Towards Erlang Verification by Term Rewriting*

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.

1 Introduction

The concurrent functional language Erlang [2] 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. For instance, [16] already introduced a translation from Erlang to *rewriting logic* [14], a unified semantic framework for concurrency. 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 they can be accurately analyzed using existing techniques.

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

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* [20], 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 [11], 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 [1]). Therefore, one could define a narrowing-based model checker by adapting this partial evaluation framework—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 [12] or Noll [16], 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 $X, Y, \ldots$), atoms (denoted by a, b, ...), process identifiers –pids– (denoted by p, p', ...), constructors (which are fixed in Erlang to lists, tuples and atoms), and defined functions (denoted by $f/n, g/m, \ldots$). 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 $f(X_1, \ldots, X_n) \to s.$ where $X_1, \ldots, X_n$ are distinct variables and the body of the function, s, can be an expression, a sequence of expressions, a case distinction, message sending (e.g., $\mathrm{main} \,!\, \{\mathrm{hello}, \mathrm{world}\}$ sends a message

$$
\begin{array}{rl}
pgm ::= & f(X_1,\ldots,X_n) \to s. \mid pgm\ pgm \\
\mathsf{ErlangExp} \ni s ::= & e \mid s_1, s_2 \mid \mathsf{case}\ e\ \mathsf{of}\ clauses\ \mathsf{end} \mid e_1 \,!\, e_2 \\
& \mid \mathsf{receive}\ clauses\ \mathsf{end} \mid pat = e \mid pat = \mathsf{self} \\
& \mid pat = \mathsf{spawn}(f(e_1,\ldots,e_n)) \\
\mathsf{Exp} \ni e ::= & f(e_1,\ldots,e_n) \mid [e_1|e_n] \mid [\,] \mid \{e_1,\ldots,e_n\} \mid \mathrm{a} \mid \mathrm{p} \mid X \\
clauses ::= & pat_1 \to s_1; \ldots; pat_n \to s_n \\
\mathsf{Pat} \ni pat ::= & [pat_1|pat_2] \mid [\,] \mid \{pat_1,\ldots,pat_n\} \mid \mathrm{a} \mid \mathrm{p} \mid X \\
\mathsf{Value} \ni v ::= & [v_1|v_2] \mid [\,] \mid \{v_1,\ldots,v_n\} \mid \mathrm{a} \mid \mathrm{p}
\end{array}
$$

Fig. 1. Erlang syntax rules

{hello, world} to the process with pid main) and receiving (e.g., $\mathsf{receive}\ \{A, B\} \to A\ \mathsf{end}$ 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., $\mathsf{spawn}(foo(1, 2))$ creates a new process[1] 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.

$$
\begin{array}{ll}
proc1 \to Pid1 = \mathsf{spawn}(proc2), & proc3 \to \mathsf{receive} \\
\quad Pid1 \,!\, \mathrm{a}. & \qquad X \to \mathrm{ok} \\
proc2 \to Pid2 = \mathsf{spawn}(proc3), & \quad \mathsf{end}. \\
\quad \mathsf{receive} & \\
\qquad X \to Pid2 \,!\, X & \\
\quad \mathsf{end}. &
\end{array}
$$

In the past, there have been several attempts to formalize the semantics of Erlang (e.g., [5, 6, 12, 15–17, 21]). In the following, we present an operational semantics for Erlang programs that mainly follows the approach of [12].

Erlang states are denoted by the parallel composition of their processes, where each process $\langle \mathrm{p}, e, q\rangle$ consists of a process identifier, an expression and a message queue: $\mathsf{Proc} ::= \mathsf{Pid} \times \mathsf{ErlangExp} \times \mathsf{Value}^*$. An *initial* state has the form $\langle \mathrm{p}, f(v_1,\ldots,v_n), []\rangle$ where f is a defined function, $v_1,\ldots,v_n$ are values, p is some

[1] Note that we consider $\mathsf{spawn}(foo(1, 2))$ rather than the original Erlang notation $\mathsf{spawn}(foo, [1, 2])$ which is sensible since we do not allow higher order functions in this paper.

$$\textsf{(seq)}\quad \frac{}{\langle \mathrm{p}, C[v, s], q\rangle \;\&\; \Pi \longrightarrow \langle \mathrm{p}, C[s], q\rangle \;\&\; \Pi}$$

$$\textsf{(self)}\quad \frac{}{\langle \mathrm{p}, C[\textsf{self}], q\rangle \;\&\; \Pi \longrightarrow \langle \mathrm{p}, C[\mathrm{p}], q\rangle \;\&\; \Pi}$$

$$\textsf{(fun)}\quad \frac{f(X_1, \ldots, X_n) \to s. \in prog}{\langle \mathrm{p}, C[f(v_1, \ldots, v_n)], q\rangle \;\&\; \Pi \longrightarrow \langle \mathrm{p}, C[\widehat{s}\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}], q\rangle \;\&\; \Pi}$$

$$\textsf{(match)}\quad \frac{\exists \sigma.\ pat\sigma = v}{\langle \mathrm{p}, C[pat = v], q\rangle \;\&\; \Pi \longrightarrow \langle \mathrm{p}, (C[v])\sigma, q\rangle \;\&\; \Pi}$$

$$\textsf{(case)}\quad \frac{\exists i.\ pat_i\sigma = v \text{ for some } \sigma \wedge \nexists \sigma'.\ pat_j\sigma' = v \text{ for any } j < i}{\langle \mathrm{p}, C[\textsf{case } v \textsf{ of } pat_1 \to s_1; \ldots; pat_n \to s_n \textsf{ end}], q\rangle \;\&\; \Pi \longrightarrow \langle \mathrm{p}, (C[s_i])\sigma, q\rangle \;\&\; \Pi}$$

$$\textsf{(spawn)}\quad \frac{\mathrm{p'} \text{ is a fresh pid}}{\langle \mathrm{p}, C[\textsf{spawn}(f(v_1, \ldots, v_n))], q\rangle \;\&\; \Pi \longrightarrow \langle \mathrm{p}, C[\mathrm{p'}], q\rangle, \langle \mathrm{p'}, f(v_1, \ldots, v_n), [\,]\rangle \;\&\; \Pi}$$

$$\textsf{(send)}\quad \frac{v_1 = \mathrm{p'} \in Pid}{\langle \mathrm{p}, C[v_1 \,!\, v_2], q\rangle \;\&\; \langle \mathrm{p'}, s, q'\rangle \;\&\; \Pi \longrightarrow \langle \mathrm{p}, C[v_2], q\rangle \;\&\; \langle \mathrm{p'}, e, q' {+\!+} [v_2]\rangle \;\&\; \Pi}$$

$$\textsf{(receive)}\quad \frac{\begin{array}{c} v_k \text{ is the first message such that} \\ (\exists i.\ pat_i\sigma = v \text{ for some } \sigma \wedge \nexists \sigma'.\ pat_j\sigma' = v \text{ for any } j < i) \end{array}}{\begin{array}{r} \langle \mathrm{p}, C[\textsf{receive } pat_1 \to s_1; \ldots; pat_n \to s_n \textsf{ end}], [v_1, \ldots, v_k, \ldots, v_m]\rangle \;\&\; \Pi \\ \longrightarrow \langle \mathrm{p}, (C[s_i])\sigma, [v_1, \ldots, v_{k-1}, v_{k+1}, \ldots, v_m]\rangle \;\&\; \Pi \end{array}}$$

Fig. 2. Basic Erlang Semantics

initial pid and [ ] denotes an empty message queue; we will use lists to denote 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 $\langle \mathrm{p}_1, v_1, q_1\rangle \;\&\; \cdots \;\&\; \langle \mathrm{p}_n, v_n, q_n\rangle$ where $v_1, \ldots, v_n$ 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 $\longrightarrow$: $\textsf{State} \times \textsf{State}$. 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:[2]

$$\begin{array}{rl} C ::= & [] \;\mid\; C, s \;\mid\; \textsf{case } C \textsf{ of } \mathit{clauses} \textsf{ end} \;\mid\; C \,!\, e \;\mid\; v \,!\, C \;\mid\; pat = C \\ \mid & \textsf{spawn}(f(v_1, \ldots, v_i, C, e_{i+2}, \ldots, e_n)) \;\mid\; f(v_1, \ldots, v_i, C, e_{i+2}, \ldots, e_n) \\ \mid & [v_1, \ldots, v_i, C|e] \;\mid\; \{v_1, \ldots, v_i, C, e_{i+2}, \ldots, e_n\} \end{array}$$

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 $\Gamma = \langle \mathrm{p}, e, q\rangle \;\&\; \Pi$ where Π denotes a (possibly empty) parallel composition of processes. The

[2] This is similar to the reduction contexts of [8] and allows us to deterministically identify the next expression to be reduced.

$\langle 1, \underline{proc1}, [\,] \rangle$
$\longrightarrow_{\mathsf{fun}}$ $\langle 1, Pid1 = \underline{\mathsf{spawn}(proc2)}, Pid1 \mathbin{!} \mathrm{a}, [\,] \rangle$
$\longrightarrow_{\mathsf{spawn}}$ $\langle 1, \underline{Pid1 = 2}, Pid1 \mathbin{!} \mathrm{a}, [\,] \rangle \;\&\; \langle 2, proc2, [\,] \rangle$
$\longrightarrow_{\mathsf{match}}$ $\langle 1, \underline{2, 2 \mathbin{!} \mathrm{a}}, [\,] \rangle \;\&\; \langle 2, proc2, [\,] \rangle$
$\longrightarrow_{\mathsf{seq}}$ $\langle 1, 2 \mathbin{!} \mathrm{a}, [\,] \rangle \;\&\; \langle 2, \underline{proc2}, [\,] \rangle$
$\longrightarrow_{\mathsf{fun}}$ $\langle 1, 2 \mathbin{!} \mathrm{a}, [\,] \rangle \;\&\; \langle 2, \underline{Pid2 = \mathsf{spawn}(proc3)}, \mathsf{receive}\ X \to Pid2 \mathbin{!} X\ \mathsf{end}, [\,] \rangle$
$\longrightarrow_{\mathsf{spawn}}$ $\langle 1, 2 \mathbin{!} \mathrm{a}, [\,] \rangle \;\&\; \langle 2, \underline{Pid2 = 3}, \mathsf{receive}\ X \to Pid2 \mathbin{!} X\ \mathsf{end}, [\,] \rangle$
$\;\&\; \langle 3, proc3, [\,] \rangle$
$\longrightarrow_{\mathsf{match}}$ $\langle 1, 2 \mathbin{!} \mathrm{a}, [\,] \rangle \;\&\; \langle 2, \underline{3}, \underline{\mathsf{receive}\ X \to 3 \mathbin{!} X\ \mathsf{end}}, [\,] \rangle$
$\;\&\; \langle 3, proc3, [\,] \rangle$
$\longrightarrow_{\mathsf{match}}$ $\langle 1, \underline{2 \mathbin{!} \mathrm{a}}, [\,] \rangle \;\&\; \langle 2, \mathsf{receive}\ X \to 3 \mathbin{!} X\ \mathsf{end}, [\,] \rangle \;\&\; \langle 3, proc3, [\,] \rangle$
$\longrightarrow_{\mathsf{send}}$ $\langle 1, \mathrm{a}, [\,] \rangle \;\&\; \langle 2, \underline{\mathsf{receive}\ X \to 3 \mathbin{!} X\ \mathsf{end}}, [\mathrm{a}] \rangle \;\&\; \langle 3, proc3, [\,] \rangle$
$\longrightarrow_{\mathsf{receive}}$ $\langle 1, \mathrm{a}, [\,] \rangle \;\&\; \langle 2, 3 \mathbin{!} \mathrm{a}, [\,] \rangle \;\&\; \langle 3, \underline{proc3}, [\,] \rangle$
$\longrightarrow_{\mathsf{fun}}$ $\langle 1, \mathrm{a}, [\,] \rangle \;\&\; \langle 2, \underline{3 \mathbin{!} \mathrm{a}}, [\,] \rangle \;\&\; \langle 3, \mathsf{receive}\ X \to \mathrm{ok}\ \mathsf{end}, [\,] \rangle$
$\longrightarrow_{\mathsf{send}}$ $\langle 1, \mathrm{a}, [\,] \rangle \;\&\; \langle 2, \mathrm{a}, [\,] \rangle \;\&\; \langle 3, \underline{\mathsf{receive}\ X \to \mathrm{ok}\ \mathsf{end}}, [\mathrm{a}] \rangle$
$\longrightarrow_{\mathsf{receive}}$ $\langle 1, \mathrm{a}, [\,] \rangle \;\&\; \langle 2, \mathrm{a}, [\,] \rangle \;\&\; \langle 3, \mathrm{ok}, [\,] \rangle$

Fig. 3. Computation for the program of Example 1

order of processes is not relevant here (i.e., $\langle \mathrm{p}, s, q \rangle$ might appear in any position within the pool of processes Γ).
- Rule self reduces the predefined atom self to the process identifier of the current process.
- Rule fun performs a function unfolding, where $\widehat{s}$ 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 σ 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., [4] 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 ϵ 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. $\mathcal{V}ar(t)$ 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 $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ 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 σ to a term t by $t\sigma$ rather than $\sigma(t)$. The identity substitution is denoted by *id*.

A set of rewrite rules $l \to r$ 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 $\to_\mathcal{R}$ as follows: given terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $s \to_\mathcal{R} t$ iff there exists a position p in s, a rewrite rule $l \to r \in \mathcal{R}$ and a substitution σ with $s|_p = l\sigma$ and $t = s[r\sigma]_p$; the rewrite step is often denoted by $s \to_{p,l\to r} t$ 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 $\to$, we denote by $\to^*$ its reflexive and transitive closure. Thus $t \to^*_\mathcal{R} s$ 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:

- 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.

– Several Erlang constructs cannot be translated with the same semantics to a rewrite system (unless a number of complex functions are instroduced). 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.
– 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 (e.g., as in [16]).
  For this purpose, we introduce some auxiliary functions that can be seen as *continuations* of the original functions (see below).

We formalize our transformation $[\![\ ]\!]$ as follows. Given an Erlang program P, we have:

$$[\![P]\!] = \{f(x_1, \ldots, x_n) \to [\![s.]\!]^V \mid f(x_1, \ldots, x_n) \to s. \in P\}$$

where $V = \{x_1, \ldots, x_n\} \cap \mathcal{FV}ar(s)$ is used for introducing auxiliary functions with appropriate parameters. In the following, $\mathcal{FV}ar(s)$ denotes the free variables of s. Now, we define the transformation function $[\![\ ]\!]$ 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:

$$[\![\mathsf{case}\ e\ \mathsf{of}\ p_1 \to s_1, \ldots, p_n \to s_n\ \mathsf{end.}]\!]^V = f(e, \overline{V})$$

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

$$\begin{array}{l} f(p_1, \overline{V}) \to [\![s_1.]\!]^{V_1} \\ \quad \ldots \\ f(p_n, \overline{V}) \to [\![s_n.]\!]^{V_n} \end{array}$$

where $V_i = \mathcal{V}ar(f(p_1, \overline{V})) \cap \mathcal{FV}ar(s_i)$, $i = 1, \ldots, n$. When the case expression is not the last statement in the right-hand side, we proceed analogously as follows:

$$[\![\mathsf{case}\ e\ \mathsf{of}\ p_1 \to s_1, \ldots, p_n \to s_n\ \mathsf{end}, s.]\!]^V = f(e, \overline{V})$$

where the auxiliary function f is now defined by

$$\begin{array}{l} f(p_1, \overline{V}) \to [\![s_1, s.]\!]^{V_1} \\ \quad \ldots \\ f(p_n, \overline{V}) \to [\![s_n, s.]\!]^{V_n} \end{array}$$

where $V_i = \mathcal{V}ar(f(p_1, \overline{V})) \cap \mathcal{FV}ar(s_i, s)$, $i = 1, \ldots, n$.

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 $\mathsf{SEND}(i, p, e, vars)$, where i is a unique identifier and $vars$ is a list of variables. We distinguish the following cases:

$$[\![e_1 \,!\, e_2.]\!]^V = \mathsf{SEND}(i, e_1, e_2, [\,]) \qquad \text{with } send(i, v, \_) \to v$$

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, SEND is a constructor symbol that will require the (global) system rules to be dealt with. Roughly speaking, the system rules will rewrite $\mathsf{SEND}(i, e_1, e_2, [\,])$ to $send(i, e_2, [\,])$—the continuation of 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

$$[\![e_1 \,!\, e_2, s.]\!]^V = \mathsf{SEND}(i, e_1, e_2, \overline{V}) \qquad \text{with } send(i, \_, \overline{V}) \to [\![s]\!]^{V'}$$

where i is a fresh constant symbol and $V' = V \cap \mathcal{FV}ar(s)$. 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 $\mathsf{AREC}(i, list, vars)$, 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:

$$[\![\mathsf{receive}\ p_1 \to s_1, \ldots, p_n \to s_n\ \mathsf{end}.]\!]^V = \mathsf{AREC}(i, [\,], \overline{V})$$

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

$$\begin{array}{ll} brec(i, p_1) \to \mathsf{True} & rec(i, p_1, \overline{V}) \to [\![s_1.]\!]^{V_1} \\ \ldots & \ldots \\ brec(i, p_n) \to \mathsf{True} & rec(i, p_n, \overline{V}) \to [\![s_n.]\!]^{V_n} \end{array}$$

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

$$[\![e, \mathsf{receive}\ p_1 \to s_1, \ldots, p_n \to s_n\ \mathsf{end}, s.]\!]^V = \mathsf{AREC}(i, [\,], \overline{V})$$

with

$$\begin{array}{ll} brec(i, p_1) \to \mathsf{True} & rec(i, p_1, \overline{V}) \to [\![s_1, s.]\!]^{V_1} \\ \ldots & \ldots \\ brec(i, p_n) \to \mathsf{True} & rec(i, p_n, \overline{V}) \to [\![s_n, s.]\!]^{V_n} \end{array}$$

where $V_j = \mathcal{V}ar(rec(i, p_j, \overline{V})) \cap \mathcal{FV}ar(s_j, s)$, $j = 1, \ldots, n$.

Loosely speaking, the system reduction rules will rewrite $\mathsf{AREC}(i, [\,], \overline{V})$ to $rec(i, m, \overline{V})$—the continuation of AREC—when $brec(i, m)$ is true, where m is the first message in the process mailbox; otherwise, the message m is moved to the second parameter of 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:

$$[\![p = e.]\!]^V = f(e, \overline{V}) \qquad \text{with } f(p, \overline{V}) \to p.$$

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

$$[\![p = e, s.]\!]^V = f(e, \overline{V}) \qquad \text{with } f(p, \overline{V}) \to [\![s.]\!]^{V'}$$

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

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

$$[\![p = \mathsf{spawn}(e).]\!]^V = \mathsf{SPAWN}(i, e, [\,]) \qquad \text{with } spawn(i, p, \_) \to p$$

where i is a fresh constant. Basically, the auxiliary function $spawn$—the continuation of 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:

$$[\![p = \mathsf{spawn}(e), s.]\!]^V = \mathsf{SPAWN}(i, e, \overline{V}) \qquad \text{with } spawn(i, p, \overline{V}) \to [\![s]\!]^{V'}$$

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

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

$$[\![p = \mathsf{self}.]\!]^V = \mathsf{SELF}(i, [\,]) \qquad \text{with } self(i, p, \_) \to p$$

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 $self$—the continuation

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

Fig. 4. State reduction rules

of SELF—with this pid as a second parameter. When the pattern matching is not the last element in a sequence, we proceed as follows:

$$[\![p = \mathsf{self}, s.]\!]^V = \mathsf{SELF}(i, \overline{V}) \qquad \text{with } self(i, p, \overline{V}) \rightarrow [\![s]\!]^{V'}$$

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

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:

$$[\![s_1, s_2.]\!]^V = [\![\mathsf{case}\ s_1\ \mathsf{of}\ \_ \rightarrow s_2\ \mathsf{end}.]\!]^V$$

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

Expressions. For the remaining expressions, we have $[\![e.]\!] = e$. Note that we assumed that no occurrence of the concurrency primitives: !, receive, self, etc., can occur in expressions.

3.3 State Reduction Rules

Processes are denoted by tuples $\langle \mathrm{p}, e, q\rangle$, 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 $\mathsf{SPAWN}(n, e, vars)$ is reduced by creating a new process initialized with the expression e, and replacing the 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.

Example 3. Let us consider the Erlang program of Example 1. This program is transformed into the following TRS (functions and variables start with a lowercase letter, while constructors start with an uppercase letter):

$$
\begin{array}{ll}
proc1 & \to \mathsf{SPAWN}(1, proc2, [\,]) \\
spawn(1, pid1, [\,]) & \to \mathsf{SEND}(2, pid1, \mathsf{A}, [\,]) \\
send(2, e, \_) & \to e \\
\\
proc2 & \to \mathsf{SPAWN}(3, proc3, [\,]) \\
spawn(3, pid2, [\,]) & \to \mathsf{AREC}(4, [\,], [pid2]) \\
brec(4, x) & \to \mathsf{True} \\
rec(4, x, [pid2]) & \to \mathsf{SEND}(5, pid2, x, [\,]) \\
send(5, e, \_) & \to e \\
\\
proc3 & \to \mathsf{AREC}(6, [\,], [\,]) \\
brec(6, x) & \to \mathsf{True} \\
rec(6, x, [\,]) & \to \mathsf{Ok}
\end{array}
$$

$[\langle 0, 2, [\,]\rangle, \langle 1, \underline{proc1}, [\,]\rangle]$
$\rightarrow [\langle 0, 2, [\,]\rangle, \underline{\langle 1, \mathsf{SPAWN}(1, proc2, [\,]), [\,]\rangle}]$
$\rightarrow [\langle 0, 3, [\,]\rangle, \underline{\langle 1, spawn(1, 2, [\,]), [\,]\rangle}, \langle 2, proc2, [\,]\rangle]$
$\rightarrow [\langle 0, 3, [\,]\rangle, \underline{\langle 1, \mathsf{SEND}(2, 2, \mathsf{A}, [\,]), [\,]\rangle}, \langle 2, proc2, [\,]\rangle]$
$\rightarrow [\langle 0, 3, [\,]\rangle, \underline{\langle 2, proc2, [\mathsf{A}]\rangle}, \langle 1, send(2, \mathsf{A}, [\,]), [\,]\rangle]$
$\rightarrow [\langle 0, 3, [\,]\rangle, \underline{\langle 2, \mathsf{SPAWN}(3, proc3, [\,]), [\mathsf{A}]\rangle}, \langle 1, send(2, \mathsf{A}, [\,]), [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 1, send(2, \mathsf{A}, [\,]), [\,]\rangle}, \langle 2, spawn(3, 3, [\,]), [\mathsf{A}]\rangle, \langle 3, proc3, [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 1, \mathsf{A}, [\,]\rangle}, \langle 2, spawn(3, 3, [\,]), [\mathsf{A}]\rangle, \langle 3, proc3, [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 2, spawn(3, 3, [\,]), [\mathsf{A}]\rangle}, \langle 3, proc3, [\,]\rangle, \langle 1, \mathsf{A}, [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 2, \mathsf{AREC}(4, [\,], [3]), [\mathsf{A}]\rangle}, \langle 3, proc3, [\,]\rangle, \langle 1, \mathsf{A}, [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 3, proc3, [\,]\rangle}, \langle 1, \mathsf{A}, [\,]\rangle, \langle 2, rec(4, \mathsf{A}, [3]), [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 3, \mathsf{AREC}(6, [\,], [\,]), [\,]\rangle}, \langle 1, \mathsf{A}, [\,]\rangle, \langle 2, rec(4, \mathsf{A}, [3]), [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 1, \mathsf{A}, [\,]\rangle}, \langle 2, rec(4, \mathsf{A}, [3]), [\,]\rangle, \langle 3, \mathsf{AREC}(6, [\,], [\,]), [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 2, rec(4, \mathsf{A}, [3]), [\,]\rangle}, \langle 3, \mathsf{AREC}(6, [\,], [\,]), [\,]\rangle, \langle 1, \mathsf{A}, [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 2, \mathsf{SEND}(5, 3, \mathsf{A}, [\,]), [\,]\rangle}, \langle 3, \mathsf{AREC}(6, [\,], [\,]), [\,]\rangle, \langle 1, \mathsf{A}, [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 3, \mathsf{AREC}(6, [\,], [\,]), [\mathsf{A}]\rangle}, \langle 1, \mathsf{A}, [\,]\rangle, \langle 2, send(5, \mathsf{A}, [\,]), [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 1, \mathsf{A}, [\,]\rangle}, \langle 2, send(5, \mathsf{A}, [\,]), [\,]\rangle, \langle 3, rec(6, \mathsf{A}, [\,]), [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 2, send(5, \mathsf{A}, [\,]), [\,]\rangle}, \langle 3, rec(6, \mathsf{A}, [\,]), [\,]\rangle, \langle 1, \mathsf{A}, [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 2, \mathsf{A}, [\,]\rangle}, \langle 3, rec(6, \mathsf{A}, [\,]), [\,]\rangle, \langle 1, \mathsf{A}, [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \underline{\langle 3, rec(6, \mathsf{A}, [\,]), [\,]\rangle}, \langle 1, \mathsf{A}, [\,]\rangle, \langle 2, \mathsf{A}, [\,]\rangle]$
$\rightarrow [\langle 0, 4, [\,]\rangle, \langle 3, \mathsf{Ok}, [\,]\rangle, \langle 1, \mathsf{A}, [\,]\rangle, \langle 2, \mathsf{A}, [\,]\rangle]$

Fig. 5. Example of reduction

The computation shown in Example 2 for the Erlang program proceeds now as shown in Fig. 5.[3] Here, we reach exactly the same final state of Fig. 5. Note, however, that due to non-determinism, other computations are also possible.

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; it is left as future work.

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*, a conservative extension of term rewriting for dealing with non-determinism and logic variables. Narrowing can be seen as a symbolic execution

[3] We underline either the expression or the selected process involved in a reduction step.

version of rewriting where pattern matching is replaced with unification (as in logic programming). Narrowing has been used as the basis of a partial evaluation framework for rewrite systems [1].

In particular, in order to produce executable programs, we consider the language Curry [11] (a conservative extension of Haskell to deal with logic variables and non-determinism).

Example 4. Consider the following Erlang program:

$$
\begin{array}{lll}
main \rightarrow Pid2 = \mathsf{spawn}(proc2), & & proc1(Pid) \rightarrow \mathsf{receive} \\
\quad Pid1 = \mathsf{spawn}(proc1(Pid2)), & & \qquad X \rightarrow Pid\ !\ X \\
\quad Pid1\ !\ \text{hello}, & & \quad \mathsf{end}. \\
\quad Pid2\ !\ \text{world}. & proc2 \rightarrow & \mathsf{receive} \\
 & & \qquad X \rightarrow \text{ok} \\
 & & \quad \mathsf{end}.
\end{array}
$$

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 *proc*3 before the message "Hello". 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
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.

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., [1]).

Therefore, we can adapt such an approach to perform symbolic execution of infinite-state systems.

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 [10] 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.

Noll [16] introduces an implementation of Erlang in *rewriting logic* [14], 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* [13] 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., [12]) or the use of equations to define abstraction mappings (e.g., [17]). We can also find some approaches where Erlang is translated to π-calculus [19] or μCRL [3].

More specific tools for Erlang verification include EVT [18], a theorem prover that requires user intervention, and the model checker McErlang [9], which implements a big-step operational semantics for dealing with concurrency as a run-time Erlang system. In these approaches, no symbolic execution mechanism for Erlang is introduced. Actually, we are only aware of the approach presented in [7], though no formalization is introduced. 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.[4] We have illustrated the practicality of the approach by verifying a simple safety property.

[4] 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 [22].

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 explore 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 would like to thank the anonymous reviewers for their useful comments to improve this paper.

References

1. E. Albert and G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Generation Computing*, 20(1):3–26, 2002.
2. Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.
3. Thomas Arts, Clara Benac Earle, and John Derrick. Development of a verified Erlang program for resource locking. *STTT*, 5(2-3):205–220, 2004.
4. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. Jürgen Giesl and Thomas Arts. Verification of Erlang Processes by Dependency Pairs. *Appl. Algebra Eng. Commun. Comput.*, 12(1/2):39–72, 2001.
11. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at http://www.curry-language.org, 2012.
12. Frank Huch. Verification of erlang programs using abstract interpretation and model mhecking. In Didier Rémi and Peter Lee, editors, *Proc. of ICFP '99*, pages 261–272. ACM, 1999.
13. Martin Leucker and Thomas Noll. Rewriting Logic as a Framework for Generic Verification Tools. *Electr. Notes Theor. Comput. Sci.*, 36:121–137, 2000.

14. José Meseguer. Conditioned Rewriting Logic as a United Model of Concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.
15. M.R. Neuhäußer and T. Noll. Abstraction and Model Checking of Core Erlang Programs in Maude. *Electr. Notes Theor. Comput. Sci.*, 176(4):147–163, 2007.
16. Thomas Noll. A Rewriting Logic Implementation of Erlang. *Electr. Notes Theor. Comput. Sci.*, 44(2):206–224, 2001.
17. Thomas Noll. Equational Abstractions for Model Checking Erlang Programs. *Electr. Notes Theor. Comput. Sci.*, 118:145–162, 2005.
18. 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.
19. 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.
20. James R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM*, 21(4):622–642, 1974.
21. 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.
22. 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.