Modeling Erlang in the Pi–Calculus

Thomas Noll
Lehrstuhl für Informatik 2
RWTH Aachen University
D–52056 Aachen, Germany
noll@cs.rwth-aachen.de

Chanchal Kumar Roy
School of Computing
Dublin City University
Dublin 9, Ireland
Chanchal.Roy@computing.dcu.ie

Abstract

This paper provides a contribution to the formal modeling and verification of programs written in the concurrent functional programming language Erlang, which is designed for telecommunication applications. It presents a mapping of Core Erlang programs into the π–calculus, a process algebra whose name–passing feature allows to represent the mobile aspects of Erlang software in a natural way.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Legal Aspects, Verification

Keywords Functional Programming Languages, Pi–Calculus, Mobile Systems

1. Introduction

In this paper we address the software verification issue in the context of the functional programming language Erlang [1], which was developed by Ericsson corporation to address the complexities of developing large–scale programs within a concurrent and distributed setting. Our interest in this language is twofold. On the one hand, it is often and successfully used in the design and implementation of telecommunication systems. On the other hand, its relatively compact syntax and its clean semantics support the application of formal reasoning methods.

Due to the presence of unbounded data structures, recursive functions, dynamic process spawning, and mobility, Erlang programs usually induce infinite–state systems. It is therefore natural to employ interactive *theorem–proving assistants* such as the Erlang Verification Tool [6, 7] to establish the desired system properties.

Here we follow an alternative approach in which we try to employ fully–automatic *model–checking techniques* [5] to establish correctness properties of communication systems implemented in Erlang. While simulation and testing explore some of the possible executions of a system, model checking conducts an exhaustive exploration of all its behaviors. In this paper we concentrate on the first part of the verification procedure, the construction of the (transition–system) model to be checked.

More concretely, our approach is based on Core Erlang [3], an intermediate language being used in the Erlang compiler which, however, is very close to the original language. We define a translation mapping from Core Erlang into the π–calculus [11], a process–algebraic model of concurrent computation which concentrates on the mobile aspects of systems. This translation exploits a strong connection between Erlang and the π–calculus: the sending of process identifier information between Erlang processes, which allows to implement dynamic mobile systems, directly corresponds to the name–passing feature of the π–calculus. Since message passing in Erlang is asynchronous, we can restrict ourselves to the asynchronous variant of this process algebra. Moreover we represent most data values such as numbers or lists by a special "unknown" value, which eliminates one potential reason for infinite state spaces. However the other sources of possible infinite–state behavior are also present in the π–calculus representation of a Core Erlang system: unbounded mailboxes, recursive function calls, and dynamic process spawning.

By employing tools supporting the π–calculus, such as the Mobility Workbench [12], the HD–Automata Laboratory [8], or Pi2Promela [15], it is possible to automatically derive the transition system of a given Erlang program (at least in the finite–state case). Thereafter model–checking tools such as Truth [9] can be used to automatically verify that the system meets certain requirements given as formulae of some mathematical logic. This, however, is outside the scope of this article.

The remainder of this paper is organized as follows. Section 2 presents the Core Erlang programming language by sketching its syntactic constructs and their intuitive meaning. Section 3 introduces the asynchronous π–calculus. Section 4 constitutes the main part of this paper in which the mapping from Erlang into the π–calculus is studied. Finally Section 5 concludes with some remarks.

2. Core Erlang

Erlang/OTP is a programming platform providing the necessary functionality for programming open distributed (telecommunication) systems: the functional language Erlang with support for communication and concurrency, and the OTP (Open Telecom Platform) middleware providing ready–to–use components (libraries) and services such as e.g. a distributed data base manager, support for "hot code replacement", and design guidelines for using the components.

Today many commercially available products offered by Ericsson are at least partly implemented in Erlang. The software of such products is typically organized into many, relatively small source modules, which at runtime are executed as a dynamically varying number of processes operating in parallel and communicating through asynchronous message passing. The highly concurrent and dynamic nature of such software makes it particularly hard to debug and test by manual methods.

In the following we consider (a fragment of) the core version of the Erlang programming language which has been introduced in [3], and which is used as an intermediate language in the Erlang compiler. It supports the implementation of dynamic networks of processes operating on data types such as atomic constants (atoms), integers, lists, tuples, and process identifiers (pids), using asynchronous, call–by–value communication via unbounded ordered message queues called mailboxes. Full Erlang has several additional features such as distribution of processes (onto nodes), and support for robust programming and for interoperation with non–Erlang code written in, e.g., C or Java.

The syntax of our Core Erlang fragment is defined by the following context–free grammar. For further explanations, please refer to Section 4 and to [3, 4].

$$
\begin{array}{rcl}
\mathit{Module} & ::= & \texttt{module}\ \mathit{atom}\ \texttt{[}\mathit{Fname}_{i_1}\texttt{,}\ldots\texttt{,}\mathit{Fname}_{i_k}\texttt{]} \\
 & & \texttt{attributes}\ \texttt{[}\mathit{atom}_1\texttt{=}\mathit{Const}_1\texttt{,}\ldots\texttt{,} \\
 & & \qquad\qquad \mathit{atom}_m\texttt{=}\mathit{Const}_m\texttt{]} \\
 & & \mathit{Fdef}_1 \ldots \mathit{Fdef}_n \\
\mathit{Fdef} & ::= & \mathit{Fname}\ \texttt{=}\ \mathit{Fun} \\
\mathit{Fname} & ::= & \mathit{atom}\texttt{/}\mathit{integer} \\
\mathit{Const} & ::= & \mathit{Lit} \mid \texttt{[}\mathit{Const}_1\,\texttt{|}\,\mathit{Const}_2\texttt{]} \\
 & \mid & \{\mathit{Const}_1\texttt{,}\ldots\texttt{,}\mathit{Const}_n\} \\
\mathit{Lit} & ::= & \mathit{integer} \mid \mathit{float} \mid \mathit{atom} \mid \mathit{char} \mid \mathit{string} \mid \texttt{[]} \\
\mathit{Fun} & ::= & \texttt{fun}\ \texttt{(}\mathit{var}_1\texttt{,}\ldots\texttt{,}\mathit{var}_n\texttt{)}\ \texttt{->}\ \mathit{Expr} \\
\mathit{Expr} & ::= & \mathit{var} \mid \mathit{Fname} \mid \mathit{Lit} \mid \mathit{Fun} \\
 & \mid & \texttt{[}\mathit{Expr}_1\,\texttt{|}\,\mathit{Expr}_2\texttt{]} \mid \{\mathit{Expr}_1\texttt{,}\ldots\texttt{,}\mathit{Expr}_n\} \\
 & \mid & \texttt{let}\ \mathit{var}\ \texttt{=}\ \mathit{Expr}_1\ \texttt{in}\ \mathit{Expr}_2 \mid \texttt{do}\ \mathit{Expr}_1\ \mathit{Expr}_2 \\
 & \mid & \texttt{apply}\ \mathit{Fname}\texttt{(}\mathit{Expr}_1\texttt{,}\ldots\texttt{,}\mathit{Expr}_n\texttt{)} \\
 & \mid & \texttt{call}\ \mathit{atom}_1\texttt{:}\mathit{atom}_2\texttt{(}\mathit{Expr}_1\texttt{,}\ldots\texttt{,}\mathit{Expr}_n\texttt{)} \\
 & \mid & \texttt{primop}\ \mathit{atom}\texttt{(}\mathit{Expr}_1\texttt{,}\ldots\texttt{,}\mathit{Expr}_n\texttt{)} \\
 & \mid & \texttt{case}\ \mathit{Expr}\ \texttt{of}\ \mathit{Clause}_1 \ldots \mathit{Clause}_n \\
 & \mid & \texttt{receive}\ \mathit{Clause}_1 \ldots \mathit{Clause}_n \\
 & & \texttt{after}\ \mathit{Expr}_1\ \texttt{->}\ \mathit{Expr}_2 \\
\mathit{Clause} & ::= & \mathit{Pat}\ \texttt{when}\ \mathit{Expr}_1\ \texttt{->}\ \mathit{Expr}_2 \\
\mathit{Pat} & ::= & \mathit{var} \mid \mathit{Lit} \mid \texttt{[}\mathit{Pat}_1\,\texttt{|}\,\mathit{Pat}_2\texttt{]} \mid \{\mathit{Pat}_1\texttt{,}\ldots\texttt{,}\mathit{Pat}_n\}
\end{array}
$$

Note that our grammar, in comparison to [3], exhibits some restrictions:

- only ordered sequences of length 1 are considered,
- local recursive function definitions (`letrec`) are omitted,
- only function names are allowed in the function position of an application,
- exception handling (`try`/`catch`) is not supported, and
- patterns of the form var = Pat are missing.

The omission of these constructs greatly simplified the development of an initial version of our π–calculus model. They will probably be considered in future versions of our modeling approach.

Figure 1 presents a short Erlang program, a simplified 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`, it therefore first needs to retrieve this information from `manager`. Having received the pid it sends a simple request to `resource`.

Hence this program represents a simplistic example of a dynamic or mobile system: the required communication channel between `client` and `resource` is not available right from the beginning; it has to be established first by passing the corresponding "handle". Traditional modeling approaches which lack this name–passing capability are therefore bound to fail in this setting.

```
-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.
```

Figure 1. Resource manager in Erlang

Figure 2 shows the essential parts of the Core Erlang module which the Erlang compiler generates as intermediate code from the above program.

3. The Asynchronous Π–Calculus

In contrast to other approaches such as [7, 13, 14], we do not directly construct the transition–system model for the given Core Erlang program. Rather we try to benefit from existing work by first translating it into a specification language for which analysis and verification methods have already been developed. However, as we indicated in the introduction, the dynamic and mobile communication structures which arise in many Erlang applications (such as, in its simplest form, in the resource manager example in Figure 1) a "static" language such as CCS [10] is not suitable for this purpose. What we employ instead is the π–calculus, a process algebra whose name–passing capability allows to represent concurrent systems with dynamically changing communication topologies. More concretely, this feature can be used to model the sending of an Erlang pid to another process in an adequate way.

We will now introduce the syntax of the asynchronous π–calculus, which is parameterized with respect to a set I of agent or process identifiers (represented by $i \in I$) and to a set X of names ($x \in X$) which serve as both communication channels and data to be transmitted along these. The syntactic categories Sys (process systems), Pdef (single process definitions), and Proc (process expressions) are defined by the following grammar.

$$
\begin{array}{rcl}
\mathit{Sys} & ::= & \mathit{Pdef}_1 \ldots \mathit{Pdef}_n \\
\mathit{Pdef} & ::= & i\texttt{(}x_1\texttt{,}\ldots x_n\texttt{)}\ \texttt{=}\ \mathit{Proc} \\
\mathit{Proc} & ::= & \texttt{nil}
\end{array}
$$

```
module 'resmgr' ['start'/0]
attributes []

'start'/0 =
  fun () ->
    let
      Rsr = call 'erlang':'spawn'
                 ('resource', [])
    in
      let
        Mgr = call 'erlang':'spawn'
                   ('manager', [Rsr])
      in  apply 'client'/1(Mgr)

'resource'/0 =
  fun () ->
    receive
      Req when 'true' ->
        'action'
      after 'infinity' ->
        'true'

'manager'/1 =
  fun (Rsr) ->
    receive
      C when 'true' ->
        call 'erlang':'!'(C, Rsr)
      after 'infinity' ->
        'true'

'client'/1 =
  fun (Mgr) ->
    let
      _cor1 = call 'erlang':'self'()
    in
      do
        call 'erlang':'!'(Mgr, _cor1)
        receive
          R when 'true' ->
            call 'erlang':'!'(R, 'request')
          after 'infinity' ->
            'true'
```

Figure 2. Resource manager in Core Erlang

$$
\begin{array}{ll}
| & x_0\,(x_1\,,\ldots,x_n)\,.\,\mathit{Proc}\\
| & \overline{x_0}\texttt{<}x_1\,,\ldots,x_n\texttt{>.nil}\\
| & \mathit{Proc}_1 \parallel \mathit{Proc}_2\\
| & \mathit{Proc}_1 + \mathit{Proc}_2\\
| & (\nu\, x)\,\mathit{Proc}\\
| & \texttt{[}x_1\texttt{=}x_2\texttt{]}\,\mathit{Proc}\\
| & \texttt{[}x_1\texttt{<>}x_2\texttt{]}\,\mathit{Proc}\\
| & i\texttt{<}x_1\,,\ldots,x_n\texttt{>}
\end{array}
$$

Thus a system in the π–calculus is a sequence of one or more process definitions of the form $i(x_1,\ldots,x_n)$ = P where the right–hand side process expression P can have the following forms:

- `nil` is the inactive or deadlocked process; it denotes a process that cannot do anything.
- The input–prefixed process $x_0(x_1,\ldots,x_n).P$ first has to execute its prefix action before being able to continue as P. This action denotes the reception of the names $x_1,\ldots,x_n$ along x_0.
- The asynchronous output process $\overline{x_0}\texttt{<}x_1,\ldots,x_n\texttt{>.nil}$ constitutes the complementary construct. It outputs the names $x_1,\ldots,x_n$ on channel x_0. Note that the output prefix is necessarily followed by the `nil` process: this is exactly the restriction which distinguishes the asynchronous π–calculus from the general form.
- $P_1 \parallel P_2$ denotes the parallel composition, representing the combined behavior of P_1 and P_2 executing concurrently. Here P_1 and P_2 may communicate if one performs an output and the other an input along the same channel. The effect of the communication is the substitution of all (unbound) occurrences of an input name by the corresponding output name, formally:

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

  Note that communication is actually synchronous since the output and the input step are simultaneously executed. However the requirement that the output prefix can only be followed by the `nil` process makes it "non–blocking", i.e., it ensures no other action in the sending process has to wait for the communication to complete. As we will see later, this means that all other activities on the sending side have to be executed in parallel with the output operation.
- A process of the form $P_1 + P_2$ denotes the non–deterministic choice between the two component processes. This construct will later be used to model Erlang's `case` and `receive` expressions.
- $(\nu\, x)P$ means that x is declared as a new name in P, being invisible outside P.
- $[x_1\texttt{=}x_2]\,P$ denotes a process that behaves as P if $x_1 = x_2$, and deadlocks otherwise.
- $[x_1\texttt{<>}x_2]\,P$ defines the opposite behavior, i.e., checks that $x_1 \neq x_2$.
- $i\texttt{<}x_1,\ldots,x_n\texttt{>}$ represents the instantiation of a defined process, and will e.g. be used to model function calls.

The (asynchronous) π–calculus has been given a formal semantics in terms of labeled transition systems. Details can be found e.g. in [11]. Thus, once the π–calculus representation is constructed for a given Core Erlang program, its transition system can be derived using existing tools such as the Mobility Workbench [12], the HD–Automata Laboratory [8], or Pi2Promela [15].

In the following we give a sketch of the π–calculus representation of the resource manager system from Figure 2. In the next section we will see how it can be systematically extracted from the Core Erlang source code.

Upon invocation of the `start/0` function, the `resource` and the `manager` process are spawned, and the initial process runs the `client` function. Thus we have a three–process system of the form

$$
(\nu\,\texttt{Rsr})(\nu\,\texttt{Mgr})(\nu\,\texttt{Client})
\begin{pmatrix}
\texttt{resource/0<Rsr>} \parallel\\
\texttt{manager/1<Rsr,Mgr>} \parallel\\
\texttt{client/1<Mgr,Client>}
\end{pmatrix}
$$

where `Rsr`, `Mgr` and `Client` represent the pids of the respective process, passed as an additional parameter. The single processes are defined by

$$
\begin{array}{l}
\texttt{resource/0(Rsr)}\\
\quad = \;\texttt{Rsr(Req).}\overline{\texttt{action}}\texttt{<>.nil}
\end{array}
$$

$$\begin{aligned}&\texttt{manager/1(Rsr,Mgr)}\\ &\quad = \;\texttt{Mgr(C)}.\overline{\texttt{C}}\texttt{<Rsr>.nil}\\ &\texttt{client/1(Mgr,Client)}\\ &\quad = \;\overline{\texttt{Mgr}}\texttt{<Client>.nil} \parallel \texttt{Client(R)}.\overline{\texttt{R}}\texttt{<request>.nil}\end{aligned}$$

Note that the initial asynchronous request to `manager` in `client` involves a parallel composition in the corresponding π–calculus version.

In the first transition of the system, `client` asks `manager` for the handle to `resource`. The resulting state is of the form

$$(\nu\,\texttt{Rsr})(\nu\,\texttt{Mgr})(\nu\,\texttt{Client})\begin{pmatrix}\overline{\texttt{resource}}\texttt{/0<Rsr>} \parallel \\ \overline{\texttt{Client}}\texttt{<Rsr>.nil} \parallel \\ \texttt{nil} \parallel \texttt{Client(R)}.\overline{\texttt{R}}\texttt{<request>.nil}\end{pmatrix}$$

Process `client` is now enabled to receive the pid of `resource`:

$$(\nu\,\texttt{Rsr})(\nu\,\texttt{Mgr})(\nu\,\texttt{Client})\begin{pmatrix}\texttt{resource/0<Rsr>} \parallel \\ \texttt{nil} \parallel \\ \texttt{nil} \parallel \overline{\texttt{Rsr}}\texttt{<request>.nil}\end{pmatrix}$$

Now `client` can send the actual request to `resource`. Note that this was not possible before since the corresponding pid (`Rsr`) was not known to `client`.

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

4. The Mapping

We will now formally explain the mapping by which Core Erlang programs are translated into π–calculus processes. This translation involves a series of functions, one for each syntactic category of the source language. Here we follow a top–down approach, starting with Core Erlang modules which are mapped into systems of process definitions. Thus the main function is of the type

$$\mathit{trans}_M[\![\cdot]\!] : \mathit{Module} \to \mathit{Sys}.$$

It ignores the exported functions and the module attributes and just associates a process definition with every function declaration:

$$\begin{aligned}&\mathit{trans}_M[\![\texttt{module}\ a\ \texttt{[...]}\ \texttt{attributes}\ \texttt{[...]}\ \mathit{fd}_1 \dots \mathit{fd}_n]\!]\\ &\qquad := \quad \mathit{trans}_F[\![\mathit{fd}_1]\!] \dots \mathit{trans}_F[\![\mathit{fd}_n]\!]\end{aligned}$$

This is accomplished by the following function which reuses the name of the Core Erlang function as the name of the associated process identifier, and also keeps the list of parameter variables. Moreover it adds two special names, `self` and `res`, which represent the Core Erlang pid of the process evaluating the respective function (that is, the value of `self()`) and the name which is used as a return channel for the result of the computation, respectively.

$$\mathit{trans}_F[\![\cdot]\!] : \mathit{Fdef} \to \mathit{Pdef}$$

$$\begin{aligned}&\mathit{trans}_F[\![f\ \texttt{=}\ \texttt{fun}\ \texttt{(}v_1\texttt{,}\dots\texttt{,}v_n\texttt{)}\ \texttt{->}\ e]\!]\\ &\quad := \quad f\texttt{(}v_1\texttt{,}\dots\texttt{,}v_n\texttt{,self,res)}\ \texttt{=}\ \mathit{trans}_E[\![e]\!]\end{aligned}$$

We now come to the central part of the translation. Since the actual task of a Core Erlang process is to evaluate an expression, the representation of this concept is crucial for our modeling approach. Here we use the following invariants:

- Every Core Erlang expression is represented by a π–calculus process:
$$\mathit{trans}_E[\![\cdot]\!] : \mathit{Expr} \to \mathit{Proc}.$$
- Upon termination of this process, the value of the corresponding expression is passed along the distinguished result channel called `res`.
- The translation abstracts from data structures such as numbers, characters, strings, lists, or tuples. All these items are represented by another special name which is denoted `unknown`.
- Higher–order features such as function names or `fun` expressions in arbitrary expressions are currently not supported.

We start with the translation of variables, literals, lists, and tuples. Here v denotes a variable, f a function name, z a number (integer or float), a an atom, c a character, s a string, and e (possibly indexed) an expression. As explained above, all data structures but atoms are represented by the name `unknown`. Atoms as well as Core Erlang variables are taken identically as π–calculus names. Since the value of the expression is passed along the `res` channel, the only thing we have to do here is to provide the corresponding output operation.

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

A `let` expression binds the given variable to the value of the first expression, e_1, and evaluates the second expression e_2 with respect to that binding. In the π–calculus representation this is modeled by passing the value of e_1 to the process simulating e_2, using a new result channel $\texttt{res}'$ and binding the value to the name x:

$$\begin{aligned}&\mathit{trans}_E[\![\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!]\\ &\quad := \quad (\nu\,\texttt{res}')(\mathit{trans}_E[\![e_1]\!] \parallel \texttt{res}'(x)\,.\,\mathit{trans}_E[\![e_2]\!])\end{aligned}$$

The sequencing expression `do` has a similar effect, but without passing any information to the second expression. Thus we just need to introduce a dummy name, `dummy`, which consumes the value produced in the computation of the first expression.

$$\begin{aligned}&\mathit{trans}_E[\![\texttt{do}\ e_1\ e_2]\!]\\ &\quad := \quad (\nu\,\texttt{res}')(\mathit{trans}_E[\![e_1]\!] \parallel \texttt{res}'\texttt{(dummy)}\,.\,\mathit{trans}_E[\![e_2]\!])\end{aligned}$$

The next part of the specification deals with function calls. Local calls (i.e., applications) are directly translated into process calls where again the additional `self` and `res` parameters have to be given.

$$\begin{aligned}&\mathit{trans}_E[\![\texttt{apply}\ f\texttt{(}e_1\texttt{,}\dots\texttt{,}e_n\texttt{)}]\!]\\ &\quad := \quad f\texttt{<}\mathit{trans}_P[\![e_1]\!]\texttt{,}\dots\texttt{,}\mathit{trans}_P[\![e_n]\!]\texttt{,self,res>}\end{aligned}$$

The only remote calls which are currently supported are asynchronous output and process spawning operations, and the `self()` call. The first can directly be represented by an output operation where the concept of asynchronicity requires the this operation to be executed concurrently with the remainder process. According to our translation invariants, the value of the sending expression, which is the value of the message, has to be provided on the `res` channel.

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

The simulation of a `spawn` operation is more involved. It basically splits control into two threads, the first evaluating the argument function and the second resuming the computation of the spawning process. Here we introduce a third concurrent component which just consumes the computation result of the spawned process, which is no longer needed. The overall result is the new Core Erlang pid of the spawned process which is represented by a new name $\texttt{self}'$.

$$\begin{aligned}
&\mathit{trans}_E[\![\texttt{call 'erlang':'spawn'(}a\texttt{, [}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{])}]\!]\\
&\quad:= \quad (\nu\,\texttt{self}')\\
&\qquad\left(\begin{array}{l}(\nu\,\texttt{res}')\\ \left(\begin{array}{l}a\texttt{<}\mathit{trans}_P[\![e_1]\!]\texttt{,}\ldots\texttt{,}\mathit{trans}_P[\![e_n]\!]\texttt{,self}'\texttt{,res}'\texttt{>}\ \|\\ \texttt{res}'\texttt{(dummy).nil}\end{array}\right)\ \|\\ \overline{\texttt{res}}\texttt{<self}'\texttt{>.nil}\end{array}\right)
\end{aligned}$$

The last type of remote call, `self()`, can simply be handled by using the `self` parameter.

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

Primitive operations are currently not supported; we just abstract from the result by employing the `unknown` name.

$$\begin{aligned}
&\mathit{trans}_E[\![\texttt{primop}\ a\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}]\!]\\
&\quad:= \quad \overline{\texttt{res}}\texttt{<unknown>.nil}
\end{aligned}$$

The translation of a branching expression introduces another kind of abstraction: the choice of a matching clause, which is deterministic in Core Erlang, is implemented by the sum operator and non–deterministic therefore.

A `case` expression is translated by first evaluation the selector expression, e, and by passing the result to the processes implementing the single clauses. Here a trick is used to unify the handling of `case` and `receive`: the translation function for clauses, $\mathit{trans}_C[\![\cdot]\!]$, is equipped with an additional name parameter which identifies the channel on which the value is expected. This is the result channel for e in the first case and the `self` channel in the second case.

$$\begin{aligned}
&\mathit{trans}_E[\![\texttt{case}\ e\ \texttt{of}\ c_1\ldots c_n]\!]\\
&\quad:= \quad (\nu\,\texttt{res}')(\mathit{trans}_E[\![e]\!]\ \|\ (\mathit{trans}_C[\![c_1]\!](\texttt{res}') + \ldots +\\
&\qquad\qquad\qquad \mathit{trans}_C[\![c_n]\!](\texttt{res}')))
\end{aligned}$$

Note that, in contrast to the original semantics of Core Erlang, the translation of the `case` construct involves non–determinism: if the patterns used in the clauses overlap, then any match (and not necessarily the first one) can be chosen. This simplifies our translation mapping; a more refined version employing mismatch operations is currently under development.

In the translation of `receive`, first the timeout value e_1 is evaluated. If this value is `infinity`, then the timeout cannot occur, and e_2 can be ignored therefore. Otherwise the evaluation of e_2 has to be added as one of the possible choices.

$$\begin{aligned}
&\mathit{trans}_E[\![\texttt{receive}\ c_1\ldots c_n\ \texttt{after}\ e_1\ \texttt{->}\ e_2]\!]\\
&\quad:= \quad (\nu\,\texttt{res}')\\
&\qquad\left(\begin{array}{l}\mathit{trans}_E[\![e_1]\!]\ \|\\ \texttt{res}'\texttt{(dummy).}\\ \left(\begin{array}{l}\texttt{[dummy=infinity]}\,(\mathit{trans}_C[\![c_1]\!](\texttt{self}) + \ldots +\\ \qquad \mathit{trans}_C[\![c_n]\!](\texttt{self})) +\\ \texttt{[dummy<>infinity]}\,(\mathit{trans}_C[\![c_1]\!](\texttt{self}) + \ldots +\\ \qquad \mathit{trans}_C[\![c_n]\!](\texttt{self}) +\\ \qquad \mathit{trans}_E[\![e_2]\!])\end{array}\right)\end{array}\right)
\end{aligned}$$

The next syntactic category to consider are clauses. They are handled by the translation mapping

$$\mathit{trans}_F[\![\cdot]\!] : \mathit{Clause} \times X \to \mathit{Proc}.$$

This mapping first distinguishes whether the given pattern is a variable. If so, it establishes the binding by setting up an appropriate input action. Otherwise it tries to match the input value against the pattern p, using the auxiliary function $\mathit{trans}_P[\![\cdot]\!]$.

The only non–`true` guard which is supported so far is a comparison of the form `v1 =:= v2`, which is used by the Erlang compiler to implement matching against bound variables. It is directly translated into a corresponding matching process.

$$\begin{aligned}
&\mathit{trans}_C[\![v\ \texttt{when 'true' ->}\ e]\!](x)\\
&\qquad := \quad x\texttt{(}v\texttt{)}\,\texttt{.}\,\mathit{trans}_E[\![e]\!]\\
&\mathit{trans}_C[\![p\ \texttt{when 'true' ->}\ e]\!](x)\\
&\qquad := \quad x\texttt{(dummy).[dummy=}\mathit{trans}_P[\![p]\!]\texttt{]}\,\mathit{trans}_E[\![e]\!]\\
&\mathit{trans}_C[\![v_1\ \texttt{when call 'erlang':'=:='(}v_1\texttt{,}v_2\texttt{) ->}\ e]\!](x)\\
&\qquad := \quad x\texttt{(}v_1\texttt{).[}v_1\texttt{=}v_2\texttt{]}\,\mathit{trans}_E[\![e]\!]
\end{aligned}$$

The auxiliary function $\mathit{trans}_P[\![\cdot]\!]$ is used to implement pattern matching, and is also employed in the translation of function arguments. It has the type

$$\mathit{trans}_P[\![\cdot]\!] : \mathit{Pat} \to \mathit{Proc}$$

and is defined by the following clauses:

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

This completes the definition of the translation mapping.

Figure 3 shows the π–calculus representation of the resource manager system which is obtained from the Core Erlang module in Figure 2. It employs the syntax of the Mobility Workbench [12], which differs from that introduced in Section 3 with regard to the following aspects.

- All process identifiers start with uppercase letters while (variable and atom) names start with lowercase letters;
- every function name of the form a`/`n is converted into a process identifier of the form a_n;
- the inactive process is denoted by `0`;
- an output channel is marked by a leading apostrophe, not by overlining;
- parallel composition of processes is denoted by `|`; and
- the "new name" operator, ν, is written as `^`, and can be followed by arbitrarily many names.

To simplify the presentation we have omitted the timeout clauses of `receive` expressions, which are all equipped with an `infinity` value. Moreover a new process is introduced, `Main`, which declares the parameters of the `Start_0` process (`self`, `res`) and the atom names used in the system (`unknown`, `action`, `request`), and then sets up the initial call to `Start_0`.

5. Conclusion

In this paper we have introduced the initial version of a process–algebraic model for the functional programming language (Core) Erlang. As the process algebra we have chosen the π–calculus, which is is particularly appropriate for specifying mobile systems

```
agent Main =
  (^self,res,unknown,action,request)
    Start_0<self,res>

agent Start_0(self,res) =
  (^res1)(
    (^self1)(
      (^res2)(
        Resource_0<self1,res2> |
        res2(dummy).0
      ) |
      'res1<self1>.0
    ) |
    res1(rsr).
      (^res1)(
        (^self2)(
          (^res2)(
            Manager_1<rsr,self2,res2> |
            res2(dummy).0
          ) |
          'res1<self2>.0
        ) |
        res1(mgr).Client_1<mgr,self,res>
      )
  )

agent Resource_0(self,res) =
  self(req).'res<action>.0

agent Manager_1(rsr,self,res) =
  self(c).('c<rsr>.0 | 'res<rsr>.0)

agent Client_1(mgr,self,res) =
  (^res1)(
    'res1<self>.0 |
    res1(_cor1).
      (^res1)(
        ('mgr<_cor1>.0 | 'res1<_cor1>.0) |
        res1(dummy).self(r).
          ('r<request>.0 | 'res<request>.0)
      )
  )
```

Figure 3. Resource manager in π–calculus

with dynamically changing communication topologies. We have seen that its name–passing feature makes it superior to other approaches which lack this capability since it allows to directly represent the sending of process identifier information between Erlang processes. This was not possible with previous attempts which were undertaken to define process–algebraic models for Erlang such as the work based on Promela/SPIN [17] or μCRL [2].

As indicated in the paper, our model abstracts from the concrete behavior of the Core Erlang system with regard to several issues. Our current work focuses on removing some of these abstractions. A first attempt in this direction has been undertaken in [16] where a refined modeling of several data structures such as lists or tuples is studied. Moreover [16] presents a way to replace the non–deterministic choice between the clauses of a `case` or `receive` expression by a deterministic top–down strategy.

References

[1] J. Armstrong, S. Virding, M. Williams, and C. Wikström. *Concurrent Programming in Erlang*. Prentice Hall International, 2nd edition, 1996.

[2] T. Arts, C. Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *International Journal on Software Tools for Technology Transfer*, 5:205–220, 2004.

[3] R. Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI '01 Erlang Workshop*, 2001. `http://www.it.uu.se/research/group/hipe/corerl/doc/cerl_intro.ps`.

[4] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0.3 language specification. Technical report, Uppsala University, Sweden, 2004. `http://www.it.uu.se/research/group/hipe/corerl/doc/core_erlang-1.0.3.pdf`.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[6] L. Fredlund, D. Gurov, and T. Noll. Semi–automated verification of Erlang code. In *16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 319–323. IEEE Computer Society Press, 2001.

[7] L.-Å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *International Journal on Software Tools for Technology Transfer*, 4(4):405–420, 2002.

[8] The HD–Automata Laboratory. `http://fmt.isti.cnr.it:8080/hal/`.

[9] M. Lange, M. Leucker, T. Noll, and S. Tobies. Truth – a verification platform for concurrent systems. In *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science, pages 150–159. Springer–Verlag Wien, 1999.

[10] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice–Hall, 1989.

[11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

[12] The Mobility Workbench. `http://www.it.uu.se/research/group/mobility/mwb/`.

[13] T. Noll. A rewriting logic implementation of Erlang. In *Proceedings of First Workshop on Language Descriptions, Tools and Applications (ETAPS/LDTA'01)*, volume 44(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[14] T. Noll. Equational abstractions for model checking erlang programs. In *Proceedings of the International Workshop on Software Verification and Validation (SVV 2003)*, volume 118 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.

[15] Pi2Promela. `http://lcs.ios.ac.cn/~wp/pi2pro.html`.

[16] C. Roy. Modelling programming languages for concurrent and distributed systems in specification languages. Master's thesis, RWTH Aachen University, Germany, 2004. http://www-i2.informatik.rwth-aachen.de/Staff/Current/noll/Teaching/Roy/.

[17] C. Wiklander. Verification of Erlang programs using SPIN. Technical report, Department of Teleinformatics, Royal Institute of Technology, Stockholm, Sweden, 1999.