computers

MDPI

Article

Program Equivalence in the Erlang Actor Model

Péter Bereczky [1,*], Dániel Horpácsi [1,*] and Simon Thompson [1,2]

[1] Department of Programming Languages and Compilers, ELTE Eötvös Loránd University, 1117 Budapest, Hungary; s.j.thompson@kent.ac.uk
[2] School of Computing, University of Kent, Canterbury CT2 7NZ, UK
[*] Correspondence: berpeti@inf.elte.hu (P.B.); daniel-h@elte.hu (D.H.)

Abstract: This paper presents the formal semantics of concurrency in Core Erlang, an intermediate language for Erlang, along with a notion of program equivalence (based on barbed bisimulation) that is able to model equivalence between programs that have different communication structures but the same observable behaviour. The novelty in our formalisation is its extent: it includes semantics for messages and exit and link signals, in addition to most of Core Erlang's sequential features. Furthermore, unlike previous studies, this work formalises message receipt using primitive operations, consistent with the standard as of Erlang/OTP 23. In this novel formalisation, we show some generally applicable program equivalences (such as process identifier renaming and silent evaluation) and present a practical case study featuring the equivalence of sequential and concurrent list processing.

Keywords: formal semantics; program equivalence; Erlang; barbed bisimulation; concurrency; Coq

check for updates

Citation: Bereczky, P.; Horpácsi, D.; Thompson, S. Program Equivalence in the Erlang Actor Model. *Computers* **2024**, *13*, 276. https://doi.org/10.3390/computers13110276

Academic Editor: Yan Liu

Received: 31 August 2024
Revised: 4 October 2024
Accepted: 16 October 2024
Published: 23 October 2024

1. Introduction

The main motivation of this work is to provide formal basis for reasoning about refactoring transformations of Erlang programs. Code refactoring is the process of improving the internal structure of a program without affecting its observable behaviour [1]. Admittedly, one of the main challenges of refactoring is correctness [2,3] as "refactoring might not always be behaviour-preserving in practice" [4]. Most refactoring tools lack a precise specification of how they affect the code, are only verified via testing, and in peculiar circumstances they can introduce bugs. Therefore, the majority of developers do not even use automated tools specifically designed for refactoring [2,3].

We aim to break the status quo and develop trustworthy refactoring tools with formal guarantees of behaviour-preservation, fostering tool-assisted refactoring. By using mathematical definitions of program behaviour and program equivalence, we can achieve high levels of assurance by carrying out machine-checked, formal proofs about behaviour-preservation of program transformations.

1.1. Our Target Language

This paper extends our previous work on the formalisation of (Core) Erlang [5,6], an impure, concurrent functional programming language featuring strict evaluation, uncurried function abstraction and application. In particular, our work here targets the concurrent subset of Core Erlang, by defining a modular formal semantics and program equivalence definitions for it, as a milestone towards reasoning about the correctness of refactoring concurrent programs. The language features we cover in this paper allow us to express and prove the behavioural equivalence of sequential and parallel algorithmic skeletons (such as *map* and *pmap*), a formal result strongly motivated and encouraged in [7].

(Core) Erlang implements and extends the actor model [8] to express concurrency. An Erlang node consists of processes (actors) which execute in their own memory. Communication between processes is achieved via asynchronous message passing; the messages sent from a process are placed at the end of the receiver's mailbox and the receiver can select messages to handle (i.e., messages do not need to be processed in the order of their arrival).

In addition to messages, Erlang processes communicate via signals such as *link*, *unlink*, or *exit* [9]. These signals potentially modify the state of the process upon their arrival, without being placed into the mailbox. Links can be established and removed between processes with *link* and *unlink* signals. These links are bidirectional and serve as a way to notify a process with an *exit* signal when a linked process has terminated. *Exit* signals express termination, and, upon their arrival, a process can terminate, drop the signal, or convert it into a message and place it at the end of its mailbox. Processes also have different process flags. In this formalisation, we address the `trap_exit` flag; whenever this flag is set, *exit* signals will be converted into messages (except in very particular circumstances). *Links* and trapping *exit* signals are the main ingredients of building fault-tolerant Erlang systems.

1.2. Contributions

In this paper, we investigate the actor model of Core Erlang with the extensions mentioned above. Namely, we make the following contributions:

- An upgrade to the modular, frame stack style semantics for concurrent Core Erlang [10,11] (including a more refined semantics of message reception and exceptions);
- A notion of program equivalence for concurrent Core Erlang, based on bisimulation, that is able to model equivalence between programs that have different communication structures but the same observable behaviour;
- Concurrent program equivalence examples: We show that silent evaluation and process identifier renaming produce equivalent programs, and that transforming lists sequentially is equivalent to doing it concurrently;
- A machine-checked formalisation of the semantics and results concerning it in the Coq proof management system (we establish the link between the code and this paper in Appendix C).

The paper is structured as follows. In Section 2, we briefly summarise related and previous work, then Section 3 presents the formal semantics of concurrent Core Erlang. Section 4 defines bisimulations to argue about program equivalence, and shows some examples. Section 5 highlights theoretical and technical challenges, discusses the Coq implementation, and compares our work to tightly related research. Finally, Section 6 concludes the paper.

2. Related Work

In this section, we briefly summarise the related and our previous work on formal semantics for (Core) Erlang and program equivalence in the concurrent setup.

2.1. Erlang Semantics

Fredlund's influential work [12] set the state of the art in the formal definition of the Erlang programming language. It follows the documentation [5] faithfully, including signals (such as *exit* and *link*), similar to our approach. However, unlike our definition, Fredlund's semantics handles signal-passing as an atomic operation, while according to the "signal ordering guarantee" [9], this is not necessarily the case.

The work of Lanese et al. [13–16] defines the semantics of Core Erlang for a small subset of sequential Core Erlang expressions, messages, primitives for message-passing, and process creation. They express message-passing as non-atomic; however, the order of messages between a source and destination is not preserved in the ether, potentially violating Erlang's semantics on signal ordering [9]. They also impose a restriction on their semantics: process identifiers can only appear as computation results (i.e., when evaluating *self* or *spawn* actions); however, the main limitation of their work is the language coverage of the semantics, for both the sequential and concurrent features.

Harrison's [17] formalisation of Core Erlang is minimal, but it is implemented in Isabelle, which aided our Coq development. There is also related research on Core Erlang with the goal of causal debugging of concurrent programs [18], which defines a semantics of similar coverage to ours.

All the related work mentioned above models `receive` expressions as language primitives, while (as of OTP 23 [19]) message receipts are expressed with primitive operations. In this work, we address this change, but this comes with a number of drawbacks which we discuss in Section 5.

2.2. Bisimulation Approaches

The original notion of bisimulation has a long history [20–22]. Since then, multiple variants of bisimulations have been proposed. We rely on the notion of barbed bisimulation [23]. Barbed bisimulation explicitly defines what should be observed, and its weak variants do not compare how many and what steps the bisimilar systems take.

This notion of observational equivalence was successfully applied to Erlang-like languages: by Lanese et al. [16] to prove that PID renaming and evaluation without message arrives provide bisimilar Core Erlang nodes, and by Bocchi et al. [24] in an actor model variant with failures. There are also a number of proof techniques based on bisimulation (e.g., bisimulation up-to) [21], some of which were also discussed in [16]. In the future, we plan to adopt these ideas to aid in bisimulation proofs for Erlang refactorings.

2.3. Previous Work

In our previous work [11], we defined a frame stack semantics for the sequential sublanguage of Core Erlang, and investigated several program equivalence concepts in the sequential setup. Moreover, we also investigated the concurrency model of Core Erlang with a minimal sublanguage [10], also based on a frame stack semantics.

A frame stack semantics is essentially a small-step [25], reduction-style [26] semantics where the reduction context is split into a stack of basic evaluation frames [27]. In this semantics style, there are explicit rules to decompose the reduction context around a redex into a stack which represents the continuation of the evaluation. This semantics style is simpler to use in proof assistants because the reduction context does not need to be inferred; the aforementioned rules construct it explicitly.

This paper unites our two previous results [10,11], extends the formalisation for better language coverage (based on Fredlund's work [12]), and adapts the techniques used by Lanese et al. [16] to argue about program equivalence. Although the theorems we discuss here were also proved by Lanese et al. [16], their proofs are mathematical, high-level proofs (for a restricted sublanguage of Core Erlang), whilst our results are also implemented in Coq as formal, machine-checked proofs.

Compared to our previous work, our semantics is extended with the formalisation of exceptions in the concurrent setup, `spawn_link` (spawning a process while creating a link) and expresses message receipt with primitive operations (as introduced in Erlang/OTP 23 [19]). We also define a more faithful representation of mailboxes and links which was necessary to handle the above-mentioned primitive operations correctly. In fact, to best of our knowledge, our formalisation is the first one to address this major change in message receipts.

3. Concurrent Formal Semantics

In this section, we describe the syntax and semantics of Core Erlang. The semantics presented here is modular and consists of three layers (see Table 1).

Table 1. The layers of the semantics.

Layer Name	Notation	Description
Inter-process (Section 3.5)	$\xrightarrow{\iota:a}_O$	System-level reductions
Process-local (Section 3.4)	$\xrightarrow{a}$	Process-level reductions
Sequential (Section 3.3)	$\longrightarrow$	Computational reductions

Reductions denoted with $\longrightarrow$ are computational steps performed by the sequential frame stack semantics defined in our previous work [11]. Informally, $\langle K, r\rangle \longrightarrow \langle K', r'\rangle$ means that in the frame stack (continuation) K, a redex r is rewritten to r' while the stack changes to K'. Reductions denoted with $p \xrightarrow{a} p'$ are the process-local steps, which involve one single process and communicate with the inter-process semantics by the actions. The inter-process semantics (denoted with $N \xrightarrow{\iota:a}_{O} N'$) describes how communication is carried out between processes.

This section is structured as follows. In Section 3.1 we describe the formal syntax of Core Erlang, then in Sections 3.3–3.5 we define the semantics following the structure in Table 1. Section 3.6 discusses an example evaluation (on which we build a program equivalence proof in Example 2), and Section 3.7 presents substantial properties of the semantics.

3.1. Language Syntax

First, we recall the syntax of Core Erlang from our previous work [11] in Figure 1. We reuse the same notations and shorthands: lists from the metatheory are denoted with $e_1, \dots, e_n$, and non-empty lists with $e_1, e_2, \dots, e_n$. Appending an element to the front of a list is denoted with $x :: l$, and for concatenation we use $l_1 \mathbin{+\!\!+} l_2$.

$$
\begin{aligned}
p \in \mathit{Pattern} &::= i \mid a \mid x \mid [p_1|p_2] \mid [] \mid \{p_1, \dots, p_n\} \mid \sim\!\{p_1^k \Rightarrow p_1^v, \dots, p_n^k \Rightarrow p_n^v\}\!\sim \\
\mathit{ps} \in \mathit{list(Pattern)} &::= <p_1, \dots, p_n> \\
\mathit{cl} \in \mathit{Clause} &::= \mathit{ps}\ \texttt{when}\ e^g \to e^b \\
\mathit{cli} \in \mathit{ClosItem} &::= f/k = \texttt{fun}(x_1, \dots, x_k) \to e \\
\mathit{fdefs} \in \mathit{list(ClosItem)} &::= \mathit{cli}_1, \dots, \mathit{cli}_n \\
v \in \mathit{Val} &::= i \mid a \mid x \mid f/k \mid \iota \mid \mathit{clos}(\mathit{fdefs}, [x_1, \dots, x_n], e) \mid [v_1|v_2] \mid [] \mid \{v_1, \dots, v_n\} \\
&\quad \mid \sim\!\{v_1^k \Rightarrow v_1^v, \dots, v_n^k \Rightarrow v_n^v\}\!\sim \\
\mathit{nv} \in \mathit{NonVal} &::= \texttt{fun}(x_1, \dots, x_n) \to e \mid <e_1, \dots, e_n> \mid [e_1|e_2] \mid \{e_1, \dots, e_n\} \\
&\quad \mid \sim\!\{e_1^k \Rightarrow e_1^v, \dots, e_n^k \Rightarrow e_n^v\}\!\sim \mid \texttt{call}\ e^m{:}e^f(e_1, \dots, e_n) \mid \texttt{primop}\ a(e_1, \dots, e_n) \\
&\quad \mid \texttt{apply}\ e(e_1, \dots, e_n) \mid \texttt{case}\ e_1\ \texttt{of}\ \mathit{cl}_1; \dots; \mathit{cl}_n\ \texttt{end} \mid \texttt{let}\ <x_1, \dots, x_n> = e_1\ \texttt{in}\ e_2 \mid \texttt{do}\ e_1\ e_2 \\
&\quad \mid \texttt{letrec}\ \mathit{fdefs}\ \texttt{in}\ e \mid \texttt{try}\ e_1\ \texttt{of}\ <x_1, \dots, x_k> \to e_2\ \texttt{catch}\ <x_{k+1}, \dots, x_{k+n}> \to e_3 \\
e \in \mathit{Exp} &::= \mathit{nv} \mid v \\
\mathit{vs} \in \mathit{ValSeq} &::= <v_1, \dots, v_n> \\
c \in \mathit{ExcClass} &::= \texttt{'throw'} \mid \texttt{'exit'} \mid \texttt{'error'} \\
\mathit{exc} \in \mathit{Exception} &:= \{c, v^r, v^d\}^X \\
\mathit{res} \in \mathit{Result} &:= \mathit{exc} \mid \mathit{vs}
\end{aligned}
$$

Figure 1. Syntax of Core Erlang.

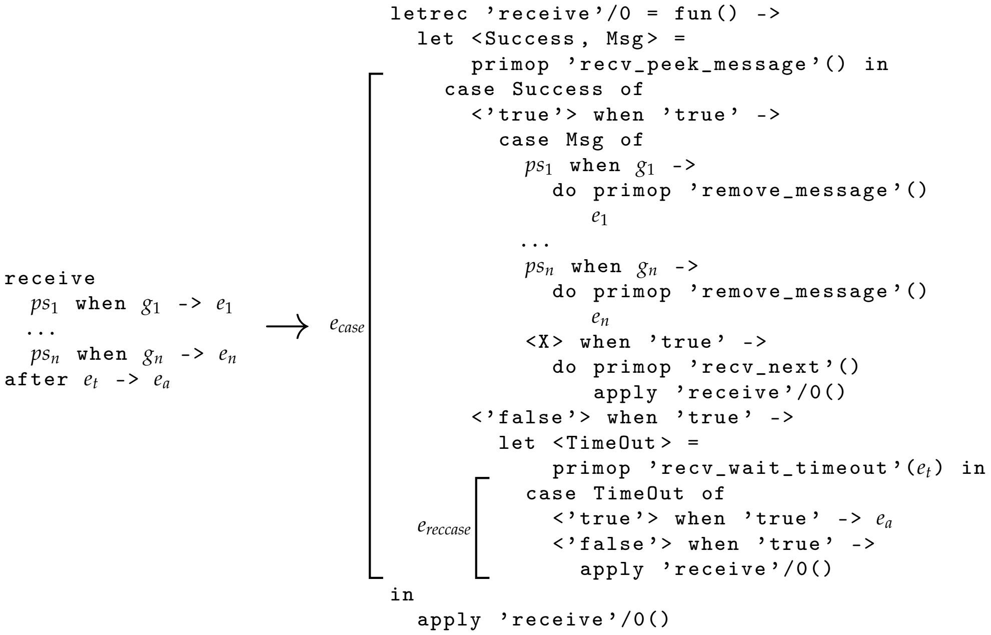
We use i to range over integers, a, f over atoms, and k over natural numbers. We use superscripts to denote the roles of expressions. Furthermore, x ranges over variables and ι over process identifiers. Integers (denoted with numbers), atoms (enclosed in single quotation marks), variables, empty and non-empty lists, tuples, and maps (tilde-enclosed tuples of key-value pairs, denoted with superscripts) form the patterns of the language. The set of values essentially consists of the same elements with the addition of function identifiers (f/k atom-arity pairs), process identifiers (PIDs), and function closures (which include a list of functions *fdefs* that can be applied recursively in the closure's body).

In Core Erlang, the result of the evaluation is either a value sequence (denoted with $<v_1, \dots, v_n>$ or vs) or an exception. While most expressions of the language evaluate to singleton value sequences, one can use value list expressions ($<e_1, \dots, e_n>$) in binding expressions (such as `case`, `let`, `letrec`, or `try`) to bind multiple names simultaneously. Pattern-matching is expressed with `case` expressions; each case clause consists of a list of patterns to be matched, a guard, and a body expression (denoted with superscripts).

The set of expressions of Core Erlang also contains the values, lists, tuples, and maps containing expressions, sequencing (`do`), uncurried function abstraction (`fun`) and application (`apply`), inter-module function calls (`call`), and primitive operations (`primop`). Currently, the module system is not formalised; thus, inter-module calls only express standard and built-in functions (BIFs) [28] and their semantics is simulated in the metatheory.

Exceptions in Erlang implementations are represented by a triple: an exception class (which is an atom), and two values describing the reason of the exception and additional details about the exception (these are denoted with superscripts). These three values are bound in a `catch` clause of a `try` expression. For further details, see the language specification [6].

Compared to our previous work [11], the only syntactical addition is the syntax for PIDs; our previous results on the sequential sublanguage [11] still hold for this extension. The other concurrent features of the language (e.g., message sending and receiving) are expressed with BIF calls and primitive operations. As of Erlang/OTP 23 [19], `receive` expressions are defined as syntactic sugar on top of primitive operations (Figure 2), which are described in Section 3.4.



```
receive
  ps1 when g1 -> e1
  ...
  psn when gn -> en
after et -> ea
```

```
letrec 'receive'/0 = fun() ->
  let <Success, Msg> =
      primop 'recv_peek_message'() in
    case Success of
      <'true'> when 'true' ->
        case Msg of
          ps1 when g1 ->
            do primop 'remove_message'()
               e1
          ...
          psn when gn ->
            do primop 'remove_message'()
               en
          <X> when 'true' ->
            do primop 'recv_next'()
               apply 'receive'/0()
      <'false'> when 'true' ->
        let <TimeOut> =
            primop 'recv_wait_timeout'(et) in
          case TimeOut of
            <'true'> when 'true' -> ea
            <'false'> when 'true' ->
              apply 'receive'/0()
in
  apply 'receive'/0()
```

Figure 2. `Receive` expression in Core Erlang since Erlang/OTP 23 [19]. Note that the actual variable names (and the function identifier for `letrec`) used in the translated version are always generated based on the expressions and patterns present in the original version to avoid name clashes. Also, note that the labelled code segments are intended to ease understanding of Example 1 later, and they are not relevant at this point.

3.2. Running Example

As an example, we show that sequential and concurrent maps over lists behave in the same way: the expression in Figure 3 computes this mapping sequentially, while the code in Figure 4 splits the list into two parts, computes the transformation concurrently, then appends the two sub-lists. Hence, the reason both expressions send the transformed list to a particular PID ι as the final step of their evaluation: in Example 2 we reason about the equivalence of these expressions based on their observable behaviour, i.e., the outward communication they carry out.

```
     letrec 'map'/2 = fun(F,L) ->
         case <L> of
cl1        <[]> when 'true' -> []
cl2        <[H|T]> when 'true' ->
              [apply F(H) | apply 'map'/2(F,T)]
       in
           call 'erlang':'!'(
             ι, apply 'map'/2(e_f, e_l)
           )
```

Figure 3. Sequential definition for list mapping for any PID ι and expressions e_f, e_l.

```
        case call 'lists':'split'(i, e_l) of
          <{L1, L2}> when 'true' ->
            let <S> = call 'erlang':'self'() in
              do call 'erlang':'spawn'(
                      fun(S, L) ->
c_child                 call 'erlang':'!'(S,
e_pmap                     apply c_map(e_f,L)),
                           [S|[L1|[]]])
                 let <M2> = apply c_map(e_f,L2) in
                   receive
                     <M1> when 'true' ->
e_let                  call 'erlang':'!'(
                          ι, call 'erlang':'++'(M1, M2)
                       )
                   after 'infinity' -> []
```

Figure 4. Concurrent definition for list mapping for any PID ι, non-negative integer i, and expressions e_f, e_l.

Note that we use c_{map} to denote the closure of the sequential list transforming function (inside `letrec`) in Figure 3. Also note that the labelled code segments are intended to ease understanding of Example 1 later, and they are not relevant at this point.

3.3. Sequential Semantics

For the sequential semantics, we reuse the frame stack semantics from our previous work [11] (also described in Appendix A) and we build the process-local and inter-process semantics on top of it, based on the prototype described in [10]. A sequential configuration consists of a frame stack and a redex. The syntax of redexes, frame stacks, and frames is described in Figure 5. Essentially, a frame is a compound expression with a hole in place of one of its subexpressions.

$$\begin{aligned}
r \in \mathit{Redex} &::= vs \mid exc \mid e \mid \square \\
id \in \mathit{FrameId} &::= tuple \mid values \mid map \mid call(v^m, v^f) \mid primop(a) \mid app(v^f) \\
F \in \mathit{Frame} &::= id(v_1, \ldots, v_{i-1}, \square, e_{i+1} \ldots, e_n) \mid [e_1|\square] \mid [\square|v_2] \\
&\mid \texttt{call}\ \square{:}e^f(e_1, \ldots, e_n) \mid \texttt{call}\ v^m{:}\square(e_1, \ldots, e_n) \mid \texttt{apply}\ \square(e_1, \ldots, e_n) \\
&\mid \texttt{case}\ \square\ \texttt{of}\ cl_1; \ldots; cl_n\ \texttt{end} \mid \texttt{let}\ {<}x_1, \ldots, x_n{>} = \square\ \texttt{in}\ e_2 \\
&\mid \texttt{case}\ vs\ \texttt{of}\ ps\ \texttt{when}\ \square \to e^b; cl_2; \ldots; cl_n\ \texttt{end} \mid \texttt{do}\ \square\ e_2 \\
&\mid \texttt{try}\ \square\ \texttt{of}\ {<}x_1, \ldots, x_n{>} \to e_2\ \texttt{catch}\ {<}x_{k+1}, x_{k+2}, x_{k+3}{>} \to e_3 \\
K \in \mathit{FrameStack} &::= \varepsilon \mid F :: K
\end{aligned}$$

Figure 5. Syntax of redexes, frames, frame stacks.

The frames for language elements, that include a list of subexpressions (i.e., tuples, value lists, maps, inter-module calls, primitive operations, and function applications) are expressed with parameter list frames $id(v_1, \ldots, v_{i-1}, \square, e_{i+1}, \ldots, e_n)$. The frame identifier id

determines which language element the frame corresponds to. This way, the evaluation of a list of parameter expressions is expressed uniformly, without repeating the rules for each expression type.

The sequential reductions are denoted with $\langle K, r\rangle \longrightarrow \langle K', r'\rangle$. In Figure 6, we only recall the evaluation rules for inter-module calls and primitive operations from our previous work. The complete definition can be found in [11] and Appendix A.

$$\langle K, \texttt{call}\ e^m{:}e^f(e_1, \dots, e_n)\rangle \longrightarrow \langle \texttt{call}\ \Box{:}e^f(e_1, \dots, e_n) :: K, e^m\rangle \qquad \text{(SCallMod)}$$

$$\langle K, \texttt{primop}\ a(e_1, \dots, e_n)\rangle \longrightarrow \langle \mathit{primop}(a)(\Box, e_1, \dots, e_n) :: K, \Box\rangle \qquad \text{(SPrimOp)}$$

$$\langle \texttt{call}\ \Box{:}e^f(e_1, \dots, e_n) :: K, {<}v^m{>}\rangle \longrightarrow \langle \texttt{call}\ v^m{:}\Box(e_1, \dots, e_n) :: K, e^f\rangle \qquad \text{(SCallFun)}$$

$$\langle \texttt{call}\ v^m{:}\Box(e_1, \dots, e_n) :: K, {<}v^f{>}\rangle \longrightarrow \langle \mathit{call}(v^m, v^f)(\Box, e_1, \dots, e_n) :: K, \Box\rangle \qquad \text{(SCallParam)}$$

$$\langle \mathit{id}(\Box, e_1, e_2, \dots, e_n) :: K, \Box\rangle \longrightarrow \langle \mathit{id}(\Box, e_2, \dots, e_n) :: K, e_1\rangle \quad (\text{if } \mathit{id} \neq \mathit{map}) \qquad \text{(SParams}_0\text{)}$$

$$\begin{aligned}&\langle \mathit{id}(v_1, \dots, v_{i-1}, \Box, e_{i+1}, e_{i+2}, \dots, e_n) :: K, {<}v_i{>}\rangle \longrightarrow \\ &\qquad \langle \mathit{id}(v_1, \dots, v_{i-1}, v_i, \Box, e_{i+2}, \dots, e_n) :: K, e_{i+1}\rangle \qquad \text{(SParams)}\end{aligned}$$

$$\langle \mathit{id}(v_1, \dots, v_n, \Box) :: K, \Box\rangle \longrightarrow \langle K, \mathit{eval}(\mathit{id}, v_1, \dots, v_n)\rangle \quad (\text{if } \mathit{id} \neq \mathit{map}) \qquad \text{(PParams}_0\text{)}$$

$$\langle \mathit{id}(v_1, \dots, v_{n-1}, \Box) :: K, {<}v_n{>}\rangle \longrightarrow \langle K, \mathit{eval}(\mathit{id}, v_1, \dots, v_n)\rangle \qquad \text{(PParams)}$$

Figure 6. Frame stack semantics rules for `calls` and `primops`.

3.4. Process-Local Semantics

The process-local semantics describes the behaviour of a single process in response to an action. First, we define Core Erlang processes.

Definition 1 (Core Erlang processes). *A process (denoted with $p \in \mathit{Process}$) is either dead or alive.*

- *A live process is a quintuple (K, r, q, L, b), where K denotes its frame stack, r is the redex currently evaluated, and q is the mailbox. L is the set of linked process identifiers, and b is a metatheoretical boolean value denoting the status of the `trap_exit` flag.*
- *A terminated (or dead) process (denoted with T) is a finite map of (linked) process identifiers to values.*

Compared to related studies [10,12,16], we do not formalise the mailbox of a process as a single list of values but, rather, split it into seen and unseen messages (denoted with $[v_1, \dots, v_m \blacktriangleright v_{m+1}, \dots, v_n]$ where v_{m+1} is the first unseen message). This change was needed to correctly express the meaning of the primitive operations of message receipts [19], while it also narrows gap between the formalisation and the standard Erlang/OTP compiler. We formally define the mailbox operations on Figure 7. Incoming messages are placed at the end of the unseen message list. We can check the first unseen message (and its existence) in a mailbox. The first unseen message can be placed into the seen section of the mailbox. When a message is received, it is removed from the mailbox, and the remaining elements should all be scanned again for the next message receipt (i.e., they all become unseen). With this formalism, the entire mailbox can be pictured as follows:

$$[v_1, \dots, v_m] \mathbin{+\!\!+} [v_{m+1}, \dots, v_n]$$

As mentioned before, the concurrent sublanguage of (Core) Erlang is based on the actor model [8]. Processes of (Core) Erlang communicate with asynchronous message-passing and signals; in fact, messages are just one particular kind of signal. In this paper, we consider four signal types, while there are several others [9].

$$
\begin{aligned}
&\mathit{push}([v_1, \ldots, v_m \blacktriangleright v_{m+1}, \ldots, v_n], v) \stackrel{\text{def}}{=} [v_1, \ldots, v_m \blacktriangleright v_{m+1}, \ldots, v_n, v] \\
&\mathit{hasNew}([v_1, \ldots, v_m \blacktriangleright v_{m+1}, \ldots, v_n]) \stackrel{\text{def}}{=} n > m \\
&\mathit{peek}([v_1, \ldots, v_m \blacktriangleright v_{m+1}, \ldots, v_n]) \stackrel{\text{def}}{=} \begin{cases} \mathit{Some}(v_{m+1}) & \mathit{if}\ n > m \\ \mathit{None} & \mathit{otherwise} \end{cases} \\
&\mathit{recvNext}([v_1, \ldots, v_m \blacktriangleright v_{m+1}, \ldots, v_n]) \stackrel{\text{def}}{=} \begin{cases} \mathit{Some}([v_1, \ldots, v_m, v_{m+1} \blacktriangleright v_{m+2}, \ldots, v_n]) & \mathit{if}\ n > m \\ \mathit{None} & \mathit{otherwise} \end{cases} \\
&\mathit{removeMsg}([v_1, \ldots, v_m \blacktriangleright v_{m+1}, \ldots, v_n]) \stackrel{\text{def}}{=} \begin{cases} \mathit{Some}([\blacktriangleright v_1, \ldots, v_m, v_{m+2}, \ldots, v_n]) & \mathit{if}\ n > m \\ \mathit{None} & \mathit{otherwise} \end{cases}
\end{aligned}
$$

Figure 7. Mailbox operations.

Definition 2 (Signals)**.**

$$s \in \mathit{Signal} \ ::= \mathit{msg}(v) \mid \mathit{exit}(v, b) \mid \mathit{link} \mid \mathit{unlink}$$

- *Messages* are values that are sent from one process and placed into the mailbox of another process.
- *Link* signals communicate that two processes should be linked. Links are bidirectional; when one of the processes terminates, it will notify the other process with an *exit* signal.
- *Unlink* signals indicate that the link between two processes should be removed.
- *Exit* signals are used to indicate runtime errors. Terminated processes send them to their links, but they can be created and sent manually (with the `'exit'/2` BIF) too. *Exit* signals include a value describing the reason of termination and a boolean flag of whether they have been triggered by a link.

Actions represent the effects that characterise concurrency; they unambiguously define the next reduction a process should take, while they also include the necessary data from the context to make this step (e.g., in the case of message sending, these data consist of PID of the sender, PID of the receiver, and the message value). The syntax of signals and actions are shown below.

Definition 3 (Actions)**.**

$$a \in \mathit{Action} ::= \mathit{send}(\iota^s, \iota^d, s) \mid \mathit{arr}(\iota^s, \iota^d, s) \mid \mathit{self}(\iota) \mid \mathit{spawn}(\iota, v_1, v_2, b) \mid \tau \mid \varepsilon$$

We call τ and $\mathit{self}(\iota)$ silent actions. We use ι^s to emphasise that this PID is a source of a signal, while ι^d denotes target (destination) PIDs.

- Signals can be sent, and they can arrive at processes. These actions include the sender's (ι^s) and receiver's (ι^d) PIDs. Note that signal-passing is not instantaneous; signals reside in "the ether" before their arrival. This behaviour is expressed with the inter-process semantics (we refer to Section 3.5).
- A process can obtain its PID with $\mathit{self}(\iota)$ from the inter-process semantics.
- A process can spawn another process to evaluate a function with $\mathit{spawn}(\iota, v_1, v_2, b)$. ι is the PID of the spawned process given by the inter-process semantics, v_1 should be a closure value, and v_2 should be a (Core) Erlang list of parameters. The flag b denotes whether the spawned process should be linked to its parent (when evaluating `spawn_link`).
- We use τ actions to denote reductions of the computational (sequential) layer and some process-local steps for which the system is strongly confluent (formally defined in Theorem 4).

- The local reduction steps are denoted with ε actions. These steps affect either the mailbox, the set of links, or the process flag (besides the frame stack and the redex) of a process, and they are not confluent with the other actions.

Before delving into discussing the rules of the process-local semantics, we introduce the following notations for readability.

- $\lambda x \Rightarrow b$ is used to denote functions of the metatheory.
- ff denotes metatheoretical false, and tt denotes metatheoretical true.
- $\mathit{map}(f, l)$ is a higher-order function of the metatheory, transforming all elements of the (metatheoretical) container l (list or set) by applying the (metatheoretical) function f to them.
- $\mathit{to\_obj}(y)$ is used to transform the metatheoretical (list or boolean) value y to a Core Erlang value (a list or `'true'` or `'false'` atoms).
- $\mathit{to\_meta}(v)$ is used to transform the Core Erlang value v to its metatheoretical counterpart. The result of this function is of option type; it is *None* for unsuccessful conversion, or the metatheoretical counterpart enclosed in *Some*.
- $M[k]$ denotes the value enclosed in *Some* associated with the key k in a finite map M, if it exists. Otherwise, the result is *None*.
- $M \setminus k$ denotes removing the value associated with the key k from the finite map M.

Next, we briefly discuss the rules of the process-local semantics, categorised into 4 groups:

1. Semantics of signal arrival (Figure 8);
2. Semantics of signal sending (Figure 9);
3. Semantics of *spawn* and *self* (Figure 10);
4. Semantics of local actions (Figure 11).

$$(K, r, q, L, b) \xrightarrow{\mathit{arr}(\iota^s, \iota^d, \mathit{msg}(v))} (K, r, \mathit{push}(q, v), L, b) \qquad (\textsc{Msg})$$

$$\frac{\iota^s \neq \iota^d \land ((b = \mathit{ff} \land v = \texttt{'normal'}) \lor (\iota^s \notin L \land b_e = \mathit{tt}))}{(K, r, q, L, b) \xrightarrow{\mathit{arr}(\iota^s, \iota^d, \mathit{exit}(v, b_e))} (K, r, q, L, b)} \qquad (\textsc{ExitDrop})$$

$$\frac{\begin{gathered}(v = \texttt{'kill'} \land b_e = \mathit{ff} \land v' = \texttt{'killed'}) \lor \\ (b = \mathit{ff} \land v = \texttt{'normal'} = v' \land \iota^s = \iota^d) \lor \\ (b = \mathit{ff} \land v \neq \texttt{'normal'} \land v' = v \ \land (b_e = \mathit{tt} \to \iota^s \in L) \land (b_e = \mathit{ff} \to v \neq \texttt{'kill'}))\end{gathered}}{(K, r, q, L, b) \xrightarrow{\mathit{arr}(\iota^s, \iota^d, \mathit{exit}(v, b_e))} \mathit{map}(\lambda \iota \Rightarrow (\iota, v'), L)} \qquad (\textsc{ExitTerm})$$

$$\frac{b = \mathit{tt} \land ((b_e = \mathit{ff} \land v \neq \texttt{'kill'}) \lor (b_e = \mathit{tt} \land \iota^s \in L))}{(K, r, q, L, b) \xrightarrow{\mathit{arr}(\iota^s, \iota^d, \mathit{exit}(v, b_e))} (K, r, \mathit{push}(q, \texttt{\{'EXIT',}\iota^s\texttt{,}v\texttt{\}}), L, b)} \qquad (\textsc{ExitTrap})$$

$$(K, r, q, L, b) \xrightarrow{\mathit{arr}(\iota^s, \iota^d, \mathit{link})} (K, r, q, \{\iota^s\} \cup L, b) \qquad (\textsc{LinkArr})$$

$$(K, r, q, L, b) \xrightarrow{\mathit{arr}(\iota^s, \iota^d, \mathit{unlink})} (K, r, q, L \setminus \{\iota^s\}, b) \qquad (\textsc{UnlinkArr})$$

Figure 8. Semantics of signal arrival (group 1).

We start the explanation with the rules about signal arrival. We note that the Erlang reference manual [9] does not express the behaviour of signal arrival precisely (although, there are no inconsistencies). Thus we determined the sufficient premises of the following rules by testing the reference implementation (also reported in [10]).

- Rule Msg shows that incoming messages are appended to the list of unseen messages in the mailbox.
- Rules ExitDrop, ExitTerm, and ExitTrap describe the arrival of an *exit* signal. Based on the set of links L, the state of the `trap_exit` flag b, the reason value v, and

the link flag b_e of the *exit* signal, and the source ι^s and destination ι^d PIDs, there are three different behaviours.

  - Rule ExitDrop drops the *exit* signal if its reason was `'normal'` or it came from an unlinked process and was triggered by a link.
  - Rule ExitTerm terminates the process, by transforming it into a dead process which will notify its links with the reason of the termination (each linked PID is associated with this reason value). This rule can be applied in three scenarios: (a) the *exit*'s reason is `'kill'` and it was sent manually (in this case, the reason is changed to `'killed'`); (b) *exits* are not trapped, and a non-`'normal'` *exit* either came from a linked process or was sent manually with a non-`'kill'` reason; (c) *exits* are not trapped, and the process terminated itself with an *exit* signal with `'normal'` reason.
  - Rule ExitTrap "traps" the *exit* signal by converting it into a message that is appended to the mailbox. This rule can be used if the process traps *exits*, the incoming signal is either triggered by a link, or its reason is not `'kill'`.

- Rule LinkArr and UnlinkArr describe the arrival of *link* and *unlink* signals, which modify the set of the linked PIDs.

$$(call(\texttt{'erlang'},\texttt{'!'})(\iota^d,\Box) :: K, {<}v{>}, q, L, b) \xrightarrow{send(\iota^s,\iota^d,msg(v))} (K, {<}v{>}, q, L, b) \qquad \text{(Send)}$$

$$(call(\texttt{'erlang'},\texttt{'exit'})(\iota^d,\Box) :: K, {<}v{>}, q, L, b) \xrightarrow{send(\iota^s,\iota^d,exit(v,\mathit{ff}))} (K, {<}\texttt{'true'}{>}, q, L, b) \qquad \text{(Exit)}$$

$$(call(\texttt{'erlang'},\texttt{'link'})(\Box) :: K, {<}\iota^d{>}, q, L, b) \xrightarrow{send(\iota^s,\iota^d,link)} (K, {<}\texttt{'ok'}{>}, q, \{\iota^d\} \cup L, b) \qquad \text{(Link)}$$

$$(call(\texttt{'erlang'},\texttt{'unlink'})(\Box) :: K, {<}\iota^d{>}, q, L, b) \xrightarrow{send(\iota^s,\iota^d,unlink)} (K, {<}\texttt{'ok'}{>}, q, L \setminus \{\iota^d\}, b) \qquad \text{(Unlink)}$$

$$\frac{T[\iota^d] = Some(v)}{T \xrightarrow{send(\iota^s,\iota^d,exit(v,tt))} T \setminus \{\iota^d\}} \qquad \text{(Dead)}$$

Figure 9. Semantics of signal sending (group 2).

Next, we discuss rules about signal sending (Figure 9). Note that all the first frames in the following rules use frame identifiers from Figure 5 inside a parameter list frame $id(v_1, \ldots, v_{i-1}, \Box)$ which includes the normal forms of its subexpressions.

- Rule Send describes message sending. This BIF reduces to the message value which is also communicated to the inter-process semantics inside the *send* action.
- Rule Exit describes manual *exit* signal sending. In this case, the result is `'true'` while the *exit* signal is communicated to the inter-process semantics. The link flag of the signal is *ff*, since it is sent manually.
- Rules Link and Unlink describe sending *link* and *unlink* signals. Both evaluate to `'ok'` while adding or removing a PID from the set of links, and communicating the corresponding action to the inter-process level.
- Rule Dead describes the communication of a dead process. Dead processes send an *exit* signal to all the linked processes with the given reason value. The flag of the signal is *tt*, since it is triggered by a link.

  Thereafter, we describe how *self* and *spawn* BIFs evaluate.

- Rule Self reduces to the PID of the current process which is obtained from the inter-process level in the action *self*.
- Rules Spawn and SpawnLink describe process spawning. The spawned process will evaluate the given function applied to the given parameters, which are communicated

to it within the *spawn* action. To be able to evaluate this function application, the parameters of *spawn* are required to be a closure and a proper Core Erlang list. The result is the PID of the spawned process which is obtained from the inter-process semantics in the *spawn* action. In the case of rule SpawnLink, the flag in the *spawn* action is also set, and a link is established to the spawned PID.

$$(\mathit{call}(\texttt{'erlang'},\texttt{'self'})(\Box) :: K, \Box, q, L, b) \xrightarrow{\mathit{self}(\iota)} (K, <\iota>, q, L, b) \qquad (\textsc{Self})$$

$$\frac{v^l = [v_1|\ldots[v_k|\texttt{[]}]\ldots] \qquad v^f = \mathit{clos}(\mathit{fdefs}, [x_1, \ldots, x_k], e)}{(\mathit{call}(\texttt{'erlang'},\texttt{'spawn'})(v^f, \Box) :: K, <v^l>, q, L, b) \xrightarrow{\mathit{spawn}(\iota, v^f, v^l, \mathit{ff})} (K, <\iota>, q, L, b)} \qquad (\textsc{Spawn})$$

$$\frac{v^l = [v_1|\ldots[v_k|\texttt{[]}]\ldots] \qquad v^f = \mathit{clos}(\mathit{fdefs}, [x_1, \ldots, x_k], e)}{\begin{array}{c}(\mathit{call}(\texttt{'erlang'},\texttt{'spawn\_link'})(v^f, \Box) :: K, <v^l>, q, L, b) \\ \xrightarrow{\mathit{spawn}(\iota, v^f, v^l, \mathit{tt})} (K, <\iota>, q, \{\iota\} \cup L, b)\end{array}} \qquad (\textsc{SpawnLink})$$

Figure 10. Semantics of *spawn* and *self* (group 3).

$$\frac{\langle K, r\rangle \rightarrow \langle K', r'\rangle}{(K, r, q, L, b) \xrightarrow{\tau} (K', r', q, L, b)} \qquad (\textsc{Seq})$$

$$\frac{\mathit{to\_meta}(v) = \mathit{Some}(b') \qquad v' = \mathit{to\_obj}(b)}{(\mathit{call}(\texttt{'process\_flag'},\texttt{'trap\_exit'})(\Box) :: K, <v>, q, L, b) \xrightarrow{\varepsilon} (K, <v'>, q, L, b')} \qquad (\textsc{Flag})$$

$$\frac{\mathit{to\_meta}(v) \neq \mathit{Some}(b')}{\begin{array}{c}(\mathit{call}(\texttt{'process\_flag'},\texttt{'trap\_exit'})(\Box) :: K, <v>, q, L, b) \\ \xrightarrow{\varepsilon} (K, \{\texttt{'error'},\texttt{'badarg'}, v\}^X, q, L, b)\end{array}} \qquad (\textsc{FlagExc})$$

$$(\varepsilon, v, q, L, b) \xrightarrow{\varepsilon} \mathit{map}(\lambda\iota \Rightarrow (\iota, \texttt{'normal'}), L) \qquad (\textsc{Term})$$

$$(\varepsilon, \{c, v^r, v^d\}^X, q, L, b) \xrightarrow{\varepsilon} \mathit{map}(\lambda\iota \Rightarrow (\iota, v^r), L) \qquad (\textsc{TermExc})$$

$$\frac{\mathit{peek}(q) = \mathit{Some}(v)}{(\mathit{primop}(\texttt{'recv\_peek\_message'})(\Box) :: K, \Box, q, L, b) \xrightarrow{\tau} (K, <\texttt{'true'}, v>, q, L, b)} \qquad (\textsc{Peek})$$

$$\frac{\mathit{peek}(q) = \mathit{None}}{(\mathit{primop}(\texttt{'recv\_peek\_message'})(\Box) :: K, \Box, q, L, b) \xrightarrow{\varepsilon} (K, <\texttt{'false'},\texttt{'error'}>, q, L, b)} \qquad (\textsc{PeekFail})$$

$$\frac{\mathit{recvNext}(q) = \mathit{Some}(q')}{(\mathit{primop}(\texttt{'recv\_next'})(\Box) :: K, \Box, q, L, b) \xrightarrow{\tau} (K, <\texttt{'ok'}>, q', L, b)} \qquad (\textsc{RecvNext})$$

$$\frac{\mathit{removeMsg}(q) = \mathit{Some}(q')}{(\mathit{primop}(\texttt{'remove\_message'})(\Box) :: K, \Box, q, L, b) \xrightarrow{\tau} (K, <\texttt{'ok'}>, q', L, b)} \qquad (\textsc{RemoveMsg})$$

$$\frac{\mathit{hasNew}(q)}{(\mathit{primop}(\texttt{'recv\_wait\_timeout'})(\Box) :: K, <\texttt{'infinity'}>, q, L, b) \xrightarrow{\tau} (K, <\texttt{'false'}>, q, L, b)} \qquad (\textsc{WaitInf})$$

$$(\mathit{primop}(\texttt{'recv\_wait\_timeout'})(\Box) :: K, <\texttt{0}>, q, L, b) \xrightarrow{\tau} (K, <\texttt{'true'}>, q, L, b) \qquad (\textsc{Wait0})$$

$$\frac{v \neq \texttt{0} \qquad v \neq \texttt{'infinity'}}{\begin{array}{c}(\mathit{primop}(\texttt{'recv\_wait\_timeout'})(\Box) :: K, <v>, q, L, b) \\ \xrightarrow{\tau} (K, \{\texttt{'error'},\texttt{'timeout\_value'}, v\}^X, q, L, b)\end{array}} \qquad (\textsc{WaitExc})$$

Figure 11. Semantics of local actions (group 4).

Finally, we describe rules about process-local steps (Figure 11). Unlike in previous work [10,12,16], we express rules about message receipts with primitive operations; since `receive` expressions have been removed from the primitives of Core Erlang in OTP 23 [19], they are automatically expanded to the primitive operations.

- Rule SEQ lifts the sequential steps to the process-local level.
- Rule FLAG changes the state of the `trap_exit` flag of the process to the provided boolean value, and its result is the original value of the flag. Rule FLAGEXC raises an exception, if a non-boolean value was provided.
- Rule TERM describes the normal termination. The links are notified with *exit* signals with `‘normal’` reason.
- Rule TERMEXC describes the behaviour when an exception terminates the process. Each linked process is notified with an *exit* signal which includes the exception's reason.
- Rule PEEK checks the first unseen message (if it exists). The result is a value sequence consisting of `‘true’` and the said message.
- Rule PEEKFAIL is used if there are no unseen messages in the mailbox. The result is a value sequence consisting of `‘false’` and an error value (which can be neither observed in the implementations, nor in the generated BEAM code).
- Rule RECVNEXT moves the first unseen message into the list of seen messages (if it exists).
- Rule REMOVEMSG removes the first unseen message from the mailbox and sets all messages as unseen hereafter.
- Rule WAITINF describes the semantics of unblocking. This rule is used if there is an unseen message, otherwise the process is blocked until one arrives. The result `‘false’` indicates that the timeout (`‘infinity’`) was not reached.
- Rule WAIT0 always evaluates to `‘true’`, indicating that a timeout was reached. Currently, we have not formalised the timing; thus, only `0` and `‘infinity’` timeouts are modelled; however, we plan to change this in the future.
- Rule WAITEXC evaluates to an exception when an invalid timeout value (i.e., non-integer and non-infinity) was used.

3.5. Inter-Process Semantics

After having the process-local semantics defined, we turn our attention to the inter-process (node-level) semantics, which defines how actions should be propagated among processes. The main advantage of this semantics is its simplicity—it is described with only four rules. First, we define the notion of process pools.

Definition 4 (Process pool)**.** *A (process) pool Π is a finite map (associative list) which assigns PIDs to processes.*

In (Core) Erlang, signal passing is not atomic. According to the reference manual [9] “The amount of time that passes between the time a signal is sent and the arrival of the signal at the destination is unspecified but positive”; thus, we need to store these signals in an ether until they arrive. Moreover, this ether should respect the signal-ordering, i.e., “if an entity sends multiple signals to the same destination entity, the order is preserved” [9].

Definition 5 (Ether)**.** *An ether is a finite map (associative list) which assigns a pair of PIDs (representing the source and destination of signals) to a list of signals. We denote ethers with Δ.*

Definition 6 (Node)**.** *A Core Erlang node N is a pair of a process pool and an ether. We use N^{Π} and N^{Δ} for the pool and ether of N.*

Before discussing the four rules of the inter-process semantics, we introduce a number of notations for ethers and process pools.

- $\iota : p \parallel \Pi$ denotes the pool consisting of process p (associated with PID ι) and pool Π.
- $\Delta[(\iota^s, \iota^d) \mapsto l]$ updates the ether Δ by binding the pair of PIDs (ι^s, ι^d) to the list of signals l.
- $\Delta[(\iota^s, \iota^d) \mapsto^+ s]$ appends a signal s to the end of the list associated with (ι^s, ι^d) in the ether Δ. If there is nothing associated with the given source and destination, this function creates a singleton list associated with the given PIDs.
- $\mathit{remFirst}(\Delta, \iota^s, \iota^d)$ removes the first signal from the list associated with (ι^s, ι^d) from Δ. Its result is of option type, if there is a signal to remove, the result is this signal and the modified ether enclosed in *Some*, otherwise *None*.
- $\mathit{PIDsOf}(\Delta)$ and $\mathit{PIDsOf}(\Pi)$ denote the PIDs that appear in Δ or Π, respectively. A PID appears in an ether if it is used as a source or destination of a signal, or if it appears syntactically in one of the signals stored in the ether. Respectively, a PID appears in a pool if there is a process associated with it, or it appears inside a process syntactically.
- $\mathit{eval}(v^f, v_1, \ldots, v_k)$ notation is taken from [11]; here, we only use it to express beta-reduction of v_f with the parameters $v_1, \ldots, v_k$. For more details, refer to Appendix A and [11].

Next, we discuss the semantics rules (shown in Figure 12). In general, all rules propagate an action to one of the processes inside the process pool, potentially changing the ether or creating new processes.

$$\frac{p \xrightarrow{\mathit{send}(\iota^s,\iota^d,s)} p'}{(\Delta, \iota^s : p \parallel \Pi) \xrightarrow{\iota^s:\mathit{send}(\iota^s,\iota^d,s)}_O (\Delta[(\iota^s, \iota^d) \mapsto^+ s], \iota^s : p' \parallel \Pi)} \quad (\text{NSend})$$

$$\frac{p \xrightarrow{\mathit{arr}(\iota^s,\iota^d,s)} p' \qquad \mathit{remFirst}(\Delta, \iota^s, \iota^d) = \mathit{Some}(s, \Delta')}{(\Delta, \iota^d : p \parallel \Pi) \xrightarrow{\iota^d:\mathit{arr}(\iota^s,\iota^d,s)}_O (\Delta', \iota^d : p' \parallel \Pi)} \quad (\text{NArrive})$$

$$\frac{\begin{array}{c} p \xrightarrow{\mathit{spawn}(\iota_2,v^f,v^l,b)} p' \\ L = \mathit{if}\ b\ \mathit{then}\ \{\iota_1\}\ \mathit{else}\ \emptyset \\ \iota_2 \notin O \cup \mathit{PIDsOf}(\Delta) \cup \mathit{PIDsOf}(\iota_1 : p \parallel \Pi) \end{array}}{(\Delta, \iota_1 : p \parallel \Pi) \xrightarrow{\iota_1:\mathit{spawn}(\iota_2,v^f,[v_1|\ldots[v_k|\texttt{[]}]\ldots],b)}_O (\Delta, \iota_2 : ([], \mathit{eval}(v^f, v_1, \ldots, v_k), [\blacktriangleright], L, \mathit{ff}) \parallel \iota_1 : p' \parallel \Pi)} \quad (\text{NSpawn})$$

$$\frac{p \xrightarrow{a} p' \qquad a \in \{\mathit{self}(\iota), \varepsilon, \tau\}}{(\Delta, \iota : p \parallel \Pi) \xrightarrow{\iota:a}_O (\Delta, \iota : p' \parallel \Pi)} \quad (\text{NLocal})$$

Figure 12. Formal semantics of communication between processes.

Remark 1. *The rules of the inter-process semantics are decorated with a set of PIDs O; these PIDs are considered as observed, and no processes can be spawned on them. The communication to the PIDs in O is used to express the observable behaviour in bisimulation definitions in Section 4.*

- Rule NSend describes signal sending. When a process (associated with ι^s) sends a signal to ι^d, this signal is placed into the ether with the source ι^s and destination ι^d.
- Rule NArrive removes the first signal from the ether (with some source ι^s) to deliver to the process with PID ι^d.
- Rule NSpawn describes process creation. When a process reduces with a *spawn* action, the inter-process semantics assigns a fresh and unobserved PID ι_2 to the new process which will evaluate the given closure applied to the given parameters of the *spawn* action. If the flag in the *spawn* action is set (i.e., `spawn_link` has been evaluated in the process), a link to the parent process is also established. Whether v^f is a closure and

the third parameter of *spawn* is a proper Core Erlang list of values are checked in the process-local semantics (in rules SPAWN and SPAWNLINK).

- Rule NLOCAL defines that every other action should be only propagated to the process-local level without modifying the ether, or creating new processes.

We use $N \xrightarrow{l}{}_{O}^{*} N'$ to denote the reflexive transitive closure of the semantics, where l is the evaluation trace of PID-action pairs. We write $N \xrightarrow{a}{}_{O}^{*} N'$ if the trace only consists of actions a (and its length is irrelevant), and omit the trace entirely if it is not relevant.

3.6. Example Evaluation

In this section, we show example evaluations for our running example presented in Figure 4, where we instantiate the metavariables in the following way (we keep ι as arbitrary):

$$e_l := \texttt{[1|[2|[3|[4|[]]]]]} \qquad i := 2$$
$$e_f := \texttt{fun(X)} \to \texttt{call 'erlang':'+'(X,1)}$$

We remind the reader that the closure of the sequential list processing (c_{map} evaluated as the result closure of the `letrec` expression in Figure 3) is already substituted correctly.

Example 1 (Concurrent *map* evaluation)**.** *We use the following shorthands for the evaluation. We note that we also labelled the code presented in Figures 2–4 with the following shorthands to ease understanding.*

- e_{pmap} *denotes the expression presented in Figure 4;*
- c_{child} *denotes the closure for the spawned process:*

$$clos(\emptyset, [\texttt{S},\texttt{L}], \texttt{call 'erlang':'!'(S,apply}\ c_{map}(e_f,\texttt{L})));$$

- e_{let} *denotes the second subexpression of the* `do` *expression from Figure 4, which processes the list suffix sequentially, and receives the result from the child process;*
- e_{case} *denotes the outermost* `case` *expression (checking the result of peeking the mailbox) from Figure 2 substituted with the two cases of the* `receive` *expression in Figure 4;*
- $e_{reccase}$ *denotes the innermost* `case` *expression (substituted with the concrete values in the* `receive` *expression of Figure 4) from Figure 2 which checks the result of the timeout;*
- cl_1, cl_2 *denote the clauses of the* `case` *expression of Figure 3.*
- v^l *is used for the transformed list* `[2|[3|[4|[5|[]]]]]`*, and* v^l_1 *for the transformed prefix* `[2|[3|[]]]`*.*

Next, we present multiple ways to evaluate concurrent list transformation. For brevity, we show the sequential configurations, since the set of linked processes and the process flag does not influence this evaluation (i.e., they can be arbitrary for both the parent and the child process), and show the mailbox separately. First, we show the configurations for the parent in which a non-silent reduction can happen or happened, and the initial state:

$$\langle \varepsilon, e_{pmap} \rangle \qquad (p_{start})$$
$$\langle call(\texttt{'erlang'},\texttt{'spawn'})(c_{child},\square) :: \texttt{do}\ \square\ e_{let} :: \varepsilon, \texttt{[1|[2|[]]]} \rangle \qquad (p_{spawn})$$
$$\langle \texttt{do}\ \square\ e_{let} :: \varepsilon, \texttt{<}\iota_c\texttt{>} \rangle \qquad (p_{do})$$
$$\langle primop(\texttt{'recv\_peek\_message'})(\square) :: $$
$$:: \texttt{let <Success,Msg>} = \square\ \texttt{in}\ e_{case} :: \varepsilon, \square \rangle \qquad (p_{peek})$$
$$\langle \texttt{let <Success,Msg>} = \square\ \texttt{in}\ e_{case} :: \varepsilon, \texttt{<'false','error'>} \rangle \qquad (p_{fail})$$
$$\langle primop(\texttt{'recv\_wait\_timeout'})(\square) :: $$
$$:: \texttt{let}\ = \square\ \texttt{in}\ e_{reccase} :: \varepsilon, \texttt{<'infinity'>} \rangle \qquad (p_{wait})$$
$$\langle call(\texttt{'erlang'},\texttt{'!'})(\iota,\square) :: \varepsilon, \texttt{<[2|[3|[4|[5|[]]]]]>} \rangle \qquad (p_{send})$$

$$\langle \varepsilon, \texttt{<[2|[3|[4|[5|[]]]]]>} \rangle \tag{p_{final}}$$

The child process can also be in three such states where non-silent actions can happen. We present these and the child's initial state:

$$\langle \varepsilon, \texttt{case [1|[2|[]]] of } cl_1; cl_2 \texttt{ end} \rangle \tag{c_{start}}$$

$$\langle call(\texttt{'erlang'}, \texttt{'!'})(\iota_p, \Box) :: \varepsilon, \texttt{<}v_1^l\texttt{>} \rangle \tag{c_{send}}$$

$$\langle \varepsilon, \texttt{<}v_1^l\texttt{>} \rangle \tag{c_{final}}$$

$$\emptyset \tag{c_{term}}$$

We highlight possible execution paths for the concurrent list transforming function. In Figure 13, we show the decision points where it matters which action to evaluate first. The semantics is strongly confluent for silent actions according to Theorems 3 and 4; thus, the order of silent reductions does not matter. For this reason, we do not discuss these steps (but refer to Appendix B for an example). In Figure 13, we use the notations below for the states of the node and denote silent reduction chains with (*). In the following list of pairs, the first component denotes the ether, the second the process pool consisting of at most the parent and child process, and the parent's mailbox is explicitly described (the child's mailbox is empty in all steps). The node's subscripts highlight the next BIF that the parent has to evaluate, while the superscripts denote the remaining actions that can happen in the given configuration.

- $N_{start} = (\emptyset, (p_{start}, [\blacktriangleright]))$
- $N_{spawn} = (\emptyset, (p_{spawn}, [\blacktriangleright]))$
- $N_{spawned} = (\emptyset, (p_{do}, [\blacktriangleright]) \parallel c_{start})$
- $N^{send}_{peek} = (\emptyset, (p_{peek}, [\blacktriangleright]) \parallel c_{send})$
- $N^{send}_{peekfail} = (\emptyset, (p_{fail}, [\blacktriangleright]) \parallel c_{send})$
- $N^{msg,\ term}_{peek} = (\emptyset[(\iota_c, \iota_p) \mapsto^+ v_1^l], (p_{peek}, [\blacktriangleright]) \parallel c_{final})$
- $N^{msg,\ term}_{peekfail} = (\emptyset[(\iota_c, \iota_p) \mapsto^+ v_1^l], (p_{fail}, [\blacktriangleright]) \parallel c_{final})$
- $N^{send}_{wait} = (\emptyset, (p_{wait}, [\blacktriangleright]) \parallel c_{send})$
- $N^{msg}_{peekfail} = (\emptyset[(\iota_c, \iota_p) \mapsto^+ v_1^l], (p_{fail}, [\blacktriangleright]) \parallel c_{term})$
- $N^{msg}_{peek} = (\emptyset[(\iota_c, \iota_p) \mapsto^+ v_1^l], (p_{peek}, [\blacktriangleright]) \parallel c_{term})$
- $N^{term}_{peek} = (\emptyset, (p_{peek}, [\blacktriangleright\ v_1^l]) \parallel c_{final})$
- $N^{term}_{wait} = (\emptyset, (p_{wait}, [\blacktriangleright\ v_1^l]) \parallel c_{final})$
- $N^{msg,\ term}_{wait} = (\emptyset[(\iota_c, \iota_p) \mapsto^+ v_1^l], (p_{wait}, [\blacktriangleright]) \parallel c_{final})$
- $N_{peek} = (\emptyset, (p_{peek}, [\blacktriangleright\ v_1^l]) \parallel c_{term})$
- $N^{term}_{send} = (\emptyset, (p_{send}, [\blacktriangleright]) \parallel c_{final})$
- $N^{msg}_{wait} = (\emptyset[(\iota_c, \iota_p) \mapsto^+ v_1^l], (p_{wait}, [\blacktriangleright]) \parallel c_{term})$
- $N_{send} = (\emptyset, (p_{send}, [\blacktriangleright]) \parallel c_{term})$
- $N_{term} = (\emptyset[(\iota_p, \iota) \mapsto^+ v^l], (p_{final}, [\blacktriangleright]) \parallel c_{final})$
- $N_{wait} = (\emptyset, (p_{wait}, [\blacktriangleright\ v_1^l]) \parallel c_{term})$
- $N_{final} = (\emptyset[(\iota_p, \iota) \mapsto^+ v^l], (p_{final}, [\blacktriangleright]) \parallel c_{term})$

The evaluation starts with the parent splitting the list in half (since $i = 2$), and spawning the child process (N_{start}, N_{spawn}, $N_{spawned}$). Next, both the parent and child process evaluate the map *function sequentially for their list segments reaching N^{send}_{peek} (for this evaluation, we refer to Appendix B, Example A1). At this point, either the child sends its result to the parent, or the parent fails to evaluate* `'recv_peek_message'`*. Actually, the child can even send the message ($N^{msg,\ term}_{peekfail}$), and terminate ($N^{msg}_{peekfail}$), while the parent still fails on* `'recv_peek_message'`*, because the message has not been delivered yet (with* Msg*).*

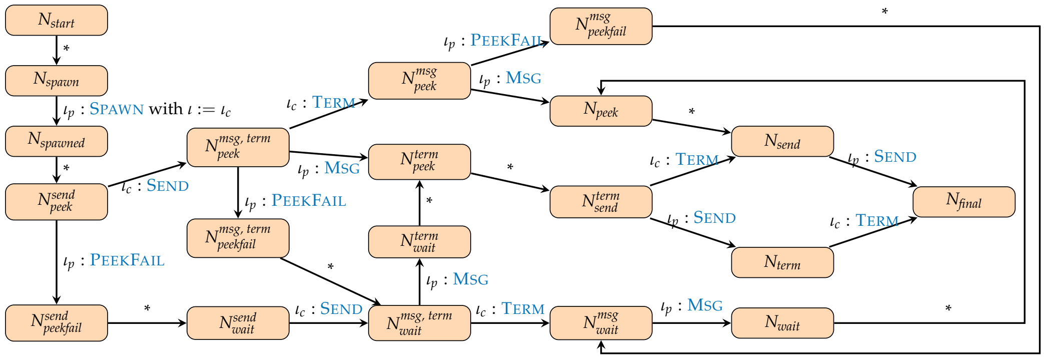


Figure 13. Evaluation of concurrent *map* (Figure 4).

If the parent failed in either of the previous steps, it becomes stuck on an infinite timeout, until the message from the child arrives. Next, the parent continues the evaluation of the message receipt recursively (see Figure 2), and retries peeking into the mailbox. This leads to N^{term}_{peek} (if the child is not yet terminated) or N_{peek} (if the child is already terminated). If the parent tried peeking into the mailbox only after the message had arrived (i.e., peeking had never failed), then one of the previous two states was reached, too.

Finally, the parent successfully receives the transformed list from the child (reaching either N^{term}_{send} or N_{send}); then it appends the two segments and sends the result to the observed PID ι.

3.7. Semantic Properties

Next, we show a number of properties of the concurrent semantics, and for their proofs, we refer to the formalisation [29], and to Appendix C, which connects the concepts of this paper, to the code. First, we state that our semantics respects the signal ordering guarantee [9].

Theorem 1 (Signal ordering guarantee)**.** *For all nodes N_1, N_2, N_3, PIDs ι, ι', and unique signals (i.e., they are different from any other signal in the starting configuration). $s_1 \neq s_2$, if $N_1 \xrightarrow{\iota:send(\iota,\iota',s_1)}_O N_2$ and $N_2 \xrightarrow{\iota:send(\iota,\iota',s_2)}_O N_3$, then for all nodes N_4 and action traces l which satisfy $N_3 \xrightarrow{l}{}^{*}_{O} N_4$ and also $(\iota', arr(\iota,\iota',s_1)) \notin l$ then $\nexists N_5 : N_4 \xrightarrow{\iota':arr(\iota,\iota',s_2)}_O N_5$ (i.e., there is no node at which s_2 can arrive).*

To reason about process creation in later sections, it is inevitable to reason about PID renaming. We use renaming when arguing about process spawns because in most cases, the freshness criteria imposed by NSPAWN is too weak as it depends only on O and the node the *spawn* is evaluated in. In some cases, there could be other PIDs that should be distinct from the spawned one (e.g., when reasoning about node equivalence).

Definition 7 (PID renaming)**.** *We denote PID renaming with $r[\iota_1 \mapsto \iota_2]$, which syntactically replaces all occurrences of ι_1 with ι_2 inside a redex r. For simplicity, we use the same notation for frame stacks, mailboxes, lists, and live processes.*

We denote PID renaming for finite maps (dead processes, ethers, process pools, and nodes) with $N[\iota_1 \leftrightarrow \iota_2]$. For these syntactical constructs, renaming is the syntactical exchange of the given two PIDs. With this notion, we avoid accidental overwriting of existing bindings in a finite map.

Given a list of PID pairs $[(\iota_1, \iota'_1), \dots, (\iota_n, \iota'_n)]$, we use $r[\iota_1 \mapsto \iota'_1, \dots, \iota_n \mapsto \iota'_n]$ (and $N[\iota_1 \leftrightarrow \iota'_1, \dots, \iota_n \leftrightarrow \iota'_n]$) for sequential renaming, i.e., $r[\iota_1 \mapsto \iota'_1] \dots [\iota_n \mapsto \iota'_n]$ ($N[\iota_1 \leftrightarrow \iota'_1] \dots [\iota_n \leftrightarrow \iota'_n]$, respectively).

PIDs cannot be renamed freely while reasoning about program equivalence; we have to make sure that we do not bind observed or used PIDs while renaming. This property is expressed with the following two compatibility definitions. Node-compatibility ensures that a renaming satisfies the freshness conditions, and action-compatibility ensures that this freshness condition is respected by the reductions with the given actions.

Definition 8 (Node-compatible renaming). *A list of PID pairs* $[(\iota_1, \iota'_1), \ldots, (\iota_n, \iota'_n)]$ *is compatible with a node N and observable PIDs O, if for all indices i the following points are all satisfied:*

- $\iota_i, \iota'_i \notin O$;
- $\iota'_i \notin \mathit{PIDsOf}(N^{\Delta}[\iota_1 \leftrightarrow \iota'_1, \ldots, \iota_{i-1} \leftrightarrow \iota'_{i-1}])$;
- $\iota'_i \notin \mathit{PIDsOf}(N^{\Pi}[\iota_1 \leftrightarrow \iota'_1, \ldots, \iota_{i-1} \leftrightarrow \iota'_{i-1}])$.

Definition 9 (Action-compatible renaming). *A list of PID pairs* $[(\iota_1, \iota'_1), \ldots, (\iota_n, \iota'_n)]$ *is compatible with an action a, if for all indices i,* $\iota'_i \notin \mathit{PIDsOf}(a[\iota_1 \mapsto \iota'_1, \ldots, \iota_{i-1} \mapsto \iota'_{i-1}])$.

We note that we can always define a compatible renaming for a node or action based on fresh PIDs. Next, we show that node and action-compatible renamings preserve the semantics.

Theorem 2 (Renaming preserves reduction). *For all lists of PID pairs* $[(\iota_1, \iota'_1), \ldots, (\iota_n, \iota'_n)]$ *which are compatible with nodes N with observable PIDs O, and actions a, all reductions* $N \xrightarrow{\iota:a}_O N'$ *are preserved by the compatible renaming, i.e.,*

$$N[\iota_1 \leftrightarrow \iota'_1, \ldots, \iota_n \leftrightarrow \iota'_n] \xrightarrow{\iota[\iota_1 \leftrightarrow \iota'_1, \ldots, \iota_n \leftrightarrow \iota'_n]:a[\iota_1 \mapsto \iota'_1, \ldots, \iota_n \mapsto \iota'_n]}_O N'[\iota_1 \leftrightarrow \iota'_1, \ldots, \iota_n \leftrightarrow \iota'_n].$$

In addition to renaming, the other important property of the semantics is confluence, on which some of our equivalence examples depend. We proved two confluence properties: a diamond property for reasoning about different processes, and a strong confluence property for reasoning about silent actions. For the first theorem, we need to define the compatibility of actions.

Definition 10 (Compatibility between actions). *Two actions are compatible, if neither of them is a spawn action, or if one is a* $\mathit{spawn}(\iota, v_1, v_2, b)$ *action, the other one is not* $\mathit{spawn}(\iota, v_3, v_4, b')$ *or* $\mathit{send}(\iota^s, \iota, v)$ *(i.e., in case of spawn or send, the used (and target) PIDs are different).*

The previous definition can always be satisfied by using PID renaming for the *spawn* actions.

Theorem 3 (Commutativity of reductions (a restricted diamond property)). *For all nodes* N_1, N_2, N'_2*, compatible actions* a_1, a_2*, and PIDs* $\iota_1 \neq \iota_2$*, if* $N_1 \xrightarrow{\iota_1:a_1}_O N_2$ *and* $N_1 \xrightarrow{\iota_2:a_2}_O N'_2$*, there exists a node* N_3*, which satisfies both* $N_2 \xrightarrow{\iota_2:a_2}_O N_3$ *and* $N'_2 \xrightarrow{\iota_1:a_1}_O N_3$.

Theorem 4 (Strong confluence with silent actions). *For all nodes* N_1, N_2, N'_2*, silent actions* a_s*, actions a, and PIDs* ι*, if* $N_1 \xrightarrow{\iota:a_s}_O N_2$ *and* $N_1 \xrightarrow{\iota:a}_O N'_2$*, then one of the following cases is satisfied:*

- $a_s = a$ *and* $N'_2 = N_2$*; or*
- *a is an arrive action, there is a node* N_3*, such that* $N_2 \xrightarrow{\iota:a}_O N_3$*, and either* $N'_2 \xrightarrow{\iota:a_s}_O N_3$ *or* $N_3 = N'_2$ *(in the latter case, the process with PID* ι *was terminated).*

4. Program Equivalence

In this section, we investigate a number of definitions for program equivalence based on barbed bisimulation [30]. This section is structured as follows. In Section 4.1, we introduce the usual notion of strong and weak (barbed) bisimulations, and argue why they are insufficient to reason about (Core) Erlang nodes. Next, in Section 4.2, we introduce a refined program equivalence concept based on barbed bisimulation, which is suitable to argue about refactoring correctness on concurrent programs. Finally, in Section 4.3, we enumerate a number of bisimulation examples which can be considered as correct concurrent refactorings.

4.1. Restrictive Notions of Program Equivalence

Practically, two concurrent programs are equivalent, if an external observer cannot distinguish their behaviour. For us, the communication (i.e., the signals sent) of a node can be observed from the outside. For barbed bisimulations, we have to characterise this property. First, we define the observable behaviour as the signals in a node's ether targeting some PID in the set O (i.e., O is the observer, and the signals sent to it define the observed behaviour). Based on this thought, we can define when two nodes agree on the observed signals:

Definition 11 (Nodes agree on observed signals)**.** *Two nodes N_1, N_2 weakly agree on O, when for all PIDs ι_s, ι_d, if $\iota_d \in O$ implies that there exists a PID ι_2^s such that $N_1^\Delta[(\iota_s, \iota_d)] \simeq N_2^\Delta[(\iota_2^s, \iota_d)]$, where $\simeq$ denotes syntactical equality up to PIDs.*

Two nodes strongly agree on O, if they weakly agree on O; moreover, $\iota_2^s = \iota_s$ in the previous definition.

The first definition we investigate is barbed strong bisimulation, which relates two nodes whenever they can make the same reductions, and they agree on the signals sent to observed PIDs.

Definition 12 (Barbed strong bisimulation)**.** *A relation $\mathcal{R}$ on nodes is a barbed strong bisimulation observing O if given two nodes N_1, N_2, $\mathcal{R}(N_1, N_2)$ implies the following:*

- *For all nodes N_1', actions a, and PIDs ι, if $N_1 \xrightarrow{\iota:a}_O N_1'$ then $\exists N_2' : N_2 \xrightarrow{\iota:a}_O N_2'$ and $\mathcal{R}(N_1', N_2')$;*
- *The converse of the previous point for reductions from N_2;*
- *N_1 and N_2 strongly agree on O (note that this property is symmetric).*

We use $N_1 \sim_O N_2$ to denote that N_1 and N_2 are related by a relation $\mathcal{R}$ which is a barbed strong bisimulation.

Proving strong bisimilarity even between simple nodes is impossible in most practical cases. Consider a node consisting of one process which computes the expression `call 'erlang':'+'(1,2)` and a node with a process computing 3. The first node can take some τ computation steps, which could not be taken by the second one. To loosen this notion of program equivalence, we can introduce weak bisimulations.

Definition 13 (Barbed weak bisimulation)**.** *A relation $\mathcal{R}$ on nodes is a weak barbed bisimulation observing O if given two nodes N_1, N_2, $\mathcal{R}(N_1, N_2)$ implies the following:*

- *For all nodes N_1', actions a, and PIDs ι, if $N_1 \xrightarrow{\iota:a}_O N_1'$ then $\exists N_2', N_2'', N_2''' : N_2 \xrightarrow{\tau}{}^*_O N_2' \xrightarrow[O]{\iota:a} N_2'' \xrightarrow{\tau}{}^*_O N_2'''$ and $\mathcal{R}(N_1', N_2''')$;*
- *The converse of the previous point for reductions from N_2;*
- *N_1 and N_2 strongly agree on O.*

We use $N_1 \approx_O N_2$ to denote that N_1 and N_2 are related by a relation $\mathcal{R}$ which is a weak barbed bisimulation.

Remark 2. *There is no need to include silent reduction steps from N_2 in the third point of Definition 13 as τ actions do not affect the ether.*

With this definition, we can prove examples like the previous one, which only differ in computational steps, but cannot prove equivalence between nodes that communicate or spawn processes differently. In particular, with this notion, we cannot prove the equivalence between the two list processing functions from the running example (Figures 3 and 4). This is because the sequential variant only communicates the result of the computation, while the parallel version spawns a child process and also performs communication between the child and parent processes.

4.2. Alternative Notion for Weak Barbed Bisimulation

We borrow another idea of (weak) barbed bisimulation used for related research on Core Erlang [16], and tailor it to our needs. In this definition, we do not compare the actions that induce the reductions made by the nodes, and only focus on the observables. If a reduction is taken by one node, the other one has to simulate it, but there is no restriction on what and how many steps this simulation should take.

Definition 14 (Barbed bisimulation)**.** *A relation $\mathcal{R}$ on nodes is a barbed bisimulation observing O, if given two nodes N_1, N_2, $\mathcal{R}(N_1, N_2)$ implies the following:*

- *For all nodes N_1', actions a, and PIDs ι, if $N_1 \xrightarrow{\iota:a}_O N_1'$ then $\exists N_2' : N_2 \longrightarrow_O^* N_2'$ and $\mathcal{R}(N_1', N_2')$;*
- *$\exists N_2' : N_2 \longrightarrow_O^* N_2'$ and N_1 and N_2' weakly agree on O;*
- *The converse of the previous points for reductions and observables of N_2.*

We use $N_1 \approx_O N_2$ to denote that N_1 and N_2 are related by a relation $\mathcal{R}$ which is a barbed bisimulation. Hereafter, whenever we write bisimulation, we mean this definition.

After defining the suitable notion of program equivalence, we state the basic properties of this relation: reflexivity, symmetry, and transitivity. We furthermore highlight that this notion is less restrictive than the previous two definitions (for the proofs, refer to the formalisation [29]).

Theorem 5 (Equivalence relation)**.** *For all sets of PIDs O, the relation $\approx_O$ is reflexive, symmetric, and transitive.*

Theorem 6 (Correspondence between bisimulations)**.** *For all sets of PIDs O, $\sim_O \subset \approx_O \subset \approx_O$.*

4.3. Bisimulation Examples

Next, we show some bisimilar node pairs. For complete proofs, we refer to the formalisation [29], while we also give some insights here. In general, to prove these lemmas, we relied on coinduction, and case distinction based on the four rules of the inter-process semantics (Figure 12). To reason about *spawn* actions, in most cases, we also had to use renaming (either with Theorem 7 or with the coinductive hypothesis).

Theorem 7 (PID-renaming is a barbed bisimulation)**.** *For all lists $[(\iota_1, \iota_1'), \ldots, (\iota_n, \iota_n')]$ containing PID pairs which are compatible with the node N and sets of PIDs O, $N \approx_O N[\iota_1 \leftrightarrow \iota_1', \ldots, \iota_n \leftrightarrow \iota_n']$.*

Proof. We proceed with coinduction. We need to prove the four conditions of the bisimulation (Definition 14). The proof of the third and fourth points is based on the symmetric properties of bisimulations, and equality; thus, we only briefly explain the proof of the first two points. For readability, we use l to denote the list of renamings $\iota_1 \leftrightarrow \iota_1', \ldots, \iota_n \leftrightarrow \iota_n'$ (or $\iota_1 \mapsto \iota_1', \ldots, \iota_n \mapsto \iota_n'$ depending on the context).

Suppose that $N \xrightarrow{\iota:a}_O N'$. We need to show $N[l] \xrightarrow{\iota[l]:a[l]}_O N'[l]$. According to Theorem 2, if the renaming is compatible with action a, we can show that the renaming preserves the reduction, and by the coinductive hypothesis, the reached nodes are bisimilar. Since the renaming is compatible with the initial node, it is also compatible with almost all actions, because the PIDs that appear in the actions originated either from the ether or the process pool. On the other hand, *spawn* actions involve a fresh PID (ι_{spawn}), which could collide with the PIDs used for renaming. In this case, we can extend the list of renamings with a new renaming: $(\iota_{spawn}, \iota_{fresh}) :: l$. Since ι_{fresh} is chosen arbitrarily, we can ensure that it does not appear in the node or the action, and this extended list is compatible with action a, thus $N[\iota_{spawn} \leftrightarrow \iota_{fresh}, l] \xrightarrow{\iota[\iota_{spawn} \mapsto \iota_{fresh}, l]:a[\iota_{spawn} \mapsto \iota_{fresh}, l]}_O N'[\iota_{spawn} \leftrightarrow \iota_{fresh}, l]$, and we can use the coinductive hypothesis for this extended list to prove that the reached nodes are bisimilar.

The proof of the second point is mostly technical. If there are signals targeting $\iota^d \in O$ in the ether of N (e.g., with the source ι^s), we can show that there are signals targeting ι^d in $N[l]$ too, with the source $\iota^s[l]$. Moreover, these signals are equal up to the PIDs. □

While PID renaming seems to be an obvious (and often implicitly used) property, in a machine-checked formalisation, we need to be rigorous and explicit both in its proof and in its uses. Without renaming, there is no way to reason about nodes that use different PIDs, and also spawn some new processes, since the spawned PID is potentially not fresh in the other node. We mitigate this problem by renaming the spawned PID to a fresh(er) one.

Next, we state two technical lemmas to reason about elements of the ether and process pool which do not affect the evaluation. With these lemmas, we can remove irrelevant signals and terminated processes from a node during bisimulation proofs. The first lemma states that signals originating from, or targeting, terminated processes do not distinguish nodes.

Lemma 1 (Ether update for terminated processes)**.** *For all nodes N, sets of PIDs O, and PIDs ι^s, ι^d which satisfies $\iota^d \notin O$, if both $N^\Pi[\iota^s]$ and $N^\Pi[\iota^d]$ are terminated processes, then for any list of signals l, $N \approx_O (N^\Delta[(\iota^s, \iota^d) \mapsto l], N^\Pi)$, and $N \approx_O (N^\Delta \setminus (\iota^s, \iota^d), N^\Pi)$.*

Proof. We proceed by coinduction. We need to prove the four conditions of the bisimulation (Definition 14). The points about observables trivially hold since the update in the ether does not affect observed PIDs.

Suppose that $N \xrightarrow{\iota:a}_O N'$, then we have to show that we can perform an equivalent step with the updated ether.

- If NLocal or NArrive was used, then the same step can be made in the updated node (since these actions could not have been taken by terminated processes), and we can use the coinductive hypothesis.
- If NSpawn was used (which could not have been performed by a terminated process), we have to rename the PID of the spawned process to a fresh PID (so that it does not appear anywhere in the modified configuration, nor in the set of PIDs used in l), and we can perform the same reduction step with the fresh PID, since it satisfies the side condition of NSpawn. We finish this case by using the transitivity of barbed bisimulation with Theorem 7 and the coinductive hypothesis.
- In the case of NSend, we can make the same reduction in the updated node, regardless of which process sent the message. Supposing that $a = \mathit{send}(\iota^s, \iota^d, s)$, we can use the coinductive hypothesis with the list of signals chosen as $l +\!\!+ [s]$ (or just $[s]$, if we prove the second part of the theorem).

For the converse direction, we can use the fact that any ether E can be expressed with two updates:

- If $E[(\iota^s, \iota^d)] = \mathit{Some}(l')$, then $E = E[(\iota^s, \iota^d) \mapsto l][(\iota^s, \iota^d) \mapsto l']$.
- If $E[(\iota^s, \iota^d)] = \mathit{None}$, then $E = E[(\iota^s, \iota^d) \mapsto l] \setminus (\iota^s, \iota^d)$.

Therefore, we can use the same train of thought as described above to conclude the proof. □

The following lemma states that adding unlinked terminated processes (we denote empty maps with $\emptyset$) to a node creates an equivalent node.

Lemma 2 (Terminated process)**.** *For all nodes N, sets of PIDs O, and PIDs ι, if $\iota \notin O$ and $\iota \notin dom(N^\Pi)$, then $N \approx_O (N^\Delta, \iota \mapsto \emptyset \parallel N^\Pi)$.*

Proof. This proof is also constructed with coinduction. The main idea is that the terminated process $\emptyset$ cannot take any reduction; thus, the same reductions can be made in both nodes, and this way, the ether is not affected either (i.e., observables remain the same). □

The next theorem states that if a node can be reduced with τ steps or *self* actions, the result is equivalent to the original node. It is useful to reason about concrete node equivalences since it reduces the problem of proving bisimulation to proving evaluation. Together with the transitivity of $\approx_O$, in bisimulation proofs, we can discharge reasoning about silent steps.

Theorem 8 (Silent evaluation)**.** *For all nodes N, N', sets of PIDs O, if $N \xrightarrow{l}^{*}_{O} N'$ for some action trace l, which contains only silent (i.e., τ or self) actions, $N \approx_O N'$ holds.*

Proof. For this theorem, it is sufficient to prove that one single silent step creates bisimilar nodes, since based on this fact and the transitivity of the barbed bisimulation (Theorem 5), we can prove the original property by induction on the action trace l.

To prove that a single silent step creates bisimilar nodes, we use coinduction, and prove the four requirements for barbed bisimulation. Suppose that $a = \tau \vee \exists \iota, a = \mathit{self}(\iota)$, and $N \xrightarrow{\iota:a}_{O} N'$. We prove $N \approx_O N'$:.

First, we check what other possible reductions can be made from N based on the first point in Definition 14. Suppose that for some action a', PID ι' and node N'', there is a reduction $N \xrightarrow{\iota':a'}_{O} N''$. We have to show that

$$\exists N_{final}, N' \rightarrow^{*}_{O} N_{final} \wedge N' \approx_O N_{final}.$$

There are the following options:

- If $\iota' = \iota$, then there are two options:
  - If $a' = a$, i.e., the same step is taken, then $N' = N''$ and we can choose $N_{final} = N'$ and use the reflexivity of the bisimulation.
  - If $a' \neq a$, then a' can only be an *arrive* action (Theorem 4). Supposing that the *arrive* action does not terminate the process, it can be postponed after making the reduction with a; thus, there is a node N''' to which $N' \xrightarrow{\iota:a'}_{O} N'''$ and $N'' \xrightarrow{\iota:a}_{O} N'''$. We choose $N_{final} = N'''$, which is reachable from N' based on the first reduction, and prove $N' \approx_O N'''$ based on the coinductive hypothesis and the second reduction.
  - If $a' \neq a$, a' is an *arrive* action, and it terminates the process, then the process is also if the action arrives in N', since silent actions do not modify either of the properties (linked PIDs, source and destination, exit reason, process flag) used in the semantics for *arrives* (see Figure 8); thus, $N' \xrightarrow{\iota:a'}_{O} N''$. We can choose $N_{final} = N''$ and use the reflexivity of the bisimulation.
- If for some action a', PID ι' and node N'', there is a reduction $N \xrightarrow{\iota':a'}_{O} N''$, then we can use Theorem 3 to derive the existence of a node N''' to which $N' \xrightarrow{\iota':a'}_{O} N'''$ and $N'' \xrightarrow{\iota:a}_{O} N'''$. We choose $N_{final} = N'''$, which is reachable from N' based on the first reduction, and prove $N' \approx_O N'''$ based on the coinductive hypothesis and the second reduction.

To prove that for all reductions $N' \xrightarrow{\iota':a'}_{O} N''$, the result N'' is bisimilar to a node which is reachable from N, we can chain this reduction after the assumption of $N \xrightarrow{\iota:a}_{O} N'$, and use the reflexivity of the bisimulation (Theorem 5).

The observables are not modified in the ether by silent actions; thus, the requirements on them of Definition 14 hold. □

We highlight that this theorem is more restrictive than the normalisation theorem of [16], which also considers message sending and process spawning in this bisimulation (we refer the reader to Section 5 for a discussion of why this is not the case for us). Finally, we show a concrete program equivalence based on barbed bisimulation.

Example 2 (Equivalence of sequential and concurrent *map*)**.** *Consider the expressions from Figure 3 (denoted with e_{map}) and Figure 4 (denoted with e_{pmap}). For all values v_l, v_f, natural numbers i, PIDs ι which appear as the metavariables in Figures 3 and 4, and for all PIDs ι_{base}, $(\emptyset, \iota_{base} \mapsto (\varepsilon, e_{map}, [\blacktriangleright], \emptyset, ff) \parallel \emptyset) \approx_{\{\iota\}} (\emptyset, \iota_{base} \mapsto (\varepsilon, e_{pmap}, [\blacktriangleright], \emptyset, ff) \parallel \emptyset)$, if the following conditions hold:*

- $\iota_{base} \neq \iota$.
- *The closure value v_f computes a metatheoretical function f, that is, for all values v, we can prove $\langle \varepsilon, \texttt{apply}\ v_f(v) \rangle \longrightarrow^* <f(v)>$ in the sequential layer.*
- *v_l is a proper Core Erlang list, i.e., it is constructed as $[v_1|[v_2|\dots[v_n|[]]]\dots]$, and $i \leq n$.*
- *v_l does not contain any PIDs; moreover, the application of v_f does not introduce any PIDs.*

Proof Sketch. The proof of this theorem is quite involved; thus, we refer to the formalisation [29] for all details, and only describe the main idea here. We avoided using the definition of the bisimulation manually since it takes many reductions to evaluate both sequential and parallel list transformation, and reasoning about the four conditions of Definition 14 generates unnecessary, tremendous overhead. Instead, we heavily rely on Theorem 8 and the transitivity of bisimulation (Theorem 5) to discharge silent evaluation steps from the reasoning.

There are two main points to prove: there is an evaluation of the parallel list processing that behaves the same way as the sequential one, and vice versa. The former point follows from the fact that the sequential *map* can be only evaluated deterministically (see Example A1 for more insights) and from Example 1 where (all) paths lead to the same final configuration. The latter point is more challenging, since all evaluation paths in Figure 13 have to be simulated by the sequential list processing. To prove this, we need to manually apply the definition of bisimulation in all states from Figure 13 from where a non-silent reduction happens. □

5. Discussion

In this section, we describe the novelty of this work with respect to our previous work we build on. We also discuss some major theoretical and technical challenges originating from the formalised signals, the absence of atomic `receive` expressions, and the fact that our formalisation is mechanized in Coq.

5.1. Novelty

As mentioned before, this paper builds on, and extends, our previous work described in [10,11]. Namely, we reused the sequential semantics of [11] (which we recall in Appendix A), and built the process-local and inter-process semantics on top of it based on [10]. However, note that we do not repeat the definitions presented there; in particular, this current paper not only extends our previous work but defines the communication primitives with a different granularity (i.e., with the new primitives implemented in Erlang/OTP version 23 [19]), which required changes in the underlying representations compared to [10] (such as that of mailboxes, process pools, and the ether), the semantic rules, and the proofs of the fundamental properties. Since the implementation of the new communication primitives does not fully conform with the old language specification, our work is essential, and it sets the state-of-the-art regarding the formal definition of the actor model of Erlang, providing a rigorous and solid basis for future research on the most recent communication model.

5.2. Theoretical Challenges

The first challenge we encountered was the definition of the inter-process semantics, which had to satisfy both the signal-ordering guarantee and the fact that message passing is not atomic (as explained in Section 3.5). These conditions make it necessary to define an ether that includes the sent but not delivered signals in an ordered way. In addition to making the semantics more complex, this decision also makes some reductions impossible, which would have been carried out if the signal-ordering guarantee was not enforced. On the other hand, this simplifies reasoning about signal arrival, since only the first signals need to be considered from the ether (from the ordered list of signals) targeting a process.

Next, the formalisation of *exit* signals comes with a number of challenges. Since *exit* signals can potentially terminate a process, they also limit the confluence properties of the semantics. This is also one of the reasons why Theorem 8 is more restricted than “normalisation” (from [16]) which also allows *send* and ε actions in the reduction chain. Whether an *exit* signal terminates a process depends on a number of factors: which are the linked PIDs, how the `trap_exit` flag is set, what is the reason value, and the source of the signal. If the process for which the *exit* is arriving can also make a step which modifies either of these, the process will behave in a different way if the signal arrives before or after this step, and this breaks the confluence property.

A similar argument can be made about the correlation between primitive operations for message receipts and arrival. For example, if a message arrives before evaluating `recv_peek_message`, then the first component of its result is the atom `'true'`, while if the message arrives later, the same component could be `'false'` (if there are no unseen messages in the mailbox). This means that Lemma 9 of [16]—saying that any reduction that can be made with a process can also be made if a new message is appended at the end of the mailbox—cannot be proved; this lemma only holds for `receive` expressions that evaluate in an atomic way. To handle the maximum number of reductions with Theorem 8 when reasoning about bisimulations, we decided to label all reductions with τ (not just steps of the sequential layer) for which the semantics is strongly confluent.

5.3. Formalisation

As mentioned before, all of our results are machine-checked with Coq [29] (around 35,000 lines of code). We briefly explain the main design decisions we made.

- We deeply embedded the syntax of Core Erlang into Coq as an inductive definition, so that we can use induction to reason about substitutions, PID renamings, and variable scoping.
- We encoded variables (and function identifiers) of Core Erlang with the nameless variable representation, i.e., all variables are de Bruijn indices. Using a nameless encoding simplifies reasoning about alpha-equivalence to checking equality, and fresh variable generation is simply expressed with addition of natural numbers. This way, the syntax is less readable for the human eye, but it is much simpler to define parallel capture-avoiding substitutions and use them in proofs [31].
- We did not use a similar, nameless encoding for PIDs. While process pools behave like binders, PIDs generally do not behave as variables: PIDs are dynamically created, and we do not apply substitutions for them. Moreover, in some cases, we might need to depend on the exact value of a PID in the program (e.g., for PID comparison). In addition, alpha-equivalence of PIDs cannot be reduced to equality checking with a nameless encoding. Suppose that we have several process spawns that can be executed in any order. After executing all process spawns in all possible orders, the result systems will be alpha-equivalent, but never equal, since the spawned PIDs (as de Bruijn indices) would depend on the particular execution path.
- We expressed all semantic layers as inductive judgements in order so that we could more easily use Coq’s tactics (`inversion` and `constructor`) to handle semantic reductions in proofs.
- We formalised process pools, ethers and dead processes as finite maps so that the used PIDs inside such constructs can be collected by a recursive function. Based on the collection result, we can come up with fresh PIDs for spawned processes. If we used functions instead of finite maps, showing that a concrete PID does not appear inside process of a process pool would have required much more complex proofs (potentially based on induction).

- For an implementation of finite maps and freshness, we relied on the Iris project's stdpp [32] library, which also includes the `set_solver` tactic that can be used to automatically discharge statements about sets and finite maps.
- The formalisation depends only on one standard axiom; we use functional extensionality for reasoning about parallel substitutions.

Having a machine-checked formalisation also comes with some drawbacks. Firstly, a proof assistant forces the proof engineer to be more precise compared to writing mathematical proofs on paper. For our project, this creates technical difficulties whenever reasoning about *spawn* actions is needed, because (in most cases) at these steps, we have to rely on renaming with fresh PIDs. After defining the necessary fresh PIDs, we have to manually simplify these renamings, and in equivalence proofs, based on the transitivity of the bisimulation (Theorem 5), we use Theorem 7 to reach the desired results.

However, relying on the transitivity of the bisimulation (or any other bisimulation-transforming proof) comes with another drawback: in a coinductive proof, the guardedness checker of Coq does not accept proofs that use transitivity on the coinductive hypothesis, since it is often unsound (for now, we turned off the guardedness checker for these proofs and relied on existing approaches [16] to carry out the proofs; we intend to investigate other approaches to this). However, the violations of the guardedness only originate from the following two proof strategies.

1. In Theorem 7, we used bisimulation's symmetry to justify the first clause of Definition 14 for the derivations starting from N_2 (i.e., for the node with renamings); however, we are certain that this could be replaced by several analogous helper lemmas, ultimately discharging the potential of invalidity.
2. For every other case (including Lemmas 1 and 2), we used bisimulation's transitivity with Theorem 7 for injective alpha-renaming (based on the soundness results of [33]).

6. Conclusions and Future Work

We have defined formal semantics for the concurrent subset of Core Erlang, building on earlier work [10,11], extending the sequential syntax with process identifiers (PIDs), and defining the concurrent (process-local and inter-process) semantics by defining the meaning of concurrent built-in functions and primitive operations.

We defined three concepts of concurrent program equivalence based on barbed bisimulations. We argued that the usual notions of strong and weak bisimulations are too restrictive to reason about program equivalence. In order to model equivalence between programs that have different communication structure but the same observable behaviour, we introduced a weaker variant of barbed bisimulation (following the footsteps of [16,24]), with which we proved a number of program equivalences (such as PID renaming, executing computation steps, list processing sequentially or concurrently), reaching beyond previous and related work. The results presented here are formalised in the Coq proof assistant.

In the future, we aim to generalize the equivalence proof for sequential and concurrent list processing to include *exit* signals and message source validation. We then aim to generalize and create new approaches for reasoning about bisimulations to make it simpler to show concrete Core Erlang programs equivalent, to fulfil our goal of verifying Erlang refactorings.

Author Contributions: Conceptualization, P.B., D.H. and S.T.; funding acquisition, P.B. and D.H.; investigation, P.B.; methodology, P.B., D.H. and S.T.; project administration, D.H. and S.T.; software, P.B.; supervision, D.H. and S.T.; original draft, P.B.; Review and editing, P.B., D.H. and S.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the ÚNKP-23-3 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund. Project no. TKP2021-NVA-29 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

Data Availability Statement: The original data presented in the study are openly available in [29].

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Rules of the Sequential Semantics

In this section, we recall the rules of the sequential semantics from our previous work [11]. The rules are categorised into four groups:

1. Rules that deconstruct an expression by extracting its first redex while putting the rest of the expression in the frame stack (Figure A1).
2. Rules that modify the top frame of the stack by extracting the next redex and putting back the currently evaluated value into this top frame (Figure A2).
3. Rules that remove the top frame of the stack and construct the next redex based on this removed frame (Figure A3). We also included rules here which immediately reduce an expression without modifying the stack (e.g., PFUN).
4. Rules that express concepts of exception creation, handling, or propagation (Figure A4).

Next, we recall some of the auxiliary definitions, which are necessary to understand the semantics rules.

- $r[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$: substitutes names (variables or function identifiers) $x_1, \dots, x_n$ in the the redex r with the given values $v_1, \dots, v_n$. In some cases, we use the same notation ($r[l]$) for substituting a list containing name–value pairs in the same way.
- $\mathit{mk\_closlist}(\mathit{fdefs})$: creates a list of function identifier–closure pairs from the list fdefs. Every function definition $f/k = \texttt{fun}(x_1, \dots, x_k) \to e$ in the list is mapped to the following pair in the result list: $(f/k, \mathit{clos}(\mathit{fdefs}, [x_1, \dots, x_k], e))$.
- $\mathit{is\_match}(\mathit{ps}, \mathit{vs})$: decides whether a pattern list ps matches pairwise a value list vs (of the same length). A pattern matches a value, when they are constructed from the same elements, while pattern variables match every value.
- $\mathit{match}(\mathit{ps}, \mathit{vs})$: defines the variable-value binding (as a list of pairs) resulted by successfully matching the patterns ps with the values vs.

We provide an informal overview of $\mathit{eval}(\mathit{id}, v_1, \dots, v_n)$ here. If

- $\mathit{id} = \mathit{app}(v)$ and $v = \mathit{clos}(\mathit{fdefs}, [x_1, \dots, x_n], e)$, then $\mathit{eval}(\mathit{app}(v), v_1, \dots, v_n) = e[\mathit{mk\_closlist}(\mathit{fdefs}), x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$.
- $\mathit{id} = \mathit{app}(v)$ and v is not a closure, or has an incorrect number of formal parameters, the result is an exception.
- $\mathit{id} = \mathit{tuple}$, then $\mathit{eval}(\mathit{tuple}, v_1, \dots, v_n) = \{v_1, \dots, v_n\}$.
- $\mathit{id} = \mathit{values}$, then $\mathit{eval}(\mathit{values}, v_1, \dots, v_n) = {<}v_1, \dots, v_n{>}$.
- $\mathit{id} = \mathit{map}$ and n is an even number, then

$$\mathit{eval}(\mathit{map}, v_1, \dots, v_n) = \sim\{v_1 \Rightarrow v_2, \dots, v_{k-1} \Rightarrow v_k\}\sim,$$

  where the $k \leq n$ result values inside the map are obtained by eliminating duplicate keys and their associated values.
- $\mathit{id} = \mathit{call}(a^m, a^f)$, then $\mathit{eval}(\mathit{call}(a^m, a^f), v_1, \dots, v_n)$ simulates the behaviour of sequential built-in functions of (Core) Erlang.
- $\mathit{id} = \mathit{primop(a)}$, then $\mathit{eval}(\mathit{primop(a)}, v_1, \dots, v_n)$ simulates the behaviour of sequential primitive operations of Core Erlang.

We highlight a few points of this semantics but refer to [11] for all further details.

$$\langle K, [e_1|e_2]\rangle \longrightarrow \langle [e_1|\square] :: K, e_2\rangle \qquad \text{(SConsTail)}$$

$$\langle K, \texttt{let}\ {<}x_1,\ldots,x_n{>} = e_1\ \texttt{in}\ e_2\rangle \longrightarrow \langle \texttt{let}\ {<}x_1,\ldots,x_n{>} = \square\ \texttt{in}\ e_2 :: K, e_1\rangle \qquad \text{(SLet)}$$

$$\langle K, \texttt{do}\ e_1\ e_2\rangle \longrightarrow \langle \texttt{do}\ \square\ e_2 :: K, e_1\rangle \qquad \text{(SSeq)}$$

$$\langle K, \texttt{apply}\ e(e_1,\ldots,e_n)\rangle \longrightarrow \langle \texttt{apply}\ \square(e_1,\ldots,e_n) :: K, e\rangle \qquad \text{(SApp)}$$

$$\langle K, \texttt{call}\ e^m{:}e^f(e_1,\ldots,e_n)\rangle \longrightarrow \langle \texttt{call}\ \square{:}e^f(e_1,\ldots,e_n) :: K, e^m\rangle \qquad \text{(SCallMod)}$$

$$\langle K, \texttt{primop}\ a(e_1,\ldots,e_n)\rangle \longrightarrow \langle \mathit{primop}(a)(\square,e_1,\ldots,e_n) :: K, \square\rangle \qquad \text{(SPrimOp)}$$

$$\langle K, {<}e_1,\ldots,e_n{>}\rangle \longrightarrow \langle \mathit{values}(\square,e_1,\ldots,e_n) :: K, \square\rangle \qquad \text{(SVals)}$$

$$\langle K, \{e_1,\ldots,e_n\}\rangle \longrightarrow \langle \mathit{tuple}(\square,e_1,\ldots,e_n) :: K, \square\rangle \qquad \text{(STuple)}$$

$$\langle K, \sim\!\{e^k_1 \Rightarrow e^v_1, e^k_2 \Rightarrow e^v_2 \ldots, e^k_n \Rightarrow e^v_n\}\!\sim\rangle \longrightarrow \langle \mathit{map}(\square, e^v_1, e^k_2, e^v_2, \ldots, e^k_n, e^v_n) :: K, e^k_1\rangle \qquad \text{(SMap)}$$

$$\langle K, \texttt{case}\ e\ \texttt{of}\ cl_1;\ldots;cl_n\ \texttt{end}\rangle \longrightarrow \langle \texttt{case}\ \square\ \texttt{of}\ cl_1;\ldots;cl_n\ \texttt{end} :: K, e\rangle \qquad \text{(SCase)}$$

Figure A1. Frame stack semantics rules of 1.

$$\langle [e_1|\square] :: K, {<}v_2{>}\rangle \longrightarrow \langle [\square|v_2] :: K, e_1\rangle \qquad \text{(SConsHead)}$$

$$\langle \texttt{call}\ \square{:}e^f(e_1,\ldots,e_n) :: K, {<}v^m{>}\rangle \longrightarrow \langle \texttt{call}\ v^m{:}\square(e_1,\ldots,e_n) :: K, e^f\rangle \qquad \text{(SCallFun)}$$

$$\langle \texttt{call}\ v^m{:}\square(e_1,\ldots,e_n) :: K, {<}v^f{>}\rangle \longrightarrow \langle \mathit{call}(v^m,v^f)(\square,e_1,\ldots,e_n) :: K, \square\rangle \qquad \text{(SCallParam)}$$

$$\langle \texttt{apply}\ \square(e_1,\ldots,e_n) :: K, {<}v{>}\rangle \longrightarrow \langle \mathit{apply}(v)(\square,e_1,\ldots,e_n) :: K, \square\rangle \qquad \text{(SAppParam)}$$

$$\begin{aligned}&\langle \texttt{case}\ \square\ \texttt{of}\ ps\ \texttt{when}\ e^g \to e^b; cl_2;\ldots;cl_n\ \texttt{end} :: K, vs\rangle \longrightarrow \\ &\qquad \langle \texttt{case}\ \square\ \texttt{of}\ cl_2;\ldots;cl_n\ \texttt{end} :: K, vs\rangle \qquad (\text{if } \neg \mathit{is\_match}(ps,vs))\end{aligned} \qquad \text{(SCaseFail)}$$

$$\begin{aligned}&\langle \texttt{case}\ \square\ \texttt{of}\ ps\ \texttt{when}\ e^g \to e^b; cl_2;\ldots;cl_n\ \texttt{end} :: K, vs\rangle \longrightarrow \\ &\qquad \langle \texttt{case}\ vs\ \texttt{of}\ ps\ \texttt{when}\ \square \to e^b[\mathit{match}(ps,vs)]; cl_2;\ldots;cl_n\ \texttt{end} :: K, e^g[\mathit{match}(ps,vs)]\rangle \\ &\qquad (\text{if } \mathit{is\_match}(ps,vs))\end{aligned}$$
$$\text{(SCaseSuccess)}$$

$$\langle \texttt{case}\ vs\ \texttt{of}\ ps\ \texttt{when}\ \square \to e^b; cl_2;\ldots;cl_n\ \texttt{end} :: K, {<}\texttt{'false'}{>}\rangle \longrightarrow \langle \texttt{case}\ \square\ \texttt{of}\ cl_2;\ldots;cl_n\ \texttt{end} :: K, vs\rangle$$
$$\text{(SCaseFalse)}$$

$$\langle \mathit{id}(\square,e_1,e_2,\ldots,e_n) :: K, \square\rangle \longrightarrow \langle \mathit{id}(\square,e_2,\ldots,e_n) :: K, e_1\rangle \qquad (\text{if } \mathit{id} \neq \mathit{map}) \qquad \text{(SParams}_0\text{)}$$

$$\langle \mathit{id}(v_1,\ldots,v_{i-1},\square,e_{i+1},e_{i+2},\ldots,e_n) :: K, {<}v_i{>}\rangle \longrightarrow \langle \mathit{id}(v_1,\ldots,v_{i-1},v_i,\square,e_{i+2},\ldots,e_n) :: K, e_{i+1}\rangle$$
$$\text{(SParams)}$$

Figure A2. Frame stack semantics rules of 2.

$$\langle K, \sim\!\{\}\!\sim\rangle \longrightarrow \langle K, {<}\sim\!\{\}\!\sim{>}\rangle \qquad \text{(PMap}_0\text{)}$$

$$\langle K, \texttt{fun}(x_1,\ldots,x_n) \to e\rangle \longrightarrow \langle K, {<}\mathit{clos}(\emptyset,[x_1,\ldots,x_n],e){>}\rangle \qquad \text{(PFun)}$$

$$\langle K, \texttt{letrec}\ \mathit{fdefs}\ \texttt{in}\ e\rangle \longrightarrow \langle K, e[\mathit{mk\_closlist}(\mathit{fdefs})]\rangle \qquad \text{(PLetRec)}$$

$$\langle K, v\rangle \longrightarrow \langle K, {<}v{>}\rangle \qquad \text{(PValue)}$$

$$\langle \mathit{id}(v_1,\ldots,v_n,\square) :: K, \square\rangle \longrightarrow \langle K, \mathit{eval}(\mathit{id},v_1,\ldots,v_n)\rangle \qquad (\text{if } \mathit{id} \neq \mathit{map}) \qquad \text{(PParams}_0\text{)}$$

$$\langle \mathit{id}(v_1,\ldots,v_{n-1},\square) :: K, {<}v_n{>}\rangle \longrightarrow \langle K, \mathit{eval}(\mathit{id},v_1,\ldots,v_n)\rangle \qquad \text{(PParams)}$$

$$\langle [\square|v_2] :: K, {<}v_1{>}\rangle \longrightarrow \langle K, {<}[v_1|v_2]{>}\rangle \qquad \text{(PCons)}$$

$$\langle \texttt{case}\ vs\ \texttt{of}\ ps\ \texttt{when}\ \square \to e^b; cl_2;\ldots;cl_n\ \texttt{end} :: K, {<}\texttt{'true'}{>}\rangle \longrightarrow \langle K, e^b\rangle \qquad \text{(PCaseTrue)}$$

$$\langle \texttt{let}\ {<}x_1,\ldots,x_n{>} = \square\ \texttt{in}\ e_2 :: K, {<}v_1,\ldots,v_n{>}\rangle \longrightarrow \langle K, e_2[x_1 \mapsto v_1,\ldots,x_n \mapsto v_n]\rangle \qquad \text{(PLet)}$$

$$\langle \texttt{do}\ \square\ e_2 :: K, {<}v_1{>}\rangle \longrightarrow \langle K, e_2\rangle \qquad \text{(PSeq)}$$

Figure A3. Frame stack semantics rules of group 3.

$$\langle \texttt{case}\ \square\ \texttt{of}\ \emptyset\ \texttt{end} :: K, vs\rangle \longrightarrow \langle K, \{error, if\_clause, \{\}\}^X\rangle \qquad (\textsc{ExcCase})$$

$$\langle K, \texttt{try}\ e_1\ \texttt{of}\ {<}x_1, \dots, x_n{>} \to e_2\ \texttt{catch}\ {<}x_{k+1}, \dots, x_{k+n}{>} \to e_3\rangle \longrightarrow \\ \langle \texttt{try}\ \square\ \texttt{of}\ {<}x_1, \dots, x_n{>} \to e_2\ \texttt{catch}\ {<}x_{k+1}, \dots, x_{k+n}{>} \to e_3 :: K, e_1\rangle \qquad (\textsc{STry})$$

$$\langle \texttt{try}\ \square\ \texttt{of}\ {<}x_1, \dots, x_n{>} \to e_2\ \texttt{catch}\ {<}x_{k+1}, \dots, x_{k+n}{>} \to e_3 :: K, {<}v_1, \dots, v_n{>}\rangle \longrightarrow \\ \langle K, e_2[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]\rangle \qquad (\textsc{PTry})$$

$$\langle \texttt{try}\ \square\ \texttt{of}\ {<}x_1, \dots, x_n{>} \to e_2\ \texttt{catch}\ {<}x_{k+1}, \dots, x_{k+3}{>} \to e_3 :: K, \{c, v^r, v^d\}^X\rangle \longrightarrow \\ \langle K, e_3[x_{k+1} \mapsto c, x_{k+2} \mapsto v^r, x_{k+3} \mapsto v^d]\rangle \qquad (\textsc{ExcTry})$$

$$\langle F :: K, \{c, v^r, v^d\}^X\rangle \longrightarrow \langle K, \{c, v^r, v^d\}^X\rangle \\ (\text{if } F \neq \texttt{try}\ \square\ \texttt{of}\ {<}x_1, \dots, x_n{>} \to e_2\ \texttt{catch}\ {<}x_{k+1}, \dots, x_{k+n}{>} \to e_3) \qquad (\textsc{ExcProp})$$

Figure A4. Frame stack semantics rules of group 4.

Appendix A.1. Parameter Lists

To avoid duplication of reduction rules for language elements involving parameter lists (value lists, tuples, maps, applications, inter-module calls, and primitive operations), we introduce parameter list frames Figure 5. These parameter lists are always evaluated in the same leftmost, innermost way, which is reflected in SPARAMS. If there are no more parameters to evaluate, we finish the evaluation with PPARAMS. Empty parameter lists are handled separately with PPARAMS$_0$ and SPARAMS$_0$, since if there are no parameters, there is no expression to be put into the second configuration cell of the semantics. This is also the reason, why □ is used as a redex in SAPPPARAM, SCALLPARAM, STUPLE, SPRIMOP.

Appendix A.2. Map Frames

Maps are handled in a special way with parameter list frames. To ensure some semantic properties, we have to make sure that the number of values and expressions in a *map* parameter list frame is an odd number, so that together with the current redex, a valid map expression can be reconstructed. This is the reason why empty maps are handled separately with PMAP$_0$.

Appendix B. Evaluation of Sequential Map

In this section, we show how to evaluate the sequential version of the *map* function (Figure 3). We denote the function inside `letrec` with map_{fun}, its body with e_b, and its closure with c_{map} (which is $clos([map_{fun}], [\texttt{F}, \texttt{L}], e_b)$), the entire expression with e_{letrec}, and the application of the *map* function closure with e_{app}. We use $(+1)$ to denote the increment function from the metatheory. As an example, we are going to use the following instantiation of the metavariables (ι is kept arbitrary):

$$\begin{aligned} e_l &:= \texttt{[1|[2|[]]]} \\ e_f &:= \texttt{fun}(\texttt{X}) \to \texttt{call 'erlang':'+'}(\texttt{X}, 1) \\ c_{inc} &:= clos(\emptyset, [\texttt{X}], \texttt{call 'erlang':'+'}(\texttt{X}, 1)) \end{aligned}$$

The steps to evaluate the sequential *map* function are all τ steps, except the very last one which is a $send(\iota_{base}, \iota, \texttt{[2|[3|[]]]})$ (supposing that the PID of the process evaluating the sequential *map* is ι_{base}). In the first steps, we evaluate the `letrec` expression with PLETREC, and next the parameters of the `call` with SCALLMOD, SCALLFUN, SCALLPARAM, SPARAMS$_0$ and SPARAMS (the latter two are general parameter list evaluation rules).

$$
\begin{array}{l}
\langle \varepsilon, e_{letrec} \rangle \longrightarrow \\
\langle \varepsilon, \texttt{call `erlang':`!'}(\iota, \texttt{apply } c_{map}(e_f, \texttt{[1|[2|[]]]})) \rangle \longrightarrow \\
\langle \mathit{call}(\square, \texttt{`!'})(\iota, \texttt{apply } c_{map}(e_f, \texttt{[1|[2|[]]]})) :: \varepsilon, \texttt{`erlang'} \rangle \longrightarrow \\
\langle \texttt{call } \square\texttt{:`!'}(\iota, \texttt{apply } c_{map}(e_f, \texttt{[1|[2|[]]]})) :: \varepsilon, <\texttt{`erlang'}> \rangle \longrightarrow \\
\langle \texttt{call `erlang':}\square(\iota, \texttt{apply } c_{map}(e_f, \texttt{[1|[2|[]]]})) :: \varepsilon, \texttt{`!'} \rangle \longrightarrow \\
\langle \texttt{call `erlang':}\square(\iota, \texttt{apply } c_{map}(e_f, \texttt{[1|[2|[]]]})) :: \varepsilon, <\texttt{`!'}> \rangle \longrightarrow \\
\langle \mathit{call}(\texttt{`erlang'}, \texttt{`!'})\square, \iota, \texttt{apply } c_{map}(e_f, \texttt{[1|[2|[]]]}) :: \varepsilon, \square \rangle \longrightarrow \\
\langle \mathit{call}(\texttt{`erlang'}, \texttt{`!'})(\square, \texttt{apply } c_{map}(e_f, \texttt{[1|[2|[]]]})) :: \varepsilon, \iota \rangle \longrightarrow \\
\langle \mathit{call}(\texttt{`erlang'}, \texttt{`!'})(\square, \texttt{apply } c_{map}(e_f, \texttt{[1|[2|[]]]})) :: \varepsilon, <\iota> \rangle \longrightarrow \\
\langle \mathit{call}(\texttt{`erlang'}, \texttt{`!'})(\iota, \square) :: \varepsilon, \texttt{apply } c_{map}(e_f, \texttt{[1|[2|[]]]}) \rangle
\end{array}
$$

We denote the current frame stack with K. Next, with the same idea, we evaluate the subexpressions of the application. For this, we use the following rules: SApp, SAppParam, SParams$_0$ and SParams (which were also used previously for the `call`'s parameter list). Note that the list `[1|[2|[]]]` evaluates to itself (i.e., it becomes a value) in multiple steps (with SConsTail, SConsHead, PCons while the integers in it evaluate with PValue).

$$
\begin{array}{l}
\langle K, \texttt{apply } c_{map}(e_f, \texttt{[1|[2|[]]]}) \rangle \longrightarrow \\
\langle \texttt{apply } \square(e_f, \texttt{[1|[2|[]]]}) :: K, c_{map} \rangle \longrightarrow \\
\langle \texttt{apply } \square(e_f, \texttt{[1|[2|[]]]}) :: K, <c_{map}> \rangle \longrightarrow \\
\langle \mathit{apply}(c_{map})(\square, e_f, \texttt{[1|[2|[]]]}) :: K, \square \rangle \longrightarrow \\
\langle \mathit{apply}(c_{map} :: K)(\square, \texttt{[1|[2|[]]]}), e_f \rangle \longrightarrow \\
\langle \mathit{apply}(c_{map})(\square, \texttt{[1|[2|[]]]}) :: K, <c_{inc}> \rangle \longrightarrow \\
\langle \mathit{apply}(c_{map})(\square) :: K, \texttt{[1|[2|[]]]} \rangle \longrightarrow^{*} \\
\langle \mathit{apply}(c_{map})(c_{inc}, \square) :: K, <\texttt{[1|[2|[]]]}> \rangle
\end{array}
$$

We denote the `case` clauses of the function in Figure 3 with cl_1, cl_2, respectively, and use e_{rec} to denote the body of the second clause, after the successful pattern matching, i.e.,

$$
[\texttt{apply } c_{inc}(\texttt{1}) | \texttt{apply } c_{map}(c_{inc}, \texttt{[2|[]]})].
$$

In the next steps, we evaluate the first application of c_{map}. In this case, the pattern matching of the first clause fails, and then the second succeeds. Thereafter, the guard `‘true’` is evaluated, followed by the body of the clause e_{rec}. Lists in Core Erlang evaluate right-to-left; thus, the next step is to evaluate the recursive application.

$$
\begin{array}{l}
\langle \mathit{apply}(c_{map})(c_{inc}, \square) :: K, <\texttt{[1|[2|[]]]}> \rangle \longrightarrow \\
\langle K, \texttt{case [1|[2|[]]] of } cl_1, cl_2 \texttt{ end} \rangle \longrightarrow \\
\langle \texttt{case } \square \texttt{ of } cl_1, cl_2 \texttt{ end} :: K, \texttt{[1|[2|[]]]} \rangle \longrightarrow \\
\langle \texttt{case } \square \texttt{ of } cl_1, cl_2 \texttt{ end} :: K, <\texttt{[1|[2|[]]]}> \rangle \longrightarrow \\
\langle \texttt{case } \square \texttt{ of } cl_2 \texttt{ end} :: K, <\texttt{[1|[2|[]]]}> \rangle \longrightarrow \\
\langle \texttt{case [1|[2|[]]] of [H|T] when } \square \rightarrow e_{rec}, cl_2 \texttt{ end} :: K, \texttt{`true'} \rangle \longrightarrow \\
\langle \texttt{case [1|[2|[]]] of [H|T] when } \square \rightarrow e_{rec}, cl_2 \texttt{ end} :: K, <\texttt{`true'}> \rangle \longrightarrow \\
\langle K, [\texttt{apply } c_{inc}(\texttt{1}) | \texttt{apply } c_{map}(c_{inc}, \texttt{[2|[]]})] \rangle \longrightarrow \\
\langle [\texttt{apply } c_{inc}(\texttt{1}) | \square] :: K, \texttt{apply } c_{map}(c_{inc}, \texttt{[2|[]]}) \rangle
\end{array}
$$

The recursive applications can be evaluated the same way as before; thus, we omit them in the following equations. We note that in the last recursive application, the first clause matches, terminating the recursion. In the following steps, we evaluate the first elements of the lists, which were kept in the frame stack, and then reassemble the result list. These steps involve the application of c_{inc} (the closure of the increment function), which we omit.

$$\begin{aligned}
&\langle[\texttt{apply}\ c_{inc}(\texttt{1})|\Box] :: K, \texttt{apply}\ c_{map}(c_{inc}, \texttt{[2|[]]})\rangle \longrightarrow^{*}\\
&\langle[\texttt{apply}\ c_{inc}(\texttt{2})|\Box] :: [\texttt{apply}\ c_{inc}(\texttt{1})|\Box] :: K, \texttt{[]}\rangle \longrightarrow\\
&\langle[\Box|\texttt{[]}] :: [\texttt{apply}\ c_{inc}(\texttt{1})|\Box] :: K, \texttt{apply}\ c_{inc}(\texttt{2})\rangle \longrightarrow^{*}\\
&\langle[\Box|\texttt{[]}] :: [\texttt{apply}\ c_{inc}(\texttt{1})|\Box] :: K, <\texttt{3}>\rangle \longrightarrow\\
&\langle[\texttt{apply}\ c_{inc}(\texttt{1})|\Box] :: K, <[\texttt{3}|\texttt{[]}]>\rangle \longrightarrow\\
&\langle[\Box|[\texttt{3}|\texttt{[]}]] :: K, \texttt{apply}\ c_{inc}(\texttt{1})\rangle \longrightarrow^{*}\\
&\langle[\Box|[\texttt{3}|\texttt{[]}]] :: K, <\texttt{2}>\rangle \longrightarrow\\
&\langle K, <[\texttt{2}|[\texttt{3}|\texttt{[]}]]>\rangle
\end{aligned}$$

Finally, the last step is to evaluate the message sending (for arbitrary set of linked processes L and process flag for trapping *exit* signals b):

$$\begin{aligned}
&(call(\texttt{`erlang'},\texttt{`!'})(\iota,\Box) :: \varepsilon, <[\texttt{2}|[\texttt{3}|\texttt{[]}]]>, q, L, b)\\
&\xrightarrow{send(\iota^{s},\iota,[\texttt{2}|[\texttt{3}|\texttt{[]}]])} (\varepsilon, <[\texttt{2}|[\texttt{3}|\texttt{[]}]]>, q, L, b)
\end{aligned}$$

We recall a theorem from previous work, saying that if a redex can be reduced in some steps, then this reduction can be done with arbitrary frame stack (continuation).

Theorem A1 (Extend frame stack)**.** *For all frame stacks K_1, K_2, K', redexes r_1, r_2, and step counters n, if $\langle K_1, r_1\rangle \longrightarrow^{n} \langle K_2, r_2\rangle$, then $\langle K_1 +\!\!+ K', r_1\rangle \longrightarrow^{n} \langle K_2 +\!\!+ K', r_2\rangle$.*

Proof. To prove this theorem, we use induction on the length of the reduction chain (n). The base case is discharged by the reflexivity of $\longrightarrow^{*}$, while in the second case, we inspect how the semantics can take the first step, and use the same rule in the conclusion, before using the induction hypothesis. □

We can use this idea to continue the evaluation without handling the `call` frame while evaluating the application. The reason we present the evaluation this way is that we can refer to the application evaluation from our bisimulation proofs (Example 2), independently of the current frame stack.

Next, we generalise this evaluation for any proper Core Erlang list and function expression.

Example A1 (Sequential *map* evaluation)**.** *Consider the expressions from Figure 3 (denoted with e_{letrec}). For all values v_l, v_f, value transforming functions f, PIDs ι which appear as the metavariables in Figure 3, we can prove*

$$\langle\varepsilon, \texttt{apply}\ c_{map}(v_f, v_l)\rangle \longrightarrow \langle\varepsilon, to\_obj(map(f, to\_meta(v_l)))\rangle,$$

if the following conditions hold:

- $\iota_{base} \neq \iota$.
- *The value v_f computes a metatheoretical function f, i.e., for all v values, we can prove $\langle\varepsilon, \texttt{apply}\ v_f(v)\rangle \longrightarrow^{*} <f(v)>$ in the sequential semantics.*
- *v_l is a proper Core Erlang list, i.e., it is constructed as $[v_1|[v_2|\ldots[v_n|[]]]\ldots]$, and $i < n$.*
- *v_l does not contain any PIDs, moreover, the application of v_f does not introduce any PIDs.*

Proof. We proved this example by induction on $to\_meta(v_l)$. In both cases, we just have to use the semantics rules to reach the result. In the inductive case, first, we evaluate the application of c_{map} for the first element of the list, then use the induction hypothesis (with the transitivity of $\longrightarrow^{*}$), and finish the evaluation by using the semantics rules. □

Appendix C. Our Results in the Coq Implementation

In Table A1, we connect the concepts, theorems and lemmas presented in the paper to the Coq implementation [29].

Table A1. Connection of our results to the Coq implementation.

Figure 1	`Syntax.v` - `Pat`, `Exp`, `Val` and `NonVal`
Figure 2	`Concurrent/ProcessSemantics.v` - `EReceive`
Figure 3	`Concurrent/MapPmap.v` - `map_clos`
Figure 4	`Concurrent/MapPmap.v` - `par_map`
Figure 5	`Syntax.v` - `Redex` and `Frames.v` - `FrameIdent` and `Frame`
Figure 6	`FrameStack/SubstSemantics.v` - `step`
Definition 1	`Concurrent/ProcessSemantics.v` - `Process`
Figure 7	`Concurrent/ProcessSemantics.v` - `removeMessage`, `peekMessage`, `recvNext`, and `mailboxPush`
Definition 2	`Concurrent/ProcessSemantics.v` - `Signal`
Definition 3	`Concurrent/ProcessSemantics.v` - `Action`
Figures 8–11	`Concurrent/ProcessSemantics.v` - `processLocalStep`
Definition 4	`Concurrent/NodeSemantics.v` - `ProcessPool`
Definition 5	`Concurrent/NodeSemantics.v` - `Ether`
Definition 6	`Concurrent/NodeSemantics.v` - `Node`
Figure 12	`Concurrent/NodeSemantics.v` - `interProcessStep`
Theorem 1	`Concurrent/NodeSemanticsLemmas.v` - `signal_ordering`
Example 1	The paths explored in the example are included in the proof of `Concurrent/MapPmap.v` - `map_pmap_empty_context_bisim`
Definition 7	`Concurrent/PIDRenaming.v`, `Concurrent/ProcessSemantics.v`, `Concurrent/NodeSemanticsLemmas.v` - Definitions with `renamePID` prefixes
Definition 8	`Concurrent/BisimRenaming.v` - `PIDsRespectNode`
Definition 9	`Concurrent/BisimRenaming.v` - `PIDsRespectAction`
Theorem 2	`Concurrent/NodeSemanticsLemmas.v` - `renamePID_is_preserved`
Definition 10	`Concurrent/NodeSemanticsLemmas.v` - We inline the uses of this definition based on `Concurrent/NodeSemanticsLemmas.v` - `compatiblePIDOf`
Theorem 3	`Concurrent/NodeSemanticsLemmas.v` - `confluence`
Theorem 4	`Concurrent/NodeSemanticsLemmas.v` - `internal_det`
Definition 11	This definition is always inlined in the code
Definition 12	`Concurrent/StrongBisim.v` - `strongBisim`
Definition 13	`Concurrent/WeakBisim.v` - `weakBisim`
Definition 14	`Concurrent/BarbedBisim.v` - `barbedBisim`
Theorem 5	`Concurrent/BarbedBisim.v` - `barbedBisim_refl`, `barbedBisim_sym`, and `barbedBisim_trans`
Theorem 6	`Concurrent/WeakBisim.v` - `strong_is_weak` and `Concurrent/BarbedBisim.v` - `weak_is_barbed`
Theorem 7	`Concurrent/BisimRenaming.v` - `rename_bisim`
Lemma 1	`Concurrent/BisimReductions.v` - `ether_update_terminated_bisim`
Lemma 2	`Concurrent/BisimReductions.v` - `terminated_process_bisim`
Theorem 8	`Concurrent/BisimReductions.v` - `silent_steps_bisim`
Example 2	`Concurrent/MapPmap.v` - `map_pmap_empty_context_bisim`

References

1. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1999; ISBN 0201485672.
2. Kim, M.; Zimmermann, T.; Nagappan, N. A field study of refactoring challenges and benefits. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Cary, NC, USA, 11–16 November 2012. [CrossRef]
3. Ivers, J.; Nord, R.L.; Ozkaya, I.; Seifried, C.; Timperley, C.S.; Kessentini, M. Industry experiences with large-scale refactoring. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022; pp. 1544–1554. [CrossRef]
4. Bagheri, A.; Hegedüs, P. Is refactoring always a good egg? exploring the interconnection between bugs and refactorings. In Proceedings of the 19th International Conference on Mining Software Repositories, Pittsburgh, PA, USA, 23–24 May 2022; pp. 117–121. [CrossRef]
5. Erlang/OTP Compiler, Version 24.0. 2021. Available online: https://www.erlang.org/patches/otp-24.0 (accessed on 15 May 2024).
6. Carlsson, R.; Gustavsson, B.; Johansson, E.; Lindgren, T.; Nyström, S.O.; Pettersson, M.; Virding, R. Core Erlang 1.0.3 Language Specification. 2004. Available online: https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf (accessed on 15 May 2024).
7. Brown, C.; Danelutto, M.; Hammond, K.; Kilpatrick, P.; Elliott, A. Cost-Directed Refactoring for Parallel Erlang Programs. *Int. J. Parallel Program.* **2014**, *42*, 564–582. [CrossRef]
8. Agha, G.; Hewitt, C. Concurrent programming using actors: Exploiting large-scale parallelism. In *Readings in Distributed Artificial Intelligence*; Bond, A.H., Gasser, L., Eds.; Morgan Kaufmann: Burlington, MA, USA, 1988; pp. 398–407. [CrossRef]
9. Erlang Documentation. Processes. 2024. Available online: https://www.erlang.org/doc/reference_manual/processes.html (accessed on 15 May 2024).
10. Bereczky, P.; Horpácsi, D.; Thompson, S. A Formalisation of Core Erlang, a Concurrent Actor Language. *Acta Cybern.* **2024**, *26*, 373–404. [CrossRef]
11. Bereczky, P.; Horpácsi, D.; Thompson, S. A frame stack semantics for sequential Core Erlang. In Proceedings of the 35th Symposium on Implementation and Application of Functional Languages (IFL 2023), Braga, Portugal, 29–31 August 2023. [CrossRef]
12. Fredlund, L.Å. A Framework for Reasoning About Erlang Code. Ph.D. Thesis, Mikroelektronik och Informationsteknik, Stockholm, Sweden, 2001.
13. Nishida, N.; Palacios, A.; Vidal, G. A reversible semantics for Erlang. In Proceedings of the International Symposium on Logic-Based Program Synthesis and Transformation, Edinburgh, UK, 6–8 September 2016; Hermenegildo, M.V., Lopez-Garcia, P., Eds.; Springer: Cham, Switzerland, 2017; pp. 259–274. [CrossRef]
14. Vidal, G. Towards symbolic execution in Erlang. In Proceedings of the International Andrei Ershov Memorial Conference on Perspectives of System Informatics, Petersburg, Russia, 24–27 June 2014; Voronkov, A., Virbitskaite, I., Eds.; Springer: Berlin/Heidelberg, Germany, 2015; pp. 351–360. [CrossRef]
15. Lanese, I.; Nishida, N.; Palacios, A.; Vidal, G. CauDEr: A causal-consistent reversible debugger for Erlang. In Proceedings of the International Symposium on Functional and Logic Programming, Nagoya, Japan, 9–11 May 2018; Gallagher, J.P., Sulzmann, M., Eds.; Springer: Cham, Switzerland, 2018; pp. 247–263. [CrossRef]
16. Lanese, I.; Sangiorgi, D.; Zavattaro, G. Playing with bisimulation in Erlang. In *Models, Languages, and Tools for Concurrent and Distributed Programming*; Boreale, M., Corradini, F., Loreti, M., Pugliese, R., Eds.; Springer: Cham, Switzerland, 2019; pp. 71–91. [CrossRef]
17. Harrison, J.R. Towards an Isabelle/HOL formalisation of Core Erlang. In Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang, Oxford, UK, 8 September 2017; pp. 55–63. [CrossRef]
18. Kong Win Chang, A.; Feret, J.; Gössler, G. A semantics of Core Erlang with handling of signals. In Proceedings of the 22nd ACM SIGPLAN International Workshop on Erlang, Seattle, WA, USA, 4 September 2023; pp. 31–38. [CrossRef]
19. Gustavsson, B. EEP 52: Allow Key and Size Expressions in Map and Binary Matching. 2020. Available online: https://www.erlang.org/eeps/eep-0052 (accessed on 15 October 2024).
20. Milner, R.; Parrow, J.; Walker, D. A calculus of mobile processes, I. *Inf. Comput.* **1992**, *100*, 1–40. [CrossRef]
21. Sangiorgi, D.; Milner, R. The problem of "weak bisimulation up to". In Proceedings of the CONCUR'92, Stony Brook, NY, USA, 24–27 August 1992; Cleaveland, W., Ed.; Springer: Berlin/Heidelberg, Germany, 1992; pp. 32–46.
22. Milner, R. *Communication and Concurrency*; Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1989.
23. Milner, R.; Sangiorgi, D. Barbed bisimulation. In Proceedings of the Automata, Languages and Programming, Wien, Austria, 13–17 July 1992; Kuich, W., Ed.; Springer: Berlin/Heidelberg, Germany, 1992; pp. 685–695. [CrossRef]
24. Bocchi, L.; Lange, J.; Thompson, S.; Voinea, A.L. A model of actors and grey failures. In *Coordination Models and Languages*; ter Beek, M.H., Sirjani, M., Eds.; Springer: Cham, Switzerland, 2022; pp. 140–158. [CrossRef]
25. Plotkin, G.D. *A Structural Approach to Operational Semantics*; Aarhus University: Aarhus, Denmark, 1981.
26. Felleisen, M.; Friedman, D.P. Control operators, the SECD-machine, and the λ-calculus. In Proceedings of the Formal Description of Programming Concepts—III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts—III, Ebberup, Denmark, 25–28 August 1986; pp. 193–222.

27. Pitts, A.M.; Stark, I.D. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*; Cambridge University Press: Cambridge, UK, 1998; pp. 227–273, ISBN 9780521631686.
28. Cesarini, F.; Thompson, S. *Erlang Programming*, 1st ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2009.
29. Core Erlang Formalization. 2024. Available online: https://github.com/harp-project/Core-Erlang-Formalization/releases/tag/v1.0.7 (accessed on 4 October 2024).
30. Amadio, R.M.; Castellani, I.; Sangiorgi, D. On bisimulations for the asynchronous π-calculus. *Theor. Comput. Sci.* **1998**, *195*, 291–324. [CrossRef]
31. Schäfer, S.; Tebbi, T.; Smolka, G. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In Proceedings of the Interactive Theorem Proving, Nanjing, China, 24–27 August 2015; Urban, C., Zhang, X., Eds.; Springer: Cham, Switzerland, 2015; pp. 359–374. [CrossRef]
32. Stdpp: An Extended "Standard Library" for Coq. 2024. Available online: https://gitlab.mpi-sws.org/iris/stdpp (accessed on 1 October 2024).
33. Sangiorgi, D. On the proof method for bisimulation. In Proceedings of the Mathematical Foundations of Computer Science 1995, Prague, Czech Republic, 28 August–1 September 1995; Wiedermann, J., Hájek, P., Eds.; Springer: Berlin/Heidelberg, Germany, 1995; pp. 479–488.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.