Towards Modelling Actor-Based Concurrency in Term Rewriting*

Adrián Palacios and Germán Vidal

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

Abstract

In this work, we introduce a scheme for modelling actor systems within sequential term rewriting. In our proposal, a TRS consists of the union of three components: the functional part (which is specific of a system), a set of rules for reducing concurrent actions, and a set of rules for defining a particular scheduling policy. A key ingredient of our approach is that concurrent systems are modelled by terms in which concurrent actions can never occur inside user-defined function calls. This assumption greatly simplifies the definition of the semantics for concurrent actions, since no term traversal will be needed. We prove that these systems are well defined in the sense that concurrent actions can always be reduced.

Our approach can be used as a basis for modelling actor-based concurrent programs, which can then be analyzed using existing techniques for term rewrite systems.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases concurrency, actor model, rewriting

Digital Object Identifier 10.4230/OASIcs.WPTE.2015.19

1 Introduction

In this work, we consider the so called *actor model* [1] of concurrency, the model underlying programming languages like Erlang [3] or Scala [7]. In this model, a program consists of a pool of processes –the actors, each one identified by a unique process identifier– that interact by exchanging messages in an asynchronous manner. Each process has a (local) mailbox where incoming messages are stored, which is not shared with the other processes. When a process receives a message, it can update its local state, send messages, or create new actors. Here, we aim at modelling an Erlang-like language within sequential term rewriting. The basic objects of the language are variables (denoted by identifiers starting with a capital letter, e.g., $X, Y, \ldots$), 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 $f/n, g/m, \ldots$). In Erlang, 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., main ! {hello, world} sends a message {hello, world} to the

* This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEOII2015/013. Adrián Palacios was partially supported by the the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores de la Secretaría de Estado de Investigación, Desarrollo e Innovación del Ministerio de Economía y Competitividad* under FPI grant BES-2014-069749.

2nd International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'15).
Editors: Yuki Chiba, Santiago Escobar, Naoki Nishida, David Sabel, and Manfred Schmidt-Schauß; pp. 19–29

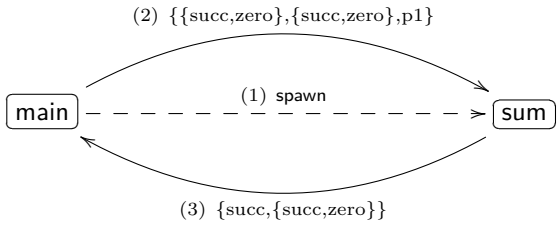
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

process with pid main) and receiving (e.g., receive $\{A, B\} \to A$ end reads a message from the process mailbox 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 that executes $foo(1, 2)$).

Consider, for instance, the following Erlang program:

$$
\begin{array}{rcl}
\mathsf{main}(X, Y) & \to & P = \mathsf{spawn}(\mathsf{sum}, [\,]), \\
& & P \mathbin{!} \{X, Y, \mathsf{self}()\}, \\
& & \mathsf{receive} \\
& & \quad Z \to Z \\
& & \mathsf{end}. \\
\\
\mathsf{sum}() & \to & \mathsf{receive} \\
& & \quad \{N, M, P\} \to P \mathbin{!} \mathsf{add}(N, M) \\
& & \mathsf{end}, \\
& & \mathsf{sum}(). \\
\\
\mathsf{add}(N, M) & \to & \mathsf{case}\ N\ \mathsf{of} \\
& & \quad \mathrm{zero} \to M \\
& & \quad \{\mathrm{succ}, X\} \to \{\mathrm{succ}, \mathsf{add}(X, M)\} \\
& & \mathsf{end}.
\end{array}
$$

Here, natural numbers are represented by zero, $\{\mathrm{succ}, \mathrm{zero}\}$, $\{\mathrm{succ}, \{\mathrm{succ}, \mathrm{zero}\}\}$, etc. When we execute this program, e.g., with main($\{\mathrm{succ}, \mathrm{zero}\}, \{\mathrm{succ}, \mathrm{zero}\}$), the function main first spawns a new process, then sends both numbers and the own process identifier –obtained with the predefined function self()– to the new process, say $\{\{\mathrm{succ}, \mathrm{zero}\}, \{\mathrm{succ}, \mathrm{zero}\}, \mathrm{p1}\}$, and receive suspends until some message arrives. The new process executes function sum which is initially suspended until a message arrives. When the message $\{\{\mathrm{succ}, \mathrm{zero}\}, \{\mathrm{succ}, \mathrm{zero}\}, \mathrm{p1}\}$ arrives, it calls to function add and returns the value $\{\mathrm{succ}, \{\mathrm{succ}, \mathrm{zero}\}\}$ to the process that requested the sum. Graphically,



We do not formally describe the semantics of Erlang here, but refer the reader to, e.g., [12]. Nevertheless, let us recall some relevant points about the semantics of Erlang:

- Erlang considers eager evaluation for function calls.
- The clauses of a case expression are tried in the textual order. Once the argument of a case expression matches a pattern, the remaning branches are discarded; note that this is the only sensible semantics when we have overlapping clauses (e.g., for defining default cases).
- Receive expressions proceed analogously to a case expression, but considers the *first* message in the process mailbox that matches some branch. If no one matches (or the mailbox is empty), the process' computation suspends until a new message arrives.

- As for message passing, Erlang's rule is that the direct messages between two processes should arrive in the same order they are sent. Nothing is said, though, when the messages follow different paths (i.e., when they traverse some other intermediate processes that resend the message). This is actually the source of many unexpected errors, and the reason to introduce a semantics like that of [11, 14].

In the following, we present an approach to model actor systems in the context of (sequential) rewriting, so that they can then be analyzed using existing techniques for term rewrite systems.

2 Term Rewriting

We assume familiarity with basic concepts of term rewriting. We refer the reader to, e.g., [4] for further 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 $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We assume that $\mathcal{F}$ always contains at least one constant $\mathsf{f}/0$. We use $\mathsf{f}, \mathsf{g}, \ldots$ to denote functions and $x, y, \ldots$ to denote variables. A *position* p in a term t is represented by a finite sequence of natural numbers, where ϵ denotes the root position. The set of positions of a term t is denoted by $\mathcal{P}os(t)$. 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 term t is *ground* if $\mathcal{V}ar(t) = \emptyset$.

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 non-variable 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 (lhs) and the right-hand side (rhs) 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 lhs's 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 $\mathcal{V}$, i.e., $\mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$. We omit $\mathcal{R}$ from $\mathcal{D}_\mathcal{R}$ and $\mathcal{C}_\mathcal{R}$ if it is clear from the context. A substitution σ is a *constructor substitution* (of $\mathcal{R}$) if $x\sigma \in \mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$ for all variables x. A TRS $\mathcal{R}$ is a *constructor system* if the lhs's of its rules have the form $\mathsf{f}(s_1, \ldots, s_n)$ where s_i are constructor terms, i.e., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, for all $i = 1, \ldots, n$.

For a TRS $\mathcal{R}$, we define the associated rewrite relation $\to_\mathcal{R}$ as the smallest binary relation satisfying the following: given terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $s \to_\mathcal{R} t$ iff there exist 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 usually denoted by $s \to_{p,l\to r} t$ to make explicit the position and rule used in this step. A term t is called *irreducible* or in *normal form* w.r.t. a TRS $\mathcal{R}$ if there is no term s with $t \to_\mathcal{R} s$. 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. We say that $s \xrightarrow{i}_{p,l\to r} t$ is an *innermost* rewrite step if $s \to_{p,l\to r} t$ and all proper subterms of $s|_p$ are irreducible.

3 Modelling Concurrency

In this section, we introduce our approach to model actor systems within the context of (sequential) rewriting. First, note that we do not intend to model the *semantics* of a

concurrent language (as it is done, e.g., in [9], where basically an interpreter of Erlang is specified in Maude [5]), but to present a proposal for specifying actor systems in term rewriting. In particular, we aim at keeping the *functional* part of the model as independent as possible from the concurrent actions. Let us introduce first the structure of the specification a *system*:

▶ **Definition 1** (system specification structure)**.** An actor system is specified as a constructor TRS $\mathcal{R} = \mathcal{E} \cup \mathcal{A} \cup \mathcal{S}$, where $\mathcal{E}$ is the functional component, $\mathcal{A}$ specifies the evaluation of concurrent actions, and $\mathcal{S}$ defines a scheduling policy. Usually, only the functional component $\mathcal{E}$ changes from one system to another.

A given system will be modelled as a term, that is then reduced using the rules of the system specification. Let us first introduce the term representation for processes:

▶ **Definition 2** (process)**.** A process is denoted by a term $\mathsf{p}(p, t, m)$, where $\mathsf{p}/3$ is a constructor symbol, p is the process identifier (a constructor constant), t is the process' term, and m is the mailbox (a list of constructor terms).

Here, we use natural numbers (built from $\mathsf{0}$ and succ) as pids. Other data structures are possible, but we choose natural numbers for simplicity. Also, the systems have a *global mailbox* which is used to model actor systems following the semantics of Erlang. Whenever a process p sends a message t to another process p', we store a term $\mathsf{m}(p, p', t)$ in the global mailbox. It is then the scheduler who determines when and in which order these messages should be dispatched to the local mailboxes of the processes. Using this strategy, one can easily ensure that all possible interleavings are explored, which is essential for, e.g., model checking techniques. A similar strategy is presented in [11, 14].

A system is basically a *pool* of processes. In our approach, we model a system by means of a *defined* function –so systems are reducible– as follows:

▶ **Definition 3** (system)**.** A system is denoted by a term $\mathsf{s}(k, m, procs)$, where $\mathsf{s}/3$ is a defined symbol, k is a natural number (used to produce fresh pids), m is a global mailbox of the system, and *procs* is a pool of processes.

In principle, there are several ways in which we can specify a *pool* of processes. For instance, we can introduce a constructor symbol $\odot/2$ and assume that it is associative and commutative (AC). Using an AC symbol greatly simplifies the presentation of the rules for concurrent actions. Note that, in this case, the system rules below are modelling all possible interleavings. In practice, though, we will be interested only in some of them. For instance, one can use a standard list to store the processes, and implement a simple Round Robin strategy that takes the process in the head of the list, performs a reduction step, then moves it to the end of the list, takes the next one, and so forth (this is the approach followed, e.g., in [13]). Here, for simplicity, we will consider the first approach.

For instance, an initial system could be represented by the following term:

$$\mathsf{s}(\mathsf{succ}(\mathsf{0}), [\,], \mathsf{p}(0, \mathsf{main}(t_1, t_2), [\,]))$$

where main is a call to a user-defined function with two arbitrary terms t_1 and t_2 as arguments. After a sequence of reduction steps, we may get a system like

$$\mathsf{s}(\mathsf{succ}(\mathsf{succ}(\mathsf{0})), [\,], \mathsf{p}(\mathsf{0}, \mathsf{self}(\ldots), [\,]) \odot \mathsf{p}(\mathsf{succ}(\mathsf{0}), \mathsf{sum}, [\,]))$$

with two processes with pids $\mathsf{0}$ and $\mathsf{succ}(\mathsf{0})$, such that the first one is *suspended* waiting for the concurrent action self to be processed by the system rules. The complete example will be shown in Section 3.3.

In the following, we consider each component of a system specification separately.

3.1 The Functional Component

The functional component of an actor system is defined by means of a constructor TRS $\mathcal{E}$. If the system performs no concurrent actions, then $\mathcal{E}$ is a standard TRS (and, moreover, there is no need for the other two components). In general, though, we consider four basic concurrent actions:

- self: this is the simplest action, and just returns the pid of the process.
- spawn: this action is used to create new processes. The argument of spawn is typically a function call that will start the execution of the new process. The pid of the new process is returned by the call to spawn.
- send: this action sends a message to a process identified by a pid. The call to send returns the value of the message.
- receive: this action consists in finding a message in the mailbox that matches some of the given patterns. If there is no such message (or the mailbox is empty), execution suspends.

In order to model these concurrent actions, we face a difficult problem. In a language like Erlang, expressions include *sequences* of actions and/or function calls. Moreover, as mentioned before, we do not want to specify an interpreter that traverses terms and executes both user-defined functions and concurrent actions.

Therefore, we took the following decision to avoid defining a complete interpreter for concurrent systems (i.e., in order to avoid the approach followed in [9], as mentioned before):

> *during the reduction of a system, concurrent actions must always occur in the topmost position of a process' term.*

Thanks to this requirement, one can model concurrent actions using a few simple rules (see Section 3.2 below). In order for this requirement to hold, we model concurrent actions using a sort of *continuations*. Basically, for each concurrent action, we have an associated auxiliary function that is used to bind some variables and to set the continuation after the concurrent action. This scheme will ensure that concurrent actions never occur below a user-defined constructor or defined function.

The constructor symbols used to represent concurrent actions are the following: $\mathsf{self}/2$, $\mathsf{spawn}/3$, $\mathsf{send}/3$, and $\mathsf{rec}/2$. For $\mathsf{self}/2$, $\mathsf{spawn}/3$ and and $\mathsf{rec}/2$, we have auxiliary functions called $\mathsf{fself}/3$, $\mathsf{fspawn}/3$ and $\mathsf{frec}/3$ (where the prefix f stands for defined "function", in contrast to constructor), which are indexed by a constructor constant. For $\mathsf{send}/3$, an auxiliary function is not needed since no additional bindings are necessary, and the continuation is set in the third argument of the constructor function. Concurrent actions are reduced to a term, but they also produce some side effect (e.g., creating a new process or sending a message). For instance, in order to model a concurrent action, say $p = \mathsf{self}()$ followed by the evaluation of a term t, we should have a term of the form $\mathsf{self}(\mathsf{f1}, vs)$, together with a rule

$$\mathsf{fself}(\mathsf{f1}, vs, p) \to t$$

The rules for the evaluation of concurrent actions will be responsible of reducing $\mathsf{self}(\mathsf{f1}, vs)$ to $\mathsf{fself}(\mathsf{f1}, vs, k)$, where k is the pid of the process where $\mathsf{self}(\mathsf{f1}, vs)$ occurs. The situation is similar for spawn and rec. The case of send is slightly different since no variables need to be bound and, thus, the continuation term is just an argument of send. E.g., sending a term t to a process with pid p, and then continue with the evaluation of t' is written as $\mathsf{send}(p, t, t')$. The rules for concurrent actions will then rewrite $\mathsf{send}(p, t, t')$ to t' also storing the message in the global mailbox.

$$
\begin{array}{rcl}
\mathsf{main}(x,y) & \to & \mathsf{spawn}(\mathsf{main1},[x,y],\mathsf{sum}) \\
\mathsf{fspawn}(\mathsf{main1},[x,y],p) & \to & \mathsf{self}(\mathsf{main1},[x,y,p]) \\
\mathsf{fself}(\mathsf{main1},[x,y,p],s) & \to & \mathsf{send}(p,\mathsf{d}(x,y,s), \\
 & & \quad \mathsf{rec}(\mathsf{main1},[x,y]) \\
 & &) \\
\mathsf{frec}(\mathsf{main1},[x,y],z) & \to & z \\
 & & \\
\mathsf{sum} & \to & \mathsf{rec}(\mathsf{sum1},[\,]) \\
\mathsf{frec}(\mathsf{sum1},[\,],\mathsf{d}(n,m,p)) & \to & \mathsf{send}(p,\mathsf{add}(n,m), \\
 & & \quad \mathsf{sum} \\
 & &) \\
 & & \\
\mathsf{frec}(h,vs,t) & \to & \mathsf{no\_match}(h,vs) \\
 & & \\
\mathsf{add}(0,m) & \to & m \\
\mathsf{add}(\mathsf{succ}(n),m) & \to & \mathsf{succ}(\mathsf{add}(n,m))
\end{array}
$$

Figure 1 Functional component $\mathcal{E}$ of an actor system.

Let us briefly explain the constructor symbols introduced to model concurrent actions:[1]

- $\mathsf{self}(h, vars)$, where h is a constructor constant that is used to identify the associated continuation function. This expression will be rewritten –by the rules of $\mathcal{A}$– to $\mathsf{fself}(h, vars, k)$, where k is the pid of the current process. Here, $vars$ is a list of variables that must be passed to the auxiliary function.
- $\mathsf{spawn}(h, vars, t)$, where h is a constructor constant that is used to identify the associated continuation function, $vars$ are the variables that must be passed to this auxiliary function, and t is the term that must be evaluated by the new process. As we will see, this term will be replaced by $\mathsf{fspawn}(h, vars, k)$, where k is a fresh pid; also, a new process will be added to the pool of processes, initialized with t and with pid k.
- $\mathsf{send}(p, t_1, t_2)$, where p is a pid and t_1, t_2 are terms. This action will be replaced with t_2 –the continuation– and will add t_1 to the mailbox of the process with pid p.
- $\mathsf{rec}(h, vars)$, where h is a constructor constant that is used to identify the associated continuation function and $vars$ are the variables that must be passed to this auxiliary function. This is the most complex action. It is replaced with $\mathsf{frec}(h, vars, m)$ where m is the first message in the process' mailbox. If there is no matching rule, the next message will be tried, and so forth.

Note that we do not introduce a special symbol for case expressions. Here, we assume that case expressions can be modelled by means of ordinary rewrite rules (as in, e.g., [13]).

For instance, the functional component $\mathcal{E}$ of the Erlang program shown in Section 1 can be specified in our context with the rewrite rules of Figure 1. Here, the rules of function add are essentially the same as in the original Erlang program since they contained no concurrent actions. Functions main and sum now use a number of auxiliary functions to deal with concurrent actions so that we get four rules for specifying the original function main and three more rules for specifying the original function sum. Moreover, some Erlang data structures are now represented using constructor functions. For instance, the three element

[1] Note, however, that the rules that deal with concurrent actions will be introduced in Section 3.2.

$$\mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{self}(h, vs), ms') \odot ps) \to \mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{fself}(h, vs, p), ms') \odot ps)$$

$$\mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{spawn}(h, vs, t), ms') \odot ps) \to \mathsf{s}(\mathsf{succ}(k), ms, \mathsf{p}(p, \mathsf{fspawn}(h, vs, k), ms') \odot \mathsf{p}(k, t, [\,]) \odot ps)$$

$$\mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{send}(p', t, t'), ms') \odot ps) \to \mathsf{s}(k, ms{+\!+}[\mathsf{m}(p, p', t)], \mathsf{p}(p, t', ms') \odot ps)$$

$$\begin{aligned}
\mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{rec}(h, vs), m : ms') \odot ps) &\to \mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{frec}(h, vs, m), ms') \odot ps) \\
\mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{no_match}(h, vs), m : ms') \odot ps) &\to \mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{frec}(h, vs, m), ms') \odot ps) \\
\mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{no_match}(h, vs), [\,]) \odot ps) &\to \mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{rec}(h, vs), [\,]) \odot ps)
\end{aligned}$$

Figure 2 Concurrent component $\mathcal{A}$: the system rules.

tuples of the original program are now represented using the constructor function $\mathsf{d}(\_, \_, \_)$. Specifying a system in this style might seem difficult, but one can generate it automatically from some higher-level language (as it is done in [13] for a similar formalism). The purpose of this paper, though, is to describe and analyze the properties of the modelling language, independently of the way the rules are produced.

The language could be more expressive by introducing additional constructs (e.g., a let expression to avoid duplicating the same term). Nevertheless, here we prefer to keep the previous minimal set of actions for simplicity.

3.2 Concurrent Actions

In this section, we present the rules of the second component $\mathcal{A}$ of an actor system. As mentioned before, we assume that concurrent actions always occur in the topmost position of a process' term, which greatly simplify the rules that deal with concurrent actions.

Let us now introduce the second component, $\mathcal{A}$, with the rules that define the evaluation of concurrent actions. Here, we assume that $\odot$ is an AC operator, so that $\mathsf{p}(p, t, m) \odot ps$ denotes a (non-empty) pool of processes, where $\mathsf{p}(p, t, m)$ is an arbitrary process.

▶ **Definition 4** (concurrent actions rules)**.** The component $\mathcal{A}$ for concurrent actions is given by the rewrite rules of Figure 2.[2]

Let us briefly explain the system rules. The first rule deals with the concurrent action self (a constructor), which is then replaced by a function call to fself also adding the pid p of the current process. The second rule first performs a *side effect* by creating a new process with pid k, and then replaces the constructor spawn with a call to the function fspawn where the new pid is also passed as argument. The third rule adds a message to the global mailbox as a side effect and, then, replaces the constructor send with the *continuation* t'. The fourth rule replaces the constructor rec by a call to function frec where the first message of the local mailbox is also passed as argument. The last two rules are used when the message does not match any pattern in the receive construct (which is denoted with the constructor

[2] We use Haskell's infix notation for lists, where $[\,]$ denotes the empty list and $x : xs$ a list with head element x and tail xs. When the number of elements is fixed, we also use the notation $[t_1, \ldots, t_n]$. Moreover, we use the operator $+\!+$ for list concatenation.

no_match), and we can either perform a new call to function frec passing the next message in the mailbox, or suspend the evaluation when the mailbox is empty. Observe that in Figure 1 we have a rule

$$\mathsf{frec}(h, vs, t) \to \mathsf{no\_match}(h, vs)$$

This rule is used as a *default* case when a call to function frec with a message does not match any of the patterns in the previous rules. In order to correctly model this behavior, one should ensure that once a redex matches a rule, all other rules are discarded (this is further discussed below in Section 3.4).

3.3 The Scheduler

Finally, we present the specification of the third component, $\mathcal{S}$, the scheduler. These rules take care of choosing a message from the global mailbox and dispatch it to the corresponding process. Here, we show a trivial scheduler that just dispatches the messages in the same order they are sent:

$$\mathsf{s}(k, \mathsf{m}(p, p', t) : ms, \mathsf{p}(p', t', ms') \odot ps) \to \mathsf{s}(k, ms, \mathsf{p}(p', t', ms'{+}{+}[t]) \odot ps)$$

In [14], for instance, we propose to first normalize a system before applying the scheduling rules. Here, we achieve the same effect by only applying the rules of the scheduler when no previous rule from $\mathcal{A}$ is applicable. Of course, more complex schedulers are possible. In particular, we might be interested in nondeterministically exploring all possible interleavings, as it is common in the context of model checking.

Consider now a system defined by the functional component shown in Figure 1 and the rules of Sections 3.2 and 3.3. Here, we consider the following initial system:

$$\mathsf{s}(1, [\,], \mathsf{p}(0, \mathsf{main}(\mathsf{succ}(0), \mathsf{succ}(0)), [\,]))$$

For clarity, we underline the selected redex at each reduction step; besides, we denote pids with 0,1,2,... instead of 0, succ(0), ... Moreover, we use different colors for each element of the system: green for the global mailbox, blue for the first process with pid 0 and red for the second process with pid 1. Then, reduction proceeds as shown in Figure 3. In this derivation, we have marked with $\overset{\alpha}{\to}$ the reduction steps where the scheduler dispatches a message to a local mailbox. Observe that this rule is only applied when no other rule is applicable.

Observe that the normal form contains defined functions –the case of s– and concurrent actions –the case of rec–. This is not unusual, since in general actor systems run forever (e.g., following a client-server architecture).

3.4 Intended Semantics

In this section, we briefly discuss the intended semantics for action systems specified with TRSs. First, we consider that reductions are performed using leftmost innermost rewriting, which essentially coincides with the reduction strategy of the functional and concurrent language Erlang. However, in order to precisely model the semantics of an Erlang-like language, one should further assume that

only the first rewrite rule that matches a redex is considered.

This is a strong requirement, but it is essential to be able to express the semantics of case and receive expressions. Overlapping cases –or default cases where the pattern is just a variable–

$$
\begin{array}{ll}
 & \mathsf{s}(1, [\,], \mathsf{p}(0, \mathsf{main}(\mathsf{succ}(0), \mathsf{succ}(0)), [\,])) \\
\to & \mathsf{s}(1, [\,], \mathsf{p}(0, \mathsf{spawn}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)], \mathsf{sum}), [\,])) \\
\to & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{fspawn}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)], 1), [\,]) \odot \mathsf{p}(1, \mathsf{sum}, [\,])) \\
\to & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{self}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0), 1]), [\,]) \odot \mathsf{p}(1, \mathsf{sum}, [\,])) \\
\to & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{fself}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0), 1], 0), [\,]) \odot \mathsf{p}(1, \mathsf{sum}, [\,])) \\
\to & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{send}(1, \mathsf{d}(\mathsf{succ}(0), \mathsf{succ}(0), 0), \ldots), [\,]) \odot \mathsf{p}(1, \mathsf{sum}, [\,])) \\
\to & \mathsf{s}(2, [\mathsf{m}(0, 1, \mathsf{d}(\mathsf{succ}(0), \mathsf{succ}(0), 0))], \mathsf{p}(0, \mathsf{rec}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)]), [\,]) \odot \mathsf{p}(1, \mathsf{sum}, [\,])) \\
\to & \mathsf{s}(2, [\mathsf{m}(0, 1, \mathsf{d}(\mathsf{succ}(0), \mathsf{succ}(0), 0))], \mathsf{p}(0, \mathsf{rec}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)]), [\,]) \\
 & \odot \mathsf{p}(1, \mathsf{rec}(\mathsf{sum1}, [\,]), [\,])) \\
\xrightarrow{\alpha} & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{rec}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)]), [\,]) \odot \mathsf{p}(1, \mathsf{rec}(\mathsf{sum1}, [\,]), [\mathsf{d}(\mathsf{succ}(0), \mathsf{succ}(0), 0)])) \\
\to & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{rec}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)]), [\,]) \odot \mathsf{p}(1, \mathsf{frec}(\mathsf{sum1}, [\,], \mathsf{d}(\mathsf{succ}(0), \mathsf{succ}(0), 0)), [\,])) \\
\to & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{rec}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)]), [\,]) \odot \mathsf{p}(1, \mathsf{send}(0, \mathsf{add}(\mathsf{succ}(0), \mathsf{succ}(0)), \mathsf{sum}), [\,])) \\
\to & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{rec}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)]), [\,]) \odot \mathsf{p}(1, \mathsf{send}(0, \mathsf{succ}(\mathsf{add}(0, \mathsf{succ}(0))), \mathsf{sum}), [\,])) \\
\to & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{rec}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)]), [\,]) \odot \mathsf{p}(1, \mathsf{send}(0, \mathsf{succ}(\mathsf{succ}(0)), \mathsf{sum}), [\,])) \\
\to & \mathsf{s}(2, [\mathsf{m}(1, 0, \mathsf{succ}(\mathsf{succ}(0)))], \mathsf{p}(0, \mathsf{rec}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)]), [\,]) \odot \mathsf{p}(1, \mathsf{sum}, [\,])) \\
\to & \mathsf{s}(2, [\mathsf{m}(1, 0, \mathsf{succ}(\mathsf{succ}(0)))], \mathsf{p}(0, \mathsf{rec}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)]), [\,]) \odot \mathsf{p}(1, \mathsf{rec}(\mathsf{sum1}, [\,]), [\,])) \\
\xrightarrow{\alpha} & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{rec}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)]), [\mathsf{succ}(\mathsf{succ}(0))]) \odot \mathsf{p}(1, \mathsf{rec}(\mathsf{sum1}, [\,]), [\,])) \\
\to & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{frec}(\mathsf{main1}, [\mathsf{succ}(0), \mathsf{succ}(0)], \mathsf{succ}(\mathsf{succ}(0))), [\,]) \odot \mathsf{p}(1, \mathsf{rec}(\mathsf{sum1}, [\,]), [\,])) \\
\to & \mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{succ}(\mathsf{succ}(0)), [\,]) \odot \mathsf{p}(1, \mathsf{rec}(\mathsf{sum1}, [\,]), [\,]))
\end{array}
$$

Figure 3 Example of the reduction of a system.

are common and we do not see any other viable alternative (note that it is also essential for the scheduling strategy mentioned above, where the rules of the scheduler are only applied when no other rule is applicable). Here, one may replace the default case alternatives by adding a rule for every non-matching constructor. However, this would only work as long as the number of atoms is finite (e.g., it will not work for integers). Of course, by ignoring this requirement, we would compute an overapproximation of the computations of the original system, but in general this is not a satisfactory solution (even more since a state explosion is a common problem when analyzing concurrent systems).

Another relevant point is that, for a system to be reduced using only the concurrent action rules $\mathcal{A}$ and the scheduling rules $\mathcal{S}$, one should further require that

> *in the rules of $\mathcal{E}$, the concurrent actions* self/2, spawn/3, send/3, *and* rec/2 *do not occur below a user-defined constructor or function symbol.*

In practice, these constructor symbols either occur in the topmost position of the right-hand side or in the *continuation* argument of send/3. Moreover, we say that a term is *safe* if it contains no occurrences of self/2, spawn/3, send/3 or rec/2 below a user-defined constructor or defined function. A TRS is safe if the right-hand sides of all its rules are safe. This static condition ensures that the rules of $\mathcal{A} \cup \mathcal{S}$ suffice to deal with concurrent actions since they can only occur in the topmost position of a process and, thus, the rules of Figure 2 suffice.

▶ **Theorem 5.** *Let $\mathcal{R} = \mathcal{E} \cup \mathcal{A} \cup \mathcal{S}$, where $\mathcal{E}$ is a safe constructor TRS, and $\mathcal{A}$ and $\mathcal{S}$ are defined as in Sections 3.2 and 3.3, respectively. Let t_0 be a term of the form* $\mathsf{s}(1, [], \mathsf{p}(0, \mathsf{f}(t_1, \ldots, t_n), []),$

where f *is a defined function of* $\mathcal{E}$ *and* $t_1, \ldots, t_n$ *are purely functional terms.*[3] *Then, for all* $t_0 \xrightarrow{i}{}^* t$ *we have that* t *is a safe term.*

Proof. We prove the claim by induction on the length of the reduction $t_0 \xrightarrow{i}{}^* t$. Since the base case $t_0 = t$ is trivial, we now consider the inductive case.

Assume that $t_0 \xrightarrow{i}{}^* t$ and t is safe. Let us consider now the innermost reduction step $t \xrightarrow{i}_{\mathcal{R}} t'$. First, we consider the case $t \xrightarrow{i}_{p,l\to r} t'$ with $l \to r \in \mathcal{E}$. By the inductive hypothesis, we have that t is safe and, thus, both $t[\,]_p$ and $t|_p = l\sigma$ are safe. Since r is also safe, we have that $t[r]_p$ is safe. It only remains to show that σ cannot introduce unsafe terms. Here, we know that $t|_p$ is rooted by a defined function of $\mathcal{E}$. Therefore, since $t|_p$ is safe, there are no occurrences of $\mathsf{self}/2$, $\mathsf{spawn}/3$, $\mathsf{send}/3$ or $\mathsf{rec}/2$. Hence, σ cannot introduce any occurrence of $\mathsf{self}/2$, $\mathsf{spawn}/3$, $\mathsf{send}/3$ or $\mathsf{rec}/2$ too, and $t[r\sigma]_p = t'$ is thus safe.

The case when $t \xrightarrow{i}_{p,l\to r} t'$ with $l \to r \in \mathcal{A} \cup \mathcal{S}$ is immediate by definition of $\mathcal{A} \cup \mathcal{S}$. ◀

4 Related Work

In this section, we review some related work on modelling actor-based concurrent systems in a language based on term rewriting. First, Giesl and Arts [6] deal with the verification of Erlang processes using dependency pairs. They transform Erlang programs to conditional rewrite systems by hand, so that termination can be analyzed. Another related approach –though for different source and target languages– is that of Albert *et al* [2], where a transformation from a concurrent object-oriented programming language based on message passing to a rule-based logic-like programming language is introduced.

As mentioned in the introduction, there are some approaches where the goal is to define a sort of interpreter in a rewriting based language like Maude [5]. This is the case of [9], who introduces an implementation of Erlang in *rewriting logic* [8], a unified semantic framework for concurrency. 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 specifying plain rewrite systems that can be analyzed using existing technologies.

This work can be seen as a continuation of [13]. While [13] focused on the transformation from programs written in a subset of Erlang to rewrite systems, in this work we focused on the modelling language within the term rewriting setting. The long-term goal is the definition of a modelling language that is rich enough to specify the main features of Erlang or Scala programming languages, so that program analysis and transformations can be designed. For instance, [13] already includes a preliminary deadlock analysis based on *narrowing* [10], an extension of rewriting to deal with logic variables.

5 Discussion

In this work, we have introduced a scheme for modelling actor-based concurrent systems in term rewriting. Our approach can be used as a basis for modelling Erlang-like programs, which can then be analyzed using existing techniques for term rewrite systems. For instance, [13] presents a translation scheme for Erlang programs to a specification which is closer to the one introduced in this paper. The transformation can be tested using the web interface from `http://kaz.dsic.upv.es/erlang2trs/`. A similar transformation can be defined from a subset of Erlang to the actor systems as specified in this paper.

[3] A *purely functional* term has no occurrences of $\mathsf{self}/2$, $\mathsf{spawn}/3$, $\mathsf{send}/3$, $\mathsf{rec}/2$, $\mathsf{p}/3$, $\mathsf{m}/3$, $\odot/2$ nor $\mathsf{s}/3$.

As a future work, we consider two main directions. On the first hand, we plan to investigate rewriting strategies to model the fact that only the first matching rule should be considered. This is the main difference with standard rewriting. For instance, one could define a transformation to the actor system –a sort of completion procedure– so that it can be reduced by standard rewriting with the desired behavior. On the other hand, and in order to be able to model integers, arithmetic operations, etc., we plan to extend the modelling language within the framework of integer rewriting. This will allow us to produce more precise models for programs written in, e.g., Erlang.

Acknowledgements. We thank the anonymous reviewers for their useful comments to improve this paper.

References

1 G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

2 E. Albert, P. Arenas, and M. Gómez-Zamalloa. Symbolic Execution of Concurrent Objects in CLP. In *PADL'12*, pages 123–137. Springer LNCS 7149, 2012.

3 Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.

4 F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

5 M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C.L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Springer LNCS 4350, 2007.

6 Jürgen Giesl and Thomas Arts. Verification of Erlang Processes by Dependency Pairs. *Appl. Algebra Eng. Commun. Comput.*, 12(1/2):39–72, 2001.

7 Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

8 José Meseguer. Conditioned Rewriting Logic as a United Model of Concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

9 Thomas Noll. A Rewriting Logic Implementation of Erlang. *Electr. Notes Theor. Comput. Sci.*, 44(2):206–224, 2001.

10 James R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

11 H. Svensson, L.-A. Fredlund, and C. Benac Earle. A unified semantics for future Erlang. In *Proc. of the 9th ACM SIGPLAN workshop on Erlang*, pages 23–32. ACM, 2010.

12 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.

13 G. Vidal. Towards Erlang Verification by Term Rewriting. In *LOPSTR'13*, pages 109–126. Springer LNCS 8901, 2014.

14 G. Vidal. Towards Symbolic Execution in Erlang (short paper). In *PSI'14*, pages 351–360. Springer LNCS 8974, 2014.