\ \mathit{Function\ Def.} \rightarrow \mathit{Process\ Def.}$$

$$
\begin{array}{l}
TrPI_{fundef}(\texttt{self},\ f\,(X_1,\ldots,X_n)\texttt{->}\,E) \\
:= \left(\ f\,(\texttt{self},\ X_1,\ldots,X_n)\ \texttt{=}\ TrPI_{exp}(\texttt{self},\ E)\ \right)
\end{array}
$$

Translation of Expressions:

$$TrPI_{exp}\texttt{:}\ \mathit{Name}\ \texttt{X}\ \mathit{Expression} \rightarrow \mathit{Process}$$

- 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

$$TrPI_{arg}\texttt{:}\ \mathit{Argument} \rightarrow \mathit{Name}$$

$$
\begin{array}{rcl}
TrPI_{arg}(n) & := & \texttt{unknown} \\
TrPI_{arg}(a) & := & a \\
TrPI_{arg}(X) & := & X \\
TrPI_{arg}(\texttt{self()}) & := & \texttt{self}
\end{array}
$$

Simple Expressions:

$$TrPI_{exp}(\texttt{self},\ A) := \overline{\texttt{res}}\texttt{<}TrPI_{arg}(A)\texttt{>.nil}$$

$$
\begin{array}{l}
TrPI_{exp}(\texttt{self},\ \{A_1,\ldots,A_n\}) \\
:= \overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_1),\ldots,TrPI_{arg}(A_n)\texttt{>.nil}
\end{array}
$$

Send Expression:

$$
\begin{array}{l}
TrPI_{exp}(\texttt{self},A!\{A_1,\ldots,A_n\}) \\
:= \left(\begin{array}{l}
\overline{TrPI_{arg}(A)}\texttt{<}TrPI_{arg}(A_1),\ldots,TrPI_{arg}(A_n)\texttt{>.nil} \\
\quad\parallel \\
\overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_1),\ldots,TrPI_{arg}(A_n)\texttt{>.nil}
\end{array}\right)
\end{array}
$$

Receive Expression: an example

$$
\begin{array}{lcl}
\texttt{receive} & & \\
\quad\texttt{ok -> a;} & & \texttt{self(in).[in=ok]}\overline{\texttt{res}}\texttt{<a>.nil +} \\
\quad\texttt{\{req, P\} -> b;} & \xrightarrow{TrPI_{exp}} & \texttt{self(in,P).[in=req]}\overline{\texttt{res}}\texttt{<b>.nil +} \\
\quad\texttt{X -> c} & & \texttt{self(X).[X<>ok]}\overline{\texttt{res}}\texttt{<c>.nil} \\
\texttt{end} & &
\end{array}
$$

8 Conclusion

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

References

[1] C. K. Roy, T. Noll, B. Roy and J.R. Cordy. Towards Automatic Verification of Erlang Programs by π-Calculus Translation. In *Proc. Erlang'06, ACM SIGPLAN 5th Erlang Workshop*, Portland, Oregon, ACM, September 2006, pp. 38-49.