

# Towards Erlang Verification by Term Rewriting<sup>\*</sup>

Germán Vidal

MiST, DSIC, Universitat Politècnica de València  
Camino de Vera, s/n, 46022 Valencia, Spain  
gvidal@dsic.upv.es

**Abstract.** This paper presents a transformational approach to the verification of Erlang programs. We define a stepwise transformation from (first-order) Erlang programs to (non-deterministic) term rewrite systems that compute an overapproximation of the original Erlang program. In this way, existing techniques for term rewriting become available. Furthermore, one can use narrowing as a symbolic execution extension of rewriting in order to design a verification technique. We illustrate our approach with some examples, including a deadlock analysis of a simple Erlang program.

## 1 Introduction

The concurrent functional language Erlang [3] has a number of distinguishing features, like dynamic typing, concurrency via asynchronous message passing or hot code loading, that make it especially appropriate for distributed, fault-tolerant, soft real-time applications. The success of Erlang is witnessed by the increasing number of its industrial applications. For instance, Erlang has been used to implement Facebook’s chat back-end, the mobile application Whatsapp or Twitterfall—a service to view trends and patterns from Twitter—, to name a few. The success of the language, however, will also require the development of powerful testing and verification techniques.

In this work, we present a transformational approach to the verification of Erlang programs. We define a stepwise transformation from (first-order) Erlang programs to (non-deterministic) term rewrite systems that compute an overapproximation of the original Erlang programs. In contrast to direct approaches, one can reuse the large body of techniques and tools for term rewriting in order to design a verification tool for Erlang. The transformation, however, is far from trivial. Previous work along these lines include, for instance, [19], where a translation from Erlang to *rewriting logic* [16]—a unified semantic framework for concurrency—is introduced. In this case, though, the aim was to provide an executable specification of the language semantics (as a basis for the development of verification tools). Therefore, in this approach, Erlang programs are seen as data objects manipulated by a sort of interpreter implemented in rewriting logic. In contrast, our aim is to produce *plain* rewrite systems that keep the structure of the original Erlang program as much as possible, so that

---

<sup>\*</sup> This work has been partially supported by the *Generalitat Valenciana* under grant PROMETEO/2011/052.

they can be accurately analyzed using existing techniques. To be precise, we produce a number of rewrite rules—a constant factor of the size of the original program—that mimic the reductions of the original Erlang programs, and only a few fixed number of *state* reductions rules that deal with global concurrency actions (process spawning, message sending and receiving, etc.), which are common to every transformed system. In particular, if an Erlang program contains no concurrency actions, we produce a purely functional rewrite system so that the state reductions rules are not necessary.

The usefulness of our approach is illustrated by using it to verify safety properties with a *symbolic execution* extension of rewriting. Luckily, such an extension already exists and has been extensively studied. It is called *narrowing* [23], and represents a conservative extension of rewriting to deal with non-determinism and logic variables—representing missing information. In fact, the rewrite systems produced by our transformation are steadily executable in a so-called functional logic language like Curry [12], which opens up many possibilities for verifying safety properties. Furthermore, there already exist well studied subsumption and abstraction operators for guaranteeing the termination of narrowing while still producing a sound overapproximation (see, e.g., the narrowing-driven partial evaluation approach of [2]). Therefore, one could define a narrowing-based model checker by using similar operators. A first step towards this direction can be found in [18], where a technique for building finite narrowing trees is introduced (though reducing the number of states to avoid a combinatorial explosion is still a challenge).

The paper is organized as follows. Section 2 presents the syntax and semantics of the considered subset of Erlang. Section 3 introduces our stepwise transformation from Erlang programs to term rewriting systems. We illustrate the usefulness of the transformation for program verification in Section 4. Finally, Section 5 presents some related work and Section 6 concludes and points out some directions for further research.

## 2 Erlang Syntax and Semantics

In this section, we present the basic syntax and semantics of a significant subset of Erlang. In particular, we consider a simplified version of the language where some features are excluded (mainly higher-order calls, predefined functions, modules and exceptions) and some other features are slightly simplified. This is similar to the language considered by Huch [13] or Noll [19], and still includes the main features of Erlang: pattern matching, process creation, message sending and receiving, etc.

The basic objects of the language are variables (denoted by  $X, Y, \dots$ ), atoms (denoted by  $a, b, \dots$ ), process identifiers –pids– (denoted by  $p, p', \dots$ ), constructors (which are fixed in Erlang to lists, tuples and atoms), and defined functions (denoted by  $f/n, g/m, \dots$ ). The syntax for programs and expressions obeys the rules shown in Figure 1.

Programs are sequences of function definitions. Each function  $f/n$  is defined by a rule  $f(X_1, \dots, X_n) \rightarrow s$ , where  $X_1, \dots, X_n$  are distinct variables and the body of the function,  $s$ , can be an expression, a sequence of expressions,

$$\begin{aligned}
& \text{pgm} ::= f(X_1, \dots, X_n) \rightarrow s. \mid \text{pgm pgm} \\
\text{ErlangExp } \ni s & ::= e \mid s_1, s_2 \mid \text{case } e \text{ of } \text{clauses} \text{ end} \mid e_1 ! e_2 \\
& \quad \mid \text{receive } \text{clauses} \text{ end} \mid \text{pat} = e \mid \text{pat} = \text{self} \\
& \quad \mid \text{pat} = \text{spawn}(f(e_1, \dots, e_n)) \\
\text{Exp } \ni e & ::= f(e_1, \dots, e_n) \mid [e_1 | e_n] \mid [] \mid \{e_1, \dots, e_n\} \mid a \mid p \mid X \\
\text{clauses} & ::= \text{pat}_1 \rightarrow s_1; \dots; \text{pat}_n \rightarrow s_n \\
\text{Pat } \ni \text{pat} & ::= [\text{pat}_1 | \text{pat}_2] \mid [] \mid \{\text{pat}_1, \dots, \text{pat}_n\} \mid a \mid p \mid X \\
\text{Value } \ni v & ::= [v_1 | v_2] \mid [] \mid \{v_1, \dots, v_n\} \mid a \mid p
\end{aligned}$$

**Fig. 1.** Erlang syntax rules

a case distinction, message sending (e.g., `main ! {hello, world}` sends a message `{hello, world}` to the process with pid `main`) and receiving (e.g., `receive {A, B} → A end` reads a message from the process queue that matches the pattern `{A, B}` and returns `A`), pattern matching where the right-hand side can be an expression, the primitive `self` (that returns the pid of the current process) or a process creation (e.g., `spawn(foo(1, 2))` creates a new process<sup>1</sup> initialized to `foo(1, 2)`). Expressions can contain function calls, lists, tuples, atoms, pids and variables. Patterns are made of lists, tuples, atoms, pids and variables. Values are similar to patterns but cannot contain variables. Note that we only allow occurrences of `self` and `spawn` in the right-hand side of pattern matching. This is not a serious restriction since occurrences in other positions can be *flattened* by introducing fresh variables and pattern matching.

The domain of pids, `Pid`, and that of atoms, `Atom`, must be disjoint. For simplicity, we consider that pids are natural numbers starting from 1.

*Example 1.* Consider the following program which simply creates a new process and sends a message. The new process receives the message and does the same. Finally, the third process receives the message and returns ok.

$$\begin{array}{ll}
\text{proc1} \rightarrow \text{Pid1} = \text{spawn}(\text{proc2}), & \text{proc3} \rightarrow \text{receive} \\
\text{Pid1} ! a. & X \rightarrow \text{ok} \\
\text{proc2} \rightarrow \text{Pid2} = \text{spawn}(\text{proc3}), & \text{end.} \\
\text{receive} & \\
X \rightarrow \text{Pid2} ! X & \\
\text{end.} &
\end{array}$$

In the past, there have been several attempts to formalize the semantics of Erlang (e.g., [6, 7, 13, 17, 19, 20, 24]). In the following, we present an operational semantics for Erlang programs that mainly follows the approach of [13].

Erlang states are denoted by the parallel composition of their processes, where each process  $\langle p, e, q \rangle$  consists of a process identifier, an expression and a message queue:  $\text{Proc} ::= \text{Pid} \times \text{ErlangExp} \times \text{Value}^*$ . An *initial* state has the form  $\langle p, f(v_1, \dots, v_n), [] \rangle$  where  $f$  is a defined function,  $v_1, \dots, v_n$  are values,  $p$  is some initial pid and  $[]$  denotes an empty message queue; we will use lists to denote

<sup>1</sup> Note that we consider `spawn(foo(1, 2))` rather than the original Erlang notation `spawn(foo, [1, 2])` which is sensible since we do not allow higher order functions in this paper.

(seq)	$\frac{}{\langle p, C[v, s], q \rangle \& \Pi \longrightarrow \langle p, C[s], q \rangle \& \Pi}$
(self)	$\frac{}{\langle p, C[\text{self}], q \rangle \& \Pi \longrightarrow \langle p, C[p], q \rangle \& \Pi}$
(fun)	$\frac{f(X_1, \dots, X_n) \rightarrow s. \in \text{prog}}{\langle p, C[f(v_1, \dots, v_n)], q \rangle \& \Pi \longrightarrow \langle p, C[\hat{s}\{X_1 \mapsto v_1, \dots, X_n \mapsto v_n\}], q \rangle \& \Pi}$
(match)	$\frac{\exists \sigma. \text{pat} \sigma = v}{\langle p, C[\text{pat} = v], q \rangle \& \Pi \longrightarrow \langle p, (C[v])\sigma, q \rangle \& \Pi}$
(case)	$\frac{\exists i. \text{pat}_i \sigma = v \text{ for some } \sigma \wedge \nexists \sigma'. \text{pat}_j \sigma' = v \text{ for any } j < i}{\langle p, C[\text{case } v \text{ of } \text{pat}_1 \rightarrow s_1; \dots; \text{pat}_n \rightarrow s_n \text{ end}], q \rangle \& \Pi \longrightarrow \langle p, (C[s_i])\sigma, q \rangle \& \Pi}$
(spawn)	$\frac{p' \text{ is a fresh pid}}{\langle p, C[\text{spawn}(f(v_1, \dots, v_n))], q \rangle \& \Pi \longrightarrow \langle p, C[p'], q \rangle \& \langle p', f(v_1, \dots, v_n), [] \rangle \& \Pi}$
(send)	$\frac{v_1 = p' \in \text{Pid}}{\langle p, C[v_1 ! v_2], q \rangle \& \langle p', s, q' \rangle \& \Pi \longrightarrow \langle p, C[v_2], q \rangle \& \langle p', s, q' ++ [v_2] \rangle \& \Pi}$
(receive)	$\frac{\begin{array}{l} v_k \text{ is the first message such that} \\ (\exists i. \text{pat}_i \sigma = v \text{ for some } \sigma \wedge \nexists \sigma'. \text{pat}_j \sigma' = v \text{ for any } j < i) \end{array}}{\langle p, C[\text{receive } \text{pat}_1 \rightarrow s_1; \dots; \text{pat}_n \rightarrow s_n \text{ end}], [v_1, \dots, v_k, \dots, v_m] \rangle \& \Pi \longrightarrow \langle p, (C[s_i])\sigma, [v_1, \dots, v_{k-1}, v_{k+1}, \dots, v_m] \rangle \& \Pi}$

**Fig. 2.** Basic Erlang Semantics

message queues, where  $[]$  denotes an empty list and  $(x : xs)$  denotes a list with head  $x$  and tail  $xs$ . A *final* state has the form  $\langle p_1, v_1, q_1 \rangle \& \dots \& \langle p_n, v_n, q_n \rangle$  where  $v_1, \dots, v_n$  are values and “ $\&$ ” denotes the parallel composition operator. Computations start with an initial state and proceed until a final state is reached or the computation is blocked (otherwise, it proceeds forever).

The operational semantics is formalized by a state transition relation  $\longrightarrow$ : **State**  $\times$  **State**. Erlang follows a leftmost innermost operational semantics. Every expression can be decomposed into a context  $C[\ ]$  with a (single) hole and a subexpression  $s$  where the next reduction can take place.<sup>2</sup>

$$\begin{aligned}
C ::= & [] \mid C, s \mid \text{case } C \text{ of } \textit{clauses} \text{ end} \mid C ! e \mid v ! C \mid \text{pat} = C \\
& \mid \text{spawn}(f(v_1, \dots, v_i, C, e_{i+2}, \dots, e_n)) \mid f(v_1, \dots, v_i, C, e_{i+2}, \dots, e_n) \\
& \mid [v_1, \dots, v_i, C ! e] \mid \{v_1, \dots, v_i, C, e_{i+2}, \dots, e_n\}
\end{aligned}$$

The definition of the operational semantics is shown in Figure 2. Let us briefly explain the rules of the semantics:

- States are denoted by sequences of processes of the form  $\Gamma = \langle p, e, q \rangle \& \Pi$  where  $\Pi$  denotes a (possibly empty) parallel composition of processes. The order of processes is not relevant here (i.e.,  $\langle p, s, q \rangle$  might appear in any position within the pool of processes  $\Gamma$ ).

<sup>2</sup> This is similar to the reduction contexts of [9] and allows us to deterministically identify the next expression to be reduced.

$$\begin{array}{l}
\langle 1, \text{proc1}, [] \rangle \\
\longrightarrow_{\text{fun}} \langle 1, \text{Pid1} = \text{spawn}(\text{proc2}), \text{Pid1} ! a, [] \rangle \\
\longrightarrow_{\text{spawn}} \langle 1, \underline{\text{Pid1} = 2}, \text{Pid1} ! a, [] \rangle \& \langle 2, \text{proc2}, [] \rangle \\
\longrightarrow_{\text{match}} \langle 1, 2 ! a, [] \rangle \& \langle 2, \text{proc2}, [] \rangle \\
\longrightarrow_{\text{seq}} \langle 1, 2 ! a, [] \rangle \& \langle 2, \text{proc2}, [] \rangle \\
\longrightarrow_{\text{fun}} \langle 1, 2 ! a, [] \rangle \& \langle 2, \text{Pid2} = \text{spawn}(\text{proc3}), \text{receive } X \rightarrow \text{Pid2} ! X \text{ end}, [] \rangle \\
\longrightarrow_{\text{spawn}} \langle 1, 2 ! a, [] \rangle \& \langle 2, \underline{\text{Pid2} = 3}, \text{receive } X \rightarrow \text{Pid2} ! X \text{ end}, [] \rangle \\
\qquad \qquad \qquad \& \langle 3, \text{proc3}, [] \rangle \\
\longrightarrow_{\text{match}} \langle 1, 2 ! a, [] \rangle \& \langle 2, 3, \text{receive } X \rightarrow 3 ! X \text{ end}, [] \rangle \\
\qquad \qquad \qquad \& \langle 3, \text{proc3}, [] \rangle \\
\longrightarrow_{\text{seq}} \langle 1, 2 ! a, [] \rangle \& \langle 2, \text{receive } X \rightarrow 3 ! X \text{ end}, [] \rangle \& \langle 3, \text{proc3}, [] \rangle \\
\longrightarrow_{\text{send}} \langle 1, a, [] \rangle \& \langle 2, \text{receive } X \rightarrow 3 ! X \text{ end}, [a] \rangle \& \langle 3, \text{proc3}, [] \rangle \\
\longrightarrow_{\text{receive}} \langle 1, a, [] \rangle \& \langle 2, 3 ! a, [] \rangle \& \langle 3, \text{proc3}, [] \rangle \\
\longrightarrow_{\text{fun}} \langle 1, a, [] \rangle \& \langle 2, 3 ! a, [] \rangle \& \langle 3, \text{receive } X \rightarrow \text{ok end}, [] \rangle \\
\longrightarrow_{\text{send}} \langle 1, a, [] \rangle \& \langle 2, a, [] \rangle \& \langle 3, \underline{\text{receive } X \rightarrow \text{ok end}}, [a] \rangle \\
\longrightarrow_{\text{receive}} \langle 1, a, [] \rangle \& \langle 2, a, [] \rangle \& \langle 3, \text{ok}, [] \rangle
\end{array}$$

**Fig. 3.** Computation for the program of Example 1

- Rule **self** reduces the predefined atom **self** to the process identifier of the current process.
- Rule **fun** performs a function unfolding, where  $\hat{s}$  denotes an expression  $s$  in which the free variables of patterns (if any) have been replaced by fresh variables to avoid name conflicts.
- Rules **match** and **case** deal with pattern matching. In both cases, we assume  $\sigma$  to be the minimal matching substitution and restricted to the variables of the pattern. For case expressions, we should select the *first* matching branch. Observe that we do not have rules for pattern matching failures, which are considered program errors and left out of this work.
- Rule **spawn** creates a new process with a fresh pid.
- Finally, rules **send** and **receive** deal with message passing and receiving. Note that **receive** should select the *first* message in the process queue that matches some pattern.

The semantics is clearly deterministic in the sense that, given a single process, there is only one applicable rule. However, we can define different strategies for selecting processes when there are more than one reducible process. In this paper, a fair selection strategy is assumed (e.g., a round-robin scheduling).

*Example 2.* Consider again the program of Example 1. A computation with this program is shown in Figure 3, where the reduced subexpression is underlined for clarity; moreover, we label the transitions with the applied rule. Therefore, the computation terminates and reaches a final state.

### 3 From Erlang Processes to Term Rewriting

In this section, we present a stepwise transformation from Erlang programs to term rewrite systems.

### 3.1 Term Rewriting

Here, we recall some basic notions and notations of term rewriting (see, e.g., [5] for more details). A *signature*  $\mathcal{F}$  is a set of function symbols. Given a set of variables  $\mathcal{V}$  with  $\mathcal{F} \cap \mathcal{V} = \emptyset$ , we denote the domain of *terms* by  $\mathcal{T}(\mathcal{F}, \mathcal{V})$ . Positions are used to address the nodes of a term viewed as a tree. A *position*  $p$  in a term  $t$  is represented by a finite sequence of natural numbers, where  $\epsilon$  denotes the root position. We let  $t|_p$  denote the *subterm* of  $t$  at position  $p$  and  $t[s]_p$  the result of *replacing the subterm*  $t|_p$  by the term  $s$ .  $\text{Var}(t)$  denotes the set of variables appearing in  $t$ . A *substitution*  $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{V})$  is a mapping from variables to terms such that  $\text{Dom}(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$  is its domain. Substitutions are extended to morphisms from  $\mathcal{T}(\mathcal{F}, \mathcal{V})$  to  $\mathcal{T}(\mathcal{F}, \mathcal{V})$  in the natural way. We denote the application of a substitution  $\sigma$  to a term  $t$  by  $t\sigma$  rather than  $\sigma(t)$ . The identity substitution is denoted by *id*.

A set of rewrite rules  $l \rightarrow r$  such that  $l$  is a nonvariable term and  $r$  is a term whose variables appear in  $l$  is called a *term rewriting system* (TRS for short); terms  $l$  and  $r$  are called the left-hand side and the right-hand side of the rule, respectively. We restrict ourselves to finite signatures and TRSs. Given a TRS  $\mathcal{R}$  over a signature  $\mathcal{F}$ , the *defined symbols*  $\mathcal{D}_{\mathcal{R}}$  are the root symbols of the left-hand sides of the rules and the *constructors* are  $\mathcal{C}_{\mathcal{R}} = \mathcal{F} \setminus \mathcal{D}_{\mathcal{R}}$ . *Constructor terms* of  $\mathcal{R}$  are terms over  $\mathcal{C}_{\mathcal{R}}$  and  $\mathcal{V}$ . We sometimes omit  $\mathcal{R}$  from  $\mathcal{D}_{\mathcal{R}}$  and  $\mathcal{C}_{\mathcal{R}}$  if it is clear from the context.

For a TRS  $\mathcal{R}$ , we define the associated rewrite relation  $\rightarrow_{\mathcal{R}}$  as follows: given terms  $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ , we have  $s \rightarrow_{\mathcal{R}} t$  iff there exists a position  $p$  in  $s$ , a rewrite rule  $l \rightarrow r \in \mathcal{R}$  and a substitution  $\sigma$  with  $s|_p = l\sigma$  and  $t = s[r\sigma]_p$ ; the rewrite step is often denoted by  $s \rightarrow_{p, l \rightarrow r} t$  to make explicit the position and rule used in this step. The instantiated left-hand side  $l\sigma$  is called a *redex*.

A *derivation* is a (possibly empty) sequence of rewrite steps. Given a binary relation  $\rightarrow$ , we denote by  $\rightarrow^*$  its reflexive and transitive closure. Thus  $t \rightarrow_{\mathcal{R}}^* s$  means that  $t$  can be reduced to  $s$  in  $\mathcal{R}$  in zero or more steps.

### 3.2 The Transformation

Our transformation is driven by the following principles:

*Keep the original structure.* We try to keep the structure of the original Erlang programs as much as possible. In particular, an Erlang program without concurrent features would be mostly untouched. This is useful to keep the analyses performed on the transformed rewrite system as accurate as possible.

*Overapproximate.* Several Erlang constructs cannot be translated to a rewrite system while preserving their original semantics (unless a number of rather complex functions are introduced, which would be a drawback for the analysis of the resulting system). This is the case, for instance, of a case expression. While Erlang only considers the first matching clause, our translation will produce an auxiliary function that considers *all* matching clauses. Therefore, in general, we will produce rewrite systems that represent overapproximations of the original Erlang programs.

As a consequence, we cannot ensure that a bug detected in the transformed rewrite system is an actual bug of the original Erlang program, i.e., false positives may occur. For instance, the analysis of the rewrite system may point out that a deadlock may occur, while this is not the case in the original Erlang program. On the other hand, if the analysis of the rewrite system allows one to conclude that no deadlock can occur, then this is surely the case in the original Erlang program.

*Concurrent actions by continuation functions.* Loosely speaking, our transformation replaces every concurrent operator with a new constructor: SPAWN, SEND, RECEIVE and SELF. Then, we define a set of rewrite rules that deal with states and take care of concurrent actions. The challenge here is to always have these constructors in a topmost position of a process so that a rule can be applied without requiring complex context rules (in contrast to, e.g., [19]).

For this purpose, we introduce some auxiliary functions that can be seen as *continuations* of the original functions (see below).

We formalize our transformation  $\llbracket \cdot \rrbracket$  as follows. Given an Erlang program  $P$ , we have:

$$\llbracket P \rrbracket = \{f(x_1, \dots, x_n) \rightarrow \llbracket s \cdot \rrbracket^V \mid f(x_1, \dots, x_n) \rightarrow s. \in P\}$$

where  $V = \{x_1, \dots, x_n\} \cap \mathcal{FVar}(s)$  is used for introducing auxiliary functions with appropriate parameters. In the following,  $\mathcal{FVar}(s)$  denotes the free variables of  $s$ . Now, we define the transformation function  $\llbracket \cdot \rrbracket$  on every program construct.

**Case Expressions.** Let us first consider the transformation of case expressions. This can easily be transformed by introducing an auxiliary function as follows:

$$\llbracket \text{case } e \text{ of } p_1 \rightarrow s_1, \dots, p_n \rightarrow s_n \text{ end.} \rrbracket^V = f(e, \bar{V})$$

where  $f$  is a fresh function symbol and  $\bar{V}$  denotes a list with the variables of set  $V$ . Here, the auxiliary function  $f$  is defined as follows:

$$f(p_1, \bar{V}) \rightarrow \llbracket s_1 \cdot \rrbracket^{V_1} \quad \dots \quad f(p_n, \bar{V}) \rightarrow \llbracket s_n \cdot \rrbracket^{V_n}$$

where  $V_i = \mathcal{Var}(f(p_i, \bar{V})) \cap \mathcal{FVar}(s_i)$ ,  $i = 1, \dots, n$ . When the case expression is not the last statement in the right-hand side, we proceed analogously as follows:

$$\llbracket \text{case } e \text{ of } p_1 \rightarrow s_1, \dots, p_n \rightarrow s_n \text{ end, } s. \rrbracket^V = f(e, \bar{V})$$

where the auxiliary function  $f$  is now defined by

$$f(p_1, \bar{V}) \rightarrow \llbracket s_1, s \cdot \rrbracket^{V_1} \quad \dots \quad f(p_n, \bar{V}) \rightarrow \llbracket s_n, s \cdot \rrbracket^{V_n}$$

where  $V_i = \mathcal{Var}(f(p_i, \bar{V})) \cap \mathcal{FVar}(s_i, s)$ ,  $i = 1, \dots, n$ .

Observe that this transformation implies that, in general, the transformed function will compute an *overapproximation* of the original Erlang program when there are overlapping patterns (since rewriting considers *all* matching patterns).

**Message Passing.** In this case, we transform an expression  $p ! e$  using a new constructor  $\text{SEND}(i, p, e, vars)$ , where  $i$  is a unique identifier and  $vars$  is a list of variables. We distinguish the following cases:

$$\llbracket e_1 ! e_2. \rrbracket^V = \text{SEND}(i, e_1, e_2, []) \quad \text{with } \text{send}(i, v, \_) \rightarrow v$$

where  $i$  is a fresh constant symbol (e.g., a number),  $v$  is a fresh variable, and “ $\_$ ” denotes an *anonymous* variable (i.e., a variable whose name is not relevant because it does not occur in the right-hand side).

In contrast to ordinary functions and the auxiliary functions introduced when transforming a case expression,  $\text{SEND}$  is a constructor symbol that will require the (global) system rules to be dealt with. Roughly speaking, the system rules will rewrite  $\text{SEND}(i, e_1, e_2, [])$  to  $\text{send}(i, e_2, [])$ —the continuation of  $\text{SEND}$ —and will also store  $e_2$  in the mailbox of the process with pid  $e_1$ .

When the message passing is not the last construct of the sequence, we have

$$\llbracket e_1 ! e_2, s. \rrbracket^V = \text{SEND}(i, e_1, e_2, \bar{V}) \quad \text{with } \text{send}(i, \_, \bar{V}) \rightarrow \llbracket s. \rrbracket^{V'}$$

where  $i$  is a fresh constant symbol and  $V' = V \cap \mathcal{FVar}(s)$ . In this case, the system rules will proceed analogously but the value of  $e_2$  is lost (as it will happen in the original Erlang program).

**Message Reception.** Here, we introduce a new constructor  $\text{AREC}(i, list, vars)$ , where  $i$  is a unique identifier,  $list$  is the list of messages already processed (initially empty), and  $vars$  is a list of variables. We transform Erlang expressions as follows:

$$\llbracket \text{receive } p_1 \rightarrow s_1, \dots, p_n \rightarrow s_n \text{ end.} \rrbracket^V = \text{AREC}(i, [], \bar{V})$$

where  $i$  is a fresh a constant symbol (e.g., a number). The following auxiliary functions are added to the program:

$$\begin{array}{ll} \text{brec}(i, p_1) \rightarrow \text{True} & \text{rec}(i, p_1, \bar{V}) \rightarrow \llbracket s_1. \rrbracket^{V_1} \\ \dots & \dots \\ \text{brec}(i, p_n) \rightarrow \text{True} & \text{rec}(i, p_n, \bar{V}) \rightarrow \llbracket s_n. \rrbracket^{V_n} \end{array}$$

where  $V_j = \mathcal{Var}(\text{rec}(i, p_j, \bar{V})) \cap \mathcal{FVar}(s_j)$ ,  $j = 1, \dots, n$ . When the receive construct is not the last expression of a sequence, we proceed analogously as follows:

$$\llbracket \text{receive } p_1 \rightarrow s_1, \dots, p_n \rightarrow s_n \text{ end, } s. \rrbracket^V = \text{AREC}(i, [], \bar{V})$$

with

$$\begin{array}{ll} \text{brec}(i, p_1) \rightarrow \text{True} & \text{rec}(i, p_1, \bar{V}) \rightarrow \llbracket s_1, s. \rrbracket^{V_1} \\ \dots & \dots \\ \text{brec}(i, p_n) \rightarrow \text{True} & \text{rec}(i, p_n, \bar{V}) \rightarrow \llbracket s_n, s. \rrbracket^{V_n} \end{array}$$

where  $V_j = \mathcal{Var}(\text{rec}(i, p_j, \bar{V})) \cap \mathcal{FVar}(s_j, s)$ ,  $j = 1, \dots, n$ .



Loosely speaking, the system reduction rules will rewrite  $\text{AREC}(i, [], \bar{V})$  to  $\text{rec}(i, m, \bar{V})$ —the continuation of  $\text{AREC}$ —when  $\text{brec}(i, m)$  is true, where  $m$  is the first message in the process mailbox; otherwise, the message  $m$  is moved to the second parameter of  $\text{AREC}$  and the traversal of the mailbox continues. When the mailbox is empty (i.e., no message matched the patterns of the receive clause), we restore the mailbox and move the process to the end of the list.

Similar to the case statements, the transformed TRS will compute an overapproximation of the original Erlang program when there are overlapping patterns.

**Pattern Matching.** First, we consider a pattern matching in which the right-hand side is an expression not including calls to `spawn` nor `self`. In this case, it is transformed analogously to a case statement with a single case:

$$\llbracket p = e. \rrbracket^V = f(e, \bar{V}) \quad \text{with } f(p, \bar{V}) \rightarrow p.$$

where  $f$  is a fresh function symbol. When the pattern matching is not the last element of a sequence, we proceed as follows:

$$\llbracket p = e, s. \rrbracket^V = f(e, \bar{V}) \quad \text{with } f(p, \bar{V}) \rightarrow \llbracket s. \rrbracket^{V'}$$

where  $f$  is a fresh function symbol and  $V' = \mathcal{V}ar(f(p, \bar{V})) \cap \mathcal{F}\mathcal{V}ar(s)$ .

**Process Creation.** Process are created using the predefined function `spawn`. Here, we introduce a new constructor  $\text{SPAWN}(i, \text{exp}, \text{vars})$ , where  $i$  is a unique identifier,  $\text{exp}$  is the function call that starts the new process, and  $\text{vars}$  is a list of variables. First, we distinguish the following case:

$$\llbracket p = \text{spawn}(e). \rrbracket^V = \text{SPAWN}(i, e, []) \quad \text{with } \text{spawn}(i, p, -) \rightarrow p$$

where  $i$  is a fresh constant. Basically, the auxiliary function  $\text{spawn}$ —the continuation of  $\text{SPAWN}$ —will be called from the system reduction rules with a second argument that contains the pid of the new process. When the pattern matching is not the last element in a sequence, we proceed as follows:

$$\llbracket p = \text{spawn}(e), s. \rrbracket^V = \text{SPAWN}(i, e, \bar{V}) \quad \text{with } \text{spawn}(i, p, \bar{V}) \rightarrow \llbracket s. \rrbracket^{V'}$$

where  $i$  is a fresh constant and  $V' = \mathcal{V}ar(\text{spawn}(i, p, \bar{V})) \cap \mathcal{F}\mathcal{V}ar(s)$ .

**The Primitive self.** We replace the occurrences of `self` with a new constructor  $\text{SELF}(i, \text{vars})$ , where  $i$  is a unique identifier and  $\text{vars}$  is a list of variables. We distinguish the following cases:

$$\llbracket p = \text{self}. \rrbracket^V = \text{SELF}(i, []) \quad \text{with } \text{self}(i, p, -) \rightarrow p$$

where  $i$  is a fresh constant symbol. Here, the system reduction rules will check the pid of the process and will call the auxiliary function  $\text{self}$ —the continuation of  $\text{SELF}$ —with this pid as a second parameter. When the pattern matching is not the last element in a sequence, we proceed as follows:

$$\llbracket p = \text{self}, s. \rrbracket^V = \text{SELF}(i, \bar{V}) \quad \text{with } \text{self}(i, p, \bar{V}) \rightarrow \llbracket s. \rrbracket^{V'}$$

where  $i$  is a fresh constant symbol and  $V' = \mathcal{V}ar(\text{self}(i, p, \bar{V})) \cap \mathcal{F}\mathcal{V}ar(s)$ .

$$\begin{aligned}
\langle \langle 0, k, [] \rangle : \langle \langle i, \text{SPAWN}(n, e, vs), m \rangle : s \rangle \rangle &\rightarrow \langle \langle 0, k + 1, [] \rangle : s \rangle \\
&\quad ++ [\langle i, \text{spawn}(n, k, vs), m \rangle, \langle k, e, [] \rangle] \\
(s_0 : \langle \langle i, \text{SEND}(n, j, e, vs), m \rangle : s \rangle) &\rightarrow (s_0 : \text{send\_msg}(j, e, s ++ [\langle i, \text{send}(n, e, vs), m \rangle])) \\
(s_0 : \langle \langle i, \text{SEND}(n, j, e, vs), m \rangle : s \rangle) &\rightarrow (s_0 : (s ++ [\langle i, \text{SEND}(n, j, e, vs), m \rangle])) \\
(s_0 : \langle \langle i, \text{AREC}(n, ms_2, vs), m : ms \rangle : s \rangle) &\rightarrow (s_0 : s ++ [\langle i, \text{rec}(n, m, vs), (ms_2 ++ ms) \rangle]) \\
&\quad \text{if } \text{brec}(n, m) \\
(s_0 : \langle \langle i, \text{AREC}(n, ms_2, vs), m : ms \rangle : s \rangle) &\rightarrow (s_0 : \langle \langle i, \text{AREC}(n, ms_2 ++ [m], vs), ms \rangle : s \rangle) \\
&\quad \text{if } \text{not}(\text{brec}(n, m)) \\
(s_0 : \langle \langle i, \text{AREC}(n, ms_2, vs), [] \rangle : s \rangle) &\rightarrow (s_0 : s) ++ [\langle i, \text{AREC}(n, ms_2, vs), [] \rangle] \\
\langle \langle 0, k, [] \rangle : \langle \langle i, \text{SELF}(n, vs), m \rangle : s \rangle \rangle &\rightarrow \langle \langle 0, k, [] \rangle : s \rangle ++ [\langle i, \text{self}(n, i, vs), m \rangle] \\
(s_0 : \langle \langle i, p, m \rangle : s \rangle) &\rightarrow (s_0 : (s ++ [\langle i, p, m \rangle]))
\end{aligned}$$

**Fig. 4.** State reduction rules

**Sequences** Most of the sequences are transformed away using the previous transformations. However, some of them may still remain in the transformed program. In this case, they are transformed as follows:

$$\llbracket s_1, s_2. \rrbracket^V = \llbracket \text{case } s_1 \text{ of } \_ \rightarrow s_2 \text{ end.} \rrbracket^V$$

so that all remaining sequences are removed from the transformed program.

**Expressions.** For the remaining expressions, we have  $\llbracket e. \rrbracket = e$ . Note that we assumed that no occurrence of the concurrency primitives: **!**, **receive**, **self**, etc., can occur in expressions. Note that this is not a real restriction since these statements could be flattened by introducing fresh variables and pattern matching.

### 3.3 State Reduction Rules

Processes are denoted by tuples  $\langle p, e, q \rangle$ , which consists of a process identifier  $p$ , an expression  $e$ , and a message queue  $q$ , as introduced in Section 2. We consider natural numbers as pids, starting from 1. Also, we have an *artificial* (first) process of the form  $\langle 0, n, [] \rangle$  that is only used for storing the first free pid  $n$ , so that we do not need to compute it every time **spawn** is called.

Basically, a *system* is represented by a list of processes, where the first process is always the one that stores the current free pid number. We consider the usual notation for lists:  $[]$  and  $(- : -)$ , where  $++$  denotes list concatenation. We consider a breadth-first exploration of the search space regarding concurrent actions (so that the considered process is always moved to the end of the current list). Let us briefly describe the rules:

- **SPAWN.** A process with a constructor call  $\text{SPAWN}(n, e, vars)$  is reduced by creating a new process initialized with the expression  $e$ , and replacing the

$proc1$	$\rightarrow SPAWN(1, proc2, [])$	$proc2$	$\rightarrow SPAWN(3, proc3, [])$
$spawn(1, pid1, [])$	$\rightarrow SEND(2, pid1, A, [])$	$spawn(3, pid2, [])$	$\rightarrow AREC(4, [], [pid2])$
$send(2, e, -)$	$\rightarrow e$	$brec(4, x)$	$\rightarrow True$
		$rec(4, x, [pid2])$	$\rightarrow SEND(5, pid2, x, [])$
		$send(5, e, -)$	$\rightarrow e$
$proc3$	$\rightarrow AREC(6, [], [])$		
$brec(6, x)$	$\rightarrow True$		
$rec(6, x, [])$	$\rightarrow Ok$		

**Fig. 5.** TRS associated to the Erlang program of Example 1

constructor call with a call to the auxiliary function  $spawn(n, k, vars)$ , where  $k$  is the pid number of the new process (which is then updated to  $k+1$ ). Note also that both the reduced process and the newly created one are moved to the end of the list.

- **SEND.** Here, and in order to explore all possible schedulings, we consider two non-deterministic alternatives. The first rule sends the message (using the auxiliary function  $send\_msg$ ), while the second rule just moves the process to the end of the queue thus delaying the message delivery. In this way, we can explore all possible process schedulings. The definition of the auxiliary function  $send\_msg$  is straightforward (and can be found in the next section).
- **AREC.** For receiving a message, we consider three possibilities. First, we check whether the first message in the mailbox matches any of the receive clauses. If so, we process the message using a call to the auxiliary function  $rec$ . Otherwise, we move the first message to the second parameter of **AREC** and continue inspecting the mailbox. When the mailbox is empty (either because no message has been received or because we have already inspected all of them), the mailbox is restored and the process is moved to the end of the list.
- Finally, we also include a rule that just moves a *finished* process to the end of the list. One could also remove it from the pool of processes, but we prefer to keep it for analysis and debugging purposes. Note that this rule does not overlap with the previous rules since the process must have a *value* not including the special constructors **SPAWN**, **SEND**, etc. For simplicity, in this work we assume that there are no non-terminating functions that are purely functional (i.e., without occurrences of **SPAWN**, **SEND**, **AREC** or **SELF**). Otherwise, one would also need to ensure that no expression in a process is rewritten infinitely.

*Example 3.* Let us consider the Erlang program of Example 1. This program is transformed into the TRS shown in Fig. 3, where functions and variables start with a lowercase letter, and constructors start with an uppercase letter.

The computation shown in Example 2 for the Erlang program proceeds now as shown in Fig. 6.<sup>3</sup> Here, we reach exactly the same final state of Fig. 3. Note, however, that due to non-determinism, other computations are also possible.

<sup>3</sup> We underline either the expression or the selected process involved in a reduction step.

```

[⟨0, 2, []⟩, ⟨1, proc1, []⟩]
→ [⟨0, 2, []⟩, ⟨1, SPAWN(1, proc2, [], []), []⟩]
→ [⟨0, 3, []⟩, ⟨1, spawn(1, 2, [], []), [], ⟨2, proc2, []⟩]
→ [⟨0, 3, []⟩, ⟨1, SEND(2, 2, A, [], []), [], ⟨2, proc2, []⟩]
→ [⟨0, 3, []⟩, ⟨2, proc2, [A]⟩, ⟨1, send(2, A, [], []), []⟩]
→ [⟨0, 3, []⟩, ⟨2, SPAWN(3, proc3, [], [A]⟩, ⟨1, send(2, A, [], []), []⟩]
→ [⟨0, 4, []⟩, ⟨1, send(2, A, [], []), [], ⟨2, spawn(3, 3, [], [A]⟩, ⟨3, proc3, []⟩]
→ [⟨0, 4, []⟩, ⟨1, A, []⟩, ⟨2, spawn(3, 3, [], [A]⟩, ⟨3, proc3, []⟩]
→ [⟨0, 4, []⟩, ⟨2, spawn(3, 3, [], [A]⟩, ⟨3, proc3, []⟩, ⟨1, A, []⟩]
→ [⟨0, 4, []⟩, ⟨2, AREC(4, [], [3]), [A]⟩, ⟨3, proc3, []⟩, ⟨1, A, []⟩]
→ [⟨0, 4, []⟩, ⟨3, proc3, []⟩, ⟨1, A, []⟩, ⟨2, rec(4, A, [3]), []⟩]
→ [⟨0, 4, []⟩, ⟨3, AREC(6, [], [], []), [], ⟨1, A, []⟩, ⟨2, rec(4, A, [3]), []⟩]
→ [⟨0, 4, []⟩, ⟨1, A, []⟩, ⟨2, rec(4, A, [3]), [], ⟨3, AREC(6, [], [], []), []⟩]
→ [⟨0, 4, []⟩, ⟨2, rec(4, A, [3]), [], ⟨3, AREC(6, [], [], []), [], ⟨1, A, []⟩]
→ [⟨0, 4, []⟩, ⟨2, SEND(5, 3, A, [], []), [], ⟨3, AREC(6, [], [], []), [], ⟨1, A, []⟩]
→ [⟨0, 4, []⟩, ⟨3, AREC(6, [], [], [A]⟩, ⟨1, A, []⟩, ⟨2, send(5, A, [], []), []⟩]
→ [⟨0, 4, []⟩, ⟨1, A, []⟩, ⟨2, send(5, A, [], []), [], ⟨3, rec(6, A, [], []), []⟩]
→ [⟨0, 4, []⟩, ⟨2, send(5, A, [], []), [], ⟨3, rec(6, A, [], []), [], ⟨1, A, []⟩]
→ [⟨0, 4, []⟩, ⟨2, A, []⟩, ⟨3, rec(6, A, [], []), [], ⟨1, A, []⟩]
→ [⟨0, 4, []⟩, ⟨3, rec(6, A, [], []), [], ⟨1, A, []⟩, ⟨2, A, []⟩]
→ [⟨0, 4, []⟩, ⟨3, Ok, []⟩, ⟨1, A, []⟩, ⟨2, A, []⟩]

```

**Fig. 6.** Example of reduction

Proving that the transformed program computes an overapproximation (i.e., that every computation of the original program can be mimicked in the transformed one) is not difficult. A sketch of the proof can be found in the appendix.

## 4 The Transformation in Practice

In this section, we show the usefulness of our transformation in the context of program verification. An implementation of the transformation has been undertaken and can be used through a web interface that can be found here:

<http://users.dsic.upv.es/~gvidal/erlang2trs/>

For verifying safety properties, we consider the execution of the rewriting system using *narrowing* [14, 23], a conservative extension of term rewriting for dealing with non-determinism and logic variables. Narrowing can be seen as a symbolic execution version of rewriting where pattern matching is replaced with unification (as in logic programming). Basically, given a TRS  $\mathcal{R}$  and two terms  $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ , we have that  $s \rightsquigarrow_{\mathcal{R}} t$  is a *narrowing step* iff there exist a nonvariable position  $p$  of  $s$ , a variant  $l \rightarrow r$  of a rule in  $\mathcal{R}$ , and a substitution  $\sigma$  which is a most general unifier of  $s|_p$  and  $l$ , with  $t = (s[r]_p)\sigma$ . For instance, narrowing has been used as the basis of a partial evaluation framework for rewrite systems [2].

In particular, in order to produce executable programs, we consider the language Curry [12] (a conservative extension of Haskell to deal with logic variables and non-determinism).

*Example 4.* Consider the following Erlang program:

```

main → Pid2 = spawn(proc2),
      Pid1 = spawn(proc1(Pid2)),
      Pid1 ! hello,
      Pid2 ! world.
proc1(Pid) → receive
                X → Pid ! X
            end.
proc2 → receive
                X → ok
            end.

```

Our transformation tool `erlang2trs` returns the following program (we use a curried notation for functions as in Curry):

```

main = (SPAWN 1 proc2 [])
spawn 1 pid2 [] = (SPAWN 2 (proc1 pid2) (pid2:[]))
spawn 2 pid1 (pid2:[]) = (SEND 3 pid1 Hello (pid1:(pid2:[])))
send 3 e (pid1:(pid2:[])) = (SEND 4 pid2 World [])
send 4 e fresh = e

proc1 pid = (AREC 5 [] (pid:[]))
brec 5 x = True
rec 5 x (pid:[]) = (SEND 6 pid x [])
send 6 e fresh = e

proc2 = (AREC 7 [] [])
brec 7 x = True
rec 7 x [] = Ok

```

together with data declarations, the system reduction rules and a few auxiliary functions:

```

data State = State Int Exp [Exp]
data Exp = I Int | SPAWN Int Exp [Exp] | SEND Int Exp Exp [Exp]
          | AREC Int [Exp] [Exp] | SELF Int [Exp]
          | World | Hello | Ok

reduce (s0 : (State i (AREC n ms2 args) (m:ms)) : s) visited
= if (brec n m)
    then reduce (s0:(s++[State i (rec n m args) (ms2++ms)])) visited
    else reduce (s0:(State i (AREC n (ms2++[m]) args) ms):s) visited
reduce (s0 : (State i (AREC n ms2 args) []) : s) visited
= reduce ((s0 : s) ++ [State i (AREC n ms2 args) []]) visited

reduce (State o (I k) l2 : (State i (SPAWN n e args) m : s)) visited
= reduce ((State o (I (k+1)) l2 : s)
++ [State i (spawn n (I k) args) m, State k e []]) visited

```

```

reduce (s0 : (State i (SEND n (I j) e args) m : s)) visited
  = reduce (s0:(send_msg j e (s++[State i (send n e args) m]))) visited
reduce (s0 : (State i (SEND n (I j) e args) m : s)) visited
  = reduce (s0 : (s ++ [State i (SEND n (I j) e args) m])) visited

send_msg _ _ [] = []
send_msg j e (State i b m : s)
  | i==j      = State i b (m++[e]) : s
  | otherwise = State i b m : (send_msg j e s)

brec 5 fresh = case fresh of
  x -> True
  _ -> False

brec 7 fresh = case fresh of
  x -> True
  _ -> False

```

The complete code of the transformed program can be found at

<http://users.dsic.upv.es/~gvidal/erlang2trs/>

Consider now that we are interested in verifying whether the message “World” can arrive to *proc3* before the message “Hello”. We can easily check this property in Curry using the following test function:

```

init = reduce [State 0 (I 2) [], State 1 main []] []

test = wrongState init
wrongState (s:ss) = case s of
  State _ Ok [Hello] -> True
  _ -> wrongState ss

```

where the state reduction rules are implemented by function `reduce` and states are represented using the constructor `State`. Here, function `init` denotes the initial state and function `test` checks if there exists a reachable final state (i.e., where the main expression is reduced to `Ok`) with the message `Hello` in the mailbox.

*Example 5.* Consider now the Erlang program shown in Fig. 7. This program represents a simplified version of the well-known dining philosophers problem. Basically, there are two processes (*left* and *right*) that compete for a couple of shared resources, which are managed by process *res*—that accepts both queries  $\{q, pid\}$  and updates  $\{u, new\_state\}$ —which is initialized with the state  $\{f, f\}$  (i.e., both resources are free). The *left* process takes the resources from left to right and the *right* process from right to left. We have not considered freeing the resources since we are only interested in illustrating the definition of a deadlock analysis in this example.

```

main → Res = spawn(res({f,f})),   res(St) → receive {q,S} → S ! St
      L = spawn(left(Res)),        end, receive {u,NSt} → res(NSt)
      R = spawn(right(Res)).       end.

left(Res) → S = self,              right(Res) → S = self,
      Res ! {q,S},                  Res ! {q,S},
      receive {f,B} →               receive {A,f} →
      Res ! {u,{b,B}},              Res ! {u,{A,b}},
      Res ! {q,S},                  Res ! {q,S},
      left2(Res)                     right2(Res)
      end.                            end.

left2(Res) → receive {b,f} →        right2(Res) → receive {f,b} →
      Res ! {u,{b,b}}              Res ! {u,{b,b}}
      end.                            end.

```

**Fig. 7.** A simple Erlang program to perform deadlock analysis

Our transformation tool `erlang2trs` returned a TRS of more than 100 lines of code (available at <http://users.dsic.upv.es/~gvidal/erlang2trs/>). Consider now that we are interested in verifying whether a deadlock is possible. We can easily verify this property in Curry using the following test function:

```

init = reduce [State 0 (I 2) [], State 1 main []] []

test = wrongState init
where
  wrongState s = and (map ws s) == True
  ws s = case s of
    State _ (I _) _ -> True
    State _ (AREC _ _ _) _ -> True
    _ -> False

```

where the function `test` checks if there exists a reachable state where all processes are either finished with a process identifier or are waiting for a message simultaneously (a deadlock):

```

deadlock> test
[(State 0 (I 5) []),(State 1 (I 4) []),
 (State 4 (AREC 19 [] [(I 2)]) []),
 (State 3 (AREC 12 [(T2 B B)] [(I 2)]) []),
 (State 2 (AREC 6 [(T2 C (I 4))] [] [])]
Result: True
More solutions? [Y(es)/n(o)/a(ll)] n
deadlock>

```

Therefore, our analysis concludes that a deadlock may occur in the original Erlang program.

Of course, for more contrived examples with an infinite number of states, narrowing has an infinite search space. Fortunately, there already exist techniques for ensuring the termination of narrowing while still producing overapproximations of the original program in the context of partial evaluation (see, e.g., [2]). Therefore, we can adapt such an approach to perform symbolic execution of infinite-state systems. A first step towards this direction can be found in [18].

Actually, our tool `erlang2trs` already produces a TRS that includes a simple memoization to avoid reducing the same state once and again.

## 5 Related Work

Giesl and Arts [11] present a verification of Erlang processes by using dependency pairs. They propose a similar idea—transforming Erlang programs to (conditional) rewrite systems—but no transformation is formalized; rather, the process is done manually. Moreover, no verification of safety properties is considered. In fact, the authors mainly focus on proposing general improvements to the termination prover for TRSs and CTRSs. Another related approach—though for different source and target languages—is that of Albert *et al* [1], where a transformation from a concurrent object-oriented programming language based on message passing to a rule-based logic-like programming language is introduced.

Noll [19] introduces an implementation of Erlang in *rewriting logic* [16], a unified semantic framework for concurrency. Although we share some ideas with this paper, the aim is different. Noll’s aim was to provide an executable specification of the language semantics that is tailored to the *Specification Language Compiler Generator* [15] in order to automatically translate the description into a verification front-end that implements the transition rules. Therefore, in this approach, Erlang programs are seen as data objects manipulated by a sort of interpreter implemented in rewriting logic. In contrast, we aim at producing plain rewrite systems that can be analyzed using existing technologies. Other approaches are based on abstract interpretation (e.g., [13]) or the use of equations to define abstraction mappings (e.g., [20]). We can also find some approaches where Erlang is translated to  $\pi$ -calculus [22] or  $\mu$ CRL [4].

More specific tools for Erlang verification include EVT [21], a theorem prover that requires user intervention, and the model checker McErlang [10], which implements a big-step operational semantics for dealing with concurrency as a run-time Erlang system. The main strengths of McErlang are that it is a robust tool that covers most of the Erlang language, explores all possible schedulings for concurrent actions, includes debugging facilities, provides mechanisms for reducing the state space, etc. On the other hand, McErlang is not intended to analyze sequential programs and, moreover, it does not allow the use of symbolic input data (in contrast to other similar tools for other programming languages, e.g., Java Pathfinder [26], a model checker for Java). This extension is far from trivial, and a symbolic execution semantics should be carefully designed. Actually, we are only aware of the approach of [8] to symbolic execution in Erlang, though no formalization is introduced in the paper (it is only explained informally). Hence we think that our approach is a promising step towards defining a symbolic execution mechanism for Erlang.



## 6 Discussion

We have introduced a novel approach to Erlang verification based on translating the original program to a term rewriting system. By keeping the original program structure as much as possible, we can effectively analyze the rewrite system and infer useful information regarding the original Erlang program using standard techniques and tools for rewrite systems.<sup>4</sup> We have illustrated the practicality of the approach with some examples.

As a future work, we would like to deal with scalability issues, e.g., defining an appropriate partial order reduction. We would also like to extend our approach to deal with the remaining features of Erlang (mainly higher-order functions, guards, modules, etc). Finally, we will also consider the generation of Prolog programs instead. In this case, we would have more mature environments available as well as a flurry of analysis techniques that could be applied to the transformed programs.

### Acknowledgements

We thank the anonymous reviewers and the participants of LOPSTR 2013 for their useful comments to improve this paper.

### References

1. E. Albert, P. Arenas, and M. Gómez-Zamalloa. Symbolic Execution of Concurrent Objects in CLP. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL'12)*, pages 123–137. Springer LNCS 7149, 2012.
2. E. Albert and G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Generation Computing*, 20(1):3–26, 2002.
3. Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.
4. Thomas Arts, Clara Benac Earle, and John Derrick. Development of a verified Erlang program for resource locking. *STTT*, 5(2-3):205–220, 2004.
5. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
6. Rafael Caballero, Enrique Martin-Martin, Adrián Riesco, and Salvador Tamarit. A Declarative Debugger for Sequential Erlang Programs. In Margus Veanes and Luca Viganò, editors, *Proc. of the 7th International Conference on Tests and Proofs (TAP 2013)*, Lecture Notes in Computer Science, pages 96–114. Springer, 2013.
7. Koen Claessen and Hans Svensson. A semantics for distributed Erlang. In Konstantinos F. Sagonas and Joe Armstrong, editors, *Proc. of the 2005 ACM SIGPLAN Workshop on Erlang*, pages 78–87. ACM, 2005.
8. Clara Benac Earle. Symbolic program execution using the Erlang verification tool. In María Alpuente, editor, *Proc. of the 9th International Workshop on Functional and Logic Programming (WFLP 2000)*, pages 42–55, 2000.

---

<sup>4</sup> Nevertheless, although our syntax-directed transformation is tailored to the functional language Erlang, one could also extend it to other programming languages by using a *semantics-driven* transformation, similarly to that of [25].

9. Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. A syntactic theory of sequential control. *Theor. Comput. Sci.*, 52:205–237, 1987.
10. Lars-Ake Fredlund and Hans Svensson. McErlang: a model checker for a distributed functional programming language. In Ralf Hinze and Norman Ramsey, editors, *Proc. of ICFP 2007*, pages 125–136. ACM, 2007.
11. Jürgen Giesl and Thomas Arts. Verification of Erlang Processes by Dependency Pairs. *Appl. Algebra Eng. Commun. Comput.*, 12(1/2):39–72, 2001.
12. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at <http://www.curry-language.org>, 2012.
13. Frank Huch. Verification of Erlang Programs using Abstract Interpretation and Model Checking. In Didier Rémi and Peter Lee, editors, *Proc. of ICFP '99*, pages 261–272. ACM, 1999.
14. Jean-Marie Hullot. Canonical forms and unification. In Wolfgang Bibel and Robert A. Kowalski, editors, *Proceedings of the 5th International Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980.
15. Martin Leucker and Thomas Noll. Rewriting Logic as a Framework for Generic Verification Tools. *Electr. Notes Theor. Comput. Sci.*, 36:121–137, 2000.
16. José Meseguer. Conditioned Rewriting Logic as a United Model of Concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.
17. M.R. Neuhäüßer and T. Noll. Abstraction and Model Checking of Core Erlang Programs in Maude. *Electr. Notes Theor. Comput. Sci.*, 176(4):147–163, 2007.
18. N. Nishida and G. Vidal. A finite representation of the narrowing space. In *Proc. of the 23th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, pages 113–128. Technical Report TR-11-13, Universidad Complutense de Madrid, 2013. Available from <http://users.dsic.upv.es/~gvidal/>.
19. Thomas Noll. A Rewriting Logic Implementation of Erlang. *Electr. Notes Theor. Comput. Sci.*, 44(2):206–224, 2001.
20. Thomas Noll. Equational Abstractions for Model Checking Erlang Programs. *Electr. Notes Theor. Comput. Sci.*, 118:145–162, 2005.
21. Thomas Noll, Lars-Ake Fredlund, and Dilian Gurov. The Erlang Verification Tool. In Tiziana Margaria and Wang Yi, editors, *Proc. of TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 582–586. Springer, 2001.
22. Chanchal Kumar Roy, Thomas Noll, Banani Roy, and James R. Cordy. Towards automatic verification of Erlang programs by pi-calculus translation. In Marc Feeley and Philip W. Trinder, editors, *Proc. of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 38–50. ACM, 2006.
23. James R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM*, 21(4):622–642, 1974.
24. Hans Svensson and Lars-Ake Fredlund. A more accurate semantics for distributed Erlang. In Simon J. Thompson and Lars-Ake Fredlund, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang*, pages 43–54. ACM, 2007.
25. Germán Vidal. Closed symbolic execution for verifying program termination. In *Proc. of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2012)*. IEEE, 2012.
26. W. Visser, K. Havelund, G.P. Brat, S.Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

## A Correctness of the Transformation

In this section, we discuss the correctness of the transformation.

Basically, for each computation of the original Erlang program, we want to prove that we have a rewriting derivation in the transformed system. The other direction, though, is not true (i.e., the rewrite system represents an *overapproximation* of the original Erlang program).

Although Erlang has an eager semantics, we assume plain rewriting to evaluate expressions with the transformed rewrite system.

First, let us extend the transformation  $\llbracket \cdot \rrbracket$  to states:

$$\llbracket \langle p_1, s_1, q_1 \rangle \& \dots \& \langle p_n, s_n, q_n \rangle \rrbracket = \langle p_1, \llbracket s_1 \rrbracket, q_1 \rangle \& \dots \& \langle p_n, \llbracket s_n \rrbracket, q_n \rangle$$

Here, we ignore the sets of variables which are only used to avoid missing the value of some variables when breaking an expression into different functions. The correctness of this fact is easy.

Now, let us consider different cases depending on the expression selected to be reduced. For simplicity, we only consider the more general cases where the selected expression is not the last one of a sequence.

- Case expression. Consider the following state:

$$\Pi = \langle p, (\text{case } e \text{ of } p_1 \rightarrow s_1, \dots, p_n \rightarrow s_n \text{ end}, s), q \rangle \& \Pi''$$

Then, according to the semantics of Erlang, we have

$$\begin{aligned} \Pi &\rightarrow^* \langle p, (\text{case } v \text{ of } p_1 \rightarrow s_1, \dots, p_n \rightarrow s_n \text{ end}, s), q \rangle \& \Pi'' \\ &\rightarrow \langle p, (s_i, s)\sigma, q \rangle \& \Pi'' \end{aligned}$$

where  $p_i\sigma = v$  for some  $\sigma$  and there is no  $\sigma'$  such that  $p_j\sigma' = v$  for any  $j < i$ . By the definition of our transformation, we have that

$$\begin{aligned} \llbracket \Pi \rrbracket &= \langle p, \llbracket \text{case } e \text{ of } p_1 \rightarrow s_1, \dots, p_n \rightarrow s_n \text{ end}, s \rrbracket, q \rangle \& \llbracket \Pi'' \rrbracket \\ &= \langle p, f(e, \bar{V}), q \rangle \& \llbracket \Pi'' \rrbracket \end{aligned}$$

with  $f(p_1, \bar{V}) \rightarrow \llbracket s_1, s \rrbracket, \dots, f(p_n, \bar{V}) \rightarrow \llbracket s_n, s \rrbracket$ . Therefore, we have

$$\llbracket \Pi \rrbracket \rightarrow \langle p, f(e, \bar{V}), q \rangle \& \llbracket \Pi'' \rrbracket \rightarrow \langle p, \llbracket (s_i, s)\sigma \rrbracket, q \rangle \& \llbracket \Pi'' \rrbracket$$

and the claim will follow by induction.

- Message sending. Consider the following state:

$$\Pi = \langle p, (e_1 ! e_2, s), q \rangle \& \langle p', s', q' \rangle \& \Pi''$$

Then, according to the semantics of Erlang, we have

$$\begin{aligned} \Pi &\rightarrow^* \langle p, (v_1 ! v_2, s), q \rangle \& \langle p', s', q' \rangle \& \Pi'' \\ &\rightarrow \langle p, (v_2, s), q \rangle \& \langle p', s', q' + + [v_2] \rangle \& \Pi'' \\ &\rightarrow \langle p, s, q \rangle \& \langle p', s', q' + + [v_2] \rangle \& \Pi'' = \Pi' \end{aligned}$$

where  $v_1 = p'$ . By the definition of our transformation, we have that

$$\begin{aligned} \llbracket II \rrbracket &= \langle p, \llbracket e_1 ! e_2, s \rrbracket, q \rangle \& \langle p', \llbracket s' \rrbracket, q' \rangle \& \llbracket II'' \rrbracket \\ &= \langle p, \text{SEND}(i, e_1, e_2, \bar{V}), q \rangle \& \langle p', \llbracket s' \rrbracket, q' \rangle \& \llbracket II'' \rrbracket \end{aligned}$$

with  $\text{send}(i, -, \bar{V}) \rightarrow \llbracket s \rrbracket$ . Therefore, we have

$$\begin{aligned} \llbracket II \rrbracket &\rightarrow \langle p, \text{SEND}(i, e_1, e_2, \bar{V}), q \rangle \& \langle p', \llbracket s' \rrbracket, q' \rangle \& \llbracket II'' \rrbracket \\ &\rightarrow^* \langle p, \text{SEND}(i, v_1, v_2, \bar{V}), q \rangle \& \langle p', \llbracket s' \rrbracket, q' \rangle \& \llbracket II'' \rrbracket \\ &\rightarrow \langle p, \text{send}(i, v_2, \bar{V}), q \rangle \& \langle p', \llbracket s' \rrbracket, q' + +[v_2] \rangle \& \llbracket II'' \rrbracket \\ &\rightarrow \langle p, \llbracket s \rrbracket, q \rangle \& \langle p', \llbracket s' \rrbracket, q' + +[v_2] \rangle \& \llbracket II'' \rrbracket \end{aligned}$$

Note that there are two rules to deal with SEND and we have used the first one above. The second rule just moves the process to the end of the queue, which is necessary to consider all possible schedulings. Therefore, the claim will follow by induction.

– Message reception. Consider the following state:

$$II = \langle p, (\text{receive } p_1 \rightarrow s_1, \dots, p_n \rightarrow s_n \text{ end}, s), [v_1, \dots, v_k, \dots, v_m] \rangle \& II''$$

Then, according to the semantics of Erlang, we have

$$II \rightarrow \langle p, (s_i, s)\sigma, [v_1, \dots, v_{k-1}, v_{k+1}, \dots, v_m] \rangle \& II''$$

where  $v_k$  is the first message such that there exists  $j$  with  $p_j\sigma = v_k$  for some  $\sigma$  and there is no  $\sigma'$  such that  $p_l\sigma' = v_k$  for any  $l < j$ . By the definition of our transformation, we have that

$$\begin{aligned} \llbracket II \rrbracket &= \langle p, \llbracket \text{receive } p_1 \rightarrow s_1, \dots, p_n \rightarrow s_n \text{ end}, s \rrbracket, [v_1, \dots, v_k, \dots, v_m] \rangle \& \llbracket II'' \rrbracket \\ &= \langle p, \text{AREC}(i, [], \bar{V}), q \rangle \& \llbracket II'' \rrbracket \end{aligned}$$

with

$$\begin{array}{ll} \text{brec}(i, p_1) \rightarrow \text{True} & \text{rec}(i, p_1, \bar{V}) \rightarrow \llbracket s_1, s. \rrbracket \\ \dots & \dots \\ \text{brec}(i, p_n) \rightarrow \text{True} & \text{rec}(i, p_n, \bar{V}) \rightarrow \llbracket s_n, s. \rrbracket \end{array}$$

where  $i$  is a fresh constant. Therefore, according to the rules of Figure 4, we will have  $k - 1$  reductions with the second rule for AREC until the message  $v_k$  is reached and  $\text{brec}(i, v_k)$  is true:

$$\begin{aligned} &\langle p, \text{AREC}(i, [], \bar{V}), [v_1, \dots, v_k, \dots, v_m] \rangle \& \llbracket II'' \rrbracket \\ &\rightarrow^* \langle p, \text{AREC}(i, [v_1, \dots, v_{k-1}], \bar{V}), [v_k, \dots, v_m] \rangle \& \llbracket II'' \rrbracket \\ &\rightarrow \langle p, \text{rec}(i, v_k, \bar{V}), [v_1, \dots, v_{k-1}, v_{k+1}, \dots, v_m] \rangle \& \llbracket II'' \rrbracket \\ &\rightarrow \langle p, \llbracket (s_i, s)\sigma \rrbracket, [v_1, \dots, v_{k-1}, v_{k+1}, \dots, v_m] \rangle \& \llbracket II'' \rrbracket \end{aligned}$$

and the claim will follow by induction.

- Pattern matching. Consider the following state:

$$\Pi = \langle p, (pat = e, s), q \rangle \& \Pi''$$

Then, according to the semantics of Erlang, we have

$$\begin{aligned} \Pi &\rightarrow^* \langle p, (pat = v, s), q \rangle \& \Pi'' \\ &\rightarrow \langle p, (v, s)\sigma, q \rangle \& \Pi'' \\ &\rightarrow \langle p, s\sigma, q \rangle \& \Pi'' \end{aligned}$$

where  $pat\sigma = v$ . By the definition of our transformation, we have that

$$\llbracket \Pi \rrbracket = \langle p, \llbracket pat = e, s \rrbracket, q \rangle \& \llbracket \Pi'' \rrbracket = \langle p, f(e, \bar{V}), q \rangle \& \llbracket \Pi'' \rrbracket$$

where  $f(pat, \bar{V}) \rightarrow \llbracket s \rrbracket$ . Therefore, we have

$$\begin{aligned} \llbracket \Pi \rrbracket &\rightarrow \langle p, f(e, \bar{V}), q \rangle \& \llbracket \Pi'' \rrbracket \\ &\rightarrow^* \langle p, f(v, \bar{V}), q \rangle \& \llbracket \Pi'' \rrbracket \\ &\rightarrow \langle p, \llbracket s\sigma \rrbracket, q \rangle \& \llbracket \Pi'' \rrbracket \end{aligned}$$

and the claim will follow by induction.

- Process creation. Consider the following state:

$$\Pi = \langle p, (pat = \text{spawn}(f(e_1, \dots, e_n)), s), q \rangle \& \Pi''$$

Then, according to the semantics of Erlang, we have

$$\begin{aligned} \Pi &\rightarrow^* \langle p, (pat = \text{spawn}(f(v_1, \dots, v_n)), s), q \rangle \& \Pi'' \\ &\rightarrow \langle p, (pat = p', s), q \rangle \& \langle p', f(v_1, \dots, v_n), [] \rangle \& \Pi'' \\ &\rightarrow \langle p, (p', s)\sigma, q \rangle \& \langle p', f(v_1, \dots, v_n), [] \rangle \& \Pi'' \\ &\rightarrow \langle p, s\sigma, q \rangle \& \langle p', f(v_1, \dots, v_n), [] \rangle \& \Pi'' \end{aligned}$$

where  $pat\sigma = p'$ . By the definition of our transformation, we have that

$$\begin{aligned} \llbracket \Pi \rrbracket &= \langle p, \llbracket pat = \text{spawn}(f(e_1, \dots, e_n)), s \rrbracket, q \rangle \& \llbracket \Pi'' \rrbracket \\ &= \langle p, \text{SPAWN}(i, f(e_1, \dots, e_n), \bar{V}), q \rangle \& \llbracket \Pi'' \rrbracket \end{aligned}$$

where  $\text{spawn}(i, pat, \bar{V}) \rightarrow \llbracket s \rrbracket$ . Therefore, we have

$$\begin{aligned} \llbracket \Pi \rrbracket &\rightarrow \langle p, \text{SPAWN}(i, f(e_1, \dots, e_n), \bar{V}), q \rangle \& \llbracket \Pi'' \rrbracket \\ &\rightarrow^* \langle p, \text{SPAWN}(i, f(v_1, \dots, v_n), \bar{V}), q \rangle \& \llbracket \Pi'' \rrbracket \\ &\rightarrow \langle p, \text{spawn}(i, p', \bar{V}), q \rangle \& \langle p', f(v_1, \dots, v_n), [] \rangle \& \llbracket \Pi'' \rrbracket \\ &\rightarrow \langle p, \llbracket s\sigma \rrbracket, q \rangle \& \langle p', f(v_1, \dots, v_n), [] \rangle \& \llbracket \Pi'' \rrbracket \end{aligned}$$

and the claim will follow by induction.

- The primitive self. Consider the following state:

$$\Pi = \langle p, (pat = \text{self}, s), q \rangle \& \Pi''$$

Then, according to the semantics of Erlang, we have

$$\begin{aligned} II &\rightarrow \langle p, (pat = p, s), q \rangle \& II'' \\ &\rightarrow \langle p, (p, s)\sigma, q \rangle \& II'' \\ &\rightarrow \langle p, s\sigma, q \rangle \& II'' \end{aligned}$$

with  $pat\sigma = p$ . By the definition of our transformation, we have that

$$\llbracket II \rrbracket = \langle p, \llbracket pat = self, s \rrbracket, q \rangle \& \llbracket II'' \rrbracket = \langle p, SELF(i, \bar{V}), q \rangle \& \llbracket II'' \rrbracket$$

where  $self(i, pat, \bar{V}) \rightarrow \llbracket s \rrbracket$ . Therefore, we have

$$\begin{aligned} \llbracket II \rrbracket &\rightarrow \langle p, SELF(i, \bar{V}), q \rangle \& \llbracket II'' \rrbracket \\ &\rightarrow \langle p, self(i, \bar{V}), q \rangle \& \llbracket II'' \rrbracket \\ &\rightarrow \langle p, \llbracket s\sigma \rrbracket, q \rangle \& \llbracket II'' \rrbracket \end{aligned}$$

and the claim will follow by induction.

- For all other cases (i.e., when the selected expression belongs to  $\text{Exp}$ ), both programs are equivalent so the claim follows trivially.