Playing with Bisimulation in Erlang

Ivan Lanese, Davide Sangiorgi, Gianluigi Zavattaro

Playing with Bisimulation in Erlang*

Ivan Lanese, Davide Sangiorgi, and Gianluigi Zavattaro

Focus Team, University of Bologna/INRIA
ivan.lanese@gmail.com, davide.sangiorgi@gmail.com,
gianluigi.zavattaro@unibo.it

Abstract. Erlang is a functional and concurrent programming language. The aim of this paper is to investigate basic properties of the Erlang concurrency model, which is based on asynchronous communication through mailboxes accessed via pattern matching. To achieve this goal, we consider Core Erlang (which is an intermediate step in Erlang compilation) and we define, on top of its operational semantics, an observational semantics following the approach used to define asynchronous bisimulation for the π-calculus. Our work allows us to shed some light on the management of process identifiers in Erlang, different from the various forms of name mobility already studied in the literature. In fact, we need to modify standard definitions to cope with such specific features of Erlang.

1 Introduction

Erlang is a message passing concurrent and functional programming language [3]. Erlang was originally a proprietary language within Ericsson, developed in 1986 to ensure high availability and fault-tolerance in distributed and massively concurrent telephony applications, but was released as open source in 1998 [2]. Along the years, it has been used not only in telephony, but also in many other high-visibility concurrent and distributed projects such as some versions of the Facebook chat [22].

Formal methods research on Erlang concentrated on defining its semantics, e.g., to precisely formalise the behaviour of Erlang implementations [15,29] and to drive future development [30]. Erlang semantics has been used also as a basis to develop tools, such as model checkers [16], static analysers [31], theorem provers for modal logics [17], declarative debuggers [9] and reversible debuggers [20,21].

Despite the remarkable interest in Erlang and its concurrency model, to the best of our knowledge, there is no research dealing with observational semantics for such language. Observational semantics represents one of the main fields of interest of Rocco De Nicola, with several pioneering contributions to which this paper (as well as many others from the authors) is strongly indebted.

The aim of this paper is to initiate the investigation of the applicability to Erlang of observational semantics already available in the literature. Instead

* This work has been partially supported by the French National Research Agency (ANR), project DCore n. ANR-18-CE25-0007.

of considering full Erlang that, e.g., includes rather expressive mechanisms for fault handling, in this work we focus on a simpler language corresponding to the functional and concurrent fragment of *Core Erlang* [10], which is an intermediate step in the compilation of Erlang.

Erlang is based on asynchronous communication, hence we have started by considering observational theories tailored to asynchrony. One of the first papers dealing with asynchronous communication in process algebra is by de Boer et al. [5], where different observation criteria are studied (bisimulation, traces and abstract traces) following the axiomatic approach typical of the process algebra ACP [4]. An alternative approach has been followed by Amadio et al. [1] who defined asynchronous bisimulation for the π-calculus [26]. They started from operational semantics (expressed as a standard labelled transition system), and then considered the largest bisimulation defined on internal steps that equates processes only when they have the same observables, and which is closed under contexts. The equivalence obtained in this way is called *barbed congruence* [24]. Notably, when asynchronous communication is considered, barbed congruence is defined assuming as observables the messages that are ready to be delivered to a potential external observer.

Merro and Sangiorgi [23] have subsequently studied barbed congruence in the context of the *Asynchronous Localised π-calculus* (ALπ), which is a fragment of the asynchronous π-calculus in which only output capabilities can be transmitted, i.e., when a process receives the name of a channel, it can only send messages along this channel (and not receive on it). Another line of research deals with the application of the testing approach to asynchronous communication; this has been investigated by Castellani and Hennessy [11] and by Boreale et al. [7,8]. These papers consider an asynchronous variant of CCS [25], and the proposed semantics turns out to be incomparable with barbed congruence for two main reasons. As usual, testing discriminates less as far as non deterministic choices are concerned, while it is able to observe divergent behaviours (while barbed congruence is not).

In Erlang, process identifiers can be passed around; this mechanism is similar to name passing in the π-calculus. Moreover, when a process receives a process identifier, it receives the capability to send messages to that process; this corresponds to the specific form of name mobility (transmission of output capability) of ALπ. For these reasons, we have adopted the approach by Merro and Sangiorgi defined for ALπ as our starting point. Differently from ALπ, in Erlang there is a unique receiver for each name, namely the process with that name.

Technically speaking, we consider the syntax of Core Erlang, and we investigate the applicability of the usual definition of barbed congruence. The direct application of standard definitions does not equate Erlang systems that are intuitively equivalent. Namely, consider an Erlang system A composed of a process that sends messages to a. Such system is expected to be equivalent to a system B in which the same process sends its messages to b instead of a, composed in parallel with a *wire* from b to a, i.e. a process with identifier b that simply forwards to a the messages that it receives.

If we apply to Erlang standard barbed congruence (denoted, as usual, with $\approx$) we have that $A \not\approx B$. The expected equivalence fails for two main reasons. First, if we put A and B in a context in which there is a process with identifier b, in the first case the obtained system is correct while in the second one it is not, because there are two distinct processes having the same identifier. The second problem is that if we put A and B in a context in which there is a message to b ready to be delivered, in the first case such message is observable while in the second one it is not because the unique receiver for such message (i.e., the wire) is already in the observed system.

We discuss a new definition of barbed congruence that solves the two above problems; we denote this new equivalence by $\approx_V^U$, where U and V are two sets of process identifiers. Both sets are used to impose limitations to the possible contexts considered in the barbed congruence definition. The set U contains names that cannot be used as identifiers for processes in the context; this can be used to solve the first of the two problems above by assuming $b \in U$, hence disallowing contexts that contain a process with identifier b. The set V contains identifiers that cannot be present in the context; this can be used to solve the second of the two problems above by assuming $b \in V$, hence disallowing contexts that contain messages to be delivered to b. Our novel bisimulation $\approx_V^U$ allows us to prove that the two above systems are equivalent.

Besides presenting the definition of this novel barbed congruence for Erlang, we use it to investigate basic properties of the concurrency model underlying such language. More precisely, we discuss the conditions under which we can rephrase in Core Erlang some typical equivalences of asynchronous name passing calculi. We also point out that some specific features of the Erlang language, like casting process identifiers to other data types or pattern matching operations on process identifiers, can break most of such equivalences.

The paper is structured as follows. In Sections 2 and 3 we present the syntax and semantics of Core Erlang. In Section 4 we present the definition of our novel barbed congruence and discuss some of its basic features, while in Section 5 we apply it to formalise some properties of Erlang. Finally, in Section 6 we discuss some possible extensions for our work, and we conclude in Section 7.

2 Erlang Syntax

In this section, we present the syntax of a first-order concurrent functional language that follows the actor model. Our language is essentially a subset of Core Erlang [10]. This material is mostly taken from [21].

The syntax of the language can be found in Figure 1. Here, a module is a sequence of function definitions, where each function name f/n (atom/arity) has an associated definition $\mathsf{fun}\ (X_1, \ldots, X_n) \to e$. We consider that a program consists of a single module for simplicity. The body of a function is an *expression*, which can include variables, literals, function names, lists, tuples, calls to built-in functions—mainly arithmetic and relational operators—, function applications, case expressions, let bindings, and receive expressions; furthermore, we also in-

$$
\begin{array}{rcl}
\mathit{module} & ::= & \mathsf{module}\ \mathit{Atom} = \mathit{fun}_1\ \ldots\ \mathit{fun}_n \\
\mathit{fun} & ::= & \mathit{fname} = \mathsf{fun}\ (\mathit{Var}_1, \ldots, \mathit{Var}_n) \to \mathit{expr} \\
\mathit{fname} & ::= & \mathit{Atom}/\mathit{Integer} \\
\mathit{lit} & ::= & \mathit{Atom} \mid \mathit{Integer} \mid \mathit{Float} \mid \mathit{Pid} \mid [\,] \\
\mathit{expr} & ::= & \mathit{Var} \mid \mathit{lit} \mid \mathit{fname} \mid [\mathit{expr}_1 | \mathit{expr}_2] \mid \{\mathit{expr}_1, \ldots, \mathit{expr}_n\} \\
& \mid & \mathsf{call}\ \mathit{Op}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \mid \mathsf{apply}\ \mathit{fname}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \\
& \mid & \mathsf{case}\ \mathit{expr}\ \mathsf{of}\ \mathit{clause}_1; \ldots; \mathit{clause}_m\ \mathsf{end} \\
& \mid & \mathsf{let}\ \mathit{Var} = \mathit{expr}_1\ \mathsf{in}\ \mathit{expr}_2 \mid \mathsf{receive}\ \mathit{clause}_1; \ldots; \mathit{clause}_n\ \mathsf{end} \\
& \mid & \mathsf{spawn}(\mathit{fname}, [\mathit{expr}_1, \ldots, \mathit{expr}_n]) \mid \mathit{expr}\,!\,\mathit{expr} \mid \mathsf{self}() \\
\mathit{clause} & ::= & \mathit{pat}\ \mathsf{when}\ \mathit{expr}_1 \to \mathit{expr}_2 \\
\mathit{pat} & ::= & \mathit{Var} \mid \mathit{lit} \mid [\mathit{pat}_1 | \mathit{pat}_2] \mid \{\mathit{pat}_1, \ldots, \mathit{pat}_n\}
\end{array}
$$

Fig. 1. Language syntax rules

clude the functions spawn, "!" (for sending a message), and self() that are usually considered built-ins in the Erlang language. As is common practice, we assume that X is a fresh variable in a let binding of the form $\mathsf{let}\ X = \mathit{expr}_1\ \mathsf{in}\ \mathit{expr}_2$.

As shown by the syntax in Figure 1, we only consider first-order expressions. Therefore, the first argument in applications and spawns is a function name. Analogously, the first argument in calls is a built-in operation Op.

In this language, we distinguish expressions, patterns, and values. Here, *patterns* are built from variables, literals, lists, and tuples, while *values* are built from literals, lists, and tuples, i.e., they are *ground*—without variables—patterns. Expressions are denoted by $e, e', e_1, e_2, \ldots$, patterns by pat, pat', pat_1, $pat_2, \ldots$ and values by $v, v', v_1, v_2, \ldots$ Atoms are typically denoted with roman letters, while variables start with an uppercase letter. As it is common practice, a *substitution* θ is a mapping from variables to expressions, and $\mathcal{D}om(\theta) = \{X \in \mathit{Var} \mid X \neq \theta(X)\}$ is its domain. Substitutions are usually denoted by sets of bindings like, e.g., $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$. Substitutions are extended to morphisms from expressions to expressions in the natural way. The identity substitution is denoted by id. Composition of substitutions is denoted by juxtaposition. Also, we denote by $\theta[X_1 \mapsto v_1, \ldots, X_n \mapsto v_n]$ the *update* of θ with the mapping $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$.

In a case expression "$\mathsf{case}\ e\ \mathsf{of}\ pat_1\ \mathsf{when}\ e_1 \to e'_1;\ \ldots;\ pat_n\ \mathsf{when}\ e_n \to e'_n\ \mathsf{end}$", we first evaluate e to a value, say v; then, we should find (if any) the first clause $pat_i\ \mathsf{when}\ e_i \to e'_i$ such that v matches pat_i (i.e., there exists a substitution σ for the variables of pat_i such that $v = pat_i\sigma$) and $e_i\sigma$—the *guard*—reduces to *true*; then, the case expression reduces to $e'_i\sigma$.

As for the concurrent features of the language, we consider that a *system* is a pool of processes that can only interact through message passing. Each process has an associated *pid* (process identifier). We consider a specific domain Pid for pids. Furthermore, in this work, we assume that pids can only be introduced in a computation from the evaluation of functions spawn and self (see below), and that there is no pid literal and no built-in function taking pids as arguments, apart for message sending. Pids however are valid values at runtime. The previous

```
main/0 = fun () → let Pid2 = spawn(target/0, [])
                  in let Pid3 = spawn(echo/1, [Pid2])
                  in let _ = Pid2 ! hello
                  in Pid3 ! world
target/0 = fun () → receive
                      M1 → receive
                             M2 → {M1, M2}
                           end
                  end
echo/1 = fun (PidT) → receive
                        M → PidT ! M
                      end
```

Fig. 2. A simple concurrent program

assumption forbids, e.g., casting from pids to strings and testing pids for equality. We further discuss this assumption in Section 6. By abuse of notation, when no confusion can arise, we refer to a process with its pid.

An expression of the form $\mathsf{spawn}(f/n, [e_1, \ldots, e_n])$ has, as a *side effect*, the creation of a new process, with a fresh pid a, initialised with the expression $\mathsf{apply}\ f/n\ (v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ are the evaluations of $e_1, \ldots, e_n$, respectively; the expression $\mathsf{spawn}(f/n, [e_1, \ldots, e_n])$ itself evaluates to the new pid a. The function $\mathsf{self}()$ just returns the pid of the current process. An expression of the form $e_1 \,!\, e_2$, where e_1 evaluates to a pid a and e_2 to a value v, also evaluates to the value v and, as a side effect, the value v—the *message*—will be stored in the queue or *mailbox* of process a at some point in the future.

Finally, an expression "$\mathsf{receive}\ pat_1\ \mathsf{when}\ e_1 \to e'_1; \ldots; pat_n\ \mathsf{when}\ e_n \to e'_n\ \mathsf{end}$" traverses the messages in the process' queue until one of them matches a branch in the receive statement; i.e., it should find the *first* message v in the process' queue (if any) such that $\mathsf{case}\ v\ \mathsf{of}\ pat_1\ \mathsf{when}\ e_1 \to e'_1; \ldots; pat_n\ \mathsf{when}\ e_n \to e'_n\ \mathsf{end}$ can be reduced to some expression e; then, the receive expression evaluates to the expression e, with the side effect of deleting the message v from the process' queue. If there is no matching message in the queue, the process suspends its execution until a matching message arrives.

Example 1. Consider the program shown in Figure 2, where the symbol "_" is used to denote an *anonymous* variable, i.e., a variable whose name is not relevant. The computation starts with "apply main/0 ()." This creates a process, say a1. Then, a1 spawns two new processes, say a2 and a3, and then sends the message hello to process a2 and the message world to process a3, which then resends world to a2. Note that we consider that variables $Pid2$ and $Pid3$ are bound to pids a2 and a3, respectively.

Given that there is no guarantee regarding which message arrives first to a3, function target/0 may return either $\{\text{hello}, \text{world}\}$ or $\{\text{world}, \text{hello}\}$. This

is coherent with the semantics of Erlang, where it is not possible to make any assumption on the order in which two messages, sent by two distinct senders, will be delivered to the same target process. In Erlang, the only expected assumption on message ordering is that if two messages are sent from a process to the same target, and both are delivered, then the order of these messages is kept. Nevertheless, current implementations only guarantee this property within the same node.

3 Erlang Semantics

In this section we formalise the semantics of the considered language. The semantics we present is equivalent to the one in [21], but allows for a simpler technical treatment.

Definition 1 (Process). *A process is denoted by a tuple $\langle a, (\theta, e), q\rangle$ where a is the pid of the process, (θ, e) is the control—which consists of an environment (a substitution) and an expression to be evaluated—and q is the process' mailbox, a queue with the sequence of messages that have reached the process.*

Given a message v and a mailbox q, we let $v : q$ denote a new mailbox with message v on top of it (i.e., v is the newer message). We also denote with $q \backslash\backslash v$ a new queue that results from q by removing the oldest occurrence of message v.

A running *system* is a pool of processes and floating messages, which we define as follows:

Definition 2 (System). *Systems, ranged over by $A, B, \ldots$, are generated by the following grammar:*

$$A := \langle a, (\theta, e), q\rangle \;\Big|\; (a, v) \;\Big|\; A_1 \mid A_2$$

that is, they are parallel compositions of processes and floating messages, where (a, v) is a floating message with content v targeting process a.

We only allow well-formed systems, in that the pids of the processes in a system are pairwise distinct. Moreover, we consider systems up-to associativity and commutativity of the parallel composition operator $\mid$. We therefore write $A = B$ to mean that the systems A and B are the same up-to associativity and commutativity of $\mid$.

In the definition above, floating messages represent messages in the system after they are sent, and before they are inserted in the target mailbox, in other terms, when they are in the network. Floating messages correspond to messages in the global mailbox of [21] or in the ether of [30].

The system representation above abstracts away from the distribution of processes over computing nodes. As a result, the only guarantee on message ordering offered by current Erlang implementations (i.e., order preserved among messages exchanged between the same pair of processes only if hosted in the same

$$(Var)\ \frac{}{\theta, X \overset{\tau}{\hookrightarrow} \theta, \theta(X)} \qquad (Tuple)\ \frac{\theta, e_i \overset{\ell}{\hookrightarrow} \theta', e_i' \quad i \in \{1, \ldots, n\}}{\theta, \{\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}\} \overset{\ell}{\hookrightarrow} \theta', \{\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}\}}$$

$$(List1)\ \frac{\theta, e_1 \overset{\ell}{\hookrightarrow} \theta', e_1'}{\theta, [e_1|e_2] \overset{\ell}{\hookrightarrow} \theta', [e_1'|e_2]} \qquad (List2)\ \frac{\theta, e_2 \overset{\ell}{\hookrightarrow} \theta', e_2'}{\theta, [v_1|e_2] \overset{\ell}{\hookrightarrow} \theta', [v_1|e_2']}$$

$$(Let1)\ \frac{\theta, e_1 \overset{\ell}{\hookrightarrow} \theta', e_1'}{\theta, \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 \overset{\ell}{\hookrightarrow} \theta', \mathsf{let}\ X = e_1'\ \mathsf{in}\ e_2} \qquad (Let2)\ \frac{}{\theta, \mathsf{let}\ X = v\ \mathsf{in}\ e \overset{\tau}{\hookrightarrow} \theta[X \mapsto v], e}$$

$$(Case1)\ \frac{\theta, e \overset{\ell}{\hookrightarrow} \theta', e'}{\theta, \mathsf{case}\ e\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \overset{\ell}{\hookrightarrow} \theta', \mathsf{case}\ e'\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end}}$$

$$(Case2)\ \frac{\mathsf{match}(\theta, v, cl_1, \ldots, cl_n) = \langle \theta_i, e_i \rangle}{\theta, \mathsf{case}\ v\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \overset{\tau}{\hookrightarrow} \theta\theta_i, e_i}$$

$$(Call1)\ \frac{\theta, e_i \overset{\ell}{\hookrightarrow} \theta', e_i' \quad i \in \{1, \ldots, n\}}{\theta, \mathsf{call}\ op\ (\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \overset{\ell}{\hookrightarrow} \theta', \mathsf{call}\ op\ (\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}})}$$

$$(Call2)\ \frac{\mathsf{eval}(op, v_1, \ldots, v_n) = v}{\theta, \mathsf{call}\ op\ (v_1, \ldots, v_n) \overset{\tau}{\hookrightarrow} \theta, v}$$

$$(Apply1)\ \frac{\theta, e_i \overset{\ell}{\hookrightarrow} \theta', e_i' \quad i \in \{1, \ldots, n\}}{\theta, \mathsf{apply}\ f/n\ (\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \overset{\ell}{\hookrightarrow} \theta', \mathsf{apply}\ f/n\ (\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}})}$$

$$(Apply2)\ \frac{\mu(f/n) = \mathsf{fun}\ (X_1, \ldots, X_n) \to e}{\theta, \mathsf{apply}\ f/n\ (v_1, \ldots, v_n) \overset{\tau}{\hookrightarrow} \theta \cup \{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}, e}$$

Fig. 3. Standard semantics: evaluation of sequential expressions

node) does not apply. Thus, we drop any assumption on the order of delivery of messages.

We write $\texttt{ppid}(P)$ (for process' pid) for the pid of the process P, and $\texttt{ppid}(A)$ for the set of the pids of the processes in A. A pid a is *fresh for* B, if a does not appear in B.

In the following, we denote by $\overline{o_n}$ a sequence of syntactic objects $o_1, \ldots, o_n$ for some n. We also write $\overline{o_{i,j}}$ for the sequence $o_i, \ldots, o_j$ when $i \leq j$ (and the empty sequence otherwise). We write $\overline{o}$ when the number of elements is not relevant.

The semantics is defined by means of two relations: $\hookrightarrow$ for expressions and $\longrightarrow$ for systems. Let us first consider the labelled transition relation

$$\hookrightarrow : (Env, Exp) \times Label \times (Env, Exp)$$

$$(Send1)\quad \frac{\theta, e_1 \overset{\ell}{\hookrightarrow} \theta', e_1'}{\theta, e_1 \,!\, e_2 \overset{\ell}{\hookrightarrow} \theta', e_1' \,!\, e_2} \qquad (Send2)\quad \frac{\theta, e_2 \overset{\ell}{\hookrightarrow} \theta', e_2'}{\theta, v_1 \,!\, e_2 \overset{\ell}{\hookrightarrow} \theta', v_1 \,!\, e_2'}$$

$$(Send3)\quad \frac{}{\theta, v_1 \,!\, v_2 \overset{\mathsf{send}(v_1,v_2)}{\hookrightarrow} \theta, v_2}$$

$$(Recv)\quad \frac{}{\theta, \mathsf{receive}\ cl_1; \ldots; cl_n\ \mathsf{end} \overset{\mathsf{rec}(\kappa,\overline{cl_n})}{\hookrightarrow} \theta, \kappa}$$

$$(Spawn1)\quad \frac{\theta, e_i \overset{\ell}{\hookrightarrow} \theta', e_i' \quad i \in \{1, \ldots, n\}}{\theta, \mathsf{spawn}(f/n, [\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}]) \overset{\ell}{\hookrightarrow} \theta', \mathsf{spawn}(f/n, [\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}])}$$

$$(Spawn2)\quad \frac{}{\theta, \mathsf{spawn}(f/n, [\overline{v_n}]) \overset{\mathsf{spawn}(\kappa, f/n, [\overline{v_n}])}{\hookrightarrow} \theta, \kappa} \qquad (Slf)\quad \frac{}{\theta, \mathsf{self}() \overset{\mathsf{self}(\kappa)}{\hookrightarrow} \theta, \kappa}$$

Fig. 4. Standard semantics: evaluation of concurrent expressions

where Env and Exp are the domains of environments (i.e., substitutions) and expressions, respectively, and $Label$ denotes an element of the set

$$\{\tau, \mathsf{send}(v_1, v_2), \mathsf{rec}(\kappa, \overline{cl_n}), \mathsf{spawn}(\kappa, a/n, [\overline{v_n}]), \mathsf{self}(\kappa)\}$$

whose meaning will be explained below. We use ℓ to range over labels. For clarity, we divide the transition rules of the semantics for expressions in two sets: rules for sequential expressions are depicted in Figure 3, while rules for concurrent ones are in Figure 4. Note, however, that concurrent expressions can occur inside sequential expressions.

Most of the rules are self-explanatory. In the following, we only discuss some subtle or complex issues. In principle, the transitions are labelled either with τ (a reduction without side effects) or with a label that identifies the reduction of an action with some side-effects. Labels are used in the system rules (Figure 5) to determine the associated side effects and/or the information to be retrieved.

We consider that the order of evaluation of the arguments in a tuple, list, etc., is fixed from left to right.

For case evaluation, we assume an auxiliary function match which selects the first clause, $cl_i = (pat_i\ \mathsf{when}\ e_i' \to e_i)$, such that v matches pat_i, i.e., $v = \theta_i(pat_i)$, and the guard holds, i.e., $\theta\theta_i, e_i' \hookrightarrow^* \theta', true$ (here $\hookrightarrow^*$ is the reflexive and transitive closure of $\hookrightarrow$). We assume that the patterns can only contain fresh variables. For simplicity, we assume here that if the argument v matches no clause then the evaluation is blocked.

Functions can either be defined in the program (in this case they are invoked by apply) or be a built-in (invoked by call). In the latter case, they are evaluated using the auxiliary function eval. In rule *Apply2*, we consider that the mapping μ stores all function definitions in the program, i.e., it maps every function name f/n to a copy of its definition $\mathsf{fun}\ (X_1, \ldots, X_n) \to e$, where $X_1, \ldots, X_n$ are (distinct) fresh variables and are the only variables that may occur free in

$$(Seq)\quad \frac{\theta, e \hookrightarrow^{\tau} \theta', e'}{\langle a, (\theta, e), q\rangle \xrightarrow{\emptyset} \langle a, (\theta', e'), q\rangle} \qquad (Send)\quad \frac{\theta, e \xhookrightarrow{\mathsf{send}(a', v)} \theta', e'}{\langle a, (\theta, e), q\rangle \xrightarrow{\emptyset} \langle a, (\theta', e'), q\rangle \mid (a', v)}$$

$$(Receive)\quad \frac{\theta, e \xhookrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q) = (\theta_i, e_i, v)}{\langle a, (\theta, e), q\rangle \xrightarrow{\emptyset} \langle a, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\backslash v\rangle}$$

$$(Spawn)\quad \frac{\theta, e \xhookrightarrow{\mathsf{spawn}(\kappa, f/n, [\overline{v_n}])} \theta', e' \quad a' \text{ is a fresh pid for } \langle a, (\theta, e), q\rangle}{\langle a, (\theta, e), q\rangle \xrightarrow{\{a'\}} \langle a, (\theta', e'\{\kappa \mapsto a'\}), q\rangle \mid \langle a', (id, \mathsf{apply}\ f/n\ (\overline{v_n})), [\,]\rangle}$$

$$(Self)\quad \frac{\theta, e \xhookrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\langle a, (\theta, e), q\rangle \xrightarrow{\emptyset} \langle a, (\theta', e'\{\kappa \mapsto a\}), q\rangle}$$

$$(Sched)\quad \frac{}{(a, v) \mid \langle a, (\theta, e), q\rangle \xrightarrow{\emptyset} \langle a, (\theta, e), v : q\rangle} \qquad (Par)\quad \frac{A \xrightarrow{U} A' \quad U \cap \mathtt{ppid}(B) = \emptyset}{A \mid B \xrightarrow{U} A' \mid B}$$

Fig. 5. Standard semantics: system rules

e. Note that we only consider first-order functions. In order to also consider higher-order functions, one should reduce the function name to a *closure* of the form $(\theta', \mathsf{fun}\ (X_1, \ldots, X_n) \to e)$. We leave this extension for future work.

Let us now consider the evaluation of expressions with side effects (Figure 4). Here, we can distinguish two kinds of rules. On the one hand, we have rules *Send1*, *Send2* and *Send3* for "!". In this case, we know *locally* what the expression should be reduced to (i.e., v_2 in rule *Send3*). For the remaining rules, this is not known locally and, thus, we return a fresh distinguished symbol, κ—by abuse, κ is dealt with as a variable—so that the system rules of Figure 5 will eventually bind κ to its correct value: the selected expression in rule *Recv* and a pid in rules *Spawn* and *Slf*. In these cases, the label of the transition contains all the information needed by system rules to perform the evaluation at the system level, including the symbol κ. This trick allows us to keep the rules for expressions and systems separated (i.e., the semantics shown in Figures 3 and 4 is mostly independent from the rules in Figure 5).

Finally, we consider the system rules, depicted in Figure 5. Reductions are of the form $A \xrightarrow{U} A'$ where U is the set of pids of processes spawned by the reduction. Also, $A \xRightarrow{U} A'$ means that A evolves into A' via a finite number of reductions in which the processes with pids in U have been spawned.

Rule *Seq* just updates the control (θ, e) of the considered process when a sequential expression is reduced using the expression rules.

Rule *Send* adds a new floating message (a', v) to the system. Adding it directly to the mailbox of the target process would not allow one to model all possible message interleavings (as discussed in Example 1). Observe that e' is usually different from v since e may have nested operators.

In rule *Receive*, we use the auxiliary function matchrec to evaluate a receive expression. The main difference w.r.t. match is that matchrec also takes a queue q and returns the selected message v. More precisely, function matchrec scans the queue q looking for the *first* message v matching a pattern of the receive statement. Then, κ is bound to the expression in the selected clause, e_i, and the environment is extended with the matching substitution. If no message in the queue q matches any clause, then the rule is not applicable and the selected process cannot be reduced (i.e., it suspends). As in case expressions, we assume that the patterns can only contain fresh variables.

Rule *Spawn* creates a new process with a fresh pid a', initialized with the application of function f/n. Its environment and queue are initially empty. The pid a' replaces κ in the process performing the spawn.

Rule *Self* simply replaces κ with the pid of the process.

Rule *Sched* delivers the message v in a pair (a, v) to the target process a. As discussed above, here we deliberately ignore the restriction mentioned in Example 1 that the messages sent –directly– between two given processes arrive in the same order they were sent, since current implementations only guarantee it within the same node. In practice, this amounts to consider that each process is potentially run in a different node. An alternative definition ensuring this restriction can be found in [27].

Rule *Par* allows one to lift a reduction to a larger system.

4 Behavioural Equivalence

In this section we initiate the study of behavioural equivalence for Erlang. Observational equivalences have been studied in the context of process calculi following the intuition that two systems should be considered equivalent only when they cannot be distinguished by any external observer. In our case — as in concurrent process calculi — an external observer is an additional pool of processes composed in parallel with the system to be observed. As discussed in the Introduction, to the best of our knowledge, the process calculus (equipped with an appropriate observational equivalence) which is closest to Erlang is the Asynchronous Localised π-calculus (ALπ) [23].

4.1 Barbed congruence

We now start the investigation of barbed congruence for the Erlang language. In ALπ, communication is based on channels identified by names: a process sends messages on a channel by indicating its name, and consumes messages by specifying the channel from which they are expected to be consumed. In Erlang, on the other hand, messages are sent to processes, and pids take the role of names. We introduce some terminology on pids.

Definition 3. *The pid a* is taken by *A if $a \in \mathtt{ppid}(A)$. The pid a* occurs untaken *in A if a appears in A (i.e., it is used in A) but is not taken by A.*

In the equivalence that we are going to define, we will equate only systems that are *pid-compatible*. The intuition is that two pid-compatible systems have the same expectations on the pids that should be taken by the environment, i.e. the context in which they will be observed. Thus a name that is taken by a system cannot occur untaken in the other one (as the latter is expecting the pid to be taken by the environment).

Definition 4. *A pair A, B of systems is* pid-compatible *if any pid that occurs untaken in A may not be taken by B, and conversely. Similarly, a relation $\mathcal{R}$ on systems is* pid-compatible *if all pairs of systems in $\mathcal{R}$ are pid-compatible.*

Only pid-compatible systems should be related. Below we implicitly assume that relations are pid-compatible. The notion of pid-compatibility is extended to reductions.

Definition 5. *Two reductions $A \stackrel{U}{\Longrightarrow} A'$ and $B \stackrel{V}{\Longrightarrow} B'$ are* pid-compatible *if A and B are pid-compatible and moreover the pids in U do not occur untaken in B, and conversely.*

As $\stackrel{U}{\longrightarrow}$ implies $\stackrel{U}{\Longrightarrow}$, the above terminology extends to one-step reductions.

Lemma 1. *If reductions $A \stackrel{U}{\Longrightarrow} A'$ and $B \stackrel{V}{\Longrightarrow} B'$ are pid-compatible, then also the derivatives A', B' are pid-compatible.*

Sometimes we write $A \longrightarrow A'$ and $A \Longrightarrow A'$ omitting the set of pids above the arrow if not important.

We are now ready to define *barbed congruence*. Such an equivalence can be defined in any calculus possessing: (i) a *reduction relation* (i.e., the 'internal steps' of process calculi), modelling the evolution of a system; and (ii) an *observability predicate* $\downarrow_a$ for each name a (pid in Erlang), which detects the possibility of a system of being observed from the outside by means of communication on a.

In Erlang, as in other calculi with asynchronous communication [1], only output messages are considered observable; this because an observer has no direct way of knowing when a message is actually received by the observed system. More precisely, we write $A \downarrow_a$ if A contains a floating message targeting a process with pid a expected to be in the context (i.e. not taken in A). Formally, $A \downarrow_a$ iff $A = A' \,|\, (a, v)$ with $a \notin \mathtt{ppid}(A')$, for some v. Also, $A \Downarrow_a$ iff there exists B and U such that $A \stackrel{U}{\Longrightarrow} B$ and $B \downarrow_a$.

We first introduce the barbed bisimulation equivalence, and then we close it by parallel contexts. The main novelty is that we need definitions that are compliant with the notion of pid-compatibility introduced above.

Definition 6 (Barbed bisimulation and congruence). *A relation $\mathcal{S}$ on systems is a U-*barbed bisimulation *if $A \,\mathcal{S}\, B$ implies:*

1. *if $A \stackrel{V}{\longrightarrow} A'$ then there exists a pid-compatible reduction $B \stackrel{W}{\Longrightarrow} B'$ with $A' \,\mathcal{S}\, B'$;*

2. *the converse of the above clause, on the reductions from B;*
3. *if $A \downarrow_a$ with $a \notin U$, then $B \Downarrow_a$;*
4. *the converse of the above clause, on the observables from B.*

*Two systems A and B are U-*barbed bisimilar*, written $A \stackrel{\cdot}{\sim}^U B$, if $A \mathcal{S} B$ for some U-barbed bisimulation $\mathcal{S}$.*

Let A and B be systems with $\mathtt{ppid}(A) \cup \mathtt{ppid}(B) \subseteq U$, and $V \subseteq U$. We say that A and B are barbed congruent *at $U; V$, written $A \approx_V^U B$, if, for each system C without occurrences of the pids in V and with $\mathtt{ppid}(C) \cap U = \emptyset$, we have $A \mid C \stackrel{\cdot}{\sim}^U B \mid C$.*

In barbed bisimulation, the parameter U contains pids that cannot be taken by the context, hence the pids in U are considered not observable. Barbed congruence has an additional parameter V; this is a set of special pids that the environment is not even allowed to know. This additional parameter compensates the absence, in Erlang, of explicit restriction operators denoting pid scopes; the scope of the pids in V is expected to be within the observed system. We preferred this approach instead of adding a restriction operator to stay closer to the semantics in [21] that we used as starting point for our development. Intuitively, it is reasonable to assume to have in V the pids of those processes that have been created within the observed system, whose name is never communicated outside.

We omit V when empty simply writing $A \approx^U B$. We also omit U both in barbed bisimulation and in barbed congruence, when not important.

4.2 A proof technique

We introduce a useful proof technique for barbed bisimilarity, based on the notion of expansion [28]. Intuitively expansion expresses the possibility for a system to match the reduction of another system using fewer reductions, i.e., more efficiently. Expansion is often used as an auxiliary relation in proof techniques for bisimulation.

We write: $A \longrightarrow_? A'$ if $A \longrightarrow A'$ or $A = A'$ (that is, A evolves into A' by means of one or zero reductions); and $A \Longrightarrow_1 A'$ to mean that A evolves into A' by means of at least one reduction. As announced, we omit below the requirements on compatibility between matching reductions, and therefore also the pid-labels decorating the reductions.

Definition 7 (barbed expansion). *A relation $\mathcal{S}$ on systems is a U-*barbed expansion *if $A \mathcal{S} B$ implies:*

1. *if $A \longrightarrow A'$ then $B \longrightarrow_? B'$ with $A' \mathcal{S} B'$;*
2. *if $B \longrightarrow B'$ then $A \Longrightarrow_1 A'$ with $A' \mathcal{S} B'$;*
3. *if $A \downarrow_a$ with $a \notin U$, then $B \downarrow_a$;*
4. *if $B \downarrow_a$ with $a \notin U$, then $A \Downarrow_a$.*

*Two systems A and B are in the U-*barbed expansion*, written $A \succeq^U B$, if $A \mathcal{S} B$ for some U-barbed expansion $\mathcal{S}$.*

As usual, we often omit the index U when empty (or not important) and simply write $A \succeq B$.

Definition 8. *A relation $\mathcal{S}$ on systems is a* U-barbed bisimulation up-to expansion *if $A\ \mathcal{S}\ B$ implies:*

1. *if $A \longrightarrow A'$ then there are B', A'', B'' such that $B \Longrightarrow B'$, $A' \succeq^U A''$, $B'' \succeq^U B'$ and $A''\ \mathcal{S}\ B''$;*
2. *the converse of the above clause, on the reductions from B;*
3. *if $A \downarrow_a$ with $a \notin U$, then $B \Downarrow_a$;*
4. *the converse of the above clause, on the observables from B.*

Lemma 2. *If $\mathcal{S}$ is a U-barbed bisimulation up-to expansion then $\mathcal{S} \subseteq \dot{\sim}^U$.*

Proof. A standard diagram-chasing argument [28]. □

The lemma above would become unsound if, in Definition 8, expansion were replaced by barbed bisimulation [28].

We conclude this section by showing a monotonicity property.

Lemma 3. *If $U \subseteq U'$ then $\dot{\sim}^U \subseteq \dot{\sim}^{U'}$ and $\succeq^U \subseteq \succeq^{U'}$.*

5 Properties

We now exploit the barbed congruence relation defined in the previous section to prove some properties of Erlang terms. The first property will be discussed in Theorem 1, where we prove that renaming a local pid a into b, is the same as adding in parallel a *wire* process that receives the messages on a and forwards them to b. In order to prove this first property, we need several preliminary results on pid renaming and system normalisation. The latter (which relies as usual on our restrictions on pid management) means that while proving two systems barbed equivalent, it is possible to focus only on those states of the system in which processes have completed their internal steps, where we assume internal all process transitions excluding the arrival of a new message in the process queue. Intuitively, normalisation holds because processes have a deterministic internal behaviour; the unique source of nondeterminism is in the order of arrival of messages.

5.1 Renaming

The first preliminary results deal with the correspondence between the transitions of the system A and that of $A\{a/b\}$. In the formalisation of these results, we use $A[a \triangleleft v]$ to mean the system obtained from A by adding the message v as the newer message in the queue of the process a of A.

Lemma 4. *Suppose $b \notin \mathtt{ppid}(A)$. We have:*

1. *if $A \xrightarrow{U} A'$ and $a \notin U$, then $A\{a/b\} \xrightarrow{U} A'\{a/b\}$;*

2. *if* $A\{a/b\} \xrightarrow{U} A''$, *with* $b \notin U$, *is a reduction derived without applying rule Sched to process* a, *then there is* A' *such that* $A \xrightarrow{U} A'$ *and* $A'' = A'\{a/b\}$;
3. *if* $A\{a/b\} \longrightarrow A''$ *is a reduction derived by applying rule Sched to process* a *and floating message* (a, v), *then either*
   (a) *there is* A' *such that* $A \longrightarrow A'$ *and* $A'' = A'\{a/b\}$; *or*
   (b) $A = (b, v') \mid B$, *with* $(b, v')\{a/b\} = (a, v)$ *and* $A'' = B\{a/b\}[a \triangleleft v]$.

Lemma 4(1) can be refined if a is fresh. We write $A[a \leftrightarrow b]$ for the system obtained from A by exchanging a and b (i.e., a simultaneous substitution).

Lemma 5. *Suppose* $b \notin \mathtt{ppid}(A)$ *and* a *is fresh for* A. *We have:*

1. *if* $A \xrightarrow{U} A'$ *and* $a \notin U$, *then* $A\{a/b\} \xrightarrow{U} A'\{a/b\}$;
2. *if* $A \xrightarrow{\{a\}} A'$ *then* $A\{a/b\} \xrightarrow{\{b\}} A'[a \leftrightarrow b]$.

Concerning the simultaneous renaming $[a \leftrightarrow b]$, we have a stronger correspondence between the transitions of A and $A[a \leftrightarrow b]$, when names a and b are already taken in A.

Lemma 6. *Suppose* $\{a, b\} \subseteq \mathtt{ppid}(A)$. *If* $A \xrightarrow{U} A'$ *then* $A[a \leftrightarrow b] \xrightarrow{U} A'[a \leftrightarrow b]$.

Corollary 1. *Suppose* $\{a, b\} \subseteq \mathtt{ppid}(A)$. *Then* $A \succeq A[a \leftrightarrow b]$.

We now move towards a result (Lemma 8) which is the Erlang analogous of the α-conversion renaming of name-passing calculi such as the π-calculus. We first need to introduce a new notation (Definition 9) and a preliminary lemma (Lemma 7).

Definition 9. *Two systems* A *and* B *are* a/b-convertible, *for pids* a, b *with* a *fresh for* B, *if* $A = B\{a/b\}$.

Lemma 7. *Suppose* A *and* B *are* a/b*-convertible, then* $A \succeq^{\{a,b\}} B$. *Moreover if* $\{a, b\} \subseteq (\mathtt{ppid}(A) \cup \mathtt{ppid}(B))$ *then also* $A \succeq B$.

Proof. If $\mathcal{S}$ is the set of all pairs (A, B) as in the assertion of the lemma, then $\mathcal{S} \cup \succeq$ is an expansion. This holds because: the reductions from two systems (A, B) as in the lemma are the same, either (Lemma 5(1)) modulo a renaming between a and b, or (Lemma 5(2)) producing derivatives that, by Corollary 1, are in the expansion relation; their observables different from a, b are the same too. □

When a, b are pids in A, B then they are not in the observables and therefore the assertion can be strengthened.

Lemma 8 (Erlang analogous of α-conversion). *Suppose* A *and* B *are* a/b*-convertible and* $\{a, b\} \subseteq U$. *Then* $A \approx^{U}_{\{a,b\}} B$.

Proof. Let C be a system in which a, b do not occur, and with $U \cap \mathtt{ppid}(C) = \emptyset$. We have to show that $A \mid C \;\dot{\sim}^{U}\; B\{a/b\} \mid C$. This follows from Lemma 7 and the inclusion $\succeq^{U} \subseteq \;\dot{\sim}^{U}$. □

5.2 Normalisation

In conjunction with the proof technique above, a crucial result that will be used afterwards is Lemma 10 below. For its proof we need Lemma 9 stating that the addition of a new message at the end of a process queue does not forbid the process from executing previously executable reductions.

Lemma 9. *Suppose* $P \xrightarrow{\emptyset} A$*; then, for any* a, v*, also* $P[a \triangleleft v] \xrightarrow{\emptyset} A[a \triangleleft v]$*.*

Proof. A case analysis on the possible rules that caused the reduction $P \longrightarrow A$. □

Essentially Lemma 10 below says that we can *normalise* any system, by letting the processes composing it evolve as long as there are messages in their queues that can be consumed (the only reduction of the system that we cannot perform in the normalisation are those that move a floating message into a queue). This includes the fetching of a message from its queue and the spawn of a new process. In Lemma 10, the reduction $P \xrightarrow{U} A$ has not been derived using rule *Sched* as it emanates from a process.

Lemma 10. *For any reduction* $P \xrightarrow{U} A$ *and system* C*, it holds that* $P \mid C \succeq A \mid C$*.*

Proof. We show that

$$\mathcal{R} \stackrel{\text{def}}{=} \{(P \mid C, A \mid C) \texttt{ s.t. } P \longrightarrow A\} \cup \succeq$$

is a barbed expansion. The interesting case is that of a challenge from $P \mid C$ in which P is involved (the case when only C is involved is trivial).

- If the reduction involving P is precisely $P \xrightarrow{U} A$ then $A \mid C$ need not move, as $\succeq$ includes the identity relation.
  If however the reduction is a spawn with, say $U = \{b\}$, then in its challenge P could choose to make a spawn on a different pid, say a. Thus the reduction is $P|C \xrightarrow{\{a\}} A\{^a/_b\}|C$. In this case we exploit Lemma 7, to derive $A\{^a/_b\}|C \succeq A|C$.
- Suppose the reduction involving P is a *Sched*, and let $P[a \triangleleft v]$ be the derivative of P, and C' the derivative of C. That is, the challenge is $P|C \longrightarrow P[a \triangleleft v]|C'$. We have $A \mid C \longrightarrow A[a \triangleleft v] \mid C'$. Moreover, by Lemma 9, $P[a \triangleleft v] \longrightarrow A[a \triangleleft v]$ (the reduction is not a *Sched* since $P[a \triangleleft v]$ is a process, hence does not contain floating messages). Hence $P[a \triangleleft v] \mid C' \; \mathcal{R} \; A[a \triangleleft v] \mid C'$. □

5.3 Wires

We prove a number of results concerning wires, which bring out fundamental properties of asynchrony for Erlang systems.

A *wire from* a *to* b is a process with pid a that sends at b all messages received at a, without modifying their content. We write $(q, a \triangleright b)$ for a wire from a to b whose queue is q.

Definition 10 (Wire). *We define* $(q, a \triangleright b)$ *as follows:*

$$(q, a \triangleright b) \stackrel{def}{=} \langle a, (\theta, \mathsf{apply}\ \mathit{wire}\ (b)), q\rangle$$

for any θ*. Furthermore, we assume to have the definition:*

$$\mathit{wire}/1 = \mathsf{fun}\ (Y) \to \mathsf{receive}\ X \to \mathsf{let}\ \_ = Y\,!\,X\ \mathsf{in}\ \mathsf{apply}\ \mathit{wire}\ (Y)\ \mathsf{end}$$

The first result that we prove on wires is that adding a wire from a into b is barbed bisimilar to renaming a into b, under the assumption that a is not already taken (hence it can be safely taken by the wire itself).

Lemma 11. *For all* C *with* $a \notin \mathtt{ppid}(C)$*, we have*

$$C \mid (\emptyset, a \triangleright b) \ \dot{\sim}\ C\{b/a\}$$

Proof. We show that the set of all such pairs is a barbed bisimulation up to expansion. The observables in the two related systems are the same: the only non-trivial case is an observable at b in $C\{b/a\}$, which corresponds to a floating message targeting a in C; this becomes an observable at b via the wire. Hence we only have to look at reductions. There are a few cases to consider. We exploit Lemma 4.

1. A reduction within C, say $C \longrightarrow C'$. By Lemma 4, we also have $C\{b/a\} \longrightarrow C'\{b/a\}$.
2. A floating message is moved into the queue of the wire, thus the reduction is $C \mid (\emptyset, a \triangleright b) \longrightarrow C' \mid (v : \emptyset, a \triangleright b)$, and $C = C' \mid (a, v)$. By Lemma 10, $C' \mid (v : \emptyset, a \triangleright b) \succeq C' \mid (b, v) \mid (\emptyset, a \triangleright b)$. Thus $C\{b/a\}$ need not move (up-to expansion), since $(C' \mid (b, v))\{b/a\} = C\{b/a\}$.
3. The case of reductions within $C\{b/a\}$ in which Lemma 4(2) or Lemma 4(3.a) can be applied is handled in a way similar to case (1) above.
   We consider the case in which Lemma 4(3.b) applies. Thus there is a floating message (b, v) in $C\{b/a\}$ that is moved into the queue of process b, with derivative C''. Let (a, v') be the corresponding floating message in C. Thus $C = C' \mid (a, v')$, and $C'' = C'\{b/a\}[b \triangleleft v]$. We also have the reductions

$$\begin{aligned} C \mid (\emptyset, a \triangleright b) &\longrightarrow C' \mid (v' : \emptyset, a \triangleright b) \\ &\Longrightarrow C' \mid (b, v') \mid (\emptyset, a \triangleright b) \\ &\longrightarrow C''' \mid (\emptyset, a \triangleright b) \end{aligned}$$

   where $C''' = C'[b \triangleleft v']$. This is sufficient, since $C'''\{b/a\} = C''$. □

We are now ready to prove our first example of barbed congruent systems, by extending to barbed congruence the previous result about barbed bisimilarity.

Theorem 1. *Suppose* $a \subseteq U$*. We have:*

$$A \mid (\emptyset, a \triangleright b) \approx^{U}_{\{a\}} A\{b/a\}$$

Proof. We have to show that for all B that does not contain a, we have

$$A \mid (\emptyset, a \triangleright b) \mid B \;\dot{\sim}\; A\{b/a\} \mid B$$

We have $A \mid (\emptyset, a \triangleright b) \mid B = (A \mid B) \mid (\emptyset, a \triangleright b)$, and $(A \mid B)\{b/a\} = A\{b/a\} \mid B$, hence we can apply Lemma 11. □

The above theorem indicates under which conditions adding a wire from a to b is barbed congruent w.r.t. applying directly a renaming of a into b: this holds only under the assumption that pid a is a restricted name that is not known by the environment. Nevertheless, under the assumption that a is known (but not taken) by the environment, we have the following result.

Theorem 2. *Suppose $a \subseteq U$. We have:*

$$A \mid (\emptyset, a \triangleright b) \approx^{U}_{\emptyset} A\{b/a\} \mid (\emptyset, a \triangleright b)$$

Proof. We have to show that for any C with $a \notin \mathtt{ppid}(C)$, we have

$$A \mid C \mid (\emptyset, a \triangleright b) \;\dot{\sim}\; A\{b/a\} \mid C \mid (\emptyset, a \triangleright b)$$

Using Lemma 11, we have $A|C|(\emptyset, a \triangleright b) \;\dot{\sim}\; (A|C)\{b/a\}|(\emptyset, a \triangleright b) \stackrel{\text{def}}{=} B_1$. Similarly, $A\{b/a\} \mid C \mid (\emptyset, a \triangleright b) \;\dot{\sim}\; (A\{b/a\} \mid C)\{b/a\} \mid (\emptyset, a \triangleright b) \stackrel{\text{def}}{=} B_2$. This completes the proof, since $B_1 = B_2$. □

As a corollary of Theorem 1 we have an additional equivalence result: a wire from a to b in parallel with a wire from b to c is equivalent to a unique wire from a to c (under the assumption that pid b is not known from the environment).

Corollary 2. *We have*

$$(\emptyset, a \triangleright b) \mid (\emptyset, b \triangleright c) \approx^{\{b\}}_{\{b\}} (\emptyset, a \triangleright c)$$

Proof. Follows from Theorem 1. □

As a corollary of Theorem 2, on the other hand, we can prove that in the presence of a wire from a to b, a floating message targeting a is equivalent to a floating message, with the same content, directly targeting b.

Corollary 3. *We have*

$$(\emptyset, a \triangleright b) \mid (a, v) \approx^{\{a\}} (\emptyset, a \triangleright b) \mid (b, v)$$

Proof. Follows from Theorem 2. □

The last equivalence that we prove still deals with wires, but used in a different way. If a process sends a pid c to the outside environment, this is equivalent to sending a pid d (unknown by the environment) under the assumption that there is a wire from d to c.

Theorem 3. *Suppose $a, d \subseteq U$ and d is fresh for $\langle a, (\theta, b\,!\,c), q\rangle$. We have:*

$$\langle a, (\theta, b\,!\,c), q\rangle \approx^{U}_{\{d\}} \langle a, (\theta, b\,!\,d), q\rangle \mid (\emptyset, d \triangleright c)$$

Proof. We have to show that for any C, with $a \notin \mathtt{ppid}(C)$ and without occurrences of pid d, we have

$$\langle a, (\theta, b\,!\,c), q\rangle \mid C \;\dot{\sim}\; \langle a, (\theta, b\,!\,d), q\rangle \mid (\emptyset, d \triangleright c) \mid C$$

Starting from the r.h.s., by using Lemma 11, we have $\langle a, (\theta, b\,!\,d), q\rangle \mid (\emptyset, d \triangleright c) \mid C = (\langle a, (\theta, b\,!\,d), q\rangle \mid C) \mid (\emptyset, d \triangleright c) \;\dot{\sim}\; (\langle a, (\theta, b\,!\,d), q\rangle \mid C)\{c/d\} \stackrel{\text{def}}{=} B$. By applying the substitution to B, we obtain $B = \langle a, (\theta\{c/d\}, b\,!\,c), q\{c/d\}\rangle \mid C\{c/d\} = \langle a, (\theta, b\,!\,c), q\rangle \mid C$ since pid d does not occur in θ, q and C. □

6 Alternative Approaches

In this section we discuss some possible alternatives to the choices made in the main development described in previous sections. As stated in Section 2 we pose various restrictions on the use of pids. Essentially, pids can only be generated by functions spawn and self, passed around, and used to send messages.

Erlang also allows, e.g., equality test on pids. This is forbidden in our context since equality would be a built-in function taking pids as arguments, which we disallow. It is not possible to have such test via pattern matching, since variables in patterns are always fresh, we forbid pid literals, and patterns cannot contain functions. Equality test would break some of ours results. For instance, in Theorem 1, we have:

$$A \mid (\emptyset, a \triangleright b) \approx^{U}_{\{a\}} A\{b/a\}$$

If we take $A = \langle c, (\theta, a!a), q\rangle$, then on the l.h.s. after a few steps b would get a message with value a, while on the r.h.s. it would get a message with value b. Comparing the two values with b would distinguish the two processes. In contrast, if b could only use the received name to send messages, no distinction would be possible, since in both cases messages would reach b, either directly or through the wire.

Even worst, built-in functions taking pids in input, such as the function `pid_to_list(Pid)`, which converts a pid to a string, would break most of our results. In the case above, just converting pids to strings and comparing the strings would break the bisimilarity. However these functions are mainly intended for debugging.

Another assumption we made is that order of messages is not preserved. Erlang specification states that if two messages are sent from a same process a to a same process b, and both are received, then the order is preserved. However, current implementations only guarantee this inside the same node, thus our approach would correspond to running each process on its own node. Adding this constraint would require changing the semantics at the system level, replacing

floating messages with explicit queues. This would impact on the theory we present. For instance, in Theorem 1, on the l.h.s. A could send a message directly to b, and another one via the wire. They could reach b in any order. On the r.h.s., both messages would go directly to b, hence the order would be preserved. We could recover the result by requiring A not to contain b.

7 Conclusion

We have investigated the definition of observational semantics for Erlang. This work has been initially conceived with the aim to honor the career of Rocco De Nicola. According to the citations received by his papers, the two main contributions of Rocco are about testing equivalences [13] and KLAIM [12]. Testing equivalences are an example of observational semantics, while KLAIM is a concurrent and distributed language that, similarly to Erlang, is based on message exchange through local repositories accessed via pattern-matching. In order to do some original work, i.e. avoid to simply replicate work already done by Rocco and his co-authors, we have followed a slightly different approach for defining observational semantics (i.e. barbed bisimulation) on a language having some differences w.r.t. KLAIM. There are two main differences between the concurrent model of Erlang and that of KLAIM: locality (when a message is created, only its expected receiver has the capability to read it) and mailbox ordering (pattern matching is applied to messages according to their order of reception).

Despite the initial celebrating objective, we think the paper contains interesting original contribution towards the development of observational theories for Erlang. As a future work, we would like to continue the study of the introduced equivalence by investigating a labelled characterisation of barbed congruence and algebraic characterisations, or transferring type-based techniques from the π-calculus (e.g. to control termination, lock-freedom and deadlock [32,14,18,19]). Moreover, we would like to investigate other approaches to observational semantics such as may and must testing. Concerning this last point, we could apply our approach to define observational equivalences following [6], where different congruences are studied simply by considering a unique definition parametric in the notion of observable. The main novelties in our proposal concern the management of Erlang pids, whereas the notion of observable (i.e. output messages) is standard. Being pid management and the observability criteria two orthogonal concepts, we are confident that our techniques could be applied also to alternative notions of observables like those studied by Boreale et al. [6].

References

1. Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-calculus. Theor. Comput. Sci. 195(2), 291–324 (1998)
2. Armstrong, J.: A history of Erlang. In: Third ACM SIGPLAN Conference on History of Programming Languages. pp. 6–1–6–26. ACM (2007)

3. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent programming in Erlang (2nd edition). Prentice Hall (1996)
4. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60(1-3), 109–137 (1984)
5. de Boer, F.S., Klop, J.W., Palamidessi, C.: Asynchronous communication in process algebra. In: LICS. pp. 137–147. IEEE Computer Society (1992)
6. Boreale, M., De Nicola, R., Pugliese, R.: Basic observables for processes. Inf. Comput. 149(1), 77–98 (1999)
7. Boreale, M., De Nicola, R., Pugliese, R.: A theory of "may" testing for asynchronous languages. In: FoSSaCS. Lecture Notes in Computer Science, vol. 1578, pp. 165–179. Springer (1999)
8. Boreale, M., De Nicola, R., Pugliese, R.: Trace and testing equivalence on asynchronous processes. Inf. Comput. 172(2), 139–164 (2002)
9. Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S.: Declarative debugging of concurrent Erlang programs. J. Log. Algebr. Meth. Program. 101, 22–41 (2018)
10. Carlsson, R., Gustavsson, B., Johansson, E., Lindgren, T., Nyström, S.O., Pettersson, M., Virding, R.: Core erlang 1.0.3. language specification (2004), available from `https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf`
11. Castellani, I., Hennessy, M.: Testing theories for asynchronous languages. In: FSTTCS. Lecture Notes in Computer Science, vol. 1530, pp. 90–101. Springer (1998)
12. De Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: A kernel language for agents interaction and mobility. IEEE Trans. Software Eng. 24(5), 315–330 (1998)
13. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. 34, 83–133 (1984)
14. Demangeon, R., Hirschkoff, D., Sangiorgi, D.: Termination in impure concurrent languages. In: Proc. 21th Conf. on Concurrency Theory. Lecture Notes in Computer Science, vol. 6269, pp. 328–342. Springer (2010)
15. Fredlund, L.A.: A Framework for Reasoning about Erlang Code. Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden (2001)
16. Fredlund, L., Earle, C.B.: Model checking Erlang programs: the functional approach. In: ACM SIGPLAN Workshop on Erlang. pp. 11–19. ACM (2006)
17. Fredlund, L., Gurov, D., Noll, T., Dam, M., Arts, T., Chugunov, G.: A verification tool for Erlang. STTT 4(4), 405–420 (2003)
18. Kobayashi, N.: A partially deadlock-free typed process calculus. Transactions on Programming Languages and Systems 20(2), 436–482 (1998)
19. Kobayashi, N., Sangiorgi, D.: A hybrid type system for lock-freedom of mobile processes. ACM Trans. Program. Lang. Syst. 32(5) (2010)
20. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: Cauder: A causal-consistent reversible debugger for Erlang. In: FLOPS. Lecture Notes in Computer Science, vol. 10818, pp. 247–263. Springer (2018)
21. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. J. Log. Algebr. Meth. Program. 100, 71–97 (2018)
22. Letuchy, E.: Erlang at Facebook. `http://www.erlang-factory.com/conference/SFBayAreaErlangFactory2009/speakers/EugeneLetuchy` (2009)
23. Merro, M., Sangiorgi, D.: On asynchrony in name-passing calculi. Mathematical Structures in Computer Science 14(5), 715–767 (2004)
24. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) Proc. 19th ICALP. Lecture Notes in Computer Science, vol. 623, pp. 685–695. Springer Verlag (1992)

25. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
26. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Inf. Comput. 100(1), 1–40 (1992)
27. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: LOPSTR. LNCS, vol. 10184, pp. 259–274. Springer (2017)
28. Sangiorgi, D., Milner, R.: The problem of "Weak Bisimulation up to". In: Cleveland, W. (ed.) Proc. CONCUR '92. Lecture Notes in Computer Science, vol. 630, pp. 32–46. Springer Verlag (1992)
29. Svensson, H., Fredlund, L.A.: A more accurate semantics for distributed Erlang. In: SIGPLAN Workshop on Erlang. pp. 43–54. ACM (2007)
30. Svensson, H., Fredlund, L.A., Benac Earle, C.: A unified semantics for future Erlang. In: ACM SIGPLAN Workshop on Erlang. pp. 23–32. ACM (2010)
31. Tóth, M., Bozó, I.: Static analysis of complex software systems implemented in Erlang. In: Central European Functional Programming School - 4th Summer School, CEFP. Lecture Notes in Computer Science, vol. 7241, pp. 440–498. Springer (2011)
32. Yoshida, N., Berger, M., Honda, K.: Strong Normalisation in the Pi-Calculus. Information and Computation 191(2), 145–202 (2004)