Towards Automatic Verification of Erlang Programs by π–Calculus Translation

Chanchal Kumar Roy
School of Computing
Queen's University
Kingston, ON, Canada K7L 3N6
croy@cs.queensu.ca

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

Banani Roy
School of Computing
Queen's University
Kingston, ON, Canada K7L 3N6
broy@cs.queensu.ca

James R. Cordy
School of Computing
Queen's University
Kingston, ON, Canada K7L 3N6
cordy@cs.queensu.ca

Abstract

ERLANG is a concurrent, dynamically typed, distributed, purely functional programming language with non-purely functional libraries that is mainly employed in telecommunication systems. This paper provides a contribution to the formal modeling and verification of programs written in Erlang. It 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.

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, Verification

Keywords Functional Programming Languages, π–Calculus, Mobile Systems, Translation Mapping

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 in a concurrent and distributed setting. Our interest in this language is twofold. On the one hand, Erlang is often used in the design and implementation of real production telecommunication systems. On the other hand, its compact syntax and clean semantics make it particularly amenable to the application of formal reasoning methods.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Erlang'06 September 16, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-490-1/06/0009…$5.00.

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 desired system properties.

In this work we follow an alternative approach in which we apply 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.

Our approach is based on a subset of Erlang, called PIErlang, which is very close to the original language. We define a translation mapping from PIErlang to the π–calculus [11], a process–algebraic model of concurrent computation which concentrates on the mobile aspects of systems. This work is an improvement over our previous work [18] where an initial translation mapping from Core Erlang [3] to π–calculus was presented with limitations. In that work we approximated matches in `case` and `receive` expressions using non–deterministic choices, and tuple based communications were not modeled at all. In this paper we provide a detailed treatment of the larger subset PIErlang using a translation mapping that deals with both of these issues.

Our new translation exploits the strong connection between Erlang and the π–calculus in a number of ways. The sending of process identifier information between Erlang processes, which supports implementation of 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 the process algebra. We represent most data values such as numbers by a special value/name `unknown`, which eliminates one potential reason for infinite state spaces. However, other potential sources of infinite–state behavior, such as unbounded mailboxes, recursive function calls and dynamic process spawning, are still present in the π–calculus representation.

By employing tools supporting the π–calculus, such as the Mobility Workbench [12], the HD–Automata Laboratory [8], Pi2Promela [15] or Spatial Logic Model Checker (SLMC) [19], it

is possible to automatically derive the transition system of a given Erlang program (at least in the finite–state case) and/or to apply the model–checking features provided by these tools. Model–checking tools such as Truth [9] can also be used to automatically verify that the system meets certain requirements expressed as logical formulae once the transition system is derived for a given Erlang program. This, however, is outside the scope of this article.

The remainder of this paper is organized as follows. Section 2 presents the PIErlang, a subset of the Erlang programming language by sketching its syntactic constructs. Section 3 introduces the polyadic asynchronous π–calculus. Section 4 presents the basic mapping from Erlang into the monadic π–calculus. In Section 5 we present a method for avoiding non–determinism in the representation of matches of `receive` and `case` expressions. Section 6 provides a detailed treatment of modeling non-nested tuples with polyadic communications. A case study with the simplified *Resource Manager* example is undertaken in Section 7. Based on several other case studies, a further refinements to `match` and `send` expressions is provided in Section 8. Section 9 discusses related work in the area of automated verification of Erlang programs, and finally Section 10 concludes the paper by pointing out the limitations of our study.

2. Erlang

Erlang/OTP is a platform for programming open distributed telecommunication systems. It consists of 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 a distributed database manager, support for "hot code replacement", and design guidelines for using the components.

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 difficult to debug and test using manual methods.

In Figure 1, we consider a subset of the Erlang programming language called PIErlang. It supports the implementation of dynamic networks of processes operating on data types such as atomic constants (atoms), integers, 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 higher–order functions, list comprehensions, distribution of processes (onto nodes), and support for robust programming and for interoperation with non–Erlang code written in, e.g., C or Java. Each of the syntactic constructs of PIErlang will be explained intuitively in the following sections in connection with the translation mapping. For further details, please refer to [1, 16].

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 Erlang program. Rather we leverage on existing work by first translating from Erlang to 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 5) make a "static" language such as CCS [10] unsuitable for this purpose. What we employ instead is the π–calculus, a process algebra whose name–passing capability allows for representation

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

where the usual meaning of different symbols are as follows:

$$
\begin{array}{rl}
\mathit{Fdef}\texttt{,}\mathit{Fdef}_1\texttt{,}\ldots\texttt{,}\mathit{Fdef}_n & \in \texttt{Function Definitions} \\
E\texttt{,}E_1\texttt{,}\ldots\texttt{,}E_n & \in \texttt{Expressions} \\
M\texttt{,}M_1\texttt{,}\ldots\texttt{,}M_n & \in \texttt{Matches} \\
P\texttt{,}P_1\texttt{,}\ldots\texttt{,}P_n & \in \texttt{Patterns} \\
A\texttt{,}A_1\texttt{,}\ldots\texttt{,}A_n & \in \texttt{Arguments} \\
a\texttt{,}f & \in \texttt{Atoms} \\
X\texttt{,}X_1\texttt{,}\ldots\texttt{,}X_n & \in \texttt{Variables}
\end{array}
$$

Figure 1. PIErlang Syntax

$$
\begin{array}{rcl}
\mathit{Sys} & ::= & \mathit{Pdef}_1 \ldots \mathit{Pdef}_n \\
\mathit{Pdef} & ::= & i\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{) = }\mathit{Proc} \\
\mathit{Proc} & ::= & \underline{\texttt{nil}} \\
 & \mid & x\texttt{(}z_1\texttt{,}\ldots\texttt{,}z_n\texttt{)}\,.\,\mathit{Proc} \\
 & \mid & \overline{x}\texttt{<}y_1\texttt{,}\ldots\texttt{,}y_n\texttt{>.}\underline{\texttt{nil}} \\
 & \mid & \mathit{Proc}_1 \parallel \mathit{Proc}_2 \\
 & \mid & \mathit{Proc}_1 + \mathit{Proc}_2 \\
 & \mid & \texttt{(}\underline{\texttt{new}}\ x\texttt{)}\,\mathit{Proc} \\
 & \mid & \texttt{[}x\texttt{=}y\texttt{]}\,\mathit{Proc} \\
 & \mid & \texttt{[}x\texttt{<>}y\texttt{]}\,\mathit{Proc} \\
 & \mid & i\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)}
\end{array}
$$

Figure 2. The syntax of the asynchronous π–calculus.

of 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.

We first 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, y, z etc.) which 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 of Figure 2. A system in the π–calculus is a sequence of one or more process definitions of the form $i\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{) = }\mathit{Proc}$ where the right–hand side process expression Proc can have the following forms:

- $\underline{\texttt{nil}}$ is an inactive or deadlock process – that is, a process that can do nothing.
- The input–prefixed process $x\texttt{(}z_1\texttt{,}\ldots\texttt{,}z_n\texttt{)}\,.\,\mathit{Proc}$ has a single capability, expressed by $x\texttt{(}z_1\texttt{,}\ldots\texttt{,}z_n\texttt{)}$. The process Proc can-

not proceed until the capability has been exercised. It is a positive prefix where x is the input port of an agent. It binds the names $z_1, \ldots, z_n$ and denotes their reception along x.

- $\overline{x}$<$y_1, \ldots, y_n$>.nil is an asynchronous output process. The prefix $\overline{x}$<$y_1, \ldots, y_n$> is a negative prefix. $\overline{x}$ can be thought of as an output port of an agent which contains it. This process outputs names $y_1, \ldots, y_n$ on port x then behaves like nil that can do nothing. We are using asynchronous π–calculus, and therefore, only nil can follow an output action. This is exactly the restriction that distinguishes the asynchronous π–calculus from the general form.
- A parallel composition $Proc_1 \parallel Proc_2$, represents the combined behavior of $Proc_1$, and $Proc_2$ executing in parallel. The components of $Proc_1$ and $Proc_2$ can act independently, and may also communicate if one performs an output and the other an input along the same port. The effect of the communication is the substitution of all (unbound) occurrences of an input name by the corresponding output name, or formally (a variant of the reaction rule):

$$\overline{x}\langle y_1, \ldots, y_n \rangle.\texttt{nil} \parallel x(z_1, \ldots, z_n).Proc \\ \rightarrow \quad Proc[z_1 \mapsto y_1, \ldots, z_n \mapsto y_n]$$

  where $Proc[z \mapsto y]$ denotes the replacement of every free occurrence of z in $Proc$ by y. Note that communication is actually synchronous since the output and the input steps 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 $Proc_1 + Proc_2$ represents a non–deterministic choice between the two component processes, that is, it non–deterministically executes either process $Proc_1$ or process $Proc_2$. This construct will later be used to model Erlang's `case` and `receive` expressions.
- (new x) $Proc$ means that x is declared as a new name local to process $Proc$ and is bound in $Proc$. It is not visible outside $Proc$.
- Matching [x=y] $Proc$ denotes a process that behaves as $Proc$ if $x = y$, and deadlocks otherwise.
- Mismatching [x <> y] $Proc$ defines the opposite behavior of matching, i.e. checks that $x <> y$.
- $i(x_1, \ldots, x_n)$ represents the instantiation of a defined process and will be used to model function calls.
- $i(x_1, \ldots, x_n)$ = $Proc$ (where $i \neq j \Rightarrow x_i \neq x_j$) represents the declaration of a process i in terms of process $Proc$. One can think of it as a procedure declaration in traditional procedural programming.

The (asynchronous) π–calculus has been given a formal semantics in terms of labeled transition systems [11]. Thus, once the π–calculus representation is constructed for a given Erlang program, its transition system can be derived using existing tools such as the Mobility Workbench [12], the HD–Automata Laboratory [8], Pi2Promela [15] or SLMC [19], allowing for use of the model–checking features of these tools.

4. Translating PIErlang Programs to the π–Calculus

We will now formally explain the mapping by which Erlang programs are translated into π–calculus. This translation uses a series of functions, one for each syntactic category of the source language. Here we follow a bottom–up approach, starting with Erlang's data types (e.g. atoms and numbers) and variables. This differs from our previous work [18] where a top–down approach was followed. We reuse some of the mappings of [18] with a detailed explanation and clarification behind the mapping. For further details the reader is referred to [16]. In this section the syntactic constructs related to tuples are not yet presented, in order to keep the mapping simple. Tuple based mappings are discussed in Section 6.

The π–calculus is a canonical process algebra especially designed for mobile concurrent computation in a very abstract way. It is a name–passing calculus that allows for the description of concurrent and distributed systems with dynamically changing interaction topology. Name communication together with the possibility of declaring and exporting local names (scope extrusion) gives this calculus great expressive power.

Because Erlang is a concurrent and distributed programming language, processes in Erlang software tend to have a complicated communication structure. Erlang processes manage concurrency through asynchronous message passing, using PIDs as the links of communication. This can be captured directly using the name passing feature of the asynchronous π–calculus. Erlang's message passing primitives have another promising feature: they can pass PIDs as messages to other process in communication. The π–calculus achieves this goal using the same idea: passing channels as messages in communication. Indirect reference and dynamic creation of new processes plays a prominent role in interaction between processes in Erlang. For instance, one process can create a new child process and then send the PID of the newly created process to a second process by using asynchronous message passing. In Erlang, links serve as both communication channel and PID. Such interactions are typically hard to model using communicating finite state machines. The asynchronous π–calculus provides a simple way to model such interactions. In π–calculus, links are primitive names of communication channels. The combination of fresh name generation (new name creation), and private channel (fresh name) passing (scope extrusion) allows faithful modeling of several complicated communication patterns between software agents.

The above striking similarities between Erlang and the π–calculus lead us to the idea of performing Erlang verification tasks using the π–calculus. Given the asynchronous message–passing nature of Erlang, we are particularly interested in checking correctness/behavioral properties (like deadlock freedom, invariant checking, message understood properties) of Erlang programs in a way similar to the Behave Project [25] of Microsoft Research. For example, the desirable correctness/behavioral properties of a simplified *Resource Locker* [26] system are: the `locker` should always be enabled to receive a new request or release (*no deadlock*), no two `clients` should gain access to the same `resource` at the same time (*mutual exclusion*), and all `clients` should eventually get their demanded access (*no starvation*). These behavioral/correctness properties can be checked using existing tools [8, 9, 12, 15, 19]. However, in order to use these tools, one must obtain a π-model for the given program. In order to obtain such a π–model for Erlang programs, several functions are defined for the mapping of Erlang programs to the π–calculus, as follows:

Two frequently used functions for translation mapping are $TrPI_{arg}$, and $TrPI_{exp}$. $TrPI_{arg}$ is used to translate any *Argument* of PIErlang. Its signature is:

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

This signature indicates that $TrPI_{arg}$ can translate any Number, Atom, Variable or the built–in–function `self()` to a name into the π–calculus when they are used as the arguments of `function calls`, `spawn calls`, and `send` expressions. This is also used when working with patterns of `receive` and `case` expressions.

$TrPI_{exp}$ is used to translate any expression of PIErlang to the π–calculus. It has the following signature:

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

This signature indicates that $TrPI_{exp}$ will take a name (normally the PID for the corresponding expression) and an Erlang expression as input and will produce a process as output. We will discuss both of these functions in the following subsections.

A number of other functions, such as $TrPI_{match}$, $TrPI_{fundef}$, and $TrPI_{prog}$, are used during the translation procedure. Each of them will also be discussed in detail below.

We use some global conventions during translation mapping. These are as follows:

- When the evaluation of the process $TrPI_{exp}$`(self,` E`)` is terminated, it sends the value of the expression E along some distinguishable channel. Normally, channel `res` is used for this purpose. However, several case studies have shown that for some expressions passing the evaluation result along `res` can be omitted.
- The evaluation result of a function `fun` will be sent along the `fun_res` channel.
- Name `dummy` will frequently be used to receive the evaluation result of an expression and will later be discarded. While working with tuples we will use `dummy1`, `dummy2`,..., `dummyn` for dummy communications.

We now come to the central part of our translation mapping. In the following we consider each of the syntactic constructs of PIErlang and define a mapping to the π–calculus. We begin with the data types of PIErlang.

4.1 Data Types

PIErlang has two simple data types, numbers (Integer and Float) and atoms. We define a corresponding mapping in the π–calculus for each of them. Although the π–calculus has only names, not constants, we must nevertheless provide a translation for any Erlang integer number. Thus we have decided not to represent constants in π–calculus semantically, rather, we have decided that an integer number should be represented by the `unknown` name in the π–calculus. In PIErlang, an integer number can be used as an argument or as an expression and so there are two kinds of mappings to the π–calculus for integer numbers.

When an integer is used as an `Argument` (cf. Figure 1) we use a direct mapping to `unknown` in the π–calculus, formally written as:

$$TrPI_{arg}(n) \quad := \quad \texttt{unknown}$$

This mapping says that $TrPI_{arg}$ takes any number as input and produces `unknown`, a name in the π–calculus as output.

When an integer is used as an expression, it is likely that it will be used by subsequent expression(s). Having this in mind, we introduce a new global name `res` in the π–calculus to store the results of such atomic expressions. This is done by sending the integer number as the argument of the `send` action of the π–calculus along the `res` channel. This can be formulated as follows:

$$\begin{aligned} TrPI_{exp}(\texttt{self},\ n) \quad &:= \quad \overline{\texttt{res}}\texttt{<}TrPI_{arg}(n)\texttt{>.}\underline{\texttt{nil}} \\ &= \quad \overline{\texttt{res}}\texttt{<unknown>.}\underline{\texttt{nil}} \end{aligned}$$

A similar mapping is defined for floating point numbers.

Atoms are constant names in Erlang. The value of an atom is its name, and two atoms are equivalent only when they have identical names. Mapping of atoms to the π–calculus can be done directly by name. We translate any atom in PIErlang to a new global name in the π–calculus with the same spelling and context without any changes. As for numbers, the translation is divided into two cases depending on the context where the atom is used.

$$\begin{aligned} TrPI_{arg}(a) \quad &:= \quad \texttt{a} \\ TrPI_{exp}(\texttt{self},\ a) \quad &:= \quad \overline{\texttt{res}}\texttt{<}TrPI_{arg}(a)\texttt{>.}\underline{\texttt{nil}} \\ &= \quad \overline{\texttt{res}}\texttt{<a>.}\underline{\texttt{nil}} \end{aligned}$$

4.2 Variables

Erlang variables are untyped, and a variable can be bound to any term. The scope of a variable extends from its first appearance in a clause through to the end of the clause in an Erlang function. After analyzing the characteristics of Erlang variables, we have found that any variable in Erlang can be translated to a name in π–calculus with the same spelling and context. However, like numbers and atoms, variables also have two different mappings depending on whether they appear as an argument or as an expression.

$$\begin{aligned} TrPI_{arg}(X) \quad &:= \quad \texttt{X} \\ TrPI_{exp}(\texttt{self},\ X) \quad &:= \quad \overline{\texttt{res}}\texttt{<}TrPI_{arg}(X)\texttt{>.}\underline{\texttt{nil}} \\ &= \quad \overline{\texttt{res}}\texttt{<X>.}\underline{\texttt{nil}} \end{aligned}$$

Erlang variables normally begin with an uppercase letter. Uppercase names are allowed in the π–calculus, and uppercase and lowercase names are distinct names with same spelling. For example `X` and `x` are two distinct names in the π–calculus.

4.3 Built-in Function `self ()`

The built-in function `self()` is another interesting construct which is frequently used in Erlang to represent the current PID of the process executing the expression. This function is used as an argument for `function calls`, `send`, `receive` and `case` expressions. It can also be used as an expression. The mappings are as follows:

$$\begin{aligned} TrPI_{arg}(\texttt{self()}) \quad &:= \quad \texttt{self} \\ TrPI_{exp}(\texttt{self},\ \texttt{self()}) \quad &:= \quad \overline{\texttt{res}}\texttt{<}TrPI_{arg}(\texttt{self()})\texttt{>.}\underline{\texttt{nil}} \\ &= \quad \overline{\texttt{res}}\texttt{<self>.}\underline{\texttt{nil}} \end{aligned}$$

Here `self` in the π–calculus is not a new name, but is a predefined global referring to the PID of the process executing the expression where it is used.

4.4 Sequence of Expressions

An expression can also be the composition of two independent sub–expressions. This is formally denoted as E_1 , E_2 where the evaluation of E_2 is independent of evaluation of E_1 but E_2 will be evaluated only after evaluation of E_1. The corresponding π–calculus mapping of such sequences of expressions is formed as follows:

$$\begin{aligned} &TrPI_{exp}(\texttt{self},\ E_1\texttt{,}\ E_2) \\ &:= (\underline{\texttt{new}}\ \texttt{res}') \begin{pmatrix} TrPI_{exp}(\texttt{self},\ E_1) \\ \quad \| \\ \texttt{res}'(\texttt{dummy}).TrPI_{exp}(\texttt{self},\ E_2) \end{pmatrix} \end{aligned}$$

The π–calculus representation makes it clear that expression E_1 must be evaluated first. In the 2nd process, we see that there is a *receive action* along a new result channel `res`$'$ before evaluating expression E_2. This *receive action* cannot be performed without evaluation of E_1, as E_1 is the only candidate that can send its result along channel `res`$'$. As soon as E_1 is evaluated, and its result is passed through channel `res`$'$, *receive action* `res`$'$`(dummy)` will be performed using the reaction rule of the π–calculus and then evaluation of E_2 will be started, thus forcing E_2 to be evaluated after E_1. Note that the received value (in `dummy`) is discarded as it is not needed in evaluation of E_2. The global name `self` is included in the input parameters list along with the expression in order to provide the PID of the process that is currently executing the expression.

4.5 Assignment Expressions

PIErlang has two forms of assignment expression, X `=` E_1 `,` E_2 where variable X can be used in expression E_2, and X `=` E, where there is no following expression. The first kind is mapped to the π–calculus as follows:

$$TrPI_{exp}(\texttt{self, } X\texttt{=}E_1\texttt{, } E_2) := (\underline{\texttt{new}}\ \texttt{res}')\begin{pmatrix} TrPI_{exp}(\texttt{self, } E_1) \\ \| \\ \texttt{res}'(X).TrPI_{exp}(\texttt{self, } E_2) \end{pmatrix}$$

As in the mapping of sequence expressions in subsection 4.4 above, the two processes are executed in parallel, one evaluating the expression E_1 and the other expecting the value of the expression in channel `res`$'$. As soon as the evaluation result of E_1 is available in `res`$'$ channel, it is received by the second process and execution of the remaining expression E_2 is started. We use the distinguished channel name `res`$'$ to express explicitly that result of the evaluation of expression E_1 will be sent along channel `res`$'$ to the second process, which receives the result in X. In this way, the evaluation result of E_1 can be used in expression E_2. The π–calculus *receive action* (here $res'(\ X)$) binds X and thus both `res`$'$ and X are bound names for the two parallel processes.

The second kind of assignment expression is denoted as X`=` E where X is a variable and E is an expression. The evaluation result of E will be assigned to X after executing the expression. We have written the corresponding translation mapping function for expression X`=` E by sending the evaluation result of expression E along the global channel `res`. Thereby, any process having *receive action* along channel `res` can receive this result. The mapping is formally defined as follows:

$$TrPI_{exp}(\texttt{self, } X\texttt{=}E) := TrPI_{exp}(\texttt{self, } E) \parallel \overline{\texttt{res}}\texttt{<}X\texttt{>.}\underline{\texttt{nil}}$$

4.6 Send Expression

The main expression for communication in PIErlang is the `send` expression. The `send` expression is denoted by A_1`!` A_2 where A_1 and A_2 are place holders for `Arguments` (numbers, atoms, variables and `self()`) and the evaluation of A_1 returns a PID to which the message A_2 is to be sent. The corresponding π–calculus translation is as follows:

$$TrPI_{exp}(\texttt{self}, A_1\texttt{!}A_2) := \begin{pmatrix} \overline{TrPI_{arg}(A_1)}\texttt{<}TrPI_{arg}(A_2)\texttt{>.}\underline{\texttt{nil}} \\ \| \\ \overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_2)\texttt{>.}\underline{\texttt{nil}} \end{pmatrix}$$

The translation of `send` expressions introduces two processes working in parallel; one is a direct mapping of the `send` expression to the π–calculus and the other sends the message A_2 to the `res` channel so that any process waiting for a message from the `res` channel can receive A_2 with a *receive action* along `res`, in particular when there is a sequence of expressions and the `send` expression is the first expression, to be executed before the execution of the second expression. In this way, the message is sent to the specific process identified by PID A_1 and along the `res` channel. However, in case studies it has been observed that use of the `res` channel for `send` expressions is not mandatory since `send` expression processes can execute in parallel with other processes. Thus it can be omitted, by contrast with the translation of [16, 18].

4.7 Function Calls

There are two different kinds of function calls in PIErlang: n–ary functions and n–ary spawn functions. An n–ary function call is an expression in Erlang. These can be translated to $(n+1)$–ary process calls in the π–calculus where the first argument is `self`, the global name representing the PID of the process executing the function expression. Translation of an n–ary function call is as follows:

$$TrPI_{exp}(\texttt{self, } f\texttt{(}A_1\texttt{, } \ldots\texttt{, }A_n\texttt{)}) := f\texttt{(self, } TrPI_{arg}(A_1)\texttt{, } \ldots\texttt{, } TrPI_{arg}(A_n)\texttt{)}$$

Only `Arguments` i.e. variables, numbers, atoms and the built-in function `self()` can be used as the arguments of the function expression. To have a proper translation depending on the types of the arguments, the $TrPI_{arg}$ translation function is applied for each of the arguments. Unlike [18] `res` is omitted here. Similarly, for 0–ary function expressions the mapping is as follows:

$$TrPI_{exp}(\texttt{self, } f\texttt{()}) := f\texttt{(self)}$$

4.8 Spawn Calls

A `spawn` call is like a function call except that a new child process is constructed to execute the call rather than the current process. The child process runs in parallel with the current process, performing the function call. A `spawn` call returns the PID of the newly created process that executes the function given as parameter of the `spawn`. If the function call terminates, the process executing it will end. If the function call returns a value, this value will be ignored. In PIErlang, `spawn` is simplified by omitting the module in which the function is placed. The syntax of this restricted form of `spawn` is as follows:

$$\texttt{spawn(}f\texttt{, [}A_1\texttt{, } \ldots\texttt{, }A_n\texttt{]) ; } n\texttt{>=}0$$

Like function call arguments, only numbers, atoms, variables and the built-in function `self()` can be used as arguments of the function in a `spawn` call. Depending on the context there can be two versions of `spawn`; one that stores the newly returned PID in a variable and another that ignores it. The mapping for the first kind of `spawn` call is as follows:

$$TrPI_{exp}(\texttt{self, } X\texttt{=spawn(}f\texttt{, [}A_1\texttt{, } \ldots\texttt{, }A_n\texttt{])}\texttt{, } E) := \left(\begin{array}{l}(\underline{\texttt{new}}\ \texttt{fpid, f\_res, p}) \\ \left(\begin{array}{l}\overline{\texttt{p}}\texttt{<fpid>.}\underline{\texttt{nil}} \parallel TrPI_{exp}(\texttt{fpid, } f\texttt{(}A_1\texttt{, } \ldots\texttt{, }A_n\texttt{))} \\ \parallel \texttt{f\_res(dummy).}\underline{\texttt{nil}} \parallel \texttt{p(X)}.TrPI_{exp}(\texttt{self, } E)\end{array}\right)\end{array}\right)$$

where process identifiers `fpid` $\neq$ `self`.

As before, `self` is included along with the `spawn` as one of the arguments of the translation function to represent that the `spawn` function is executed by the current process i.e. the process which has PID `self()`. After executing the `spawn`, the newly created PID will be stored in the variable X and the function f will be executed by the newly created process in parallel with the process containing the `spawn` call.

In the translation mappings, three new names, `fpid`, `p` and `f_res`, are created. `fpid` represents the PID of the newly created process which will execute function f, `p` is used to send/receive the `fpid` and `f_res` is the channel through which the result of the evaluation of the `spawn` will be sent. Bound name `X` is used to receive the `fpid` along `p` for further use. One process ($\overline{\texttt{p}}\texttt{<fpid>.}\underline{\texttt{nil}}$) sends the PID `fpid` along `p` and the same PID is used in the translation mapping of the function $f(A_1, \ldots, A_n)$ by the second parallel process ($TrPI_{exp}(\texttt{fpid,}f(A_1, \ldots, A_n))$) which clearly specifies that the newly created process with PID `fpid` will execute the function f. There are also two more processes working in parallel; one of which waits to receive something (the return value of the function call) from the `f_res` channel and then discards it, and another that waits to receive `fpid` in `X`. After receiving `fpid` in `X`, the current process (here `self`) will execute the remaining expression E. In this way, using the reaction rule of the π–calculus, any use of variable X in expression E will be replaced by the PID

fpid which meets the semantics of the simplified spawn function and the intention of our translation mapping.

In the second form of spawn, the newly created PID is ignored rather than storing it to a variable, and it is assumed that the PID of the newly created process will no longer be used in subsequent expression(s). Here is the translation mapping:

$$TrPI_{exp}(\texttt{self, spawn}(f\texttt{, [}A_1\texttt{,}\ldots\texttt{,}A_n\texttt{])}\texttt{,}\ E) := \left(\begin{array}{l}(\underline{\texttt{new}}\ \texttt{fpid, f\_res}) \\ \left(\begin{array}{l} TrPI_{exp}(\texttt{fpid,}\ f\texttt{(}A_1\texttt{,}\ldots\texttt{,}A_n\texttt{))} \parallel \\ \quad \texttt{f\_res(dummy).}\underline{\texttt{nil}} \parallel TrPI_{exp}(\texttt{self,}\ E)\end{array}\right)\end{array}\right)$$

where process identifiers fpid $\neq$ self.

The only major difference between this mapping and the mapping above is that the remaining expression E is executed without waiting to receive the new PID fpid along p, since the evaluation of E does not depend on the PID fpid. However, case studies have shown that in both cases the use of f_res can be omitted with no effect on the translated system.

4.9 Receive Expression

The counterpart of the send expression is the receive expression and the two jointly implement communication between two processes. It has the branching feature of a case expression and also has to match the reception semantics of PIErlang. Because of the limited set of statements available in the π–calculus, our naive translation is not a completely accurate reflection of Erlang reception semantics. Because we have chosen the asynchronous π–calculus as our target specification language, order of the received messages can not be directly respected. Instead we use non–deterministic choice among the translated matches of the receive statement as the translation of receive statements.

$$TrPI_{exp}(\texttt{self, receive}\ M_1\texttt{;}\ldots\texttt{;}M_n\ \texttt{end}) := TrPI_{match}(\texttt{self,}\ M_1)\texttt{+}\ldots\texttt{+}TrPI_{match}(\texttt{self,}\ M_n)$$

The receive expression is modeled using a non–deterministic choice between the individual match expressions where $TrPI_{match}$, a function for mapping matches (c.f. subsection 4.10), is applied to each of the matches. self, the PID of the process that executes the receive is used in the matches as well.

4.10 Match

Pattern matching is used for assigning values to variables and for controlling the flow of a program. Erlang is a single assignment language, which means that once a variable has been assigned a value, the value can never be changed. Pattern matching is used to match patterns with terms. If a pattern and term have the same shape then the match will succeed and any variables occurring in the pattern will be bound to the data structures which occur in the corresponding positions in the term. In PIErlang, a match is of the form P -> E, where pattern P can be a variable, an atom or a number. To translate such a match to the π–calculus, a $TrPI_{match}$ function is used with the following signature:

$$TrPI_{match}\texttt{:}\ Name\ \texttt{X}\ Match \rightarrow Process$$

A name and the match are used as the input and a process in the π–calculus is produced as output. The name will be self, the process identifier of the process executing the receive expression and hence the match.

We apply the name matching feature of the π–calculus to handle pattern matching in PIErlang. First, consider a match of the form a -> E, where the pattern a is an atom and the expression E can be any valid PIErlang expression. The match a -> E and the name self are provided to the $TrPI_{match}$ function as input. Along channel self the target message will be received in the bound name input which will be matched against the pattern (here the atom a) of the given match, and similarly for numbers as patterns.

Now consider the match X -> E where variable X is used as pattern. As a variable can be matched with any term, no name matching is required for a variable as pattern. We use the same variable name X as the received name along the channel self. The intention of doing so is that variable X can then be used in the evaluation of E. The complete mapping for atoms, numbers and variables as patterns can be defined as follows:

$$TrPI_{match}(\texttt{self,}\ P\ \texttt{->}\ E) := \begin{cases} \texttt{self(input).[input=}TrPI_{arg}(P)\texttt{]}\ TrPI_{exp}(\texttt{self,}\ E) & \\ \quad \texttt{if}\ P \in \texttt{Numbers} \cup \texttt{Atoms} & \\ & \\ \texttt{self(P).}\ TrPI_{exp}(\texttt{self,}\ E) & \\ \quad \texttt{if}\ P \in \texttt{Variables} & \end{cases}$$

4.11 Case Expression

Another useful construct of PIErlang is the case expression, whose syntax resembles the receive expression in the sense of using matches. Matches in the case expression can be tackled with the same way as presented for receive expressions above. Therefore, we can map the case expression to the receive expression and use the translation mapping of receive expression.

$$TrPI_{exp}(\texttt{self, case}\ E\ \texttt{of}\ M_1\texttt{;}\ldots\texttt{;}M_n\ \texttt{end}) := \left(\begin{array}{l}(\underline{\texttt{new}}\ \texttt{cres}) \\ \left(\begin{array}{l} TrPI_{exp}(\texttt{self,}\ E) \\ \quad \parallel \\ TrPI_{exp}(\texttt{cres, receive}\ M_1\texttt{;}\ldots\texttt{;}M_n\ \texttt{end})\end{array}\right)\end{array}\right)$$

$$= \left(\begin{array}{l}(\underline{\texttt{new}}\ \texttt{cres}) \\ \left(\begin{array}{l} TrPI_{exp}(\texttt{self,}\ E) \\ \quad \parallel \\ \left(\begin{array}{l} TrPI_{match}(\texttt{cres,}\ M_1)\texttt{+}\ldots \\ \texttt{+}\ TrPI_{match}(\texttt{cres,}\ M_n)\end{array}\right)\end{array}\right)\end{array}\right)$$

One process evaluates the case head expression and sends the result along fresh channel cres, which is then used in evaluating the matches of the case expression in place of self in the translation of receive expressions. Using this rule any expression can be the case head provided that evaluation rule(s) are available for it.

4.12 Function Definitions

Every PIErlang program is a sequence of one or more function definitions. Each function definition is translated into a corresponding process definition in the π–calculus. The signature of the mapping function is:

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

Each n–ary function definition in Erlang is translated to an $(n+1)$–ary process definition in the π–calculus, where the first argument is self, the current process executing the function definition. Unlike our previous method [18], res is omitted here. The mappings for n–ary and 0–ary function definitions are as follows:

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

$$TrPI_{fundef}(\texttt{self,}\ f\texttt{()->}\ E) := \left(\ f\texttt{(self) =}\ TrPI_{exp}(\texttt{self,}\ E)\ \right)$$

4.13 PIErlang Programs

Now that we have given the translation mappings for each of the syntactic constructs (except tuple related constructs) of PIErlang, we can describe the translation of a complete PIErlang

`program` into a π–calculus `system`. We define a translation function $TrPI_{prog}$ that takes a name (here `self`) and a PIErlang `program` as input and will return a `system` in the π–calculus as output. Its signature and formal definition are as follows:

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

$$TrPI_{prog}(\texttt{self},\ F_1, \ldots, F_n)$$
$$:= \begin{pmatrix} \texttt{Main=(}\underline{\texttt{new}}\ \texttt{self, OtherNames)}\ TrPI_{exp}(\texttt{self},\ f\_0), \\ TrPI_{fundef}(\texttt{self},\ F_1), \ldots, TrPI_{fundef}(\texttt{self},\ F_n) \end{pmatrix}$$

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

A PIErlang `program` is composed of a sequence of `function definitions`. When a `program` is translated to a π–calculus `system`/π–model, each of the `function definitions` becomes a `process definition` of the `system/model`. Along with the sequence of process definitions for the corresponding function definitions, a new process `Main` is introduced, which declares the parameters (e.g. `self`) of the *f_0* process and other atoms/names (e.g. `res`, `unknown`, `dummy`, `cres` etc.) used in the system.

In PIErlang *f_0* is a *0*–ary function and normally the left hand side of the first function definition F_1. For observing the behavior of the obtained π–model (System) of a PIErlang program, the initial call is set to the process `Main`. For details about this we refer to the case studies undertaken in [16]. However, in [16] a different representation was made for mapping a complete PIErlang program to π–calculus system.

4.14 Verification of the Obtained π-Model

The standard approach to evaluating correctness of a π–calculus specification is through the use of a bisimulation equivalence. A more complex process representing an implementation is shown to be bisimilar to a simpler process representing a specification. The simpler process should be so clear that it can be regarded as satisfying correctness requirements in an intuitive sense, without rigorous mathematical proof. Once the desired safety, liveness and/or fairness properties can be expressed in a suitable temporal logic (equipped with model–checking algorithm), it possible to determine whether or not those properties hold for a given π–calculus specification with existing tools [8, 9, 12, 15, 19]. However, as all numbers (and data in general) in our translation are modeled by a special name `unknown`, any faults in an Erlang program due to incorrect manipulation of data will not be picked up by our translation.

A question still remains in terms of generating counter–examples. Model–checking tools should provide counter example(s) if a given property (specified in a suitable temporal logic formula) is shown not to be satisfied. Although most of the tools mentioned above can generate (a kind of) counter–examples, none of them can provide counter example directly in Erlang. For example, HAL [8] generates counter–examples in the ordinary automata world and hence to have the counter example in Erlang, a mapping from ordinary automata to Erlang would be required. Similar restrictions apply to MWB [12]. Once it finds a deadlocked process/agent, it shows the agent and the transition trace that leads to the deadlocked agent. To get the counter example to Erlang, a tool would be required to providing a mapping from the π–calculus to Erlang.

However, once the transition–system is obtained for a given π–calculus specification, it would be most suitable to use Truth [9] for verification/model–checking and generating counter–examples. Truth is a verification platform for concurrent systems. It aims to serve as a kind of building set of verification tools in which new concepts for system specification, models of concurrency, property logics, and model–checking algorithms can easily be tested. As usual, in order to get counter–examples in Erlang an improvement to Truth is required. Moreover, the abstraction of numbers to `unknown` will have side effects on the generated counter–examples. The actual details of the verification/model–checking of the π–model for a given Erlang program, however, is beyond the scope of this article.

5. Dealing with Non–determinism Among Matches

In this section we present a way to replace the non–deterministic choice among the clauses of a `case` or `receive` expression by a deterministic top–down strategy that more accurately reflects the semantics of Erlang. This section also explores the similarity and dissimilarity between the mappings of `receive` and `case` expressions in an indirect way.

We have addressed the issues of sending and receiving mechanisms of PIErlang along with their mappings to π–calculus in subsections 4.6 and 4.9 respectively. Our mapping of `send` expression is sound enough to meet its semantics in Erlang. However, the receiving mechanism in Erlang is not so straightforward. The general syntax for a `receive` expression is as follows:

```
receive
      Pattern1 ->Body1;
      ...;
      PatternN ->BodyN;
end
```

The `receiver` process receives messages sent to its mailbox by a sender process with the send operator (!). The patterns $Pattern_i$ are sequentially matched against the first message in time order in the mailbox, then the second, and so on. If a match succeeds, the corresponding Body is evaluated. The matching message is consumed (removed from the mailbox), while any other messages in the mailbox remain unchanged. The return value of $Body_i$ is the return value of the `receive` expression. `Receive` never fails. Execution is suspended, possibly indefinitely, until a message arrives that matches one of the patterns.

This `receive` construct is mainly used to allow processes to wait for messages from other processes. The `receive` expression mapping to π-calculus presented in subsection 4.9 is not sufficient to meet the reception semantics of PIErlang. In Erlang the evaluation order of the clauses of the `receive` is a major issue but this order is not preserved by our mapping to the π–calculus since the clauses are represented as non–deterministic choices. We have not explicitly modeled the mailbox of the Erlang process in the π–calculus, rather it is modeled implicitly by considering the PID of the receiver process (PID of the destination process to which the message is to be sent) as a channel and sending the potential message over this channel asynchronously. This asynchronous message-passing and the non–deterministic choices among the `matches` essentially breaks the send–receive semantics of Erlang programs in the π–calculus representation.

Similar situations arise with the mapping of `case` expressions discussed in subsection 4.11, since `case` expressions are mapped using the `receive` expression mapping. Although there is no question of modeling mailboxes for `case` expression mapping, the semantics of `case` expressions cannot be preserved in the π-calculus as the matches are represented by non–deterministic choices. To clarify this, let us consider the general form of a `case` expression.

```
case Expr of
      Pattern1 ->Body1;
      ...;
      PatternN ->BodyN;
end
```

First, `case head` $Expr$ is evaluated, then the value of $Expr$ is sequentially matched against the patterns $Pattern1, \ldots, PatternN$

```
receive
     true -> work;
     false -> rest;
     Y -> Y
end.
```

Figure 3. A **receive** expression code segment in Erlang

```
case Status of
     true -> work;
     false -> rest;
     Y -> Y
end.
```

Figure 4. A **case** expression code segment in Erlang

until a match is found. If a match is found, then the corresponding call $Body$ is evaluated. The value of the `case` primitive is then the value of the selected $Body$. As in the `receive` expression, the order of the clauses is crucial.

If there is only one matching pattern in each execution of the `receive` or `case` expression then the non–deterministic mappings discussed in subsections 4.9 and 4.11 handle the Erlang semantics accurately. However, if more than one patterns match in an execution then the corresponding `receive/case` body evaluation choice would be non–deterministic in our π-calculus representation. For example, consider the mapping of the `receive` expression of Figure 3.

$$TrPI_{exp}(\texttt{self, (Code segment of Figure 3)})$$
$$= \begin{pmatrix} TrPI_{match}(\texttt{self, } true \texttt{ -> } work) + \\ TrPI_{match}(\texttt{self, } false \texttt{ -> } rest) + \\ TrPI_{match}(\texttt{self, } Y \texttt{ -> } Y) \end{pmatrix}$$
$$= \begin{pmatrix} \texttt{self(input).[input=true]}\overline{\texttt{res}}\texttt{<work>.}\underline{\texttt{nil}} + \\ \texttt{self(input).[input=false]}\overline{\texttt{res}}\texttt{<rest>.}\underline{\texttt{nil}} + \\ \texttt{self(Y).}\overline{\texttt{res}}\texttt{<Y>.}\underline{\texttt{nil}} \end{pmatrix}$$

The mapping of this example demonstrates the problem. Let us suppose that there is a sending process $\overline{\texttt{self}}\texttt{<false>.}\underline{\texttt{nil}}$ that sends the atom `false` along channel `self`. If both the receiver process and this sender process execute in parallel then sender process will non–deterministically communicate with either the second or the third `match` since the second matches `false` directly and the third matches any input term. This is not consistent with the semantics of Erlang, in which only the second `match` should succeed since it appears before the thrid.

Similarly, consider the mapping of the `case` expression of Figure 4.

$$TrPI_{exp}(\texttt{self, (Code segment of Figure 4)})$$
$$= \left(\begin{array}{l} \texttt{(}\underline{\texttt{new}}\texttt{ cres)} \\ \left(\begin{array}{l} TrPI_{exp}(\texttt{self, } Status) \\ \quad \| \\ \texttt{(}TrPI_{match}(\texttt{cres, } true \texttt{ -> } work) + \\ \ TrPI_{match}(\texttt{cres, } false \texttt{ -> } rest) + \\ \ TrPI_{match}(\texttt{cres, } Y \texttt{ -> } Y)\texttt{)} \end{array} \right) \end{array} \right)$$
$$= \left(\begin{array}{l} \texttt{(}\underline{\texttt{new}}\texttt{ cres)} \\ \left(\begin{array}{l} \overline{\texttt{cres}}\texttt{<}Status\texttt{>.}\underline{\texttt{nil}} \\ \quad \| \\ \texttt{(cres(input).[input=true]}\overline{\texttt{res}}\texttt{<work>.}\underline{\texttt{nil}} + \\ \ \texttt{cres(input).[input=false]}\overline{\texttt{res}}\texttt{<rest>.}\underline{\texttt{nil}} + \\ \ \texttt{cres(Y).}\overline{\texttt{res}}\texttt{<Y>.}\underline{\texttt{nil}}\texttt{)} \end{array} \right) \end{array} \right)$$

This mapping is also problematic. Let us suppose that atom `true` is bound with the variable $Status$ of *case head* somewhere in the body of the corresponding function. In Erlang, only the first `match` ($true$ `->` $work$) succeeds because of ordered matching. But in the π–calculus translation, we have used non–determinism among different matches and hence, there is a possibility that variable $Status$ will match with either the first or the third `match`.

In order to avoid such non–determinism between the matching clauses, we have designed a refined mapping using negated (mismatch) conditions for the preceding codematch clauses in each following `match`, so that a clause/match that comes after another clause in order will not be selected if the preceding one can be selected. To implement this, the `receive` and `case` expression mappings of subsections 4.9 and 4.11 respectively are refined by providing the index number of the matches as follows:

$$TrPI_{exp}(\texttt{self, receive } M_1\texttt{; } \ldots \texttt{; } M_n \texttt{ end})$$
$$:= TrPI_{match_1}(\texttt{self, } M_1)\texttt{+} \ldots \texttt{+} TrPI_{match_n}(\texttt{self, } M_n)$$

$$TrPI_{exp}(\texttt{self, case } E \texttt{ of } M_1\texttt{; } \ldots \texttt{; } M_n \texttt{ end})$$
$$:= \left(\begin{array}{l} \texttt{(}\underline{\texttt{new}}\texttt{ cres)} \\ \left(\begin{array}{l} TrPI_{exp}(\texttt{self, } E) \\ \quad \| \\ \left(\begin{array}{l} TrPI_{match_1}(\texttt{cres, } M_1)\texttt{+} \ldots \\ \texttt{+ } TrPI_{match_n}(\texttt{cres, } M_n) \end{array} \right) \end{array} \right) \end{array} \right)$$

where the mapping of the i^{th} match, $TrPI_{match_i}$ is as follows:

$$TrPI_{match_i}(\texttt{self, } P_i \texttt{ -> } E_i)$$
$$:= \begin{cases} \texttt{self(input).[input<>}TrPI_{arg}(P_1)\texttt{]} \ldots \\ \quad \texttt{[input<>}TrPI_{arg}(P_{i-1})\texttt{][input=}TrPI_{arg}(P_i)\texttt{]} \\ \quad TrPI_{exp}(\texttt{self, } E_i)\texttt{; if } P_i \in \texttt{Numbers} \cup \texttt{Atoms} \\ \\ \texttt{self(}P_i\texttt{).[}P_i\texttt{<>}TrPI_{arg}(P_1)\texttt{]} \ldots \texttt{[}P_i\texttt{<>}TrPI_{arg}(P_{i-1})\texttt{]} \\ \quad TrPI_{exp}(\texttt{self, } E_i)\texttt{; if } P_i \in \texttt{Variables} \end{cases}$$

The mapping of the `receive` expression of Figure 3 can now be rewritten as follows, avoiding non–determinism among the matches:

$$\begin{pmatrix} \texttt{self(input).[input=true]}\overline{\texttt{res}}\texttt{<work>.}\underline{\texttt{nil}} + \\ \texttt{self(input).[input<>true][input=false]} \\ \qquad\qquad\qquad\qquad \overline{\texttt{res}}\texttt{<rest>.}\underline{\texttt{nil}} + \\ \texttt{self(Y).[Y<>true][Y<>false]}\overline{\texttt{res}}\texttt{<Y>.}\underline{\texttt{nil}} \end{pmatrix}$$

It is now clear that the sender process $\overline{\texttt{self}}\texttt{<false>.}\underline{\texttt{nil}}$ can communicate only with the second match, avoiding the problems introduced by non–determinism. Similarly, the refined mapping of the `case` expression of Figure 4 is as follows:

$$\left(\begin{array}{l} \texttt{(}\underline{\texttt{new}}\texttt{ cres)} \\ \left(\begin{array}{l} \overline{\texttt{cres}}\texttt{<}Status\texttt{>.}\underline{\texttt{nil}} \\ \quad \| \\ \left(\begin{array}{l} \texttt{cres(input).[input=true]}\overline{\texttt{res}}\texttt{<work>.}\underline{\texttt{nil}} + \\ \texttt{cres(input).[input<>true][input=false]} \\ \qquad\qquad\qquad\qquad \overline{\texttt{res}}\texttt{<rest>.}\underline{\texttt{nil}} + \\ \texttt{cres(Y).[Y<>true][Y<>false]}\overline{\texttt{res}}\texttt{<Y>.}\underline{\texttt{nil}} \end{array} \right) \end{array} \right) \end{array} \right)$$

6. Mapping Tuples with Polyadic Communications

In our previous work Erlang tuples were simply abstracted away using an `unknown` name [18]. In this section we outline a mapping for tuples appearing in messages of `send` expressions and match `patterns` of `receive` and `case` expressions to polyadic communication in the π-calculus. A detailed treatment is provided for all non–nested tuple-based expressions.

PIErlang can create compound data structures using tuples. Tuples are used for storing a fixed number of items and are written as sequences of items enclosed in curly brackets. Tuples are similar to records or structures in conventional programming languages.

There is a strong similarity between the polyadic sending of multiple names/channels over a channel in the π–calculus and tuple-based expressions in PIErlang.

6.1 Tuple Expressions

Erlang programs commonly use tuples as expressions. Tuples are data structures for grouping variables and values and therefore, an intrinsic part of the expressions in which they are used. In PIErlang, only numbers, atoms, variables and the built-in function `self()` can be used as tuple elements. Using the polyadic π–calculus such tuple expressions are mapped as follows:

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

Using this mapping, tuple elements are translated using the $TrPI_{arg}$ function and then sent as `names` along the `res` channel in polyadic form. The result channel `res` is refined to handle the polyadic communication.

6.2 Tuple-based Sequences of Expressions and Spawn Calls

Our original mapping of `sequence of expressions` in subsection 4.4 cannot be used for tuple expressions, because the new channel $\texttt{res}'$ in that mapping is in monadic form, whereas the evaluation of the expression E_1 could be polyadic when handling tuples. However, if we refine the mapping to allow the channel $\texttt{res}'$ to deal with polyadic communication then the evaluation result of the tuple-based expression E_1 can be sent along it and received along the same channel for dummy communication in polyadic form. The refined mapping of the `sequence of expressions` is as follows:

$$TrPI_{exp}(\texttt{self}, E_1, E_2) := \left(\begin{array}{l}(\underline{\texttt{new}}\ \texttt{res}') \\ \left(\begin{array}{l} TrPI_{exp}(\texttt{self}, E_1) \\ \quad \| \\ \texttt{res}'\texttt{(dummy1,}\ldots\texttt{,dummyn)}.TrPI_{exp}(\texttt{self}, E_2)\end{array}\right)\end{array}\right)$$

Similarly, we can refine the mapping of `spawn` calls to use polyadic communication along the `f_res` channel, as follows:

$$TrPI_{exp}(\texttt{self}, X\texttt{=spawn(}f\texttt{, [}A_1, \ldots, A_n\texttt{])}, E) := \left(\begin{array}{l}(\underline{\texttt{new}}\ \texttt{fpid, f\_res, p}) \\ \left(\begin{array}{l}\overline{\texttt{p}}\texttt{<fpid>.}\underline{\texttt{nil}} \parallel TrPI_{exp}(\texttt{fpid}, f(A_1, \ldots, A_n)) \\ \parallel \texttt{f\_res(dummy1,}\ldots\texttt{,dummyn).}\underline{\texttt{nil}} \\ \parallel \texttt{p(X)}.TrPI_{exp}(\texttt{self}, E)\end{array}\right)\end{array}\right)$$

$$TrPI_{exp}(\texttt{self, spawn(}f\texttt{, [}A_1, \ldots, A_n\texttt{])}, E) := \left(\begin{array}{l}(\underline{\texttt{new}}\ \texttt{fpid, f\_res}) \\ \left(\begin{array}{l}TrPI_{exp}(\texttt{fpid}, f(A_1, \ldots, A_n)) \\ \parallel \texttt{f\_res(dummy1,}\ldots\texttt{,dummyn).}\underline{\texttt{nil}} \\ \parallel\ TrPI_{exp}(\texttt{self}, E)\end{array}\right)\end{array}\right)$$

Note that only the dummy communications need be refined to deal with polyadic communication. The value n in `dummy` communication is based on the evaluation of the `spawn` function calls (for `spawn` calls) or the evaluation of expression E_1 (for the sequence of expressions E_1, E_2). This contrasts with our previous work [16], where a modification to the π–calculus semantics was proposed to deal with this case.

6.3 Tuple-based Send Expressions

In Erlang tuples are frequently used as messages in `send` expressions. The `send` expression syntax is A `!` $\{A_1, \ldots, A_n\}$. From this syntax, it is clear that a tuple–based `send` expression can send a non–nested tuple of number(s), atom(s), variable(s) and the built-in function `self()` as a message to a process's mailbox, which has a PID identified by A. Polyadic π–calculus can be used to model such tuple-based `send` expressions as follows:

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

As before, the passing of the tuple message through the `res` channel can be omitted.

6.4 Tuple-based Receive and Case Expressions: Refinements of Match

With the change to the mapping of `send`, a codesender process can now send tuple-based messages to a `receiver` process. To refine `receive/case` expressions to receive such tuple-based messages, we need only refine `receive/case` to deal with polyadic reception of messages against the patterns of their matches. We handle this pattern matching for a `match` tuple pattern using the name matching feature of the polyadic π–calculus. We can receive a tuple of message(s) using the *receive action* translation rule in the polyadic π–calculus along a channel (say `self`) where the message elements are bound to corresponding input names and then applying the name matching feature on the bound names against the corresponding patterns of the *receive action*. Of course, we must use $TrPI_{arg}$ to get a corresponding translation of the patterns (elements of the tuple pattern) before applying name matching. This general scenario can be formally presented as follows for atoms or numbers as tuple pattern elements:

$$TrPI_{match}(\texttt{self, \{}P_1, \ldots, P_n\texttt{\}->}E) := \left(\begin{array}{l}\texttt{self(input1,}\ldots\texttt{,inputn).[input1=}TrPI_{arg}(P_1)\texttt{]} \\ \quad \ldots\texttt{[inputn=}TrPI_{arg}(P_n)\texttt{]}\ TrPI_{exp}(\texttt{self}, E) \\ \texttt{if}\ P_1, \ldots, P_n \in \texttt{Numbers} \cup \texttt{Atoms}\end{array}\right)$$

As before, if there is a variable in the tuple pattern then name matching is not required for that element and the variable itself can be used in the polyadic *receive action* along the `self` channel. For example:

$$TrPI_{match}(\texttt{self, \{}a\texttt{, }X\texttt{, }b\texttt{, }Y\texttt{\}->}E) := \left(\begin{array}{l}\texttt{self(input1, X, input2, Y).[input1=a]} \\ \quad \texttt{[input2=b]}\ TrPI_{exp}(\texttt{self}, E)\end{array}\right)$$

In this example, pattern variables X and Y are directly used in the *received action* and their corresponding name matchings are omitted. This general scenario can be formally expressed to support numbers, atoms and variables as pattern elements of a tuple pattern where the complete mapping is obtained by taking into account all the P_i's:

$$TrPI_{match}(\texttt{self, \{}P_1, \ldots, P_i, \ldots, P_n\texttt{\}->}E) := \begin{cases}\texttt{self(..,input\_i,..)}\ldots\texttt{.[input\_i=}TrPI_{arg}(P_i)\texttt{]}\ldots \\ \quad TrPI_{exp}(\texttt{self}, E)\texttt{; if}\ P_i \in \texttt{Numbers} \cup \texttt{Atoms} \\ \\ \texttt{self(..,}P_i\texttt{,..)}\ldots.\ TrPI_{exp}(\texttt{self}, E) \\ \quad \texttt{if}\ P_i \in \texttt{Variables}\end{cases}$$

7. Resource Manager Example: A Case Study

As a case study in the application of our new representation, let us consider the simplified resource manager example from [18] but making use of our new mapping to polyadic communication. Figure 5 shows such a small PIErlang resource manager program using tuple–based communication.

In Figure 5 the `start` function first spawns a `resource` and a `manager` process and then invokes the `client` function. The

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

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

manager(Rsr) ->
  receive
    {C, grantaccess} ->
      C!Rsr
  end.

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

Figure 5. Simple resource manager in PIErlang

```
Main
  =  (new self)(start(self))
start(self)
  =  (new rPID, mPID, cPID, p, q)
     (p̄<rPID>.nil ‖ resource(rPID) ‖
      p(Rsr).(q̄<mPID>.nil ‖ manager(mPID,Rsr) ‖
      q(Mgr).client(cPID, Mgr)))
resource(self)
  =  self(Req).res̄<action>.nil
manager(self, Rsr)
  =  self(C,input).[input=grantaccess]C̄<Rsr>.nil
client(self, Mgr)
  =  Mgr̄<self, grantaccess>.nil ‖
     self(R).R̄<request>.nil
```

Figure 6. Simple resource manager in the π–calculus

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`. This program is a simple example of a dynamic or mobile system: the required communication channel between `client` and `resource` is not available from the beginning, it must first be established by passing the corresponding "handle". Traditional modeling approaches, which lack this name–passing capability, are doomed to fail in this setting.

Figure 6 shows the result of applying our new translation mappings to the simple resource manager example of Figure 5. For simplicity, we have avoided showing all the new names in the system. To examine the behavior of this generated π–calculus model, we start from the `Main` process.

$$\texttt{Main} \;:=\; (\underline{\texttt{new}}\ \texttt{self})(\texttt{start(self)})$$

Following instantiation of the `start` process and applying the π–calculus reaction rule on channels `p` and `q`, we omit the $\underline{\texttt{nil}}$ processes and get,

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

This means that upon invocation of the `start` process, the `resource` and `manager` processes are spawned and the initial process runs the `client` function. Upon instantiation of the `manager` and `client` processes we then get,

$$\left(\begin{array}{l}(\underline{\texttt{new}}\ \texttt{rPID, mPID, cPID})\\ \left(\begin{array}{l}\texttt{resource(rPID)}\\ \quad\|\\ \texttt{mPID(C,input).[input=grantaccess]}\overline{\texttt{C}}\texttt{<rPID>.}\underline{\texttt{nil}}\\ \quad\|\\ \overline{\texttt{mPID}}\texttt{<cPID, grantaccess>.}\underline{\texttt{nil}}\ \|\\ \texttt{cPID(R).}\overline{\texttt{R}}\texttt{<request>.}\underline{\texttt{nil}}\end{array}\right)\end{array}\right)$$

In the first transition of the system, `client` asks the `manager` for the handle to `resource`. The resulting state is obtained by applying the reaction rule on channel `mPID`. As a result `C` is bound with `cPID` and `input` with `grantaccess`.

$$\left(\begin{array}{l}(\underline{\texttt{new}}\ \texttt{rPID, mPID, cPID})\\ \left(\begin{array}{l}\texttt{resource(rPID)}\\ \quad\|\\ \texttt{[grantaccesst=grantaccess]}\overline{\texttt{cPID}}\texttt{<rPID>.}\underline{\texttt{nil}}\\ \quad\|\\ \underline{\texttt{nil}}\ \|\ \texttt{cPID(R).}\overline{\texttt{R}}\texttt{<request>.}\underline{\texttt{nil}}\end{array}\right)\end{array}\right)$$

The name matching `grantaccess = grantaccess` is found and consequently, process `client` is now enabled (react on `cPID`) to receive the PID of the `resource` sent by `manager`. The received PID is bound to the name `R`.

$$\left(\begin{array}{l}(\underline{\texttt{new}}\ \texttt{rPID, mPID, cPID})\\ \left(\begin{array}{l}\texttt{resource(rPID)}\\ \quad\|\\ \underline{\texttt{nil}}\\ \quad\|\\ \underline{\texttt{nil}}\ \|\ \overline{\texttt{rPID}}\texttt{<request>.}\underline{\texttt{nil}}\end{array}\right)\end{array}\right)$$

After invoking the `resource` process we get,

$$\left(\begin{array}{l}(\underline{\texttt{new}}\ \texttt{rPID, mPID, cPID})\\ \left(\begin{array}{l}\texttt{rPID(Req).}\overline{\texttt{res}}\texttt{<action>.}\underline{\texttt{nil}}\\ \quad\|\\ \underline{\texttt{nil}}\\ \quad\|\\ \underline{\texttt{nil}}\ \|\ \overline{\texttt{rPID}}\texttt{<request>.}\underline{\texttt{nil}}\end{array}\right)\end{array}\right)$$

Now the `client` process can send the actual request (react on `rPID`) to `resource`.

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

After ignoring the $\underline{\texttt{nil}}$ processes we get,

$$(\underline{\texttt{new}}\ \texttt{rPID, mPID, cPID})\ \overline{\texttt{res}}\texttt{<action>.}\underline{\texttt{nil}}$$

Note that this was not possible before since the corresponding PID (`rPID`) of `resource` was not known to `client`. After having the request from `client`, process `resource` can now perform the request. In this simplified *Resource Manager System*, the atom

`action` is used as an expression to indicate the requested job. The global name `res` is used for the mapping of this atomic expression.

This model of the communication between the `client` and `manager` then between `client` and `resource` accurately reflects the semantics of the corresponding PIErlang program.

8. Further Refinements to Match and Send Expressions

Several case studies have been performed based on the mappings discussed above. Unfortunately, it was observed that the mappings of `match` discussed in subsections 4.10 and 6.4 are inadequate when handling Erlang bound variables. In Erlang a bound variable has the same behavior as an atom when it is used as a pattern or as an element of a tuple pattern in `case` and `receive` expressions and should retain this same meaning in the translated π–calculus system. Initially, we handled this situation by identifying all of the bound variables in a function before beginning its mapping and then treating them as atoms while translating. However, this was found to be infeasible for larger Erlang systems.

Instead, it is also possible to handle the problem of bound and unbound variables as part of translation mapping by maintaining a set of bound names while mapping. When a variable is used in the translation, we can then check whether there is a name with the same spelling and context in the Bound Name Set (BNS). If the name of the variable is already in the BNS, we consider that it is already bound with some value and can treat it as an atom rather than an unbound variable in match and tuple patterns. Initially, for each function in Erlang, the BNS is empty and a name `y` is added whenever `y` is used in a *receive action* (as in `y` in `x(y)`) or with the `new` operator (as in `z` in `( new z )`) in the π–calculus perspective.

However, we have observed that it is not necessary to include those added due to the `new` operator because we are modeling the binding of a variable in PIErlang, and this can only be done with a *receive action* in the π–calculus. We have also observed that we need not consider all names bound by the *receive action*, we need only to consider those that are original variables of the PIErlang program. Thus, we can refine the definition of BNS and have renamed the Bound Name Set (BNS) as Bound Names Set with *π–calculus Receive Actions of Erlang Variables* (BNSRAV).

The mapping of the `match` expression can now be formally refined using the BNSRAV as follows, where the complete mapping is obtained by taking into account all the P_i's:

$$TrPI_{match}(\texttt{self, \{}\ P_1\texttt{,...,}P_i\texttt{,...,}P_n\ \texttt{\}->}\ E) := \begin{cases} \texttt{self(..,input\_i,..)....[input\_i=}TrPI_{arg}(P_i)\texttt{]}\ ... \\ \quad TrPI_{exp}\texttt{(self,}\ E\texttt{);}\ \texttt{if}\ P_i \in \texttt{Numbers} \cup \texttt{Atoms} \\ \quad \texttt{or if}\ P_i \in \texttt{Variables and name}P_i \in \texttt{BNSRAV} \\ \\ \texttt{self(..,}P_i\texttt{,..)....}\ TrPI_{exp}\texttt{(self,}\ E\texttt{)} \\ \quad \texttt{if}\ P_i \in \texttt{Variables and name}P_i \notin \texttt{BNSRAV} \end{cases}$$

where `BNSRAV={X|y(X)` is a *receive action* in the π–calculus and `X` is a variable in the Erlang function for which `X` is considered as a name in the π–calculus.}

This new mapping covers both *monadic* (if `n=1`) and *polyadic* communications, so it can be used for all pattern matching in `matches` of `receive` and `case` expressions. The non–determinism among matches can be avoided using the same approach as discussed in Section 5 and a further improvement to this approach is currently under development.

Another important issue left to consider is the equality semantics (w.r.t. tuple size) of `send–receive` expressions. The translation mappings for `send` and `receive` expressions with tuples discussed above must meet the equality semantics of `send–receive` expressions with respect to the size of tuples used i.e., if a tuple of size `n` is sent with a `send` expression, then on the receiving side (`receive/case` expression), there must be at least one matching tuple pattern of size `n`. But in Erlang, the situation is different as a variable pattern can be matched with anything. If this semantics of Erlang cannot be preserved in the translated π–calculus system, the system will reach a deadlock state.

In order to overcome this problem of semantic matching between Erlang and the π–calculus, the mapping of the tuple-based `send` expression is refined as follows:

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

Along with the previous definition, two more subprocesses are added using non–deterministic choices. Whenever a tuple–based message is sent, a name `unknownTuple` is also sent non–deterministically so that the problem mentioned above can be solved. The reason for sending the message tuple along the `res` channel has already been discussed. For the same reason name `unknownTuple` is also sent along the `res` channel for the second non–determinism. As before, dealing with `res` can be avoided unless explicitly required to maintain the sequence of evaluation of expressions.

9. Related Work

Our work in using the π–calculus for verifying Erlang programs should be seen in the context of several similar projects. A number of similar approaches using different tools have been proposed by others.

This work is directly related and an extension to our previous work [18], a translation from a subset of Core Erlang [3, 4] to the asynchronous π–calculus with monadic communications, meta–level definitions of processes and name comparison. However, in that work the order of clauses in the evaluation of `receive` and `case` expressions was ignored and tuples were not modeled at all. In this work, we present an approach that accurately preserves the order of clauses in `receive` and `case` expressions and provides a detailed treatment of non–nested tuple–based expressions using *polyadic communication* in the π–calculus. Our initial approach in [18] handled only a subset of Core Erlang (an intermediate language in the Erlang compiler) whereas in this work handles a significant subset of Erlang [1] that can be further extended to cover the whole language in a more direct way. In order to provide a complete new framework for the mapping of Erlang, in this paper we have repeated some of the mappings of [18] using a more detailed presentation.

In [2, 20], a translation from Erlang to μCRL [23] is presented. The language μCRL is a process algebra with data. Several verification tools (e.g. CADP [24]) are available for μCRL, including a tool to create labelled transition systems from μCRL specifications. Using a translation from Erlang to μCRL, verification tools for process algebras and labelled transition systems can be applied to industrial code.

In [17], a translation of Erlang to Promela and model–checking of Promela using SPIN [22] has been prototyped. The main outcome of this experiment was that Promela (and hence SPIN) seems too semantically distant from Erlang to effectively use them together. Both these process–algebraic models [2, 17] for Erlang lack the capability to directly represent the sending of process identifier information between Erlang processes. Our approach overcomes this limitation using the name–passing feature of the π–calculus.

10. Conclusion

In this paper we have introduced a process–algebraic model for the functional programming language Erlang. We have chosen the π–calculus as a process algebra which is particularly appropriate for specifying mobile systems 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 direct representation of the sending of process identifier information between Erlang processes. This was not possible with previous attempts to use process–algebraic models for Erlang such as the work based on Promela/SPIN [17] or μCRL [2, 20].

The techniques of this paper improve on our previous work [18] in that we now support (non–nested) tuples as communication messages and deterministic pattern matching. However one has to keep in mind that we still have not dealt with all the syntactic constructs of the full Erlang language such as nested tuples and lists, which we abstract using the special `unknown` value. An attempt to refine our model can be found in [16]. Moreover we have not explicitly represented the mailbox aspect of Erlang processes, thereby abstracting away the ordering of messages. Currently, we are investigating possible ways of covering the full Erlang language. We are also working on real time Erlang [21] with a view to having a complete translation system for the Erlang language.

Acknowledgments

The authors would like to thank Simon Thompson and the other three anonymous referees for their useful comments and suggestions.

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 (HAL). `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 (MWB). `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. K. 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.

[18] T. Noll and C. K. Roy. Modeling Erlang in the Pi-Calculus. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, Tallinn, Estonia, 2005.

[19] Spatial Logic Model Checker (SLMC). `http://www-ctp.di.fct.unl.pt/SLMC/`

[20] T. Arts, C. Earle, and J. Penas. Translating Erlang to μCRL. In *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD2004)*. IEEE Computer Society Press, June 2004.

[21] M. Castro. *Erlang in Real Time*. RMIT University, 2001.

[22] The SPIN Model Checker. `http://spinroot.com/spin/whatispin.html`

[23] The μCRL Language. `http://homepages.cwi.nl/~mcrl/`

[24] Construction and Analysis of Distributed Processes(CADP) Tool. `http://www.inrialpes.fr/vasy/cadp/`

[25] The Behave Project. `http://research.microsoft.com/behave/`

[26] M. Leucker and T. Noll. A Distributed Model Checking Tool Tailored to Erlang. In *Proc. of Erlang Workshop at PLI'01*, 2001.