Modeling Erlang in the π–Calculus

Chanchal K. Roy and James R. Cordy
School of Computing, Queen's University

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.

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Fdef}_1 \ldots \mathit{Fdef}_n \text{ ; } n>0 \\
\mathit{Fdef} & ::= & \mathtt{f}\,(X_1,\ldots,X_n)\ \texttt{->}\ E \text{ ; } n>=0 \\
E & ::= & n \mid a \mid X \\
 & \mid & X{=}E \mid E_1, E_2 \\
 & \mid & \mathit{self}\,() \mid \mathtt{f}\,(A_1,\ldots,A_n) \text{ ; } n>=0 \\
 & \mid & \mathtt{spawn}\,(f,\ [A_1,\ldots,A_n]) \text{ ; } n>=0 \\
 & \mid & \{A_1,\ldots,A_n\} \text{ ; } n>0 \\
 & \mid & A_1\,!\,A_2 \mid A\,!\{A_1,\ldots,A_n\} \text{ ; } n>0 \\
 & \mid & \mathtt{receive}\ M_1;\ldots;M_n\ \mathtt{end} \text{ ; } n>0 \\
 & \mid & \mathtt{case}\ E\ \mathtt{of}\ M_1;\ldots;M_n\ \mathtt{end} \text{ ; } n>0 \\
M & ::= & P\texttt{->}E \mid \{P_1,\ldots,P_n\}\texttt{->}E \text{ ; } n>0 \\
P & ::= & n \mid a \mid X \\
A & ::= & n \mid a \mid X \mid \mathit{self}\,()
\end{array}
$$

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.

The syntactic categories Sys (process systems), Pdef (single process definitions), and Proc (process expressions) are defined by the grammar below:

$$
\begin{array}{rcll}
\mathit{Sys} & ::= & \mathit{Pdef}_1 \ldots \mathit{Pdef}_n & \text{\% system} \\
\mathit{Pdef} & ::= & i\,(x_1,\ldots,x_n) = \mathit{Proc} & \text{\% process definition} \\
\mathit{Proc} & ::= & \mathtt{nil} & \text{\% inactive process} \\
 & \mid & x_0\,(x_1,\ldots,x_n)\,.\,\mathit{Proc} & \text{\% input} \\
 & \mid & \overline{x_0}\texttt{<}x_1,\ldots,x_n\texttt{>}.\mathtt{nil} & \text{\% asynchronous output} \\
 & \mid & \mathit{Proc}_1 \parallel \mathit{Proc}_2 & \text{\% parallel composition} \\
 & \mid & \mathit{Proc}_1 + \mathit{Proc}_2 & \text{\% non–deterministic choice} \\
 & \mid & (\nu\,x)\,\mathit{Proc} & \text{\% new name} \\
 & \mid & [x_1{=}x_2]\,\mathit{Proc} & \text{\% match} \\
 & \mid & [x_1\texttt{<>}x_2]\,\mathit{Proc} & \text{\% mismatch} \\
 & \mid & i\texttt{<}x_1,\ldots,x_n\texttt{>} & \text{\% process instantiation}
\end{array}
$$

Reaction Rule: communication in the π–calculus

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

- 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:

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

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

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

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

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

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

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

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

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

Invoking the `resource` process, we get

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

`client` can send actual request to `resource`

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

7 The 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`)

Complete PIErlang Program:

$$\mathit{TrPI}_{prog}\colon\ \mathit{Name} \times \mathit{Program} \rightarrow \mathit{System}$$

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

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

Function Definitions:

$$\mathit{TrPI}_{fundef}\colon\ \mathit{Name} \times \mathit{Function\ Def.} \rightarrow \mathit{Process\ Def.}$$

$$
\begin{array}{l}
\mathit{TrPI}_{fundef}(\mathtt{self},\ f\,(X_1,\ldots,X_n)\texttt{->}\ E) \\
\quad := \big(\ f\,(\mathtt{self},\ X_1,\ldots,X_n) = \mathit{TrPI}_{exp}(\mathtt{self},\ E)\ \big)
\end{array}
$$

Expressions:

$$\mathit{TrPI}_{exp}\colon\ \mathit{Name} \times \mathit{Expression} \rightarrow \mathit{Process}$$

- 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

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

$$
\begin{array}{rcl}
\mathit{TrPI}_{arg}(n) & := & \mathtt{unknown} \\
\mathit{TrPI}_{arg}(a) & := & a \\
\mathit{TrPI}_{arg}(X) & := & X \\
\mathit{TrPI}_{arg}(\mathtt{self}\,()) & := & \mathtt{self}
\end{array}
$$

Simple Expressions:

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

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

Send Expression:

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

Receive Expression: an example

$$
\begin{array}{l}
\mathtt{receive} \\
\quad \mathtt{ok\ \texttt{->}\ a;} \\
\quad \mathtt{\{req,\ P\}\ \texttt{->}\ b;} \\
\quad \mathtt{X\ \texttt{->}\ c} \\
\mathtt{end}
\end{array}
\;\overset{\mathit{TrPI}_{exp}}{\longrightarrow}\;
\begin{array}{l}
\mathtt{self(in).[in{=}ok]}\overline{\mathtt{res}}\texttt{<a>.nil} + \\
\mathtt{self(in,P).[in{=}req]}\overline{\mathtt{res}}\texttt{<b>.nil} + \\
\mathtt{self(X).[X\texttt{<>}ok]}\overline{\mathtt{res}}\texttt{<c>.nil}
\end{array}
$$

8 Conclusion

Done:

- Developed a π–calculus model which reflects "essential" behaviour of an Erlang program
- 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.