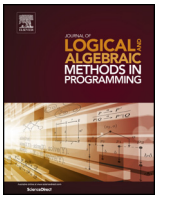
Journal of Logical and Algebraic Methods in Programming 107 (2019) 1–37

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



A core Erlang semantics for declarative debugging

Rafael Caballero [a,*], Enrique Martin-Martin [a], Adrián Riesco [a], Salvador Tamarit [b]

[a] Dpto. de Sistemas Informáticos y Computación, Fac. Informática, Universidad Complutense de Madrid, C/Profesor José García Santesmases, 9, 28040 Madrid, Spain

[b] Dept. de Sistemes Informàtics i Computació, Universitat Politècnica de València, Camí de Vera, s/n. 46022 València, Spain

ARTICLE INFO

Article history:
Received 22 October 2018
Received in revised form 15 March 2019
Accepted 1 May 2019
Available online 22 May 2019

ABSTRACT

One of the main advantages of declarative languages is their clearly established formal semantics, that allows programmers to reason about the properties of programs and to establish the correctness of tools. In particular, declarative debugging is a technique that analyses the proof trees of computations to locate bugs in programs. However, in the case of commercial declarative languages such as the functional language Erlang, sometimes the semantics is only informally defined, and this precludes these possibilities. Moreover, defining semantics for these languages is far from trivial because they include complex features needed in real applications, such as concurrency. In this paper we define a semantics for Core Erlang, the intermediate language underlying Erlang programs. We focus on the problem of concurrency and show how a medium-sized-step calculus, that avoids the details of small-step semantics but still captures the most common program errors, can be used to define an algorithmic debugger that is sound and complete.

1. Introduction

Declarative debugging, also known as *algorithmic debugging* [1], is a semi-automatic debugging technique that asks questions to an external oracle, usually the user, to obtain information about the validity of subcomputations that took place in an execution leading to an unexpected result. The computation is usually represented as a *debugging tree*, a data structure that omits the execution details to focus on the subcomputation results. In the case of declarative languages, this tree corresponds to some simplification of the *proof tree* that represents the erroneous computation in a formal semantic calculus. This close relation between the debugger and the semantics allows us to prove the soundness (any function identified as buggy really contains an error) and completeness (a buggy function is always found) of the approach, assuming that the oracle answers the questions appropriately.

More specifically, declarative debugging consists of two stages: tree generation and tree navigation. The first one builds the debugging tree by keeping those nodes from the proof tree that are related to relevant events (e.g. function calls). The rest of the inferences in the proof tree (e.g. substituting a variable by its corresponding value in the current context) can be discarded as far as we can prove that, if any of these inferences lead to an invalid result, then the source of the error is among the nodes we have decided to keep. During the second stage, the debugger traverses the debugging tree marking the nodes as valid or invalid according to the user answers, until a so-called *buggy node*, an invalid node with all its children valid, is found. The intuitive meaning of a buggy node is that, if "something" is producing an incorrect result from correct

* Corresponding author.
E-mail addresses: rafacr@ucm.es (R. Caballero), emartinm@ucm.es (E. Martin-Martin), ariesco@fdi.ucm.es (A. Riesco), stamarit@dsic.upv.es (S. Tamarit).

https://doi.org/10.1016/j.jlamp.2019.05.002
2352-2208/

subcomputations, then it contains an error. More detailed information about computing and traversing the tree and possible answers is available in [2], while a generalized approach to these concepts is presented in [3].

Since the number and the complexity of the questions asked to the user depend on the debugging tree, the calculus chosen to build this tree is very important. In this sense, it is usually easier to answer questions related to the final result obtained from a function call,[1] so *big-step semantics* are preferred when working in this field. Nonetheless, these semantics are not suitable when dealing with concurrent systems, because:

- Functions in concurrent programs can be non-terminating, so final values might not be reached in some cases. This is the case for example of servers, that are often coded as infinite loops that attend and dispatch messages.
- Even for terminating functions, it might be the case that, due to some bug, the function waits for some message that never arrives, and thus the function cannot finish and return a value.
- From the declarative debugging point of view, questions involving too many messages (it might be up to hundreds of messages) would be impossible to answer, making the approach unfeasible.

On the other hand, small-step semantics present to the user too specific fragments of the execution, which might be difficult to analyze and increases the number of questions beyond usability. This has prevented declarative debugging from being applied in general to concurrent programs.

We present in this paper a semantics for concurrent Core Erlang, the intermediate language used by Erlang, that solves these problems by introducing the notion of *medium-sized step*. A medium-sized step is the biggest step we can afford in a concurrent setting, because a declarative debugging approach based on this semantics:

- Asks meaningful questions that refer to steps that the user has in mind when designing the system (mainly transitions inside a process after a message is consumed).
- Reduces the number of questions with respect to small-step semantics.

The main contributions of this paper are:

1. We describe our medium-sized-step semantics in detail. We focus on the concurrent rules, since the rules corresponding to the sequential subset of Erlang are standard big-step semantics.
2. We prove the soundness and completeness of declarative debugging with respect to the semantics. In fact, this approach has been implemented in the Erlang Declarative Debugger EDD [4–6], available at https://github.com/tamarit/edd.
3. We prove the correctness of our medium-size-step semantics with respect to the semantics of Lanese et al. [7]. We choose this semantics because (i) it is used to debug and (ii) it is one of the most recent semantics published for Erlang.
4. We discuss other frameworks that would benefit from this semantics to formally prove their underlying foundations.

The rest of the paper is structured as follows: Section 2 presents our semantics and compares it with [7]. Section 3 relates the semantics with the declarative debugging schema, and describes the theory underlying the declarative debugger in [6]. Section 4 introduces some other applications that can take advantage of the theoretical foundations of our semantics. Section 5 discusses the extension of the language to support tuples, lists, anonymous functions, and higher-order, focusing on the syntactical and semantic changes, as well as their impact on the theoretical results. Finally, Section 6 concludes and presents some lines of future work.

This paper is related to [4–6] as they are publications devoted to the tool EDD in different phases of its development. Concretely, [4] presents EDD only for sequential Erlang programs, and [5] contains the technical details. Based on them, [6] presents the tool for concurrent Erlang programs, including a wide set of benchmarks assessing the scalability of the system but omitting technical details. The present paper fills that gap by presenting the semantic calculus used by EDD when debugging concurrent programs, its soundness and completeness w.r.t. other semantic calculus, and its adequacy for declarative debugging.

2. Syntax and semantic calculus

In this section we introduce the syntax of the simplified Erlang programs considered in this paper, as well as the proposed semantics calculus to evaluate them. We also prove the soundness and completeness of our semantics calculus with respect to the small-step Erlang semantics proposed in [7].

2.1. Syntax

We consider Core Erlang programs [8,9], a simplified version of Erlang programs without syntactic sugar that is used as an intermediate step during compilation. In order to keep the setting as simple as possible and improve readability, we

[1] As we will present later, declarative debugging can be used to debug other structures besides function calls. However, we focus on this simpler notion for the sake of simplicity.

$$
\begin{array}{rcl}
\mathit{module} & ::= & \texttt{module}\ \mathit{Atom} = \mathit{fun}_1\ \ldots\ \mathit{fun}_n \\
\mathit{fun} & ::= & \mathit{fname} = \texttt{fun}(\overline{X_n}) \rightarrow\ e \\
\mathit{fname} & ::= & \mathit{Atom} / \mathit{Integer} \\
v & ::= & \mathit{Atom} \mid \mathit{Integer} \mid \mathit{Float} \mid p \mid \mathit{fname} \\
vv & ::= & X \mid v \\
vp & ::= & X \mid p \\
e & ::= & X \mid v \mid \texttt{let}\ X = e_1\ \texttt{in}\ e_2 \mid \texttt{call}\ \mathit{fname}(\overline{vv_n}) \\
 & \mid & \texttt{apply}\ \mathit{fname}(\overline{vv_n}) \mid \texttt{case}\ vv\ \texttt{of}\ b_1\texttt{;}\ \ldots\ \texttt{;}\ b_n\ \texttt{end} \\
 & \mid & \texttt{spawn}(\mathit{fname}, [\overline{vv_n}]) \mid vp\ \texttt{!}\ vv \mid \texttt{receive}\ b_1\texttt{;}\ \ldots\ \texttt{;}\ b_n\ \texttt{end} \\
b & ::= & \mathit{pat}\ \texttt{when}\ e_1 \rightarrow e_2 \\
\mathit{pat} & ::= & X \mid v \\
 & & \\
\mathit{mnf} & ::= & v \mid \mathit{let\_receive} \\
\mathit{let\_receive} & ::= & \texttt{receive}\ b_1\texttt{;}\ \ldots\ \texttt{;}\ b_n\ \texttt{end} \mid \texttt{let}\ \mathrm{X} = \mathit{let\_receive}\ \texttt{in}\ e
\end{array}
$$

Fig. 1. Syntax of concurrent Core Erlang programs and *medium-sized normal forms*.

have focused on those elements of the language that provide concurrency, namely process creation, message submission, and message consumption. Therefore, we have omitted some features of Erlang like higher-order, anonymous functions, lists, tuples, and conditional expressions. Nevertheless, all these missing features can be integrated into the syntax and semantics without invalidating the theoretical results here, see Section 5 for more details.

Fig. 1 shows the syntax of Core Erlang programs, where the notation $\overline{o_n}$ represents a sequence of n elements of type o. A Core Erlang program consists of a module name followed by several function definitions. Each function definition has a name with arity (*fname*), a sequence of parameters ($\overline{X_n}$), and a body (e). Values (v) are atoms, integer and float numbers, process identifiers (p), and function names (*fname*). An expression e can be a variable, a value, a `let` expression, an invocation to a built-in function (`call`)[2] or to a function defined in the current module (`apply`), a `case` expression, the creation of a new process (`spawn`), a message submission (`!`), or a message reception (`receive`). Both `case` and `receive` expressions are composed by several branches b, each one with a pattern (*pat*), a guard (e_1), and a body (e_2). Notice that the parameters in our expressions are always variables or values (vv and vp). This type of syntax, known as *A-normal form* [10] (ANF), is different from the official Core Erlang syntax, where parameters can be any expression. The main reason for this decision is dealing with simpler expressions with a more predictable shape in our semantic calculus and the declarative debugger. This is not a limitation, as any Core program can be transformed to ANF by extracting the evaluation of parameters in chained `let` expressions. Indeed, we have checked that the Erlang/OTP compiler (version 21) applies this simplification in almost all the cases when processing Erlang programs. Additionally, ANF presents other benefits like simplifying some compiler optimizations [10], although it is not relevant in this work.

Fig. 1 also contains the definition of *mnf*, which stands for *medium-sized normal forms* (MNF in short), i.e., expressions that cannot progress either because they are already values or because they are expressions stuck in a `receive` expression that need a message to continue. These expressions play an important role in the semantic calculus, as we explain shortly.

Example 2.1. Fig. 2 presents a Core Erlang program for the concurrent version of the *Fibonacci* function, called `cfib` and implemented in Lines 1–19. It receives two arguments, a natural number standing for the Fibonacci number to be computed and the identifier of the process that required the computation. The function first distinguishes whether we are computing a base case (Lines 3–4) and it just needs to return the result to the parent, or it is a recursive case (Lines 5–18). In this case we compute some values that will be required latter (remember that functions are applied to completely reduced values); in particular, it stores the value of `self()` (Line 6), `M-1` (Line 7), and `M-2` (Line 9). It also makes two recursive calls inside `spawn` (Lines 8 and 10) and collects the results by means of two `receive` expressions (Lines 11–18). Finally, the final result is obtained by using the function `add` and storing the result in `Res`, which is sent to the parent (Line 16).

The function `add` should add the two values received as arguments but it is buggy and multiplies them. Hence, `cfib` is a buggy function that always returns `0` (except for the base case `1`). We will discuss in the upcoming sections how this function could be debugged.

2.2. Message order between processes

Before introducing our semantic calculus, we discuss a point that different Erlang semantics handle in several ways: the *arrival order* of the messages submitted among processes. The majority of semantics [11–14] consider that every process contains an inbox where incoming messages are stored. In these semantics, when a message m is sent to a process p, m is *immediately* enqueued into the inbox of process p. In *single-node* environments this behavior is acceptable, because the communication delay is insignificant. However, in a distributed system with multiple nodes, messages might suffer different delays. One of the fundamental ideas behind Erlang is that *message passing between a pair of processes is assumed*

[2] In the Core Erlang specification [9] the `call` keyword represents inter-module invocations, that can be built-in functions. As in [7], we assume only one module, so our `call` expressions are applied to built-in functions.

```
1  'cfib'/2 = fun (N,Parent) ->
2    case N of
3      0 when 'true' -> Parent ! 0
4      1 when 'true' -> Parent ! 1
5      M when 'true' ->
6         let Self = call 'self'/0() in
7         let N1 = call '-'/2(M, 1) in
8         let _ = call spawn ('cfib'/2, [N1,Self]) in
9         let N2 = call '-'/2(M, 2) in
10        let _ = call spawn('cfib'/2, [N2,Self]) in
11        receive
12          A when 'true' ->
13            receive
14              B when 'true' ->
15                let Res = apply 'add'/2(A, B) in
16                Parent ! Res
17            end
18        end
19   end
20
21 'add'/2 = fun (X,Y) -> call '*'/2(X, Y)
```

Fig. 2. Concurrent Fibonacci in Core Erlang.

to be ordered [15,16],[3] but the behavior of the rest of messages is not guaranteed, so any possibility must be supported by the semantics. This distributed situation is explained in detail in [19,7] by means of an example: suppose three processes p_1, p_2, and p_3, each one running in a different node. Process p_1 sends `hello` to p_2 *and then* sends `world` to p_3. Process p_3 simply resends to p_2 any message that it receives. The key question is: in this scenario, which message will arrive first to p_2? In a single node setting, if the communication delay is so insignificant that messages are delivered immediately, the message `hello` will arrive before `world`, but in a multi-node setting any of the two messages can be received first.

This problem has been solved in different semantics by using additional message queues. One possibility is to use a message queue per node, holding all messages currently "in transit" to that node [19]. Another possibility is considering a global mailbox [20,7] (also called *ether*) that stores all the sent messages before delivering them to the destination process. In all these semantics message submission and message delivery are separated in two different stages, therefore supporting some reordering of the messages. In our semantic calculus we use *outboxes* in each process, i.e., queues that store all the messages submitted by the process that have not been consumed yet. When a process needs to consume a message, it selects any process and looks *in order* for any message addressed to it. This allows us to keep the order of messages inside processes while supporting interleaving between messages from different processes. Outboxes are less restrictive than inboxes, where messages are ordered by their timestamp, while more restrictive than a global outbox, which supports any reordering of messages. The next subsection explains this solution in detail.

2.3. A calculus for concurrent Core Erlang

We propose the calculus called CEC (*Concurrent Erlang Calculus*) to evaluate sets of processes. These sets of processes are named *configurations* and are denoted as Π. Inside configurations, each process is represented by a tuple $\ll p, \langle e, \theta \rangle, l \gg$, where:

- p is the process identifier.
- e is the expression to be evaluated in the process.
- θ is a substitution mapping variables to values and standing for the *context* where e appears. We denote by id the empty substitution, and the operator $\uplus$ stands for the disjoint union of substitutions.
- l is the *outbox* with the messages sent by the process and not received by the addressee yet. It has the form $l \equiv [p_1 \,!\, v_1, \ldots, p_m \,!\, v_m]$, indicating the order of the messages, which is important in the case of repetitions of the same process identifier p_i in the list. We use the notation $|l|$ to represent the number of elements in an outbox list, $l + l'$ to indicate the concatenation, and $l|_p$ to indicate the restriction of l to the messages for process p.

The *CEC* calculus proves three kinds of statements:

1. $\Pi \Rightarrow \Pi'$: indicates that the configuration Π evolves into Π' by evaluating one processes in Π to its next medium-sized normal form (possibly creating new processes). Several statements $\Rightarrow$ can be combined in a single $\Rightarrow^+$ statement, as presented later.

[3] As noted in [17,7], this may not be guaranteed in current implementations; and is a controversial topic in the *erlang-questions* mailing list [18].

$$\text{(PROC)}\ \frac{\langle e,\theta\rangle \to (\langle mnf,\theta'\rangle, l', \Pi') \quad e \text{ is not MNF}}{\Pi\cdot \ll p, \langle e,\theta\rangle, l \gg \Rightarrow \Pi\cdot \ll p, \langle mnf,\theta'\rangle, l+l' \gg \cdot\Pi'}$$

$$\text{(CONSUME}_1\text{)}\ \frac{\langle mnf,\theta\rangle \xrightarrow{l'_2,j} (\langle mnf',\theta'\rangle, l', \Pi')}{\begin{array}{c}\Pi\cdot \ll p_1, \langle mnf,\theta\rangle, l_1 \gg \cdot \ll p_2, \langle mnf_2,\theta_2\rangle, l_2 \gg \Rightarrow \\ \Pi\cdot \ll p_1, \langle mnf',\theta'\rangle, l_1 + l' \gg, \ll p_2, \langle mnf_2,\theta_2\rangle, l''_2 \gg \cdot\Pi'\end{array}}$$

where $l'_2 \equiv l_2|_{p_1}$, $1 \le j \le |l'_2|$, i is the position of the jth message of l'_2 in l_2, and l''_2 is l_2 after removing the ith message.

$$\text{(CONSUME}_2\text{)}\ \frac{\langle mnf,\theta\rangle \xrightarrow{l',j} (\langle mnf',\theta'\rangle, l_1, \Pi')}{\Pi\cdot \ll p, \langle mnf,\theta\rangle, l \gg \Rightarrow \Pi\cdot \ll p, \langle mnf',\theta'\rangle, l'' + l_1 \gg \cdot\Pi'}$$

where $l' \equiv l|_p$, $1 \le j \le |l'|$, i is the position of the jth message of l' in l, and l'' is l after removing the ith message.

$$\text{(ENDLOCK)}\ \frac{\langle mnf_1,\theta_1\rangle \xrightarrow{l'_1,0} lock \quad \ldots \quad \langle mnf_k,\theta_k\rangle \xrightarrow{l'_k,0} lock}{\Pi \Rightarrow endlock}$$

where $k \ge 1$,

$$\Pi \equiv \ll p_1, \langle mnf_1,\theta_1\rangle, l_1 \gg, \ldots, \ll p_k, \langle mnf_k,\theta_k\rangle, l_k \gg, \ll p_{k+1}, \langle v_{k+1},\theta_{k+1}\rangle, l_{k+1} \gg, \ldots, \ll p_n, \langle v_n,\theta_n\rangle, l_n \gg$$

and $l'_i \equiv (l_1 + \ldots + l_n)|_{p_i}$ for $i = 1 \ldots k$.

$$\text{(Tr)}\ \frac{\Pi_0 \Rightarrow \Pi_1 \quad \Pi_1 \Rightarrow \Pi_2 \quad \ldots \quad \Pi_{n-1} \Rightarrow \Pi_n \quad n \ge 1}{\Pi_0 \Rightarrow^+ \Pi_n}$$

Fig. 3. Rules for evaluating configurations ($\Rightarrow$ and $\Rightarrow^+$).

2. $\langle e,\theta\rangle \to (\langle mnf,\theta'\rangle, l, \Pi)$, where e is not a medium-sized normal form. This medium step indicates that e with context θ has reached the medium-sized normal form mnf with context θ', sending the messages in l, and creating the new processes in Π.
3. $\langle mnf,\theta\rangle \xrightarrow{l,i} (\langle mnf',\theta'\rangle, l', \Pi)$: the medium-sized normal form mnf with context θ is evaluated to mnf' with context θ' after consuming the ith message from the outbox l. This statement indicates that the process has sent the messages stored in the outbox l' and created the new processes in Π.

A special case of the statements of type 1 is $\Pi \Rightarrow endlock$ and $\Pi \Rightarrow^+ endlock$, meaning that the final configuration is blocked in `receive` expressions with no possibility to continue, while the rest of processes have computed a value and finished. The third statement type also has a special case with the form $\langle mnf,\theta\rangle \xrightarrow{l,0} lock$, which indicates that no message from the outbox l can be consumed by mnf with context θ. Although the three statements occur in the calculus, only statements two and three are used in the debugging trees since the validity of statements of type 1 can be inferred from the validity of their premises.

Before presenting the rules of CEC, we will introduce three auxiliary operations related to message matching. The predicate $fails(mnf, l, \theta)$ accepts a medium-sized normal form mnf, a list of messages l, and a substitution θ. This predicate succeeds if the `receive` expression in mnf cannot consume any message in l considering the substitution θ for its free variables. On the other hand, the function $succeeds(mnf, l, j, \theta)$ accepts a medium-sized normal form mnf, a list of messages l, a position j, and a substitution θ. This function succeeds if the first message that the `receive` expression in mnf can consume is the jth message of l, considering the substitution θ for its free variables. It returns a pair (e,θ') containing the expression e that corresponds to the body of the first branch of the `receive` expression that matches the mentioned message, as well as the substitution θ' that results from extending θ with the matching substitution. Both functions, *fails* and *succeeds*, represent the evaluation of a `receive` Erlang expression (see Fig. 1) in our calculus, where all the available messages are tried in order of submission. Each message is matched against all the branches in the order they appear, and the first branch whose pattern matches the expression and the guard evaluates to `true` is selected, executing the corresponding body. Finally, the function $succeeds\_case(\theta, v, \overline{b_n})$ is used to evaluate `case` expression. It accepts a context substitution θ, a value v, and a sequence of branches $\overline{b_n}$, returning a pair (e',θ'). The expression e' is the body of the first branch matching v, and θ' is the substitution that extends θ with the matching substitution.

Fig. 3 presents the rules for evaluating configurations using $\Rightarrow$ and $\Rightarrow^+$, which focus mainly on message consumption. As we will see, when new processes are created they may not contain an MNF, so they are evaluated using rule (PROC) to reach it. From that moment, they will be evaluated using (CONSUME_1) or (CONSUME_2) to consume one message and reach its next MNF, which can be a final value. Additionally, locked configurations are detected and evolved to the special configuration *endlock*. More specifically:

$$(\mathsf{RCV}_1)\ \frac{\langle mnf,\theta\rangle \xrightarrow{l_1,j} (\langle v,\_\rangle, l, \Pi) \qquad \langle e,\theta''\rangle \to (\langle mnf',\theta'\rangle, l', \Pi')}{\langle \texttt{let}\ X = mnf\ \texttt{in}\ e,\theta\rangle \xrightarrow{l_1,j} (\langle mnf',\theta'\rangle, l + l', \Pi \cdot \Pi')}$$

where $\theta'' \equiv \theta \uplus \{X \mapsto v\}$.

$$(\mathsf{RCV}_2)\ \frac{\langle mnf,\theta\rangle \xrightarrow{l_1,j} (\langle mnf',\theta'\rangle, l, \Pi) \qquad mnf' \text{ is not a value}}{\langle \texttt{let}\ X = mnf\ \texttt{in}\ e,\theta\rangle \xrightarrow{l_1,j} (\langle \texttt{let}\ X = mnf'\ \texttt{in}\ e,\theta'\rangle, l, \Pi)}$$

$$(\mathsf{RCV}_3)\ \frac{succeeds(mnf,\ l,\ j,\ \theta) = (e',\theta') \qquad \langle e',\theta'\rangle \to (\langle mnf',\theta''\rangle, l', \Pi)}{\langle mnf,\theta\rangle \xrightarrow{l,j} (\langle mnf',\theta''\rangle, l', \Pi)}$$

$$(\mathsf{LOCK})\ \frac{fails(mnf,\ l,\ \theta)}{\langle mnf,\theta\rangle \xrightarrow{l,0} lock}$$

Fig. 4. Rules for consuming messages from queues ($\xrightarrow{l,j}$).

- (PROC) shows how configurations evolve when a process p is evaluating a non-MNF expression. The evaluation step yields a tuple $(\langle mnf,\theta'\rangle, l', \Pi')$, where the values l' and Π' refer to the list of messages sent and the new processes created during the computation step, respectively. These elements are integrated into the new configuration.
- ($\mathsf{CONSUME}_1$) and ($\mathsf{CONSUME}_2$) are employed when a process is evaluating an MNF expression and consumes a message from an outbox. ($\mathsf{CONSUME}_1$) indicates that this message is taken from the outbox l_2 of another process p_2, while ($\mathsf{CONSUME}_2$) illustrates the case where the message is taken from the outbox l of the same process that consumes the message (a so called *self* message). Notice that in both cases the outboxes are restricted to those messages addressed to the consuming process and that, also in both cases, the consumed message is removed from the outbox of the sender process.
- (ENDLOCK) characterizes a locked configuration. It requires that all the processes are finished, i.e., either they contain a value or are MNF expressions locally locked, that is, there are no messages in the whole configuration (note that l'_i collects all the messages addressed to p_i from all the outboxes) that can be accepted by the `receive` expression that must be evaluated next. There must be at least one process locally locked in order to apply this rule.
- (Tr) is the transitivity rule that combines inferences in order to evaluate the configuration by evolving processes and consuming messages an arbitrary number of times. This rule is relevant for declarative debugging because we need a debugging tree, and this rule generates the root $\Rightarrow^+$ that is the beginning of the debugging process.

It is important to note that the ($\mathsf{CONSUME}_*$) rules guarantee the order of messages between a pair of processes since they first select a process and then take in order all the messages submitted to p using the outbox restriction $l|_p$. Then the function *succeeds* processes the messages in order, selecting the first matching branch. This technique solves the problem shown in Section 2.2. In that scenario, a possible configuration after p_1 sends message `hello` to p_2, `world` to p_3, and p_3 consumes the message from the outbox of p_1 is:

$$\ll p_1, mnf_1, [p_2\ !\ \texttt{hello}] \gg \ll p_2, mnf_2, [\,] \gg \ll p_3, mnf_3, [p_2\ !\ \texttt{world}] \gg$$

In this configuration p_2 has two possible messages to consume, one in p_1 and one in p_3. Using ($\mathsf{CONSUME}_1$) p_2 can continue consuming any message, therefore consuming first `hello` and then `world` or vice versa.

Fig. 4 contains the rules for message consumption in a process evaluating an MNF within a certain context θ:
- (RCV_1) handles an MNF where the argument in the `let` expression is evaluated to a value after consuming the message. In this case, it continues the evaluation of the body by using the appropriate substitution θ'' until the next MNF.
- (RCV_2) evaluates an MNF where the argument in the `let` expression reaches another MNF that is not a value after consuming the message. Then, the `let` expression cannot be evaluated and no binding for X is generated.
- (RCV_3) acts once the `receive` expression inside an MNF is reached. Then, *succeeds* returns the body e' of the computed `receive` branch that accepts the jth message (otherwise this rule cannot be applied), with θ' as an extension of θ including the binding due to the matching. The right-hand side premise of the inference evaluates the expression until it reaches the next MNF.
- (LOCK) is in charge of checking that a particular process cannot receive any of the messages from l, i.e., it is locally blocked with respect to l.

Finally, Fig. 5 shows the rules of $\to$ that evaluate a process without consuming any message, including message submission and process creation. These rules evolve a process from an expression until it reaches the next MNF. Concretely:
- (MNF) evaluates an MNF expression to itself, without sending any message and creating no new processes.
- (VAR) reduces a variable to the value stored in the context θ. We assume that the original program is well-formed, so every variable used in any expression will have a value stored in the context.
- (APPLY) evaluates an invocation to a function defined in the current module. First we need to evaluate all arguments to values. As arguments $\overline{vv_n}$ are values or variables, they will be evaluated using (MNF) or (VAR) so the resulting context θ will not change and they will not send messages or create new processes. Then, it obtains a fresh variant of the rule of

$$\text{(MNF)}\ \frac{}{\langle mnf, \theta\rangle \to (\langle mnf, \theta\rangle, [\,], \emptyset)}$$

$$\text{(VAR)}\ \frac{}{\langle X, \theta\rangle \to (\langle \theta(X), \theta\rangle, [\,], \emptyset)}$$

$$\text{(APPLY)}\ \frac{\langle vv_1, \theta\rangle \to (\langle v_1, \theta\rangle, [\,], \emptyset) \quad \ldots \quad \langle vv_n, \theta\rangle \to (\langle v_n, \theta\rangle, [\,], \emptyset) \quad \langle e, \theta'\rangle \to (\langle mnf, \theta''\rangle, l, \Pi)}{\langle \texttt{apply}\ fname(vv_1, \ldots, vv_n), \theta\rangle \to (\langle mnf, \theta''\rangle, l, \Pi)}$$

where $fname = \texttt{fun}(\overline{X_n}) \to e$ fresh and $\theta' \equiv \theta \uplus \{\overline{X_n \mapsto v_n}\}$.

$$\text{(CALL)}\ \frac{\langle vv_1, \theta\rangle \to (\langle v_1, \theta\rangle, [\,], \emptyset) \quad \ldots \quad \langle vv_n, \theta\rangle \to (\langle v_n, \theta\rangle, [\,], \emptyset) \quad eval(fname, v_1, \ldots, v_n) = v}{\langle \texttt{call}\ fname(vv_1, \ldots, vv_n), \theta\rangle \to (\langle v, \theta\rangle, [\,], \emptyset)}$$

where *fname* is a built-in Erlang function.

$$\text{(CASE)}\ \frac{\langle vv, \theta\rangle \to (\langle v, \theta\rangle, [\,], \emptyset) \quad succeeds\_case(\theta, v, \overline{b_n}) = (e', \theta') \quad \langle e', \theta'\rangle \to (\langle mnf, \theta''\rangle, l, \Pi)}{\langle \texttt{case}\ vv\ \texttt{of}\ b_1; \ldots; b_n\ \texttt{end}, \theta\rangle \to (\langle mnf, \theta''\rangle, l, \Pi)}$$

$$\text{(LET}_1\text{)}\ \frac{\langle e_1, \theta\rangle \to (\langle v, \theta'\rangle, l, \Pi) \quad \langle e_2, \theta''\rangle \to (\langle mnf, \theta'''\rangle, l', \Pi')}{\langle \texttt{let}\ X = e_1\ \texttt{in}\ e_2, \theta\rangle \to (\langle mnf, \theta'''\rangle, l + l', \Pi \cdot \Pi')}$$

where $\theta'' \equiv \theta' \uplus \{X \mapsto v\}$.

$$\text{(LET}_2\text{)}\ \frac{\langle e_1, \theta\rangle \to (\langle mnf, \theta'\rangle, l, \Pi) \quad mnf \text{ is not a value}}{\langle \texttt{let}\, X = e_1\, \texttt{in}\, e_2, \theta\rangle \to (\langle \texttt{let}\, X = mnf\, \texttt{in}\, e_2, \theta'\rangle, l, \Pi)}$$

$$\text{(SPAWN)}\ \frac{\langle vv_1, \theta\rangle \to (\langle v_1, \theta\rangle, [\,], \emptyset) \quad \ldots \quad \langle vv_n, \theta\rangle \to (\langle v_n, \theta\rangle, [\,], \emptyset)}{\langle \texttt{spawn}(fname, [vv_1, \ldots, vv_n]), \theta\rangle \to (\langle p, \theta\rangle, [\,], \ll p, \langle e, \theta'\rangle, [\,] \gg)}$$

where $fname = \texttt{fun}(\overline{X_n}) \to e$ fresh, $\theta' \equiv \{\overline{X_n \mapsto v_n}\}$, and p is fresh.

$$\text{(BANG)}\ \frac{\langle vp, \theta\rangle \to (\langle p, \theta\rangle, [\,], \emptyset) \quad \langle vv, \theta\rangle \to (\langle v, \theta\rangle, [\,], \emptyset)}{\langle vp\,!\,vv, \theta\rangle \to (\langle v, \theta\rangle, [p\,!\,v], \emptyset)}$$

Fig. 5. Rules for evaluating an Erlang process ($\to$).

fname with new parameter variables $\overline{X_n}$. Finally, the function body e is evaluated using the context $\theta' = \{\overline{X_n \mapsto v_n}\}$ that maps parameters to values. The use of fresh variables is important to avoid possible collisions between variables defined in `let` expressions and function parameters.

- (CALL) is similar to (APPLY) but using a built-in function of the language, for example arithmetic or Boolean operators. Once all the arguments are evaluated to values, the special *eval* operator calculates the result of the built-in function We assume that built-in functions do not send messages or create new processes.

- (CASE) evaluates a `case` expression. First the expression vv is evaluated to a value, which is matched against the branches. The selected body branch e' is then evaluated to the next MNF using the extended context θ' that includes the matching substitution.

- (LET_1) is employed when the definition part in the `let` expression evaluates to a value. Then it binds the value to the appropriate variable, and continues by evaluating the body of the expression with the extended context. Notice that the messages sent in the definition part and the body are combined in $l + l'$, as well as the created processes in both parts ($\Pi \cdot \Pi'$).

- (LET_2) evaluates a `let` expression when the definition is evaluated to an MNF that is not a value. Hence, it just updates the expression of the definition part but does not continue with the body of the `let` expression.

- (SPAWN) reduces a *spawn* expression to the new process identifier, creating a new process as a side effect. As in rule (APPLY), the arguments $\overline{vv_n}$ must be evaluated to values before creating the new process, but these evaluations can only use rules (MNF) or (VAR) and therefore cannot create new processes or send any message. We also consider a fresh variant of the function definition to avoid collisions, and the new process starts with the context $\{\overline{X_n \mapsto v_n}\}$ mapping the parameters to values.

- (BANG) evaluates a message submission by evaluating the destination process vp to a process identifier p and the message body vv to a value v. Similarly to (APPLY) these derivations use (MNF) or (VAR), so they cannot create new processes or send messages. The expression is evaluated to the message body v, and the message $p\,!\,v$ is added to the outbox.

In the following, those statements of the form $\langle e, \theta\rangle \to (\langle mnf, \theta'\rangle, l, \Pi)$ or $\langle mnf, \theta\rangle \xrightarrow{l,i} (\langle mnf', \theta'\rangle, l', \Pi)$ will be called *evaluations* and denoted by $\mathcal{E}$. Additionally, we use $CEC \models_{(P,T)} \mathcal{E}$ to indicate that the evaluation $\mathcal{E}$ can be proven with respect to the program P with proof tree T in CEC, while $CEC \not\models_P \mathcal{E}$ indicates that $\mathcal{E}$ cannot be proven in CEC.

2.4. Soundness and completeness of the semantic calculus

There are several different formulations of Erlang semantics [11–14,19,7]. All of them are similar, showing differences only in the syntax of the language (full Erlang or Core Erlang) and the imposed guarantees on the order of the messages in distributed environments, as explained in Section 2.2. Moreover, all of them are small-step semantics, so they cannot be easily used for declarative debugging, as sketched in the introduction and further discussed in the next section. In order to state and prove the soundness and completeness of our semantic calculus we will focus on the semantics proposed in [7]. There are several reasons for this choice: it has been proposed recently, it supports a similar syntax, and it has been used for debugging Erlang programs [21] (concretely using causal-consistent reversibility).

The semantics presented in [7] considers *processes* of the form $\langle p, (\theta, e), q\rangle$, where the only difference with the processes defined in Section 2.3 is that q is the process mailbox, i.e., it contains the messages *sent* to the process. Since using process mailboxes can alter the message order in distributed environments, as noted in Section 2.2, the semantics in [7] uses a multiset Γ called *global mailbox* to store submitted messages that have not been yet delivered to the destination process mailbox. Therefore, a *system* in [7] is denoted by $\Gamma; \Pi$, where Π is a pool of processes[4] separated by the associative and commutative operator $|$.

The semantics in [7] defines a small-step relation $\hookrightarrow$ that performs one evaluation step on a system. This semantics presents some differences with respect to the semantic calculus presented in Section 2.3 that must be addressed in order to define and prove soundness and completeness:

- When extracting a message from the global mailbox to be inserted in a process mailbox, it does not keep the order of messages between pairs of processes because that information was lost when inserting in the global mailbox. Therefore, the global mailbox solves the lack of valid message reordering in distributed systems, but sacrifices the order of messages between a pair of processes. As a result, there will be $\hookrightarrow$-derivations that cannot be reproduced by $\Rightarrow^+$ because $\hookrightarrow$ delivers messages between processes in an unordered manner.
- When spawning a process with $\texttt{spawn}(fname, [\overline{v_n}])$, $\hookrightarrow$ adds a new process $\langle p, (id, \texttt{apply}\ fname(\overline{v_n})), [\ ]\rangle$ to the system. Our semantic calculus follows an alternative but equivalent approach: if $fname$ is defined as $\texttt{fun}(\overline{X_n}) \to e$ then the added process is $\ll p, \langle e, \{\overline{X_n \mapsto v_n}\}\rangle, [\ ] \gg$. Both expressions represent the same computation (one is one step ahead of the other), but this difference must be taken into account when translating between systems and configurations.
- The $\hookrightarrow$ relation uses internally a small-step relation $\xrightarrow{lb}$ that evaluates the expression inside a process, similar to the one presented in Fig. 5. This relation displays a label lb that can be τ, `self`, `receive`, `send`, or `spawn` together with the values involved. The labels `send` and `spawn` store the same information as the messages sent and processes spawned in a derivation $\langle e, \theta\rangle \to (\langle e', \theta'\rangle, l, \Pi)$. Therefore, this information must be linked in order to create $\hookrightarrow$-derivations from $\Rightarrow^+$ and vice versa.

Because of the first difference, our semantic calculus can be proved sound but not complete with respect to [7]. However, completeness can also be proved if we restrict how messages are delivered from the global mailbox in [7], as the authors did in a preliminary version of their semantics presented in [22]. Concretely, we will consider a small-step relation $\hookrightarrow_{mop}$ (from *message-order-preserving*) that is equal to $\hookrightarrow$ but restricts message delivery: instead of delivering any message from the global mailbox Γ, it first selects any pair of processes (p, p') and then takes from Γ the *oldest message from p to p'*.

In order to state the soundness and completeness results, we need a way to relate configurations and systems. This will be done by two translations: $\|\cdot\|$ from configurations to systems and $(\!|\cdot|\!)$ from systems to configurations.

Definition 2.1 *(Translation of configurations, $\|\cdot\|$).* A configuration

$$\Pi \equiv \ \ll p_1, \langle e_1, \theta_1\rangle, l_1 \gg \cdots \ll p_n, \langle e_n, \theta_n\rangle, l_n \gg$$

is translated into a system

$$\| \Pi \| = \bigcup_{i=1}^{n} \Gamma_i; \langle p_1, (\theta_1, e_1), q_1\rangle | \ldots | \langle p_n, (\theta_n, e_n), q_n\rangle$$

where $(l_1 + \ldots + l_n)|_{p_i} = \Gamma_i \cup\ q_i$, i.e., the messages sent to process p_i are divided between the global mailbox (through Γ_i) and the local mailbox q_i. Additionally, these local mailboxes q_i are *message-order-preserving*: a) For every process p, all the

[4] We overload notation and use Π to represent a pool of processes of [7] and also configurations as defined in Section 2.3. The context will solve any ambiguity.

messages in any q_i submitted by p are older than the messages from p in Γ, and b) for every process p, the messages from p in any mailbox q_i appear in the order they were sent. Notice that, as any message-order-preserving split of messages between Γ_i and q_i is valid, the translation of a configuration is not unique.

For the translation $(\!|\cdot|\!)$ from systems to configurations we will need some additional information in order to reliably represent the same information. First, we need to reconstruct outboxes from the messages in the global and local mailboxes. Therefore, we will assume that every message $p!v$ in a global or local mailbox is decorated with the sender process p_s and an incremental timestamp t indicating the moment when it has been sent: $p!v^t_{p_s}$. Second, we need to detect those processes newly spawned, as they will contain an `apply` expression in systems but the body of the spawned function in configurations. Therefore, we will also assume that newly created processes in systems are decorated with a $*$ mark—$\langle p, (\theta, \texttt{apply}^* fname(\overline{v_n})), [\ ]\rangle$—which disappears as soon as they are evaluated one step. Considering these decorations we can define $(\!|\cdot|\!)$ as:

Definition 2.2 *(Translation of systems, $(\!|\cdot|\!)$).* The translation of a system $(\!|\Gamma; \langle p_1, (\theta_1, e_1), q_1\rangle | \ldots | \langle p_1, (\theta_n, e_n), q_n\rangle|\!)$ is defined as

$$\ll p_1, \langle e'_1, \theta'_1\rangle, l_1 \gg \cdots \ll p_n, \langle e'_n, \theta'_n\rangle, l_n \gg$$

where:

- l_i contains those messages sent by p_i that occurs in Γ and $\overline{q_n}$, sorted by timestamp;
- if $e_i = \texttt{apply}^* fname(\overline{v_n})$ then $e'_i = e$ and $\theta'_i = \{\overline{X_n \mapsto v_n}\}$, where $fname = \texttt{fun}(\overline{X_n}) \to e$;
- if $e_i \neq \texttt{apply}^* fname(\overline{v_n})$ then $e'_i = e_i$ and $\theta'_i = \theta_i$.

Intuitively, this translation combines all the messages sent by a process that are in Γ or in local mailboxes and generates its outbox (ordered by sent time) and adapts the expression and context in the case of new processes. Note that, unlike $\| \cdot \|$, the translation of a system is unique.

Before stating the soundness and completeness results, we need to define some particular types of systems that play an important role:

Definition 2.3. A system

$$\Gamma; \Pi = \langle p_1, (\theta_1, e_1), q_1\rangle | \ldots | \langle p_n, (\theta_n, e_n), q_n\rangle$$

is called:

- *blocked*, if it cannot be evaluated using $\hookrightarrow_{mop}$ but at least one e_i is not a value.
- *empty-mailbox*, if every $q_i = [\ ]$.
- *medium-sized normal form (MNF)*, if every e_i is in MNF. A particular case of an *MNF* system is a *finished* system, identified because every e_i is a value.

Now we can present the soundness of $\Rightarrow^+$ with respect to $\hookrightarrow_{mop}$, that states that any step $\Pi_0 \Rightarrow \Pi_1$ can be mimicked by several steps $\hookrightarrow_{mop}$ from the translated initial configuration, reaching a translated destination configuration. If the destination configuration is the special term *endlock*, then it is possible to perform a $\hookrightarrow_{mop}$-derivation to a blocked system from the translated initial configuration. The proof of this theorem can be found in Appendix A.

Theorem 2.1 *(Soundness of $\Rightarrow^+$ w.r.t. $\hookrightarrow$). Consider a derivation $\Pi_0 \Rightarrow^+ \Pi_1$ and an empty-mailbox system $\Gamma; \Pi$ such that $\| \Pi_0 \| = \Gamma; \Pi$. Then:*

a) $\Gamma; \Pi \hookrightarrow^+_{mop} \| \Pi_1 \|$, *if* $\Pi_1 \neq$ *endlock*
b) $\Gamma; \Pi \hookrightarrow^*_{mop} \Gamma'; \Pi'$ *such that* $\Gamma'; \Pi'$ *is blocked, if* $\Pi_1 =$ *endlock*

Note that restricting $\Gamma; \Pi$ to be an empty-mailbox implies that the translation $\| \Pi_0 \|$ is deterministic, as all non consumed messages are placed in the global mailbox. It is important to start the $\Rightarrow^+$-derivation from an empty-mailbox system so that the $\hookrightarrow_{mop}$ trace could deliver messages from different processes in an order compatible with the original derivation. Starting from arbitrary message-order-preserving systems could prevent some `receive` expressions from consuming the appropriate message because there is a previous message from a different process that matches with the branches. On the other hand, the final system $\| \Pi_1 \|$ can contain messages in the local mailboxes provided they are message-order-preserving.

Similarly, we can prove a completeness result stating that any $\hookrightarrow_{mop}$ derivation reaching a MNF system can be reproduced with $\Rightarrow^+$:

Theorem 2.2 *(Completeness of $\Rightarrow^+$ w.r.t. $\hookrightarrow$). If $\Gamma; \Pi \hookrightarrow^+_{mop} \Gamma'; \Pi'$ and $\Gamma'; \Pi'$ is a MNF system then $(\!|\Gamma; \Pi|\!) \Rightarrow^+ (\!|\Gamma'; \Pi'|\!)$ or $(\!|\Gamma; \Pi|\!) = (\!|\Gamma'; \Pi'|\!)$.*

Our semantic calculus advances processes to their next MNF, and $\hookrightarrow_{mop}$ can perform a small step on any process, reaching non MNF expressions like `let X=3 in X` or `apply add/2(1,2)`. Therefore our completeness result is restricted to derivations finishing in MNF systems, as the granularity of our semantic calculus could not reproduce those small steps if they do not end in an MNF expression. The case $(\!|\Gamma; \Pi|\!) = (\!|\Gamma'; \Pi'|\!)$, although surprising at first, is a situation that appears naturally: a sequence of $\hookrightarrow_{mop}$ steps that do not evaluate any process expression but delivers messages from Γ to local mailboxes. This derivation reaches an MNF system if the original system was in MNF, but the steps do not change the translation of the system, so $(\!|\Gamma; \Pi|\!) = (\!|\Gamma'; \Pi'|\!)$. Another aspect to mention with respect to the completeness result is that the syntax considered in this paper is smaller than the one used in [7]. Concretely, we consider Core Erlang programs in ANF (all the parameters are values or variables instead of general expressions, see Section 2.1) and we do not support tuples or lists. This is not a severe limitation, as any Core Erlang program in [7] without tuples and lists can be converted to our syntax by extracting the evaluation of the parameters in `let` expressions. Moreover, evaluating an ANF system using $\hookrightarrow_{mop}$ generates a new ANF system.

Additionally, from the completeness result in Theorem 2.2 we can extract an immediate corollary about finished and blocked derivations, as in both cases the final system is MNF:

Corollary 2.1. *Consider a derivation $\Gamma; \Pi \hookrightarrow^+_{mop} \Gamma'; \Pi'$.*

- *If $\Gamma'; \Pi'$ is finished, then there is a derivation $(\!|\Gamma; \Pi|\!) \Rightarrow^+ (\!|\Gamma'; \Pi'|\!)$.*
- *If $\Gamma'; \Pi'$ is blocked, then there is a derivation $(\!|\Gamma; \Pi|\!) \Rightarrow^+$ endlock.*

As a final comment, our semantic calculus would be sound and complete with respect to $\hookrightarrow$ in [7] if we relax the restriction on the order of messages between processes. This change would require modifying the *succeeds* function so that it could match a message with any from a list of messages, ignoring their order. The proofs in Appendix A would be easily adapted as every $\hookrightarrow_{mop}$ step is a $\hookrightarrow$ step (for soundness) and the freedom when delivering messages would be the same as in the relaxed *succeeds* function.

3. Declarative debugging of concurrent Erlang programs

In this section we first relate declarative debugging with the calculus in the previous section. Then, we formally present the theoretical foundations of EDD, a declarative debugger for concurrent Erlang programs [4–6].

3.1. Relation between declarative debugging and the semantics

Let us recall the intuitions on how declarative debugging works; we present the formal details below in this section. Declarative debuggers first generate a debugging tree corresponding to an erroneous computation and then ask the user about the subcomputations in the tree; the user compares these computations with the results he/she expected and marks the nodes as correct/incorrect. The aim is to find a *buggy node*, an incorrect node with all its children correct, which points out an error in the code. It is important to note that defining a debugging framework is a subtle task, even when a semantics is available. From the practical point of view, the following issues must be addressed:

1. Decide the errors the debugger will detect. In our case we decided the debugger should detect *wrong functions* and *wrong* `receive` *expressions*.
2. Choose what nodes should correspond to questions. Although it would be possible to use the CEC proof tree corresponding to an erroneous computation as debugging tree, it would contain many nodes whose correctness does not depend on the user's code and, hence, it does not make sense to ask about them. For example, consider the rule (BANG) in Fig. 5: intuitively, the conclusion can only be wrong if one of the premises is wrong, and hence this node will never be buggy. On the other hand, consider the rule (APPLY): it might be the case that a function call does not return the expected result but the body executes as expected. Take the `add` function from Example 2.1: the result of `add(1,1)` returns `1`, which is unexpected, although the execution of the body, `1 * 1`, returns 1, which is expected; in this case we would say that `add` is a wrong function.
   In our case we keep those nodes related to the inference rules (APPLY), (RCV_1), (RCV_3), and (LOCK). When an (APPLY) node is buggy it will point out a *wrong function*, while the rest of nodes will point out *wrong* `receive` *expressions*.

Example 3.1. Fig. 6 depicts a simplified debugging tree for the erroneous call `cfib(2, self())`, assuming the implementation shown in Example 2.1. We omit some parts of the nodes to make them readable; in particular, we omit substitutions

$$\frac{T_1 \quad T_2 \quad T_3 \quad T_4 \quad T_5}{\ll p_1, \texttt{cfib}(2, p_0), [\,] \gg \Rightarrow^{+} \ll p_1, 0, [p_0 \,!\, 0] \gg \ll p_2, 0, [\,] \gg \ll p_3, 1, [\,] \gg}$$

where :

$$T_1 \equiv \quad \frac{}{\texttt{cfib}(2, p_0) \to (\texttt{receive A...end}, [\,], (p_2, p_3))}$$

$$T_2 \equiv \quad \frac{}{\texttt{cfib}(0, p_1) \to (0, [\texttt{p}_1 \,!\, 0], \emptyset)}$$

$$T_3 \equiv \quad \frac{}{\texttt{cfib}(1, p_1) \to (1, [\texttt{p}_1 \,!\, 1], \emptyset)}$$

$$T_4 \equiv (\mathsf{RCV}_3)\frac{}{\texttt{receive A...end} \xrightarrow{[0],0} (\texttt{receive B...end}, [\,], \emptyset)}$$

$$T_5 \equiv (\mathsf{RCV}_3)\frac{(\mathsf{APPLY})\ \dfrac{}{\texttt{add}(0,1) \to (0, [\,], \emptyset)}}{\texttt{receive B...end} \xrightarrow{[1],0} (0, [p_0 \,!\, 0], \emptyset)}$$

Fig. 6. Simplified debugging tree for `cfib(2, p0)` (top), and the corresponding subtrees.

and the calls in `spawn` expressions. Moreover, we only depict those inference rules that might reveal bugs in the code. In the following we present the details of the tree.

The tree is presented on the top of the figure, where the root shows how the configuration evolved from the initial call (that we assume takes place in a process identified by p_1) to the final configuration, where more processes have been created and a message is in the outbox of process p_1 (two other messages were sent and consumed, as we will see). The first premise, that corresponds to an (APPLY) inference rule, indicates that the expression in the initial call is evaluated until it reaches the first `receive` expression. In this evaluation it creates two new processes, p_2 and p_3, that are required to evaluate `fib(0, p1)` and `fib(1, p1)`, respectively. The next two premises, that also correspond to an (APPLY) inference rule, evaluate the expressions in the new processes, sending a message to p_1 and reaching a final value (which corresponds to the sent value).

The next premise corresponds to the tree depicted as T_4, which appears in the middle of the figure. It corresponds to the evaluation of the `receive` expression reached in the first premise, which was waiting for a message (we assume the consumed message is `0`, sent from p_2). Note that the root of this premise corresponds to a (RCV_3) inference rule that directly evaluates the `receive` expression.

Lastly, the premise T_5 is shown in the bottom of the figure. It follows a similar pattern as T_4 but in this case the (RCV_3) inference rule has an extra premise, corresponding to the evaluation of `add(0,1)`. It is easy to see that all nodes in T_5 are incorrect, because they correspond to the end of the evaluation of `fib(2, p0)` but they return `0`. In particular, the node corresponding to the evaluation of `add(0,1)` is erroneous and, hence, buggy (it has no children), so it points out `add` as a wrong function.

It is important to note that this debugging tree is much smaller than the proof tree, that would contain many nodes that are not related to the debugging process.

As the example above illustrates, note that (i) we need to keep exactly those nodes that reveal the errors previously chosen and (ii) the nodes shape the questions. In our case the questions have the form:

- *Is this evaluation correct?* Given a function and its arguments, the user must decide whether the medium-sized normal form reached, the sent messages, and the created processes are correct. This question corresponds to an (APPLY) node.
- *Is this transition correct?* Given a function waiting in a `receive` expression (possibly nested in a `let` structure), its context, and a list of messages sent by a certain process (possibly itself), the user must decide whether the reached medium-sized normal form, the sent messages, and the created processes are correct. This question corresponds to a (RCV_1) node.
- *Is this `receive` expression correctly executed?* Given a `receive` expression, its context, and a list of messages sent by a certain process (possibly itself), the user must decide whether the correct message has been consumed, the correct branch has been taken, and the correct values (medium-sized normal form, messages sent, and processes created) have been obtained. This question corresponds to a (RCV_3) node.
- *Did you expect to reach a deadlock?* Given a `receive` expression, its context, and a list of messages sent to this process by all processes (including itself), the user must decide whether it is correct that no message can be consumed. This question corresponds to a (LOCK) node.

More details about these questions are available in [6]. Note the importance of the semantics here: a small-step semantics would not allow us to ask, for example, about the evaluation of functions beyond one step. For this reason, we require a semantics with inference rules that have the appropriate abstraction level, so buggy nodes point out relevant errors while the questions are not too difficult to answer.

Discussion: would it be possible to implement a declarative debugger by transforming a small-step semantics proof tree? In fact, some debuggers rely in tree transformations that put together several steps to improve the questions asked to the user. This kind of transformation has been used in general to deal with laziness issues [23,24], so they take a valid proof tree with respect to the semantics and return another valid tree in the same semantics. Other transformations of this kind (e.g. [25]) keep some nodes that are, in general, useless for the debugging process, but that can point out bugs by "accumulating" the errors from their children. However, the problem in our case needs a more complex solution than the proposed for the transformations above, because it requires to introduce nodes that do not correspond to inference rules in the semantics. For this reason, for each new node used in the debugging process we would need to prove:

- The reached values are appropriately propagated. In our case we would need to prove that the reached medium-sized normal form can be reached in a stand-alone way (it might be the case that other processes were executed and interacted with the one executing the expression, which would be inconvenient for the debugging process), while the sent messages and the created processes of each step are adequately put together.
- The premises are appropriately placed. As explained above, declarative debugging tries to find a buggy node, an incorrect node with all its children correct. If the premises are not correctly relocated the debugger might identify as buggy a node that in fact has an incorrect child somewhere else.
- If found buggy, the node points out a particular error. If too many small steps are put together, it might be the case that the error in the node is due to more than one bug, hence worsening the granularity of the debugging process.
- The question related to the node is not too difficult. Putting together too many small steps also might make the corresponding questions more difficult to answer, which in practice prevents users from using the tool.

Note that this transformation is not easy because small steps related to different processes can be interleaved. Moreover, new transformations (and their corresponding proofs) are required if the designer wants to find new errors. Finally, it is not sure that all errors can be replicated via a transformation. For this reason, we consider that it is worth designing a semantics that naturally fits the declarative debugging scheme.

Discussion: why not using a big-step semantics? As we sketched in the introduction, big-step semantics have not been designed for Erlang because some processes might not terminate. Moreover, after the declarative debugging description above it is also apparent that a big step semantics would produce too difficult questions: it would require the user to answer evaluations where many messages are received, only some of them are processed, many others are sent, several processes are created, and a final result is finally reached. To answer these questions the user would in fact divide the computation into smaller steps, that we consider would roughly correspond to our medium-sized steps.

3.2. Intended interpretations

Thus far we have presented the intuitions behind declarative debugging, but we have not defined formally when a node is *correct* or *incorrect*. To define these ideas, we need first the concept of *intended interpretation*, represented as $\mathcal{I}$, which corresponds to the behavior expected by the oracle (in the following, the user) for the questions posed by the debugger. Based on the decisions we took above about the nodes that might point out errors in the Erlang code, this intended interpretation requires information about user functions and user-defined `receive` expressions. In the following we denote by f a user function, by r a `receive` expression, and by p either f or r.

We assume that the user knows the expected behavior of these pieces. This behavior is then extrapolated to complete programs by a suitable calculus *ICEC* presented in this section. The discrepancies between the results produced by the program, represented by *CEC* and the results expected by the user, represented by *ICEC*, define the buggy parts of the program.

The intended interpretation of a program P is defined as the union of two sets:

$$\mathcal{I} = \mathcal{I}_{fun} \cup \mathcal{I}_{rcv}$$

The first set defines the expected values for function calls and has the form:
$\mathcal{I}_{fun} = \{\ldots,\ \langle f, \theta\rangle \to (mnf, l, \Pi),\ \ldots\}$ where

- f is a function defined as `f/n = fun(`X_1`, ... ,` X_n`)` $\to e$.
- The domain of the substitution θ must be $\{X_1, \ldots, X_n\}$.
- l is the list of messages sent by this function.
- Π is the set processes created by this function.

Thus, $\mathcal{I}_{fun}$ contains the results expected for any possible function call to program functions. This notion can be easily extended to anonymous functions, which are not discussed here for simplicity but will be considered in Section 5. The other set that is part of $\mathcal{I}$ refers to the intended behavior of `receive` expressions and it is a set union of the form:

$$\mathcal{I}_{rcv} = \bigcup_{\langle\, p.r,\, l,\, \theta\,\rangle} \mathcal{I}_{\langle\, p.r,\, l,\, \theta\,\rangle}$$

with p any program piece of code, r any `receive` statement reachable from p during a computation, θ a variable substitution, and l a list of incoming messages. $\mathcal{I}_{\langle\, p.r,\, l,\, \theta\,\rangle}$ represents the expected behavior of p when stopped in r and in presence of the incoming messages l. As usual, θ represents the context, that is, the binding of variables in $p.r$ occurred so far.

Assuming that l is the incoming list for the process computing $p.r$, we can distinguish two possible forms for each set $\mathcal{I}_{\langle\, p.r,\, l,\, \theta\,\rangle}$:

1. $Ipr = \{\mathcal{I}fails(r, l, \theta)\}$ if the user expected the process to be blocked.
2. $Ipr = \{\mathcal{I}succeeds(p.r,\ l,\ j,\ \theta) \to (\langle mnf, \theta'\rangle, l', \Pi)\}$ if the computation of p is expected to continue consuming the jth message of l, reaching the new medium-sized normal form mnf with new context θ', and producing in between the list of output messages l' and the new processes Π. It is worth noticing that l is a list of input messages to this process of the form $[v_1, \ldots, v_n]$, while l' is an output box list of the form:

$$l' \equiv [p_1!v_1, \ldots, p_m!v_m]$$

Given that the intended interpretation is potentially infinite, it might seem unfeasible for the user to keep it in mind. Note, however, that in practice he/she is only required to answer questions related to the particular computations that took place to obtain the current result, which greatly reduces the size and the complexity of the intended interpretation in practice.

Example 3.2. The intended interpretation for the program in Example 2.1 would consist of the infinite set of all possible evaluations and transitions for `cfib(N, Parent)` and `add(X, Y)`, for `N`, `X`, and `Y` natural numbers and `P` a process identifier, and for the two `receive` expressions in `cfib` in all the possible contexts. For example, $\langle \mathtt{add},\ [X \mapsto 7, Y \mapsto 12]\rangle \to (19, [\ ], \emptyset)$ and $\langle \mathtt{cfib},\ [N \mapsto 1, Parent \mapsto p_1]\rangle \to (1, [p_1\ !\ 1], \emptyset)$ would be part of the intended interpretation but $\langle \mathtt{add},\ [X \mapsto 0, Y \mapsto 1]\rangle \to (0, [\ ], \emptyset)$ would not be part of it.

However, as shown in Example 3.1, the user is only required to have in mind a reduced number of elements to answer the questions related to a particular debugging session (in that case, the 6 elements corresponding to the 6 nodes in the tree, excluding the root).

3.3. Intended semantic calculus

The validity of the nodes in a *CEC*-proof tree is obtained by defining the *intended interpretation calculus*. This calculus, called *ICEC*, is defined as follows:

Definition 3.1. The calculus *ICEC* contains the same inference rules as *CEC*, prefixing the label of each rule by $\mathcal{I}$ to avoid confusion. The definitions of the rules are also the same in both calculi, except by:

1. ($\mathcal{I}$APPLY), which adds the following additional side condition to the *CEC* rule (APPLY):

$$\langle f, \theta'\rangle \to (mnf, l, \Pi) \in \mathcal{I}$$

   where θ' is defined in (APPLY) as the substitution binding the variables in the definition of the function f to the values computed for the arguments.
2. ($\mathcal{I}\mathsf{RCV}_1$) also adds to ($\mathsf{RCV}_1$) a new side condition of the form

$$\mathcal{I}succeeds(p.r,\ l_1,\ j,\ \theta) \to (\langle mnf', \theta'\rangle, l + l', \Pi \cdot \Pi') \in \mathcal{I}$$

   where:
   - $(\langle mnf', \theta'\rangle, l + l', \Pi \cdot \Pi')$ is the conclusion of the inference rule (RCV_1) in Fig. 4.
   - r is the leftmost, innermost `receive` in the left-hand side of the conclusion of the inference rule (RCV_1).
   - p is the smallest (closest) piece of code (either a function or a `receive` statement) that contains the `let` statement.
3. ($\mathcal{I}\mathsf{RCV}_3$) adds to ($\mathsf{RCV}_3$), the new side condition:

$$\mathcal{I}succeeds(r.r,\ l,\ j,\ \theta) \to (\langle mnf', \theta''\rangle, l', \Pi) \in \mathcal{I}.$$

   where $(\langle mnf', \theta''\rangle, l', \Pi)$ is conclusion of the inference rule.

4. ($\mathcal{I}$LOCK), which adds a new side condition $\mathcal{I}fails(r, l, \theta) \in \mathcal{I}$ to (LOCK).

The differences between *ICEC* and *CEC* are easy to understand: The first point forces ($\mathcal{I}$APPLY) to check whether the first step of a function call is a valid statement in $\mathcal{I}$. Analogously, ($\mathcal{I}\mathsf{RCV}_1$) and ($\mathcal{I}\mathsf{RCV}_3$) only allow expected transitions. In the rule ($\mathcal{I}\mathsf{RCV}_1$) we only check the outer `let` expression, but other inner expressions are also checked while evaluating the premises of the inference rule. It is worth observing that the closest piece of code can be either the function f that contains the associated Erlang code (the code that has given rise to the `let`), or a `receive` statement defined in f, which contains the `let` in the body of some rule. Finally, ($\mathcal{I}$LOCK) ensures that failures are only accepted if expected in the intended interpretation.

Summarizing, we can say that *ICEC* only computes intended values. Analogously to the case of *CEC*, the notation

$$ICEC \models_{(P,\mathcal{I},T)} \mathcal{E}$$

indicates that the evaluation $\mathcal{E}$ can be proven with respect to the program P and the intended interpretation $\mathcal{I}$ with proof tree T in *ICEC*, while $ICEC \not\models_{(P,\mathcal{I})} \mathcal{E}$ indicates that $\mathcal{E}$ cannot be proven in *ICEC*. The tree T, the program P, and the intended interpretation $\mathcal{I}$ are only made explicit when they are needed.

ICEC determines the validity of computations as indicates the following definition.

Definition 3.2. Let P be a Core Erlang Program, $\mathcal{I}$ an (intended) interpretation, T be a *CEC* computation tree with respect to P, and N be a node in T containing an evaluation $\mathcal{E}$.

1. N is *valid* with respect to *ICEC* when $ICEC \models_{(P,\mathcal{I})} \mathcal{E}$, and *invalid* when $ICEC \not\models_{(P,\mathcal{I})} \mathcal{E}$.
2. N is called *buggy* with respect to *ICEC* if $\mathcal{E}$ is invalid with all its children valid (in both cases with respect to *ICEC*).

This notion of valid/invalid evaluation gives rise to buggy nodes and hence it is the basis of declarative debugging, so it is worth revisiting the intuition. We have the following ingredients: an evaluation $\mathcal{E}$ the user wants to debug, a proof tree for $\mathcal{E}$ built in *CEC*, which relies on the code, and the behavior the user had in mind, *ICEC*. We can traverse the proof tree marking nodes with respect to *ICEC* to find the differences between what happened and what the user expected to happen; once a buggy node is found a bug is pointed out and the program can be fixed.

Now we are ready to define the errors detected by our tool.

3.4. Errors in core Erlang programs

Next we define precisely the errors detected with our technique:

Definition 3.3. Let P be a Core Erlang program, $\mathcal{I}$ its intended interpretation, and T a *CEC* computation tree. Let r be a `receive` expression statement in P and f a program function. Then, we say that:

1. r is *erroneously failing* if it is the `receive` statement executed in a buggy node of T rooted by the label (LOCK).
2. r is *erroneously succeeding* if it is the `receive` statement executed in a buggy node of T rooted by the label (RCV_3).
3. r (respectively f) contains an *erroneous transition* if its body (in the case of r the body of the branch reached accepting a previous message) is the conclusion of a buggy (RCV_1) inference rule in T.
4. f contains an *erroneous first step* if it is the function executed in a buggy (APPLY).

The errors are detected as discrepancies between the actual computations represented by *CEC* and the 'ideal' computations represented by *ICEC*.

Observe, for instance the definition of erroneously failing `receive`. Looking to the inference rule (LOCK) and considering the definition of buggy node (invalid conclusion with valid premises) it says in other words that:

- $fails(r, l, \theta)$, but
- $ICEC \not\models_{(P,\mathcal{I})} \mathcal{I}fails(r, l, \theta)$, that is, $\mathcal{I}fails(r, l, \theta) \notin \mathcal{I}$.

We say that a Core Erlang program is a *wrong program* when one of the associated computation trees contains any of the errors mentioned in the previous definition. We also say that an Erlang program is a *wrong program* if its Core representation is wrong. The next proposition ensures that only the inference rules mentioned in the previous definition can be buggy:

Proposition 3.1. *Let P be a Core Erlang program and T be a CEC computation tree with invalid root. Then:*

1. *T contains at least one buggy node.*
2. *Every buggy node corresponds to either an* (APPLY), (RCV_1), (RCV_3), *or* (LOCK) *inference.*

The first item is a general property of computation trees which can be proved easily by induction on the size of the tree: in every tree with invalid root it is possible to find at least one invalid node with all its children valid.

The second item is a direct consequence of Definition 3.1 since only (APPLY), (RCV_1), (RCV_3), and (LOCK) are different in both calculi. In particular, the *ICEC* version of the rules is more restrictive since they add new side conditions. Then it is straightforward to check that only inferences rooted by these rules can be buggy.

3.5. Soundness and completeness of the debugger

Thus far we have defined our infrastructure for Core Erlang programs. However, EDD is a tool for locating bugs in standard Erlang programs, so we need a way to relate them and their Core Erlang translation. In order to do that, we assume the existence of a transformation $|\cdot|$ that, given an Erlang program, returns its Core Erlang representation, appropriately annotated so the errors found in Core Erlang can be traced back to the original program.[5] This idea is detailed in the following assumption:

Assumption 1. Let P be an Erlang program and $|\cdot|$ the transformation that converts an Erlang expression into a Core expression. Then:

1. An evaluation $\mathcal{E} \equiv \langle e, \theta\rangle \to (\langle mnf, \theta'\rangle, l, \Pi)$ is computed by some Erlang system[6] with respect to P iff $CEC \models_P \langle |e|, \theta\rangle \to (\langle |mnf|, \theta'\rangle, l, |\Pi|)$.
2. An evaluation $\mathcal{E} \equiv \langle e, \theta\rangle \to (\langle mnf, \theta'\rangle, l, \Pi)$ computed by some Erlang system is considered unexpected with respect to *ICEC* by the user iff $ICEC \not\models_P \langle |e|, \theta\rangle \to (\langle |mnf|, \theta'\rangle, l, |\Pi|)$.

Analogous considerations are valid for $\langle e, \theta\rangle \to lock$, $\langle e, \theta\rangle \xrightarrow{l,i} (\langle mnf, \theta'\rangle, l, \Pi)$, and $\ll p, \langle e, id\rangle, [\ ] \gg\Rightarrow \Pi$.

Now we are ready to state our main theoretical result, which *lifts* the result of the Proposition 3.1 to the case of Erlang Programs.

Intuitively, given a debugging tree for an invalid evaluation (hence with an invalid root) we need to prove (completeness) it contains a buggy node and (soundness) this buggy node points out one of the errors defined in our framework: either an erroneous function or an erroneous `receive` expression.

Theorem 3.1. *Let P be an Erlang program, $\mathcal{I}$ its intended interpretation, and $\mathcal{E}$ an unexpected evaluation in P. Then:*

1. *P is a wrong program.*
2. *A debugger examining the computation tree representing $\mathcal{E}$ in Core Erlang will find at least one of the errors described in Definition 3.3.*

Proof (sketch). Consider an evaluation $\mathcal{E} \equiv \langle e, \theta\rangle \to (\langle mnf, \theta'\rangle, l, \Pi)$ (it will be similar for an evaluation $\langle mnf, \theta\rangle \xrightarrow{l,i} (\langle mnf', \theta'\rangle, l', \Pi)$). To prove the first item, it is enough to check that the Core version of the program is a wrong program. According to Assumption 1 there is a computation tree T such that

$$CEC \models_{P,T} \langle |e|, \theta\rangle \to (\langle |mnf|, \theta'\rangle, l, |\Pi|)$$

since $\langle e, \theta\rangle$ is evaluated by the program P. Moreover, the same assumption ensures that since the result obtained is unexpected we have

$$ICEC \not\models_P \langle |e|, \theta\rangle \to (\langle |mnf|, \theta'\rangle, l, |\Pi|)$$

Then, applying the Definition 1 we have that the root of T, which is the node $\langle |e|, \theta\rangle \to (\langle |mnf|, \theta'\rangle, l, |\Pi|)$ is invalid. Then, by Proposition 3.1, T contains a buggy node which corresponds to either an (APPLY), (RCV_1), (RCV_3), or (LOCK) inference rule, which in turn means that T contains some of the errors described in Definition 3.3, and thus P is a wrong program.

The proof of the second item is straightforward because a top-down navigation strategy (that is, looking for an invalid child of the root and recursively considering the root this invalid child until all children are correct) can find at least one buggy node, since the tree is finite. □

Thus, our debugger based on the technique depicted in this paper is suitable for finding the errors in Erlang programs with unexpected behavior.

[5] Core Erlang abstract syntax trees generated by Erlang/OTP contain node annotations storing the file and line in the original Erlang program. They can be accessed using `cerl:get_ann()`, see https://erldocs.com/r16a/compiler/cerl.html#get_ann-1.

[6] In our context this sentence must be understood as "by executing e with the values indicated by θ in Erlang we reach mnf with the extended substitution θ', generating during the execution the messages in l and creating the processes in Π."

4. Other applications of the semantic calculus

The operational semantics presented in this paper is mainly aimed to perform declarative debugging over concurrent executions of Erlang programs. However, it can also be applied in other contexts. In this section we explain some of them. In concrete, we explain how it can be used to perform runtime verification and behavior comparison.

4.1. Runtime verification

This section explains how we can use our operational semantics to check user-defined properties that should hold during each execution of a concurrent program. Concretely, the proposed properties define the order in which the concurrent events should occur between steps of the $\Rightarrow$ relation. There are two levels where these properties can be defined: 1) global, where the defined order involves several processes of the system, and 2) local, where the order is defined for a single process. Another aspect that must be considered is that, sometimes, we are not interested in (or simply we do not know) all the concurrent events that should happen, but just a small subset of them. Similarly, it could happen that there are steps of the $\Rightarrow$ relation which are unknown or irrelevant. For this reason, we need a way to define properties in a form that allows us to express this omitted information.

Concurrent events are defined with a tuple of patterns named *concurrent event pattern*. Their elements are special patterns that allow us to define only the known/relevant information. The tuple has the form *(p, consumed, sent, spawned)*, where:

- p is a pattern that represents the process identifier where the concurrent events should occur.
- *consumed* is a pair (p, msg) where p and *msg* are the patterns representing the sender and content of the consumed message. It can also be *none*, in case that no message should be consumed in this step.
- *sent* is a list that represents the outbox of the messages sent by the process. The elements of this list are tuples of the same form as in *consumed*. Additionally, the list can contain the symbol $*$, which stands for 0 to n elements, or $+$ which stands for 1 to n elements. The list can also contain lists of messages. A nested list indicates that the order in which its items occur is not relevant.
- *spawned* is a list where each element is either a pattern representing a process identifier, the symbols $*/+$, or a list (for undefined-order events).

Consider an example of one of these tuples:

$$(p_1, none, [*, (\_, \{activated, \_\}), (p_2, ready), *], [p_2, [p_3, p_4]]).$$

This pattern represents a state where the process p_1 has not consumed any message, it has sent two consecutive messages preceded and followed by an unknown number of messages, and it has created first the process p_2, and then p_3 and p_4 (in an undefined order). The first message is sent to an unknown process, and its content is a tuple whose first element is the atom *activated*. The second message is sent to process p_2 and its content is the atom *ready*.

A historic of events is a list that contains concurrent event patterns and the symbols $*/+$. It defines the order in which the concurrent event patterns should happen. Then, we define a system property P as a mapping of type $(p \mid global) \mapsto His$, where p is a process identifier, *global* is a special identifier which defines a global property, and *His* is a historic of events. Thus, $P(p_1)$ returns the historic of events of process p_1.

We can integrate the runtime property-checking by modifying the rules of Fig. 3 to add the system property to each rule state. The result can be found in Fig. 7, where we use the notation $l[i]$ to access the ith element of a list of messages l. The idea is the following: we use a function named *check*[7] to know whether the observed events in each step are the expected ones according to the user-defined property. If it can be checked, then the step is correct and a new system property P' is generated by removing the observed event. The property keeps being consumed until the end of the execution.

We have several scenarios to detect an unexpected behavior. First of all, when no rule can be fired, we can know what is wrong by consulting the current system state and the property that the semantics has not been able to validate. We could also increase this information by storing the last concurrent event that was tried to match against the property. Another interesting scenario is when the execution finishes but there are still events in the property that have not been checked. This means that the user expected the system to do more things than it actually has done. Similarly, when the property is empty and the execution of the semantics continues, we have an unexpected behavior that indicates that the system should have stopped but it has not.

The runtime verification system approach sketched here can be integrated into several tools that are extensively used by Erlang practitioners. One of these candidates is *PropEr* [26], a property-based testing tool based on *QuickCheck* [27]. With *PropEr* users provide first-order properties—for example `∀ L:list(). is_sorted(sort(L))` checking that after sorting a list `L` the result is sorted—and the system generates several random values to test the property. The runtime verification described here would allow users to extend those properties by integrating concurrent events. In order to

[7] We do not show here the definition of this function because it is a straightforward matching of patterns against actual expressions.

$$(\text{PROC})\ \frac{\langle e,\theta\rangle \to (\langle mnf,\theta'\rangle, l', \Pi') \quad e \text{ is not MNF}}{(\Pi\cdot \ll p, \langle e,\theta\rangle, l \gg, P) \Rightarrow (\Pi\cdot \ll p, \langle mnf,\theta'\rangle, l+l' \gg \cdot \Pi', P')}$$

where $P' = check(P, (p, none, l', \Pi'))$

$$(\text{CONSUME}_1)\ \frac{\langle mnf,\theta\rangle \xrightarrow{l'_2.j} (\langle mnf',\theta'\rangle, l', \Pi')}{\begin{array}{l}(\Pi\cdot \ll p_1, \langle mnf,\theta\rangle, l_1 \gg \cdot \ll p_2, \langle mnf_2,\theta_2\rangle, l_2 \gg, P) \Rightarrow \\ (\Pi\cdot \ll p_1, \langle mnf',\theta'\rangle, l_1 + l' \gg, \ll p_2, \langle mnf_2,\theta_2\rangle, l''_2 \gg \cdot \Pi', P')\end{array}}$$

where $l'_2 \equiv l_2|_{p_1}$, $1 \le j \le |l'_2|$, i the position of the jth message of l'_2 in l_2,
l''_2 is l_2 after removing the ith message,
and $P' = check(P, (p_1, (p_2, l_2[i]), l', \Pi'))$.

$$(\text{CONSUME}_2)\ \frac{\langle mnf,\theta\rangle \xrightarrow{l'.j} (\langle mnf',\theta'\rangle, l_1, \Pi')}{(\Pi\cdot \ll p, \langle mnf,\theta\rangle, l \gg, P) \Rightarrow (\Pi\cdot \ll p, \langle mnf',\theta'\rangle, l'' + l_1 \gg \cdot \Pi', P')}$$

where $l' \equiv l|_p$, $1 \le j \le |l'|$, i the position of the jth message of l' in l,
l'' is l after removing the ith message
and $P' = check(P, (p, (p, l[i]), l_1, \Pi'))$.

$$(\text{Tr})\ \frac{(\Pi_0, P_0) \Rightarrow (\Pi_1, P_1) \quad \ldots \quad (\Pi_{n-1}, P_{n-1}) \Rightarrow (\Pi_n, P_n) \quad n \ge 1}{(\Pi_0, P_0) \Rightarrow^+ (\Pi_n, P_n)}$$

Fig. 7. Modified rules for processes.

integrate this feature, an instrumentation of the evaluation process would be needed in such a way that in each evaluation step the corresponding properties would be checked.

The approach could also be integrated into *Concuerror* [28,29], a model checking tool for testing and verifying concurrent Erlang programs. This tool calculates all the interesting interleavings that a test, which should run a finite computation, can generate. Our proposed runtime verification approach could be used in this tool to obtain those interleavings for which the user-defined properties do not hold. The way *Concuerror* internally works, creating their own scheduler, would ease the integration of our approach.

Finally, *EDD* [4–6], the declarative debugger for Erlang which this work is aimed for, could also benefit of the proposed runtime verification. An extension of the debugger that took into account the user-defined properties would allow *EDD* to start a debugging session in the exact moment where a property does not hold. This will reduce the size of the debugging tree, which means less questions to be asked to the user. Additionally, the debugger will know the answer of some questions by using the previously verified parts of a property. *EDD* can integrate this approach by replacing its current evaluator, i.e., the standard one, by an instrumented one that as a side-effect checks the user-defined properties. Additionally, *EDD* could synthesize properties from a debugging session and store them, e.g. as *PropEr* tests.

4.2. Behavior comparison

A program evolves during its lifetime to incorporate new functionality or to correct or improve an existent one. When a complex system is changed, users wish to check that some basic functionality still works by comparing the old and the new version of the system. In a concurrent scenario, this implies a way to force the same interleaving when running a concrete test in both versions. In this section we demonstrate that our semantics can be used to execute a concrete interleaving instead of a random choice, enabling in this way this feature.

The first step is to get a fixed interleaving. This is done by an augmentation of the semantics very similar to the one shown in Fig. 7. Indeed, we could use exactly the same rules and conditions but instead of using the function *check*, we should use a new function *add*. This function produces a new property by adding the observed concurrent event to the history of the current process and of the *global*. At the end of the execution, it has been generated a completely defined property of the system, i.e., no patterns, lists, or symbols $*/+$ are inside it, instead all of its items are fully-defined expressions.

After recording the complete history of concurrent events for each process and for the whole system, we can run the new version by forcing exactly the same interleaving as the recorded property defines. The modification of the rules shown in Fig. 7 can be used to check that the new version is following the same interleaving. If the new version is able to finish its execution reaching the end with an empty property, then the system behaves in the same way as with the previous version.

However, it is not always possible nor desirable to use an interleaving that is too detailed. Sometimes, we are interested in keeping only a part of the behavior but not all. In order to achieve this, the recorded property can be abstracted, e.g. by a function that removes the constraints over the sent messages by replacing that sent message information by a wildcard pattern. Therefore, by the automated or personalized abstraction of the recorded properties, we obtain a more flexible behavior-preservation checking. This flexibility allows us to compare the behavior of very different program versions.

$$\text{(TUPLE)}\ \frac{\langle vv_1,\theta\rangle \to (\langle v_1,\theta\rangle,[\,],\emptyset) \quad \ldots \quad \langle vv_n,\theta\rangle \to (\langle v_n,\theta\rangle,[\,],\emptyset)}{\langle \{vv_1,\ldots,vv_n\},\theta\rangle \to (\langle \{v_1,\ldots,v_n\}\ ,\theta\rangle,[\,],\emptyset)}$$

$$\text{(LIST)}\ \frac{\langle vv_1,\theta\rangle \to (\langle v_1,\theta\rangle,[\,],\emptyset) \quad \langle vv_2,\theta\rangle \to (\langle v_2,\theta\rangle,[\,],\emptyset)}{\langle [vv_1,\,|\,vv_2],\theta\rangle \to (\langle [v_1,\,|\,v_2],\theta\rangle,[\,],\emptyset)}$$

$$\text{(CALL)}\ \frac{\begin{array}{c}\langle vv_1,\theta\rangle \to (\langle v_1,\theta\rangle,[\,],\emptyset) \quad \ldots \quad \langle vv_n,\theta\rangle \to (\langle v_n,\theta\rangle,[\,],\emptyset)\\ \langle vf,\theta\rangle \to (\langle fname,\theta\rangle,[\,],\emptyset) \qquad eval(fname,v_1,\ldots,v_n)=v\end{array}}{\langle \texttt{call}\ vf(vv_1,\ldots,vv_n),\theta\rangle \to (\langle v,\theta\rangle,[\,],\emptyset)}$$

where *fname* is a built-in Erlang function

$$\text{(APPLY)}\ \frac{\begin{array}{c}\langle vv_1,\theta\rangle \to (\langle v_1,\theta\rangle,[\,],\emptyset) \quad \ldots \quad \langle vv_n,\theta\rangle \to (\langle v_n,\theta\rangle,[\,],\emptyset)\\ \langle vff,\theta\rangle \to (\langle v,\theta\rangle,[\,],\emptyset) \qquad \langle e,\theta'\rangle \to (\langle mnf,\theta''\rangle,l,\Pi)\end{array}}{\langle \texttt{apply}\ vff(vv_1,\ldots,vv_n),\theta\rangle \to (\langle mnf,\theta''\rangle,l,\Pi)}$$

where v is an anonymous function or a function name defined as $\texttt{fun}(\overline{X_n}) \to e$ fresh, and $\theta' \equiv \theta \uplus \{\overline{X_n \mapsto v_n}\}$

Fig. 8. Extended rules for $\to$ supporting tuples, lists and higher-order.

The application sketched in this section could be integrated in other tools with similar goals. *PropEr* is also a candidate here, because the user could define properties that indicate that the behavior of two versions of a program should be the same or similar, by specifying what is the degree of similarity accepted. An instrumented evaluation that builds and compares properties would be an option to integrate this feature into *PropEr*.

Similarly, *EDD* could incorporate a feature that allow users to indicate that a new version of a program is not behaving as expected. Then, *EDD* could build a debugging tree focused on comparing the observed behavior in the buggy version and the observed behavior in a previous version which works as expected. This enhancement of the tool would modify it from the root. For instance, it would be needed to change even the type of questions asked to the user. However, it would constitute a very interesting and novel way to debug regression faults.

Finally, the presented application could be easily integrated into *SecEr* [30], a behavior comparison tool for Erlang. The tool uses two versions of a program and a set of observation points to synthesize tests that check whether the expected behavior has been preserved. The behavior comparison proposed in this section might be linked to the observational points in such a way that each time the behavior of a point is observed, their current properties (or a defined abstraction of them) are also observed. This enhancement could be achieved by an augmented tracer that stores the observed values along with the current properties calculated by an instrumented evaluator.

5. Discussion: extending the language

The syntax presented in Section 2.1 is very simple to focus on the concurrency aspects of Erlang and simplify the presentation. In this section we will discuss how to extend it with three relevant constructions (tuples, lists, and anonymous functions[8]) as well as the ability to store functions in variables/parameters and call them (known as *higher-order*).

The first step is to extend the syntax with these new constructions. Besides expressions these constructions can generate values and patterns, so all these categories must be extended:

$$\begin{array}{rcl} v & ::= & \ldots \mid \{v_1,\ldots,v_n\} \mid [\,] \mid [v_1\,|\,v_2] \mid \texttt{fun}(\overline{X_n}) \to e \\ vf & ::= & X \mid fname \\ vff & ::= & X \mid fname \mid \texttt{fun}(\overline{X_n}) \to e \\ pat & ::= & X \mid v \mid \{pat_1,\ldots,pat_n\} \mid [\,] \mid [pat_1\,|\,pat_2] \\ e & ::= & \ldots \mid \texttt{call}\ vf(\overline{vv_n}) \mid \texttt{apply}\ vff(\overline{vv_n}) \mid \{vv_1,\ldots,vv_n\} \\ & & \mid [\,] \mid [vv_1\,|\,vv_2] \mid \texttt{fun}(\overline{X_n}) \to e \end{array}$$

The new categories *vf* and *vff* are used in `call` and `apply` expressions to support higher-order (note that `call` cannot accept an anonymous function, as by definition it is used to invoke built-in functions). As in Section 2.1, we have followed A-normal forms, so tuples and lists contain values or variables.

The next step is to extend the semantic calculus presented in Section 2 with rules for tuples and lists, as well as modify the rules for function application in order to support higher-order. These changes appear in Fig. 8. The new rules (TUPLE) and (LIST) simply evaluate the arguments and generate the final value without sending messages or creating processes. (CALL) has been extended to support higher-order: *vf* can be a *fname* or a variable, but in both cases it is evaluated to a *fname* that is called with *eval*. The same modification appears in rule (APPLY), but in this case the function to call can

[8] Also known as lambda abstractions or `fun` expressions.

be a *fname*, a variable, or an anonymous function. The evaluation of the *vff* parameter generates a *fname* or an anonymous function whose body is used to obtain the next MNF.

From the point of view of the soundness and completeness of the semantic calculus presented in Section 2.4, tuples and lists will not present any problem: the new rules behave exactly as their counterparts in [7]. Therefore they guarantee both soundness and completeness. The changes made to provide higher-order, on the other hand, would invalidate soundness because higher-order is not supported in the syntax of [7]. Then, derivations with $\Rightarrow$ using higher-order could not be $\hookrightarrow_{mop}$ derivations, whereas derivations $\hookrightarrow_{mop}$ (which cannot use higher-order functions) will be still valid with respect to $\Rightarrow$. Nevertheless, $\hookrightarrow_{mop}$ could be trivially extended to support higher-order (as explained in [7]) thus guaranteeing both soundness and completeness.

Regarding declarative debugging, we would need to take into account that the modified (APPLY) rule can reveal an error in higher-order and anonymous functions, while rules for tuples and list do not affect the framework. We would modify the intended interpretation $\mathcal{I}$ to include these functions, so the *ICEC* calculus would use this information in an extended ($\mathcal{I}$APPLY) rule. Then, the definition of *wrong function* would include these functions and the soundness result would compare the results in the new rules (APPLY) and ($\mathcal{I}$APPLY); the completeness result would not be modified.

6. Concluding remarks and ongoing work

In this paper we have presented a medium-sized-step semantics for concurrent Core Erlang programs. This semantics focuses on message exchanges and hence it is specially well suited for declarative debugging, which requires meaningful steps to generate the questions asked to the user. In fact, we use this semantics to check the soundness and completeness of the declarative debugger EDD. Furthermore, it can be also useful for reasoning about the soundness and/or completeness of other Erlang tools devoted to analyze concurrent applications.

We have also proved the soundness and completeness of our semantic calculus with respect to a recently proposed Erlang semantics [7] that supports debugging using causal-consistent reversibility. Concretely, we have proved that when the order of messages between processes is preserved, both semantics produce the same traces leading to medium-sized normal forms.

As future work, we consider how to define a *predictive* semantics. This semantics would know a priori the messages that each process will receive, hence allowing us to define big-step semantics for each process. However, it also requires a verification mechanism that checks whether the messages generated by the processes correspond to the ones initially given to the processes and the interleaving used in the tree is possible. A straightforward way to prove this is to obtain the predictive proof tree from the proof tree generated by the semantics presented in this paper.

Declaration of Competing Interest

There is no competing interest.

Acknowledgements

Research supported by the Comunidad de Madrid projects S2018/TCS-4339 and P2018/TCS-4314, the MINECO projects TIN2015-66471-P, TIN2015-67522-C3-3-R, TIN2016-76843-C4-1-R, and TIN2015-69175-C4-2-R.

Appendix A. Proofs

A.1. Preliminary notions and results

Definition A.1 *(Submits and spawns).* Given a list of labels $L \equiv [l_1, \ldots, l_n]$ and a list of process identifiers $P \equiv [p_1, \ldots, p_m]$ with $m \leq n$ we define the list of submissions and spawns as:

$$submits(L) = \begin{cases} [\,] & \text{if } L = [\,] \\ (p \,!\, v) : submits(L') & \text{if } L = send(p, v) : L' \\ submits(L') & \text{if } L = l : L' \text{ and } l \neq send \end{cases}$$

$$spawns(L, P) = \begin{cases} \emptyset & \text{if } L = P = [\,] \\ \ll p, \langle e, \{\overline{X_n \mapsto v_n}\}, [\,]\rangle \gg \cdot S & \text{if } L = \texttt{spawn}(\kappa, fname, [\overline{v_n}]) : L', \\ & \quad P = p : P', \\ & \quad fname = \texttt{fun}(\overline{X_n}) \to e, \\ & \quad \text{and } S = spawns(L', P') \\ spawns(L', P) & \text{if } L = l : L' \text{ and } l \neq \texttt{spawn} \end{cases}$$

We use the notation $x : xs$ to express a list with head x and tail xs.

Definition A.2 *(Message-order-preserving traces).* We say a trace $\Gamma_0; \Pi_0 \hookrightarrow^+ \Gamma_n; \Pi_n$ is *message-order-preserving* (MOP for short) if for every system $\Gamma_i; \Pi_i$, and every process $\langle p_j, \langle \theta_j, e_j \rangle, q_j \rangle$ in Π_i, the messages submitted from a process p occurring in the mailbox q_j have been delivered maintaining the order in which they were sent. Concretely:

- For every process p, all the messages in q_j submitted by p are older than the messages from p stored in Γ.
- For every process p, the messages from p in the mailbox q_j appear in the order they were sent.

Proposition A.1 *(Equivalence of case evaluation). match*$(\theta, v, \overline{b_n}) = (\theta', e)$ *iff succeeds_case*$(\theta, v, \overline{b_n}) = (e, \theta \uplus \theta')$.

Proof. Straightforward, as both functions express the selection of a branch in a `case` expression. The only difference is that *match* generates the substitution θ' resulting from the matching of the pattern, whereas *succeeds* returns the substitution θ extended with the matching substitution. □

Proposition A.2 *(Receive evaluation). If succeeds*$(e, m, j, \theta) = (e', \theta')$ *where e is a* `receive` *expression with branches* $\overline{b_n}$ *then matchrec*$(\theta, \overline{b_n}, m) = (\theta_i, e', v)$ *where the message* $m[j]$ *has value v and* $\theta' = \theta \uplus \theta_i$.

Proof. Straightforward, since *succeeds* and *mathcrec* express the same behavior for Erlang message matching against `receive` expressions. □

Proposition A.3 *(Receive failure). If fails*(mnf, m, θ) *and mnf is a medium-sized normal form with a nested* `receive` *expression with branches* $\overline{b_n}$ *then matchrec*$(\theta, \overline{b_n}, q)$ *fails for any queue q that contains a subset of the messages in m.*

Proof. As the proof of Proposition A.2. □

Definition A.3 *(Minimum-delivery trace).* A trace $\Gamma_1; \Pi_1 \hookrightarrow \Gamma_2; \Pi_2 \hookrightarrow \ldots \hookrightarrow \Gamma_n; \Pi_n$ is a minimum-delivery trace (minimum-delivery in short) if $\Gamma_1; \Pi_1$ and $\Gamma_n; \Pi_n$ are empty-mailbox systems and in the rest of systems $\Gamma_2; \Pi_2, \ldots, \Gamma_{n-1}; \Pi_{n-1}$ there is at most one message delivered considering all the local mailboxes of the processes.

Proposition A.4. $\Gamma; \Pi \hookrightarrow^+ \Gamma'; \Pi'$ *is a MOP trace* $\iff \Gamma; \Pi \hookrightarrow^+_{mop} \Gamma'; \Pi'$.

Proof. Straightforward, as the (Sched) steps in a MOP trace delivers the oldest message from a process. □

As a final comment, in the following theoretical results and proofs we will sometimes use tm to denote lists of messages instead of l. The reason is avoid any confusion with labels as in $\xrightarrow{l}$.

A.2. Soundness proof

Lemma A.1 *(Soundness of* $\rightarrow$ *w.r.t.* $\xrightarrow{l}$*). Consider a derivation* $\langle e, \theta \rangle \rightarrow (\langle mnf, \theta' \rangle, m, \Pi)$. *Then there are* $n \geq 0$ *steps* $(\theta, e) \xrightarrow{l_1} (\theta_1, e_1)$, $(\theta_1, e_1\theta^*) \xrightarrow{l_2} (\theta_2, e_2), \ldots, (\theta_{n-1}, e_{n-1}\theta^*) \xrightarrow{l_n} (\theta', e_n)$ *using* $\xrightarrow{l}$ *[7] such that:*

a) $e_n\theta^* = mnf$
b) $l_1, \ldots, l_n$ *are* τ, *send*(p, v) *or spawn*$(\kappa, fname, [\overline{v_n}])$ *labels*
c) *submits*$([l_1, l_2, \ldots, l_n]) = m$
d) *spawns*$([l_1, l_2, \ldots, l_n], [\overline{p_j}]) = \Pi$

where $\overline{p_j}$ *are the fresh ps generated in* Π *(in the same order that they were created) and* $\theta^* \equiv \{\overline{\kappa_j \mapsto p_j}\}$ *maps the variables* $\overline{\kappa_j}$ *introduced in the* $\xrightarrow{spawn}$ *steps to those* $\overline{p_j}$.

Proof. By induction on the size of the derivation $\langle e, \theta \rangle \rightarrow (\langle mnf, \theta' \rangle, l, \Pi)$:

Base Case: The derivation has used rules (MNF) or (VAR) rules:

(**MNF**) This step does not change the expression or the substitution, so the lemma holds trivially by applying zero $\xrightarrow{l}$ steps.

(**VAR**) We have a derivation:

$$\overset{\text{(VAR)}}{\dfrac{}{\langle X, \theta \rangle \rightarrow (\langle \theta(X), \theta \rangle, [\,], \emptyset)}}$$

Then we can perform a (Var) step:

$$\text{(Var)}\ \frac{}{(\theta, X) \xrightarrow{\tau} (\theta, \theta(X))}$$

with $\theta^* = id$. Therefore $(\theta(X))\theta^* = \theta(X)$ (a), the only label is τ (b), $submits([\tau]) = [\,]$ (c), and $spawns([\tau], [\,]) = \emptyset = \Pi$ (d).

Inductive Step: The derivation has used rules (CASE), (APPLY), (CALL), (LET_1), (LET_2), (SPAWN), or (BANG).

- (CASE): We have the following derivation

$$\text{(CASE)}\ \frac{\begin{array}{c}\langle vv, \theta\rangle \to (\langle v, \theta\rangle, [\,], \emptyset) \quad succeeds\_case(\theta, v, \overline{b_n}) = (e, \theta') \\ \langle e, \theta'\rangle \to (\langle mnf, \theta''\rangle, m, \Pi)\end{array}}{\langle \texttt{case}\ vv\ \texttt{of}\ b_1; \ldots; b_n\ \texttt{end}, \theta\rangle \to (\langle mnf, \theta''\rangle, m, \Pi)}$$

If vv is a value we do not need to perform any $\xrightarrow{l}$ step, otherwise is a variable and we can evaluate then to a value using (Case1) with (Var): $(\theta, \texttt{case}\ vv\ \texttt{of}\ b_1; \ldots; b_n\ \texttt{end}) \xrightarrow{\tau} (\theta, \texttt{case}\ v\ \texttt{of}\ b_1; \ldots; b_n\ \texttt{end})$. By Proposition A.1 we have that $match(\theta, v, \overline{b_n}) = (\hat{\theta}, e)$ such that $\theta \uplus \hat{\theta} = \theta'$. Then we can evaluate the case expression using the rule (Case2), obtaining:

$$\text{(Case2)}\ \frac{match(\theta, v, \overline{b_n}) = (\hat{\theta}, e)}{(\theta, \texttt{case}\ v\ \texttt{of}\ b_1; \ldots; b_n\ \texttt{end}) \xrightarrow{\tau} (\theta', e)}$$

Then by IH we have that there are $b \geq 0$ steps $(\theta', e) \xrightarrow{l_1} (\theta_1, e_1)$, $(\theta_1, e_1\theta^*) \xrightarrow{l_2} (\theta_2, e_2)$, …, $(\theta_{b-1}, e_{b-1}\theta^*) \xrightarrow{l_n} (\theta'', e_b)$ such that:

a) $e_b\theta^* = mnf$
b) $l_1, \ldots, l_b$ are τ, $send(p, v)$ or $spawn(\kappa, fname, [\overline{v_n}])$ labels
c) $submits([l_1, l_2, \ldots, l_b]) = m$
d) $spawns([l_1, l_2, \ldots, l_b], [\overline{p_j}]) = \Pi$

Therefore, we can concatenate the first (Case1) step, followed by the (Case1) step and finally the last b steps to reach a state satisfying the conditions of the lemma—recall that the first two steps have a τ label. Moreover, in the first $\xrightarrow{\tau}$-steps applying the substitution θ^* does not change the expression, as the variables κ are not introduced in this kind of steps.

- (APPLY): We have a derivation:

$$\text{(APPLY)}\ \frac{\begin{array}{c}\langle vv_1, \theta\rangle \to (\langle v_1, \theta\rangle, [\,], \emptyset) \quad \ldots \quad \langle vv_n, \theta\rangle \to (\langle v_n, \theta\rangle, [\,], \emptyset) \\ \langle e, \theta'\rangle \to (\langle mnf, \theta''\rangle, m, \Pi)\end{array}}{\langle \texttt{apply}\ fname(vv_1, \ldots, vv_n), \theta\rangle \to (\langle mnf, \theta''\rangle, m, \Pi)}$$

where $fname$ is a function defined as $\texttt{fun}(\overline{X_n}) \to e$ and $\theta' \equiv \{\overline{X_n \mapsto v_n}\}$. For every vv_i, if it is a value then we do not perform any $\xrightarrow{\tau}$-step. Otherwise, it is a variable $X \in dom(\theta)$ so we can perform a $\xrightarrow{\tau}$-step using rule (Apply1) with (Var). Finally, we will reach a state:

$$\theta, \texttt{apply}\ fname(vv_1, \ldots, vv_n) \xrightarrow{\tau}{}^{*} \theta, \texttt{apply}\ fname(v_1, \ldots, v_n)$$

From this state we can apply rule (Apply2) to reach:

$$\text{(Apply2)}\ \frac{}{\theta, \texttt{apply}\ fname(v_1, \ldots, v_n)) \xrightarrow{\tau} (\theta \uplus \theta', e)}$$

Then by IH we have that there are $b \geq 0$ steps $(\theta', e) \xrightarrow{l_1} (\theta_1, e_1)$, $(\theta_1, e_1\theta^*) \xrightarrow{l_2} (\theta_2, e_2)$, …, $(\theta_{b-1}, e_{b-1}\theta^*) \xrightarrow{l_n} (\theta'', e_b)$ such that:

a) $e_b\theta^* = mnf$
b) $l_1, \ldots, l_b$ are τ, $send(p, v)$ or $spawn(\kappa, fname, [\overline{v_n}])$ labels
c) $submits([l_1, l_2, \ldots, l_b]) = m$
d) $spawns([l_1, l_2, \ldots, l_b], [\overline{p_j}]) = \Pi$

As in the previous case, we can concatenate the first (Apply1) steps, followed by the (Apply2) step—all with labels τ—and finally the last b steps to reach a state satisfying the conditions of the lemma.

- (CALL): A simplification of the (APPLY) case but using rules (Call1) and (Call2) and omitting the last sequence of $\xrightarrow{l}$ steps.

- (LET_1): We have a derivation:

$$(\mathsf{LET}_1)\ \frac{\langle e,\theta\rangle \to (\langle v,\theta'\rangle, m, \Pi) \qquad \langle e', \theta' \uplus \{X \mapsto v\}\rangle \to (\langle mnf, \theta'''\rangle, m', \Pi')}{\langle \mathtt{let}\ X = e\ \mathtt{in}\ e', \theta\rangle \to (\langle mnf, \theta'''\rangle, m + m', \Pi \cdot \Pi')}$$

where $\Pi \equiv \overline{\ll p_j, \langle e_j^p, \theta_j^p\rangle, [\,]\gg}$ and $\Pi' \equiv \overline{\ll p'_{j'}, \langle e'^{p}_{j'}, \theta'^{p}_{j'}\rangle, [\,]\gg}$. Then by IH there are $b \geq 0$ steps $(\theta, e) \xrightarrow{l_1} (\theta_1, e_1)$, $(\theta_1, e_1\theta^*) \xrightarrow{l_2} (\theta_2, e_2)$, …, $(\theta_{b-1}, e_{b-1}\theta^*) \xrightarrow{l_b} (\theta', e_b)$ such that:

a) $e_b\theta^* = v$
b) $l_1, \ldots, l_n$ are τ, $send(p, v)$ or $spawn(\kappa, fname, [\overline{v_n}])$ labels
c) $submits([l_1, l_2, \ldots, l_b]) = m$
d) $spawns([l_1, l_2, \ldots, l_b], [\overline{p_j}]) = \Pi$

where $\theta^* \equiv \{\overline{\kappa_j \mapsto p_j}\}$. Then again by IH there are $b' \geq 0$ steps $(\theta' \uplus \{X \mapsto v\}, e') \xrightarrow{l'_1} (\theta'_1, e'_1)$, $(\theta'_1, e'_1\theta'^*) \xrightarrow{l'_2} (\theta'_2, e'_2)$, …, $(\theta'_{b'-1}, e'_{b'-1}\theta'^*) \xrightarrow{l'_{b'}} (\theta''', e'_{b'})$such that:

a) $e'_{b'}\theta'^* = mnf$
b) $l'_1, \ldots, l'_{b'}$ are τ, $send(p, v)$ or $spawn(\kappa, fname, [\overline{v_n}])$ labels
c) $submits([l'_1, l'_2, \ldots, l'_{b'}]) = m'$
d) $spawns([l'_1, l'_2, \ldots, l'_{b'}], [\overline{p'_j}]) = \Pi'$

where $\theta'^* \equiv \{\overline{\kappa'_j \mapsto p'_j}\}$. Considering a combined mapping $\theta^*_g = \theta^* \uplus \theta'^*$ we can raise the first b steps to (Let1) steps: $b \geq 0$ steps $(\theta, e) \xrightarrow{l_1} (\theta_1, e_1)$, $(\theta_1, e_1\theta^*) \xrightarrow{l_2} (\theta_2, e_2)$, …, $(\theta_{b-1}, e_{b-1}\theta^*) \xrightarrow{l_b} (\theta', e_b)$

$$\begin{aligned}
&\theta, \mathtt{let}\ X = e\ \mathtt{in}\ e' && \xrightarrow{l_1}\ \theta_1, \mathtt{let}\ X = e_1\ \mathtt{in}\ e'\\
&\theta_1, (\mathtt{let}\ X = e_1\ \mathtt{in}\ e')\theta^*_g && \xrightarrow{l_2}\ \theta_2, \mathtt{let}\ X = e_2\ \mathtt{in}\ e'\\
& && \vdots\\
&\theta_{b-1}, (\mathtt{let}\ X = e_{b-1}\ \mathtt{in}\ e')\theta^*_g && \xrightarrow{l_2}\ \theta', \mathtt{let}\ X = e_b\ \mathtt{in}\ e'
\end{aligned}$$

As $e_b\theta^*_g = v$ we can perform a (Let2) step:

$$\theta', (\mathtt{let}\ X = e_b\ \mathtt{in}\ e')\theta^*_b \xrightarrow{\tau} \theta' \uplus \{X \mapsto v\}, e'$$

and continue with the b' steps from the IH using θ^*_g, as $e'\theta^*_g = e'$.

- (LET_2): A simplification of the (LET_1) case using only (Let1) steps.
- (SPAWN): We have a derivation:

$$(\mathsf{SPAWN})\ \frac{\langle vv_1,\theta\rangle \to (\langle v_1,\theta\rangle, [\,], \emptyset) \quad \ldots \quad \langle vv_n,\theta\rangle \to (\langle v_n,\theta\rangle, [\,], \emptyset)}{\langle \mathtt{spawn}(fname, [vv_1, \ldots, vv_n]), \theta\rangle \to (\langle p, \theta\rangle, [\,], \ll p, \langle e, \theta'\rangle, [\,]\gg))}$$

where *fname* is defined as $\mathtt{fun}(\overline{X_n}) \to e$, $\theta' \equiv \{\overline{X_n \mapsto v_n}\}$ and p is fresh. Since the evaluation of vv_i does not create any process, then by IH we have that (as in the (APPLY_1) case) there are 0 or 1 steps such that $(\theta, vv_i) \xrightarrow{\tau} (\theta, v_i)$. Then we can raise these steps over the parameters to (Spawn1) steps and reach:

$$\theta, \mathtt{spawn}(fname, [vv_1, \ldots, vv_n]) \xrightarrow{\tau}{}^{*} \theta, \mathtt{spawn}(fname, [v_1, \ldots, v_n])$$

From this state we can follow applying rule (Spawn2)

$$(\mathsf{Spawn2})\ \frac{}{\theta, \mathtt{spawn}(fname, [v_1, \ldots, v_n]) \xrightarrow{spawn(\kappa, fname, [\overline{v_n}])} \theta, \kappa}$$

Considering $\theta^* = \{\kappa \mapsto p\}$, we can perform the first $\xrightarrow{\tau}$-steps applying that substitution to the expression. Then the conditions of the lemma are satisfied with

a) $\kappa\theta^* = p$
b) The only labels in the derivation are τ followed by one *spawn*
c) $submits([\overline{\tau}, spawn(\kappa, fname, [\overline{v_n}])]) = [\,]$

d) $spawns([\overline{\tau}, spawn(\kappa, fname, [\overline{v_n}])], [p]) = \ll p, \langle e, \theta' \rangle, [\,] \gg$

- (BANG): The derivation is:

$$(\text{BANG})\frac{\langle vp, \theta\rangle \to (\langle p, \theta\rangle, [\,], \emptyset) \qquad \langle vv, \theta\rangle \to (\langle v, \theta\rangle, [\,], \emptyset)}{\langle vp\,!\,vv, \theta\rangle \to (\langle v, \theta\rangle, [p\,!\,v], \emptyset)}$$

By IH we have that $(\hat{\theta}, vp) \xrightarrow{\tau} (\hat{\theta}, p)$ and $(\hat{\theta}, vv) \xrightarrow{\tau} (\hat{\theta}, v)$ in 0 or 1 steps. If the first derivation performs one step, we can raise it to a (Send1) step:

$$(\text{Send1})\frac{\theta, vp \xrightarrow{\tau} \theta, p}{\theta, vp\,!\,vv \xrightarrow{\tau} \theta, p\,!\,vv}$$

If the second derivation performs one step, we can raise it to a (Send2) step:

$$(\text{Send2})\frac{\theta, vv \xrightarrow{\tau} \theta, v}{\theta, p\,!\,vv \xrightarrow{\tau} \theta, p\,!\,v}$$

Finally, we perform a (Send3) step:

$$(\text{Send3})\frac{}{(\hat{\theta}, p!v) \xrightarrow{send(p,v)} (\hat{\theta}, v)}$$

The original derivation does not generate new processes, so $\theta^* = id$ and $v\theta^* = v$ (a). The labels are $[\overline{\tau}, send(p, v)]$ (b), and the only submitted message is $[p!v] = submits([send(p, v)])$. Finally, there are not *spawn* steps, so $spawns([send(p, v)], [\,]) = \emptyset$. □

The next lemma is similar to Lemma A.1 but considering one step $\xrightarrow{m,j}$ that consumes the jth message from a list of messages m. In this case, the resulting sequence of $\xrightarrow{l}$ steps contain exactly one $\xrightarrow{rec(\kappa,\overline{b_n})}$ that consumes the message $m[j]$.

Lemma A.2 *(Soundness of $\xrightarrow{m,j}$ w.r.t. $\xrightarrow{l}$). Consider a derivation $\langle e, \theta\rangle \xrightarrow{m,j} (\langle mnf', \theta'\rangle, m, \Pi)$ where $\Pi \equiv \overline{\ll p_j, \langle e_j^p, \theta_j^p\rangle, [\,] \gg}$. Then there are $n > 0$ steps $(\theta, e) \xrightarrow{l_1} (\theta_1, e_1)$, $(\theta_1, e_1\theta^*) \xrightarrow{l_2} (\theta_2, e_2)$, …, $(\theta_{i-1}, e_{i-1}\theta^*) \xrightarrow{rec(\kappa,\overline{b_n})} (\theta_i, \kappa)$, $(\theta_i \uplus \{\kappa \mapsto e_i\} \uplus \theta_i', e_i) \xrightarrow{l_{i+1}} (\theta_{i+1}, e_{i+1})$, …, $(\theta_{n-1}, e_{n-1}\theta^*) \xrightarrow{l_n} (\theta', e_n)$ using $\xrightarrow{l}$ [7] such that:*

a) $e_n\theta^* = mnf'$
b) $l_1, \ldots, l_n$ *are* τ, $send(p, v)$ *or* $spawn(\kappa, fname, [\overline{v_n}])$ *labels*
c) $submits([l_1, l_2, \ldots, l_n]) = m$
d) $spawns([l_1, l_2, \ldots, l_n], [\overline{p_j}]) = \Pi$
e) $matchrec(\theta_{i-1}, \overline{b_n}, m) = (\theta_i', e_i, v)$

where $\overline{p_j}$ are the fresh process identifiers s generated in Π (in the same order they were created) and $\theta^* \equiv \{\overline{\kappa_j \mapsto p_j}\}$ maps the variables $\overline{\kappa_j}$ introduced in the $\xrightarrow{spawn}$ steps to those $\overline{p_j}$.

Proof. The proof of this lemma is similar to the one for Lemma A.1, but in this case we proceed by induction on the size of the $\xrightarrow{m,j}$ derivation. In the base case, (RECEIVE_3) processes a `receive` expression e with branches $\overline{b_n}$ using the jth message of list m, returning an expression and a substitution: $succeeds(e, m, j, \theta) = (e', \theta')$. Then we can perform a $\xrightarrow{rec(\kappa,\overline{b_n})}$ step using (Receive) (using Proposition A.2) to retrieve the jth message and by Lemma A.1 we can follow this first step with more steps $\xrightarrow{l}$. Therefore, we can concatenate all the $\xrightarrow{l}$ steps and obtain a sequence satisfying the conditions of the Lemma.

In the inductive case—(RECEIVE_1) or (RECEIVE_2)—by IH we have a (possibly empty) sequence of $\xrightarrow{l}$ steps followed by a $\xrightarrow{rec(\kappa,\overline{b_n})}$ and another (possibly empty) sequence of $\xrightarrow{l}$. From the last (θ, e) pair, by Lemma A.1 we have a sequence of $\xrightarrow{l}$ satisfying the conditions of the lemma. □

Lemma A.3 *(mnf-irreducibility). Let mnf be a medium-sized normal form:*

a) *If mnf is a value v then $\nexists(\theta', e)$ such that $(\theta, v) \xrightarrow{l} (\theta', e)$.*

b) *If mnf is not a value and* $(\theta, mnf) \xrightarrow{l} (\theta', e)$ *then* $l \equiv rec(\kappa, [\overline{b_n}])$.

Proof. a) Values cannot be evaluated by $\xrightarrow{l}$.

b) By induction on the size of the derivation of the step $\xrightarrow{l}$. If *mnf* is a `receive` expression (base case), then it can only be reduced using rule (Receive), thus generating a label $rec(\kappa, \overline{b_n})$. If *mnf* is a `let` expression, then it must be evaluated by rule (Let1), as the right-hand side of the equality is an *mnf* different from a value by definition of medium-sized normal form. Then by IH we have that the right-hand side of the equality has been reduced with a label $rec(\kappa, \overline{b_n})$, as the whole (Let1) step. □

Lemma A.4. *Consider a step* $\Pi_0 \Rightarrow \Pi_1$*:*

a) *If* $\Pi_1 \neq$ *endlock, there is a minimum-delivery trace* $\| \Pi_0 \| \hookrightarrow^+ \| \Pi_1 \|$

b) *If* $\Pi_1 =$ *endlock then* $\| \Pi_0 \| \overset{(\text{Sched})}{\hookrightarrow}{}^* \Gamma; \Pi$*, where* $\| \Pi_0 \|$ *is an empty-mailbox system and* $\Gamma; \Pi$ *is blocked*

Proof. We first prove by case distinction the case a) when $\Pi_1 \neq$ *endlock*. The derivation has used rules (PROC), (CONSUME_1) or, (CONSUME_2). Rule (ENDLOCK) could not be used in this case because $\Pi_1 \neq$ *endlock*.

- (PROC): Then we have a derivation:

$$\text{(PROC)}\ \frac{\langle e, \theta\rangle \to (\langle mnf, \theta'\rangle, m', \Pi') \qquad e \neq mnf}{\Pi \cdot \ll p, \langle e, \theta\rangle, m \gg \Rightarrow \Pi \cdot \ll p, \langle mnf, \theta'\rangle, m + m' \gg \cdot \Pi'}$$

We assume $\Pi' \equiv \overline{\ll p_j, \langle e_j^p, \theta_j^p\rangle, [\ ] \gg}$. Then by Lemma A.1 there are $n > 0$ steps[9] $(\theta, e) \xrightarrow{l_1} (\theta_1, e_1)$, $(\theta_1, e_1\theta^*) \xrightarrow{l_2} (\theta_2, e_2)$, …, $(\theta_{n-1}, e_{n-1}\theta^*) \xrightarrow{l_n} (\theta', e_n)$ using $\xrightarrow{l}$ [7] such that:

a) $e_n\theta^* = mnf$
b) $l_1, \ldots, l_n$ are τ, $send(p, v)$ or $spawn(\kappa, fname, [\overline{v}])$ labels
c) $submits([l_1, l_2, \ldots, l_n]) = m'$
d) $spawns([l_1, l_2, \ldots, l_n], [\overline{p_j}]) = \Pi'$

where $\theta^* \equiv \{\overline{\kappa_j \mapsto p_j}\}$ maps the variables $\overline{\kappa_j}$ introduced in the $\xrightarrow{spawn}$ steps to the fresh process identifiers generated in Π', in the same order they were created. On the other hand, we have that:

$$\| \Pi_0 \| = \| \Pi \cdot \ll p, \langle e, \theta\rangle, l \gg \| = \Gamma_t; \langle p, (\theta, e), [\ ]\rangle | \Pi_t$$

where Π_t is the translation of Π and its mailboxes are empty. Depending of the label of each step $\xrightarrow{l_i}$ we can generate different $\hookrightarrow$ steps and concatenate them:

- $l_i \equiv \tau$)
  Then we can perform a (Seq) step on process p:

$$\text{(Seq)}\ \frac{(\theta_{i-1}, e_{i-1}) \xrightarrow{\tau} (\theta_i, e_i)}{\Gamma_t; \langle p, (\theta_{i-1}, e_{i-1}), [\ ]\rangle | \Pi_t \hookrightarrow \Gamma_t; \langle p, (\theta_i, e_i), [\ ]\rangle | \Pi_t}$$

  where $e_i\theta^* = e_i$ because no κ variable has been introduced.
- $l_i \equiv send(p', v')$
  Then we can perform a (Send) step on process p:

$$\text{(Send)}\ \frac{(\theta_{i-1}, e_{i-1}) \xrightarrow{send(p', v')} (\theta_i, e_i)}{\Gamma_t; \langle p, (\theta_{i-1}, e_{i-1}), [\ ]\rangle | \Pi_t \hookrightarrow \Gamma_t \cup (p', v'); \langle p, (\theta_i, e_i), [\ ]\rangle | \Pi_t}$$

  where $p'!v' \in submits([l_1, l_2, \ldots, l_n]) = m'$ and $e_i\theta^* = e_i$ because no κ variable has been introduced.
- $l_i \equiv spawn(\kappa, fname, [\overline{v_n}])$

[9] Since $e \neq mnf$ then the derivation does not use the rule (MNF), the only rule that requires 0 $\xrightarrow{l}$-steps.

Then we can perform a (Spawn) step on process p

$$\text{(Spawn)}\ \frac{(\theta_{i-1}, e_{i-1}) \xrightarrow{spawn(\kappa, fname, [\overline{v_n}])} (\theta_i, e_i) \qquad p' \text{ fresh}}{\Gamma_t;\ \langle p, (\theta_{i-1}, e_{i-1}), [\,]\rangle | \Pi_t \hookrightarrow \Gamma_t;\ \langle p, (\theta_i, e_i\{\kappa \mapsto p'\}), [\,]\rangle | \langle p', (id, \texttt{apply}\ fname(\overline{v_n})), [\,]\rangle | \Pi_t}$$

where p' is the ith process identifier generated the $\Rightarrow$ derivation. Then we can perform a (Seq) step in process p' using (Apply2):

$$\text{(Seq)}\ \frac{\mu(fname) = \texttt{fun}(\overline{X_n}) \to e}{\Gamma_t;\ \langle p, (\theta_i, e_i[\kappa \mapsto p']), [\,]\rangle | \langle p', (id, \texttt{apply}\ fname\ (\overline{v_n})), [\,]\rangle | \Pi_t \hookrightarrow \Gamma_t;\ \langle p, (\theta_i, e_i\{\kappa \mapsto p'\}), [\,]\rangle | \langle p', (\theta', e), [\,]\rangle | \Pi_t}$$

where $\theta' = \{\overline{X_n \mapsto v_n}\}$ as in the process generated by $\Rightarrow$ in Π'. Moreover, $e_i\{\kappa \mapsto p'\} = e_i\theta^*$ because θ^* collects the renaming of all κ variables during the $\Rightarrow$ derivation. Notice that $\langle p', (\theta', e), [\,]\rangle \in \| \Pi_1 \|$.

- (CONSUME$_1$): Then we have a derivation:

$$\text{(CONSUME}_1\text{)}\ \frac{\langle mnf, \theta\rangle \xrightarrow{m'_2, c} (\langle mnf', \theta'\rangle, m', \Pi')}{\Pi \cdot \ll p_1, \langle mnf, \theta\rangle, m_1 \gg, \ll p_2, \langle mnf_2, \theta_2\rangle, m_2 \gg \Rightarrow \Pi \cdot \ll p_1, \langle mnf', \theta'\rangle, m_1 + m' \gg, \ll p_2, \langle mnf_2, \theta_2\rangle, m''_2 \gg \cdot \Pi'}$$

where $m'_2 = m_2|_{p_1}$, m''_2 is the result of removing from m_2 the message consumed (the cth considering all the messages to p_1 in m_2).

Using Lemma A.2, there are $n > 0$ steps $(\theta, e) \xrightarrow{l_1} (\theta_1, e_1)$, $(\theta_1, e_1\theta^*) \xrightarrow{l_2} (\theta_2, e_2)$, …, $(\theta_{i-1}, e_{i-1}\theta^*) \xrightarrow{rec(\kappa, \overline{b_n})} (\theta_i, \kappa)$, $(\theta_i \uplus \{\kappa \mapsto e_i\} \uplus \theta'_i, e_i) \xrightarrow{l_{i+1}} (\theta_{i+1}, e_{i+1})$, …, $(\theta_{n-1}, e_{n-1}\theta^*) \xrightarrow{l_n} (\theta', e_n)$ using $\xrightarrow{l}$ [7] such that:

a) $e_n\theta^* = mnf'$
b) $l_1, \ldots, l_n$ are τ, $send(p, v)$ or $spawn(\kappa, fname, [\overline{v_n}])$ labels
c) $submits([l_1, l_2, \ldots, l_n]) = m'$
d) $spawns([l_1, l_2, \ldots, l_n], [\overline{p_j}]) = \Pi'$
e) $matchrec(\theta_{i-1}, \overline{b_n}, m'_2) = (\theta'_i, e_i, v)$

where θ^* is the substitution that maps the variables $\overline{\kappa_j}$ introduced in the $\xrightarrow{spawn(\kappa, fname, [\overline{v}])}$ steps to the fresh process identifiers generated in Π, in the same order they were created.

The translation of the initial configuration is

$$\begin{aligned} &\| \Pi \cdot \ll p_1, \langle mnf, \theta\rangle, m_1 \gg, \ll p_2, \langle mnf_2, \theta_2\rangle, m_2 \gg \| \\ =\ &\Gamma_t;\ \Pi_t | \langle p_1, (\theta, mnf), [\,]\rangle | \langle p_2, (\theta_2, mnf_2), [\,]\rangle \end{aligned}$$

where Γ_t contains the messages in m_1, m_2 and the rest of processes in Π, and Π_t is the translation of Π, so all the processes have empty mailboxes. As in the previous case, we can raise the first $i-1$ $\xrightarrow{l}$-steps to chained $\hookrightarrow$-steps using (Seq), (Send), or (Spawn)+(Seq). In the resulting system every process will have an empty mailbox, and the message $m'_2[c]$ consumed in the $\Rightarrow$ derivation will be in the global mailbox. Then we can perform a (Sched) step to place that message in the mailbox of process p_2, and then a (Receive) step to consume that message from the unitary mailbox of process p_2 (notice that by Proposition A.2 we know that *matchrec* in (Receive) will succeed). After that, we can chain the rest of $\xrightarrow{l}$ steps as $\hookrightarrow$-steps using (Seq), (Send), or (Spawn)+(Seq) (similar to the previous case) and reach a final system:

$$\begin{aligned} &(\Gamma_t \cup m') \setminus m''_2[c];\ \Pi_t | \langle p_1, (\theta', mnf'), [\,]\rangle | \langle p_2, (\theta_2, mnf_2), [\,]\rangle | \Pi'_t \\ =\ &\| \Pi \cdot \ll p_1, \langle mnf', \theta'\rangle, m_1 + m' \gg, \ll p_2, \langle mnf_2, \theta_2\rangle, m''_2 \gg \cdot \Pi' \| \end{aligned}$$

- (CONSUME$_2$): Similar to the (CONSUME$_1$) case, but omitting the second p_2 process.

In the case b) $\Pi_1 = endlock$ the only possible derivation is

$$\text{(ENDLOCK)}\ \frac{\text{(LOCK)}\ \dfrac{fails(mnf_1, m'_1, \theta_1)}{\langle mnf_1, \theta_1\rangle \xrightarrow{l'_1, 0} lock} \quad \ldots \quad \text{(LOCK)}\ \dfrac{fails(mnf_k, m'_k, \theta_k)}{\langle mnf_k, \theta_k\rangle \xrightarrow{l'_k, 0} lock}}{\Pi_0 \Rightarrow endlock}$$

$$\text{where } \Pi_0 \equiv \ll p_1, \langle mnf_1, \theta_1\rangle, m_1 \gg, \ldots, \ll p_k, \langle mnf_k, \theta_k\rangle, m_k \gg, \\ \ll p_{k+1}, v_{k+1}, m_{k+1} \gg, \ldots, \ll p_n, v_n, m_n \gg$$

and $m'_i \equiv m_1 + \ldots + m_n$ restricted to messages addressed for p_i for $i = 1 \ldots k$. The translation of Π_0 is

$$\| \Pi_0 \| = \Gamma_t; \langle p_1, (\theta_1, mnf_1), [\,]\rangle | \ldots | \langle p_k, (\theta_k, mnf_k), [\,]\rangle \\ | \langle p_{k+1}, (\theta_{k+1}, v_{k+1}), [\,]\rangle | \langle p_n, (\theta_n, v_n), [\,]\rangle |$$

where $\Gamma_t = \{c \mid c \in m_1 + \ldots + m_n\}$. If every outbox m_i is empty then this system cannot progress because by Lemma A.3 we know that a) $(\theta_{k+1}, v_{k+1}) \ldots (\theta_n, v_n)$ cannot be reduced, so those processes cannot progress by applying any rule for $\hookrightarrow$; and b) $(\theta_1, mnf_1) \ldots (\theta_k, mnf_k)$ can only be reduced using a $\xrightarrow{red(\kappa, \overline{b_m})}$ step that cannot be raised to a $\hookrightarrow$ step using (Receive) because all the mailboxes are empty.

If Γ_t contains some messages, it is possible to perform some $\hookrightarrow$ steps using (Sched) but any combination will lead to a system that cannot progress. As before, by Lemma A.3 we know that processes with values cannot progress (a), and $(\theta_1, mnf_1) \ldots (\theta_k, mnf_k)$ can only be reduced using a $\xrightarrow{red(\kappa, \overline{b_m})}$. However, these steps cannot be raised to $\hookrightarrow$ steps using the (Receive) rule because by Proposition A.3 the derivations $\langle mnf_i, \theta_1\rangle \xrightarrow{m'_i, 0} lock$ implies that $matchrec(\theta_1, \overline{b_i}, q)$ will fail for every queue q with a subset of the messages in m'_i. □

Lemma A.5. *Consider the derivation* $\Pi_0 \Rightarrow^+ \Pi_1$*:*

a) *If* $\Pi_1 \neq endlock$*, there is a minimum-delivery trace* $\| \Pi_0 \| \hookrightarrow^+ \| \Pi_1 \|$
b) *If* $\Pi_1 = endlock$ *then* $\| \Pi_0 \| \hookrightarrow^* \Gamma; \Pi$*, where* $\| \Pi_0 \|$ *is an empty-mailbox system and* $\Gamma; \Pi$ *is blocked*

Proof. If $\Pi_0 \Rightarrow^+ \Pi_1$ we have a derivation:

$$\text{(Tr)}\ \frac{\Pi_0 \Rightarrow \Pi'_1 \quad \Pi'_1 \Rightarrow \Pi'_2 \quad \ldots \quad \Pi'_{n-1} \Rightarrow \Pi_1 \quad n \geq 1}{\Pi_0 \Rightarrow^+ \Pi_1}$$

If $n = 1$ then there is only one $\Rightarrow$-step, so directly the Lemma A.4 proves the result. If $n \geq 1$ then Π_0, …, Π'_{n-1} are not *endlock*, as this configuration is final. Then by Lemma A.4 we have minimum-delivery traces $\| \Pi_0 \| \hookrightarrow^+ \| \Pi'_1 \|$, $\| \Pi'_1 \| \hookrightarrow^+ \| \Pi'_2 \|$, …, $\| \Pi'_{n-2} \| \hookrightarrow^+ \| \Pi_{n-1} \|$. Applying again Lemma A.4 to the last step, we have the minimum-delivery trace $\| \Pi_0 \| \hookrightarrow^+ \| \Pi_1 \|$, if $\Pi_1 \neq endlock$; or $\| \Pi_0 \| \overset{\text{(Sched)}}{\hookrightarrow}{}^* \Gamma; \Pi$ such that $\Gamma; \Pi$ is blocked, if $\Pi_1 = endlock$. In both cases we can concatenate the $\hookrightarrow$-steps and obtain the expected trace. □

Using the previous lemma we can address the proof of the soundness result that relates our big-step semantics $\Rightarrow$ to the small-step semantics $\hookrightarrow$.

Theorem 2.1 *(Soundness of* $\Rightarrow^+$ *w.r.t.* $\hookrightarrow_{mop}$*). Consider a derivation* $\Pi_0 \Rightarrow^+ \Pi_1$ *and a empty-mailbox system* $\Gamma; \Pi$ *such that* $\| \Pi_0 \| = \Gamma; \Pi$*. Then:*

a) $\Gamma; \Pi \hookrightarrow^+_{mop} \| \Pi_1 \|$*, if* $\Pi_1 \neq endlock$
b) $\Gamma; \Pi \hookrightarrow^*_{mop} \Gamma'; \Pi'$ *such that* $\Gamma'; \Pi'$ *is blocked, if* $\Pi_1 = endlock$

Proof. In both cases by Lemma A.5 we have a trace $\hookrightarrow^+$ such that every (Receive) step processes a unitary mailbox. If this trace is message-order-preserving then we are done; otherwise, we will transform it into a MOP trace in two steps:

1. For every (Receive) step in process p consuming a message m from process p', we will precede it with (Sched) steps delivering to p's mailbox those messages in Γ from p' submitted to p that are older than m and are not already in the mailbox. This process is made chronologically in the trace, and the resulting trace will satisfy the MOP conditions in Definition A.2. If the (Receive) steps still consume the same message in all the trace considering the new extended mailboxes, then we are done. Otherwise, we can to swap some steps.
2. Find the first (Receive) such that its evaluation has changed, i.e., in the original trace it consumed a message m from p but now consumed a previous message m' in its local mailbox. This message m' cannot proceed from p, as it should be older than m and that could not happen in the $\Rightarrow$ derivation. Then, applying Lemmas A.6 and A.7 we can move forward the (Sched) step delivering m (possibly with the (Send) and other previous steps of p) until the considered (Receive) step consumes m. We can repeat this process until all the (Receive) steps behave as in the original trace. □

A.3. Completeness proof

The completeness result states that any $\hookrightarrow_{mop}$ trace can be reproduced using $\Rightarrow^{+}$. As the considered syntax lacks tuples and lists compared to the one in [7], those expressions cannot appear in the systems. Therefore, there are rules presented in [7]—(Tuple), (List1) and (List2)—that cannot be applied, thus they are not considered in the proofs when distinguishing cases.

Lemma A.6 *(Swaping $\hookrightarrow$ steps of a process). Consider a two-step MOP trace $\Gamma_0; \Pi_0 \overset{p}{\hookrightarrow}_{(\mathsf{Sched})} \Gamma_1; \Pi_1 \overset{p}{\hookrightarrow}_{b} \Gamma_2; \Pi_2$ where b is* (Seq), (Send) *or* (Spawn). *Then the two steps can be swapped without changing the result, i.e., $\Gamma_0; \Pi_0 \overset{p}{\hookrightarrow}_{b} \Gamma'; \Pi' \overset{p}{\hookrightarrow}_{(\mathsf{Sched})} \Gamma_2; \Pi_2$, and the result is a MOP trace.*

Proof. By case distinction. In all the cases the first step is as follows:

$$(\mathsf{Sched})\ \frac{}{\Gamma \cup \{p!v\}; \langle p, (\theta, e), q\rangle|\Pi' \hookrightarrow \Gamma; \langle p, (\theta, e), v : q\rangle|\Pi'}$$

- $b = (\mathsf{Seq})$. The second step has the derivation:

$$(\mathsf{Seq})\ \frac{\theta, e \to \theta', e'}{\Gamma; \langle p, (\theta, e), v : q\rangle|\Pi' \hookrightarrow \Gamma; \langle p, (\theta', e'), v : q\rangle|\Pi'}$$

  Then we can swap the steps and obtain the following sequence

$$\begin{array}{rl} \Gamma \cup \{p!v\}; \langle p, (\theta, e), q\rangle|\Pi' & \hookrightarrow_{(\mathsf{Seq})} \ \Gamma \cup \{p!v\}; \langle p, (\theta', e'), q\rangle|\Pi' \\ & \hookrightarrow_{(\mathsf{Sched})} \ \Gamma; \langle p, (\theta', e'), v : q\rangle|\Pi' \end{array}$$

  Note that the inner system is MOP because it contains the same global and local mailbox as the first one.
- $b = (\mathsf{Send})$. The second step has the derivation:

$$(\mathsf{Send})\ \frac{\theta, e \xrightarrow{send(p', v')} \theta', e'}{\Gamma; \langle p, (\theta, e), v : q\rangle|\Pi' \hookrightarrow \Gamma \cup \{p'!v'\}; \langle p, (\theta', e'), v : q\rangle|\Pi'}$$

  Then we can swap the steps and reach the same final system with the following derivations:

$$(\mathsf{Send})\ \frac{\theta, e \xrightarrow{send(p', v')} \theta', e'}{\Gamma \cup \{p!v\}; \langle p, (\theta, e), q\rangle|\Pi' \hookrightarrow \Gamma \cup \{p!v, p'!v'\}; \langle p, (\theta', e'), q\rangle|\Pi'}$$

  and

$$(\mathsf{Sched})\ \frac{}{\Gamma \cup \{p!v, p'!v'\}; \langle p, (\theta', e'), q\rangle|\Pi' \hookrightarrow \Gamma \cup \{p'!v'\}; \langle p, (\theta', e'), v : q\rangle|\Pi'}$$

  Note that the message $p'!v'$ is newer than $p!v$, so the inner system is MOP.
- $b = (\mathsf{Spawn})$. Similar to the previous case. Note that the (Spawn) step creates a new process p' which is different from the message p receiving the message in (Sched). Also, the inner system is MOP because the global mailbox and local mailbox of process p are the same, and the new local mailbox of p' is empty. □

Lemma A.7 *(Swaping $\hookrightarrow$ steps). Consider a two-step MOP trace $\Gamma_0; \Pi_0 \overset{p_1}{\hookrightarrow}_{b_1} \Gamma_1; \Pi_1 \overset{p_2}{\hookrightarrow}_{b_2} \Gamma_2; \Pi_2$ such that $p_1 \neq p_2$ and b_i is the rule used in each step. Then the two steps can be swapped to create a MOP trace $\Gamma_0; \Pi_0 \overset{p_2}{\hookrightarrow}_{b_2} \Gamma'; \Pi' \overset{p_1}{\hookrightarrow}_{b_1} \Gamma_2; \Pi_2$ if they do not verify that:*

- *The first step in p_1 is* (Send) *submitting the message $p_2!v$, and the second step is* (Sched) *delivering the same message $p_2!v$ to p_2.*
- *The first step is* (Spawn) *creating process p_2.*

Proof. By case distinction on the rules used. There are 5 different rules, so potentially there are 25 combinations. The proof in all the cases is similar, so we will detail only some of them:

- (Send) and (Sched), considering that the message sent ($p'!v'$) is not the one delivered by (Sched), i.e., $p_2!v$. In this case the trace if composed by this two steps:

$$(\mathsf{Send})\ \frac{\theta_1, e_1 \xrightarrow{send(p'!v')} \theta_1', e_!'}{\begin{array}{l}\Gamma \cup \{p_2!v\}; \langle p_1, (\theta_1, e_1), q_1\rangle|\langle p_2, (\theta_2, e_2), q_2\rangle|\Pi' \overset{p_1}{\hookrightarrow} \\ \Gamma \cup \{p_2!v, p'!v'\}; \langle p_1, (\theta_1', e_1'), q_1\rangle|\langle p_2, (\theta_2, e_2), q_2\rangle|\Pi'\end{array}}$$

and

$$\text{(Sched)}\ \frac{}{\begin{array}{l}\Gamma \cup \{p_2!v, p'!v'\};\ \langle p_1, (\theta_1', e_1'), q_1\rangle \mid \langle p_2, (\theta_2, e_2), q_2\rangle \mid \Pi' \overset{p_2}{\hookrightarrow} \\ \Gamma \cup \{p'!v'\};\ \langle p_1, (\theta_1', e_1'), q_1\rangle \mid \langle p_2, (\theta_2, e_2), v : q_2\rangle \mid \Pi'\end{array}}$$

If we swap the steps we reach the same final system, because:

$$\text{(Sched)}\ \frac{}{\begin{array}{l}\Gamma \cup \{p_2!v\};\ \langle p_1, (\theta_1, e_1), q_1\rangle \mid \langle p_2, (\theta_2, e_2), q_2\rangle \mid \Pi' \overset{p_2}{\hookrightarrow} \\ \Gamma;\ \langle p_1, (\theta_1, e_1), q_1\rangle \mid \langle p_2, (\theta_2, e_2), v : q_2\rangle \mid \Pi'\end{array}}$$

and

$$\text{(Send)}\ \frac{\theta_1, e_1 \xrightarrow{send(p'!v')} \theta_1', e_1'}{\begin{array}{l}\Gamma;\ \langle p_1, (\theta_1, e_1), q_1\rangle \mid \langle p_2, (\theta_2, e_2), v : q_2\rangle \mid \Pi' \overset{p_1}{\hookrightarrow} \\ \Gamma \cup \{p'!v'\};\ \langle p_1, (\theta_1', e_1'), q_1\rangle \mid \langle p_2, (\theta_2, e_2), v : q_2\rangle \mid \Pi'\end{array}}$$

The inner system is MOP because the message delivered to the local mailbox of p_2 is older than the newly submitted message $p'!v'$.

- (Spawn) and (Seq), when the spawned process is $p' \neq p_2$. In this case the two steps are:

$$\text{(Spawn)}\ \frac{\theta_1, e_1 \xrightarrow{spawn(\kappa, fname, [\overline{v_n}])} \theta_1', e_1' \qquad p' \text{ fresh}}{\begin{array}{l}\Gamma;\ \langle p_1, (\theta_1, e_1), q_1\rangle \mid \langle p_2, (\theta_2, e_2), q_2\rangle \mid \Pi \overset{p_1}{\hookrightarrow} \Gamma;\ \langle p_1, (\theta_1', e_1'\{\kappa \mapsto p'\}), q_1\rangle \\ \quad \mid \langle p_2, (\theta_2, e_2), q_2\rangle \mid \langle p', (id, \texttt{apply}\ fname(\overline{v_n})), [\,]\rangle \mid \Pi\end{array}}$$

and

$$\text{(Seq)}\ \frac{\theta_2, e_2 \xrightarrow{\tau} \theta_2', e_2'}{\begin{array}{l}\Gamma;\ \langle p_1, (\theta_1', e_1'\{\kappa \mapsto p'\}), q_1\rangle \mid \langle p_2, (\theta_2, e_2), q_2\rangle \mid \langle p', (id, \texttt{apply}\ fname(\overline{v_n})), [\,]\rangle \mid \Pi \overset{p_2}{\hookrightarrow} \\ \Gamma;\ \langle p_1, (\theta_1', e_1'\{\kappa \mapsto p'\}), q_1\rangle \mid \langle p_2, (\theta_2', e_2'), q_2\rangle \mid \langle p', (id, \texttt{apply}\ fname(\overline{v_n})), [\,]\rangle \mid \Pi\end{array}}$$

Then we can swap the steps as follows:

$$\text{(Spawn)}\ \frac{\text{(Seq)}\ \dfrac{\theta_2, e_2 \xrightarrow{\tau} \theta_2', e_2'}{\begin{array}{l}\Gamma;\ \langle p_1, (\theta_1, e_1), q_1\rangle \mid \langle p_2, (\theta_2, e_2), q_2\rangle \mid \Pi \overset{p_2}{\hookrightarrow} \\ \Gamma;\ \langle p_1, (\theta_1, e_1), q_1\rangle \mid \langle p_2, (\theta_2', e_2'), q_2\rangle \mid \Pi\end{array}} \qquad \theta_1, e_1 \xrightarrow{spawn(\kappa, fname, [\overline{v_n}])} \theta_1', e_1' \qquad p' \text{ fresh}}{\begin{array}{l}\Gamma;\ \langle p_1, (\theta_1, e_1), q_1\rangle \mid \langle p_2, (\theta_2', e_2'), q_2\rangle \mid \Pi \overset{p_1}{\hookrightarrow} \\ \Gamma;\ \langle p_1, (\theta_1', e_1'\{\kappa \mapsto p'\}), q_1\rangle \mid \langle p_2, (\theta_2', e_2'), q_2\rangle \mid \langle p', (id, \texttt{apply}\ fname(\overline{v_n})), [\,]\rangle \mid \Pi\end{array}}$$

The inner system is MOP because the local mailboxes are the same as the first system of the derivation.

- (Sched) and (Receive). In this case the first step delivers a message to the local mailbox of p_1, and the second step consumes a message from the local mailbox of p_2, so they are independent. The inner system is MOP because the local mailboxes are the same as in the first system, but with the local mailbox of p_2 reduced by one message. □

Lemma A.8 *(Grouping $\hookrightarrow$ steps). Consider a MOP trace $\Gamma_1; \Pi_1 \hookrightarrow^+ \Gamma_{m+1}; \Pi_{m+1}$ such that $\Gamma_{m+1}; \Pi_{m+1}$ is MNF, then it can be converted to a* grouped MOP trace

$$\Gamma_1; \Pi_1 \overset{p^1}{\hookrightarrow}^+ \Gamma_2; \Pi_2 \overset{p^2}{\hookrightarrow}^+ \ldots \overset{p^m}{\hookrightarrow}^+ \Gamma_{m+1}; \Pi_{m+1}$$

where each group of steps $\Gamma_i; \Pi_i \overset{p^i}{\hookrightarrow}^+ \Gamma_{i+1}; \Pi_{i+1}$ contains steps on the same process p^i such that (a) they do not change the expression, or (b) they evaluate the expression to the next MNF.

Proof. Consider a trace that is not grouped. Then it will contain a suffix that can be divided as:

$$\Gamma_1'; \Pi_1' \overset{p}{\hookrightarrow}^+ \Gamma_2'; \Pi_2' \hookrightarrow^+ \Gamma_3'; \Pi_3' \overset{p}{\hookrightarrow}_b \Gamma_4'; \Pi_4' \hookrightarrow^+ \Gamma_{m+1}; \Pi_{m+1}$$

where the expression for process p in Π'_2 is not MNF, and the derivation $\Gamma'_2; \Pi'_2 \hookrightarrow^+ \Gamma'_3; \Pi'_3$ does not contain any step on process p. Then:

- If the step $\overset{p}{\hookrightarrow}_b$ can be swapped using repeatedly Lemma A.7 with all the steps in $\Gamma'_2; \Pi'_2 \hookrightarrow^+ \Gamma'_3; \Pi'_3$ then we can create the equivalent trace

$$\Gamma'_1; \Pi'_1 \overset{p}{\hookrightarrow}^+ \Gamma'_2; \Pi'_2 \overset{p}{\hookrightarrow}_b \Gamma'; \Pi' \hookrightarrow^+ \Gamma'_4; \Pi'_4 \hookrightarrow^+ \Gamma_{m+1}; \Pi_{m+1}$$

  Regardless of the expression for the process p in Π' (it can be in MNF or not) we can repeat this process in the rest of the trace $\Gamma'; \Pi' \hookrightarrow^+ \Gamma'_4; \Pi'_4 \hookrightarrow^+ \Gamma_{m+1}; \Pi_{m+1}$ until it forms a grouped trace.
- According to Lemma A.7, if the step cannot be swapped is because the trace $\Gamma'_2; \Pi'_2 \hookrightarrow^+ \Gamma'_3; \Pi'_3$ contains a (Send) step on a process $p' \neq p$ that submits a message (p, v) to the global mailbox, and the step $\Gamma'_3; \Pi'_3 \overset{p}{\hookrightarrow}_b \Gamma'_4; \Pi'_4$ uses (Sched) to deliver that message to the local mailbox of process p. As the expression of process p in Π'_3 is not in MNF and it is not changed in Π'_4 because the step is (Sched), then in the derivation $\Gamma'_4; \Pi'_4 \hookrightarrow^+ \Gamma_{m+1}; \Pi_{m+1}$ there will be a first step b' on p different from (Sched) and (Receive)–the latter needs a MNF to be applied. Then using Lemmas A.7 and A.6 we can swap that step and place it next to the steps for p:

$$\Gamma'_1; \Pi'_1 \overset{p}{\hookrightarrow}^+ \Gamma'_2; \Pi'_2 \overset{p}{\hookrightarrow}_{b'} \Gamma'; \Pi' \hookrightarrow^+ \Gamma''; \Pi'' \overset{p}{\hookrightarrow}_b \Gamma'''; \Pi''' \hookrightarrow^+ \Gamma_{m+1}; \Pi_{m+1}$$

  As before, regardless of the expression for the process p in Π' (it can be in MNF or not) we can repeat this process in shorter trace $\Gamma'; \Pi' \hookrightarrow^+ \Gamma''; \Pi'' \overset{p}{\hookrightarrow}_b \Gamma'''; \Pi''' \hookrightarrow^+ \Gamma_{m+1}; \Pi_{m+1}$ until it forms a grouped trace.
  Another possibility that could prevent the swapping of the step $\overset{p}{\hookrightarrow}_b$ according to Lemma A.7 is that $\Gamma'_2; \Pi'_2 \hookrightarrow^+ \Gamma'_3; \Pi'_3$ contains a (Spawn) step creating process p. This cannot happen either, as the previous derivation $\Gamma'_1; \Pi'_1 \overset{p}{\hookrightarrow}^+ \Gamma'_2; \Pi'_2$ operates on process p, so it already exists.

The resulting trace is still MOP because the relative order of the (Sched) steps for each process has not changed. □

Lemma A.9. *If* $(\theta_1, e_1 \xrightarrow{\tau} \theta_2, mnf)$ *then* $\langle e_1, \theta_1 \rangle \to (\langle mnf, \theta_2 \rangle, [\,], \emptyset)$.

Proof. By induction on the size of the $\xrightarrow{\tau}$ derivation. The destination expression must be in MNF, so the only applicable rules are (Var), (Let1), (Let2), (Case2), (Apply2), and (Call2).

Base Case: derivation of size $s = 1$. There are several cases.

- (Var): In this case we have a derivation

$$\text{(Var)}\ \frac{}{\theta_1, X \xrightarrow{\tau} \theta_1, \theta_1(X)}$$

  Then we can create the following derivation using the (VAR) rule:

$$\text{(VAR)}\ \frac{}{\langle X, \theta_1 \rangle \to (\langle \theta_1(X), \theta_1 \rangle, [\,], \emptyset)}$$

- (Let2): The derivation is:

$$\text{(Let2)}\ \frac{}{\theta_1, let\ X = v\ in\ mnf \xrightarrow{\tau} \theta_1 \uplus \{X \mapsto v\}, mnf}$$

  Then we can generate a derivation with (LET_1) and (MNF):

$$\text{(LET_1)}\ \frac{\text{(MNF)}\ \dfrac{}{\langle v, \theta_1 \rangle \to (\langle v, \theta_1 \rangle, [\,], \emptyset)} \qquad \text{(MNF)}\ \dfrac{}{\langle mnf, \theta_2 \rangle \to (\langle mnf, \theta_2 \rangle, [\,], \emptyset)}}{\langle let\ X = v\ in\ mnf, \theta_1 \rangle \to (\langle mnf, \theta_2 \rangle, [\,], \emptyset)}$$

  where $\theta_2 = \theta_1 \uplus \{X \mapsto v\}$.
- (Case2): We have the following derivation:

$$\text{(Case2)}\ \frac{match(\theta_1, v, \overline{b_n}) = (\theta', mnf)}{\theta_1, \texttt{case}\ v\ \texttt{of}\ \overline{b_n}\ \texttt{end} \xrightarrow{\tau} \theta_1 \uplus \theta', mnf}$$

By Proposition A.1 we have that $succeeds\_case(\theta_1, v, \overline{b_n}) = (mnf, \theta_1 \uplus \theta')$. Then we can build the following step using (CASE) and (MNF):

$$\text{(CASE)}\ \frac{\langle v, \theta_1\rangle \to (\langle v, \theta\rangle, [\,], \emptyset) \quad succeeds\_case(\theta_1, v, \overline{b_n}) = (mnf, \theta_1 \uplus \theta') \quad \langle mnf, \theta_1 \uplus \theta'\rangle \to (\langle mnf, \theta_1 \uplus \theta'\rangle, [\,], \emptyset)}{\langle \texttt{case}\ v\ \texttt{of}\ \overline{b_n}\ \texttt{end}, \theta_1\rangle \to (\langle mnf, \theta_1 \uplus \theta'\rangle, [\,], \emptyset}$$

- (Apply2): The derivation is

$$\text{(Apply2)}\ \frac{}{\theta_1, \texttt{apply}\ fname(\overline{v_n}) \xrightarrow{\tau} \theta_1 \uplus \{\overline{X_n \mapsto v_n}\}, mnf}$$

where $fname = \texttt{fun}(\overline{X_n}) \to mnf$. Then we can create a derivation using rule (APPLY) and (MNF):

$$\text{(APPLY)}\ \frac{\langle v_i, \theta_i\rangle \to (\langle v_i, \theta_i\rangle, [\,], \emptyset) \quad \ldots \quad \langle v_n, \theta_n\rangle \to (\langle v_n, \theta_n\rangle, [\,], \emptyset) \quad \langle mnf, \theta_2\rangle \to (\langle mnf, \theta_2\rangle, [\,], \emptyset)}{\langle \texttt{apply}\ fname(\overline{v_n}), \theta_1\rangle \to (\langle mnf, \theta_2\rangle, [\,], \emptyset)}$$

where $\theta_2 = \theta_1 \uplus \{\overline{X_n \mapsto v_n}\}$.
- (Call2): similar to the (Apply2) case.

Inductive Step: derivation of size $s > 1$. The only possible step is (Let1), so we have a derivation:

$$\text{(Let1)}\ \frac{\theta_1, e \xrightarrow{\tau} \theta_2, mnf}{\theta_1, let\ X = e\ in\ e' \xrightarrow{\tau} \theta_2, let\ X = mnf\ in\ e'}$$

Then by the Induction Hypothesis we have that $\langle e, \theta_1\rangle \to (\langle mnf, \theta_2\rangle, [\,], \emptyset)$, and we can create a derivation using rule (LET_2):

$$\text{(LET}_1\text{)}\ \frac{\langle e, \theta_1\rangle \to (\langle mnf, \theta_2\rangle, [\,], \emptyset) \quad mnf \text{ is not a value}}{\langle let\ X = e\ in\ e', \theta_1\rangle \to (\langle let\ X = mnf\ in\ e', \theta_2\rangle, [\,], \emptyset)}$$

Note that *mnf* cannot be a value because the final expression $let\ X = mnf\ in\ e'$ is in MNF. □

Proposition A.5. *If $matchrec(\theta, \overline{b_n}, q) = (\theta_i, e, v)$, the message v was submitted by process p, m is a sequence of messages from p such that starts with the messages in q from p* in the same order, *and mnf contains a* `receive` *expression with branches $\overline{b_n}$ in the active position, then there is a position j such that $succeeds(mnf, m, j, \theta) = (e, \theta')$, $m[j] = v$, and $\theta' = \theta \uplus \theta_i$.*

Proof. Straightforward, since *succeeds* and *mathcrec* express the same behav- ior for Erlang message matching against receive expressions. □

Lemma A.10. *Consider a MOP $\hookrightarrow$-derivation on the same process p*

$$\Gamma_1; \langle p, (\theta, e), q_1\rangle|\Pi \overset{p}{\hookrightarrow} \Gamma_2; \langle p, (\theta, e), q_2\rangle|\Pi \overset{p}{\hookrightarrow} \ldots \overset{p}{\hookrightarrow} \Gamma_n; \langle p, (\theta, e), q_n\rangle|\Pi$$

Then $(\!|\Gamma_1; \langle p, (\theta, e), q_1\rangle|\Pi|\!) = (\!|\Gamma_n; \langle p, (\theta, e), q_n\rangle|\Pi|\!)$.

Proof. The only possible steps that do not modify the expression of a process are (Sched). These steps move messages from the global mailbox to the local mailbox. As the translation in Definition 2.2 collects all the messages independently of the original (global or local mailbox), the translation is the same. □

Lemma A.11. *If $(\theta, e) \xrightarrow{\tau} (\theta', e')$ and $\langle e', \theta'\rangle \to (\langle mnf, \theta''\rangle, l, \Pi)$ then $\langle e, \theta\rangle \to (\langle mnf, \theta''\rangle, l, \Pi)$.*

Proof. By induction on the size of the derivation $(\theta, e) \xrightarrow{\tau} (\theta', e')$.

Base Case:

- (Var): In this case the expression *mnf* must be a value v, $e = X$, $\theta' = \theta$, $\theta(X) = v$, and the $\to$-step used is (MNF)—i.e., $\theta'' = \theta$, $l = [\,]$, and $\Pi = \emptyset$). Then we can create a derivation:

$$\text{(VAR)}\ \frac{}{\langle X, \theta\rangle \to (\langle \theta(X), \theta\rangle, [\,], \emptyset)}$$

- (Let2): We have a derivation

$$\text{(Let2)}\ \frac{}{\theta, \texttt{let}\ X = v\ \texttt{in}\ e_2 \xrightarrow{\tau} \theta \uplus \{X \to v\}, e_2}$$

and $\langle e_2, \theta \uplus \{X \to v\}\rangle \to (\langle mnf, \theta''\rangle, l, \Pi)$. Then we can build the following derivation:

$$\text{(LET}_1\text{)}\ \frac{\text{(MNF)}\ \dfrac{}{\langle v, \theta\rangle \to (\langle v, \theta\rangle, [\,], \emptyset)} \qquad \langle e_2, \theta \uplus \{X \to v\}\rangle \to (\langle mnf, \theta''\rangle, l, \Pi)}{\langle \texttt{let}\ X = v\ \texttt{in}\ e_2, \theta\rangle \to (\langle mnf, \theta''\rangle, l, \Pi)}$$

- (Case2): We have the following derivation:

$$\text{(Case2)}\ \frac{match(\theta_1, v, \overline{b_n}) = (\theta', e)}{\theta_1, \texttt{case}\ v\ \texttt{of}\ \overline{b_n}\ \texttt{end} \xrightarrow{\tau} \theta_1 \uplus \theta', e}$$

By Proposition A.1 we have that $succeeds\_case(\theta_1, v, \overline{b_n}) = (mnf, \theta_1 \uplus \theta')$. Then we can build the following step using (CASE) and (MNF):

$$\text{(CASE)}\ \frac{\begin{array}{c}\langle v, \theta_1\rangle \to (\langle v, \theta\rangle, [\,], \emptyset) \qquad succeeds\_case(\theta_1, v, \overline{b_n}) = (e, \theta_1 \uplus \theta') \\ \langle e, \theta_1 \uplus \theta'\rangle \to (\langle mnf, \theta_1 \uplus \theta'\rangle, l, \Pi)\end{array}}{\langle \texttt{case}\ v\ \texttt{of}\ \overline{b_n}\ \texttt{end}, \theta_1\rangle \to (\langle mnf, \theta_1 \uplus \theta'\rangle, l, \Pi)}$$

- (Apply2): In this case the derivation is

$$\text{(Apply2)}\ \frac{}{\theta, \texttt{apply}\ fname(\overline{v_n}) \xrightarrow{\tau} \theta \uplus \{\overline{X_n \to v_n}\}, e}$$

where $fname = \texttt{fun}(\overline{X_n}) \to e$. On the other hand, we have that $\langle e, \theta \uplus \{\overline{X_n \to v_n}\}\rangle \to (\langle mnf, \theta''\rangle, l, \Pi)$. Then we can create a derivation using the rule (APPLY):

$$\text{(APPLY)}\ \frac{\text{(MNF)}\ \dfrac{}{\langle v_1, \theta\rangle \to (\langle v_1, \theta\rangle, [\,], \emptyset)} \ \ldots\ \text{(MNF)}\ \dfrac{}{\langle v_n, \theta\rangle \to (\langle v_n, \theta\rangle, [\,], \emptyset)} \qquad \langle e, \theta \uplus \{\overline{X_n \mapsto v_n}\}\rangle (\langle mnf, \theta''\rangle, l, \Pi)}{\langle \texttt{apply}\ fname(\overline{v_n}), \theta\rangle \to (\langle mnf, \theta''\rangle, l, \Pi)}$$

- (Call2): similar to the (Apply2) case.

Inductive Step:

- (Let1): In this case we have the following $\xrightarrow{\tau}$-derivation.

$$\text{(Let1)}\ \frac{\textbf{(A)}\ \theta, e_1 \xrightarrow{\tau} \theta', e_1'}{\theta, \texttt{let}\ X = e_1\ \texttt{in}\ e_2 \xrightarrow{\tau} \theta', \texttt{let}\ X = e_1'\ \texttt{in}\ e_2}$$

The expression $\texttt{let}\ X = e'_1\ \texttt{in}\ e_2$ can be evaluated using (LET_1) or (LET_2), depending on the result of evaluating e'_1 (a value or an MNF). Suppose it is evaluated to a value:

$$\text{(LET}_1\text{)}\ \frac{\textbf{(B)}\ \langle e_1', \theta'\rangle \to (\langle v, \theta^*\rangle, l, \Pi) \qquad \langle e_2', \theta^* \uplus \{X \mapsto v\}\rangle \to (\langle mnf, \theta''\rangle) l', \Pi')}{\langle \texttt{let}\ X = e'_1\ \texttt{in}\ e_2, \theta'\rangle \to (\langle mnf, \theta''\rangle, l + l', \Pi \cdot \Pi')}$$

By Induction Hypothesis using (A) and (B) we have that $\langle e_1, \theta\rangle \to (\langle v, \theta^*\rangle, l, \Pi)$. Then we can build a derivation

$$\text{(LET}_1\text{)}\ \frac{\langle e_1, \theta\rangle \to (\langle v, \theta^*\rangle, l, \Pi) \qquad \langle e_2', \theta^* \uplus \{X \mapsto v\}\rangle \to (\langle mnf, \theta''\rangle) l', \Pi')}{\langle \texttt{let}\ X = e_1\ \texttt{in}\ e_2, \theta\rangle \to (\langle mnf, \theta''\rangle, l + l', \Pi \cdot \Pi')}$$

If e'_1 is evaluated to a MNF different from a value, the reasoning is similar but using (LET_2).

- (Call1): If this rule is applied is because the ith parameter of the call is the first one that is not a value, so the $\xrightarrow{\tau}$ derivation will be:

$$\text{(Call1)}\ \frac{\text{(Var)}\ \dfrac{}{\textbf{(A)}\ \theta, X_i \xrightarrow{\tau} \theta, v_i}}{\theta, \texttt{call}\ fname(\overline{v_{1,i-1}}, X_i, \overline{vv_{i+1,n}}) \xrightarrow{\tau} \theta, \texttt{call}\ fname(\overline{v_{1,i-1}}, v_i, \overline{vv_{i+1,n}})}$$

where $\theta(X_i) = v_i$. The expression $\texttt{call}\ fname(\overline{v_{1,i-1}}, v_i, \overline{vv_{i+1,n}})$ can only be evaluated with the rule (CALL), so we have:

$$\text{(CALL)}\ \frac{\begin{array}{c}\langle v_1,\theta\rangle \to (\langle v_1,\theta\rangle,[\,],\emptyset)\\ \vdots\\ \textbf{(B)}\ \langle v_i,\theta\rangle \to (\langle v_i,\theta\rangle,[\,],\emptyset)\\ \vdots\\ \langle vv_n,\theta\rangle \to (\langle v_n,\theta\rangle,[\,],\emptyset)\\ eval(fname,v_1,\ldots,v_n)=v\end{array}}{\langle \texttt{call}\ fname(\overline{v_{1,i-1}},v_i,\overline{vv_{i+1,n}}),\theta\rangle \to (\langle v,\theta''\rangle,[\,],\emptyset)}$$

Note that in this case *mnf* is a value v and $\theta''=\theta$. By the Induction Hypothesis from (A) and (B) we have $(\langle X_i,\theta\rangle)\to(\langle v_i,\theta\rangle,[\,],\emptyset)$. Then we can build the derivation:

$$\text{(CALL)}\ \frac{\begin{array}{c}\langle v_1,\theta\rangle \to (\langle v_1,\theta\rangle,[\,],\emptyset)\\ \vdots\\ \langle X_i,\theta\rangle \to (\langle v_i,\theta\rangle,[\,],\emptyset)\\ \vdots\\ \langle vv_n,\theta\rangle \to (\langle v_n,\theta\rangle,[\,],\emptyset)\\ eval(fname,v_1,\ldots,v_n)=v\end{array}}{\langle \texttt{call}\ fname(\overline{v_{1,i-1}},X_i,\overline{vv_{i+1,n}}),\theta\rangle \to (\langle v,\theta''\rangle,[\,],\emptyset)}$$

- (Case1), (Apply1), (Send1), (Send2), and (Spawn1) are similar to the (Call1) case. □

Lemma A.12. *Consider a MOP $\hookrightarrow$-derivation on the same process p*

$$\begin{array}{l}\Gamma_1;\langle p,(\theta_1,e_1),q_1\rangle|\Pi_1\\ \overset{p}{\hookrightarrow}\ \Gamma_2;\langle p,(\theta_2,e_2),q_2\rangle|\Pi_2\\ \overset{p}{\hookrightarrow}\ \Gamma_3;\langle p,(\theta_3,e_3),q_3\rangle|\Pi_3\\ \vdots\\ \overset{p}{\hookrightarrow}\ \Gamma_n;\langle p,(\theta_n,mnf),q_n\rangle|\Pi_n\end{array}$$

where $e_1\neq mnf$ and the sequence of steps evaluates e_1 to the next MNF (i.e., only e_1 and mnf can be MNF expressions in $\{\overline{e_i}\}$). Then it is possible to perform a step $(\!|\Gamma_1;\langle p,(\theta_1,e_1),q_1\rangle|\Pi|\!)\Rightarrow(\!|\Gamma_n;\langle p,(\theta_n,mnf),q_n\rangle|\Pi_n|\!)$ or $(\!|\Gamma_1;\langle p,(\theta_1,e_1),q_1\rangle|\Pi|\!)=(\!|\Gamma_n;\langle p,(\theta_n,mnf),q_n\rangle|\Pi_n|\!)$.

Proof. By induction on the number k of steps:
Base Case: $k=1$, i.e., $\Gamma_1;\langle p,(\theta_1,e_1),q_1\rangle|\Pi_1\overset{p}{\hookrightarrow}\Gamma_1;\langle p,(\theta_2,mnf),q_2\rangle|\Pi_2$. We perform a case distinction:

- (Seq): Then e_1 cannot be in MNF—because MNFs are evaluated using (Receive)—and we have a derivation:

$$\text{(Seq)}\ \frac{\theta_1,e_1\xrightarrow{\tau}\theta_2,mnf}{\Gamma_1;\langle p,(\theta_1,e_1),q_1\rangle|\Pi_1\hookrightarrow\Gamma_1;\langle p,(\theta_2,mnf),q_1\rangle|\Pi_1}$$

because (Seq) steps do not modify the global or local mailbox, and they do not generate new processes. If the step evaluates a newly created process p marked with * then $\theta_1=id$, $e_1=\texttt{apply}^*\ fname(\overline{v_n})$, $fname=\texttt{fun}(\overline{X_n}\to mnf)$, $e_2=mnf$, and $\theta_2=\{\overline{X_n\mapsto v_n}\}$. In this case.

$$(\!|\Gamma_1;\langle p,(id,\texttt{apply}^*\ fname(\overline{v_n}),q_1\rangle|\Pi_1|\!)=(\!|\Gamma_1;\langle p,(\{\overline{X_n\mapsto v_n}\},mnf),q_1\rangle|\Pi_1|\!)$$

Otherwise, the translation of the original system will be:

$$(\!|\Gamma_1;\langle p,(\theta_1,e_1),q_1\rangle|\Pi_1|\!)=\ll p,\langle e_1,\theta_1\rangle,l_1\gg\cdot\Pi'$$

by Lemma A.9 we have that $\langle e_1,\theta_1\rangle\to(\langle mnf,\theta_2\rangle,[\,],\emptyset)$, so we can create a $\Rightarrow$-step using (PROC):

$$\text{(PROC)}\ \frac{\langle e_1,\theta_1\rangle\to(\langle mnf,\theta_2\rangle,[\,],\emptyset)\qquad e_1\text{ is not MNF}}{\ll p,\langle e_1,\theta_1\rangle,l_1\gg\cdot\Pi'\Rightarrow\ll p,\langle mnf,\theta_2\rangle,l_1\gg\cdot\Pi'}$$

Clearly

$$(\!|\Gamma_1;\langle p,(\theta_2,mnf),q_1\rangle|\Pi_1|\!)=\ll p,\langle mnf,\theta_2\rangle,l_1\gg\cdot\Pi'$$

- (Send): In this case we have a derivation where *mnf* is a value:

$$\text{(Send)}\ \frac{\text{(Send3)}\ \dfrac{}{\theta_1, p'' \,!\, mnf \xrightarrow{send(p'', mnf)} \theta_1, mnf}}{\Gamma_1; \langle p, (\theta_1, p'' \,!\, mnf), q_1\rangle | \Pi_1 \hookrightarrow \Gamma_1 \cup (p'', mnf); \langle p, (\theta_1, mnf), q_1\rangle | \Pi_1}$$

  The translation of the system will be:

$$(\!|\Gamma_1; \langle p, (\theta_1, p'' \,!\, mnf), q_1\rangle | \Pi_1|\!) = \ll p, \langle p''!mnf, \theta_1\rangle, l_1 \gg \cdot \Pi'$$

  Then we can create a $\Rightarrow$-step using (PROC) and (BANG) as:

$$\text{(PROC)}\ \frac{\text{(BANG)}\ \dfrac{\langle p'', \theta_1\rangle \to (\langle p'', \theta_1\rangle, [\,], \emptyset) \qquad \langle mnf, \theta_1\rangle \to (\langle mnf, \theta_1\rangle, [\,], \emptyset)}{\langle p''!mnf, \theta_1\rangle \to (\langle mnf, \theta_2\rangle, [p''!mnf], \emptyset)}}{\ll p, \langle p''!mnf, \theta_1\rangle, l_1 \gg \Rightarrow \ll p, \langle mnf, \theta_2\rangle, l_1 + [p''!mnf] \gg \cdot \Pi'}$$

  Finally, the obtained configuration is the translation of the destination system as the expression and context is the same, and the message $p''!mnf$ will be moved to the outbox of the sender:

$$(\!|\Gamma_1 \cup (p'', mnf); \langle p, (\theta_2, mnf), q_1\rangle | \Pi_1|\!) = \ll p, \langle mnf, \theta_2\rangle, l_1 + [p''!mnf] \gg \cdot \Pi'$$

  Note that the step used to evaluate the expression cannot be (Let1) because in that case the resulting expression will not be MNF (after sending a message the result is a value, not a sequence of let expressions finished with a `receive` expression).
- (Receive): If this rule is used then e_1 must be MNF. There are two cases, e_1 is a `receive` expression or it is a compound MNF where the `receive` appears inside a chain of `let` expressions. In the firs case we have the following derivation:

$$\text{(Receive)}\ \frac{\text{(Receive)}\ \dfrac{}{\theta_1, \texttt{receive}\ \overline{b_n}\ \texttt{end} \xrightarrow{rec(\kappa, \overline{b_n})} \theta_1, \kappa}}{\Gamma_1; \langle p, (\theta_1, \texttt{receive}\ \overline{b_n}\ \texttt{end}), q_1\rangle | \Pi_1 \hookrightarrow \Gamma_1; \langle p, (\theta_1 \uplus \theta_i, mnf), q_1 \backslash\backslash v\rangle | \Pi_1}$$

  where $matchrec(\theta_1, \overline{b_n}, q_1) = (\theta_i, mnf, v)$. Then translation of the origin system is

$$(\!|\Gamma_1; \langle p, (\theta_1, \texttt{receive}\ \overline{b_n}\ \texttt{end}), q_1\rangle | \Pi_1|\!) = \ll p, \langle \texttt{receive}\ \overline{b_n}\ \texttt{end}, \theta_1\rangle, l_1 \gg \cdot \Pi'$$

  Assuming that the consumed message comes from the same process p then we can generate the following $\Rightarrow$-step:

$$\text{(CONSUME}_2\text{)}\ \frac{\text{(RCV}_3\text{)}\ \dfrac{\begin{array}{c} succeeds(\texttt{receive}\ \overline{b_n}\ \texttt{end}, l_1', j, \theta_1) = (mnf, \theta') \\ \langle mnf, \theta'\rangle \to (\langle mnf, \theta'\rangle, [\,], \emptyset) \end{array}}{\langle \texttt{receive}\ \overline{b_n}\ \texttt{end}, \theta_1\rangle \xrightarrow{l_1', j} (\langle mnf, \theta'\rangle, [p''!mnf], \emptyset)}}{\ll p, \langle \texttt{receive}\ \overline{b_n}\ \texttt{end}, \theta_1\rangle, l_1 \gg \Rightarrow \ll p, \langle mnf, \theta'\rangle, l_1'' \gg \cdot \Pi'}$$

  where $l_1' \equiv l_1|_p$, $1 \le j \le |l_1'|$, i is the position of the jth message of l_1' in l_1, and l_1'' is l_1 after removing the ith message. As the original derivation is MOP, then l_1' is starts with the same messages of q from p in the same order, so by Proposition A.5 we have that there is a position j such that $succeeds(\texttt{receive}\ \overline{b_n}\ \texttt{end}, l_1', j, \theta) = (mnf, \theta')$ such that $l_1'[j] = v$ and $\theta' = \theta_1 \uplus \theta_i$. Therefore,

$$(\!|\Gamma_1; \langle p, (\theta_1 \uplus \theta_i, mnf), q_1 \backslash\backslash v\rangle | \Pi_1|\!) = \ll p, \langle mnf, \theta_1 \uplus\rangle, l_1'' \gg \cdot \Pi'$$

  because the removed message v from the mailbox q_1 is the same as the one removed from the outbox l_1.
  If e_1 is a compound MNF then we can build a similar derivation but using a chain of rules (RCV_2) finished in a rule (RCV_3) instead of one (RCV_3) rule.
  The reasoning is the same if the message consumed comes from a different process p', in that case using (CONSUME_1) instead of (CONSUME_2).
- (Spawn): Assuming that the inner step uses rule (Spawn2) we have the following derivation:

$$\text{(Spawn)}\ \frac{\text{(Spawn}_2\text{)}\ \dfrac{}{\theta_1, \texttt{spawn}(fname, [\overline{v_n}]) \xrightarrow{spawn(\kappa, fname, \overline{v_n})} \theta_1, \kappa} \qquad p' \text{ fresh}}{\begin{array}{c} \Gamma_1; \langle p, (\theta_1, \texttt{spawn}(fname, [\overline{v_n}])), q_1\rangle | \Pi_1 \hookrightarrow \\ \Gamma_1; \langle p, (\theta_1, mnf), q_1\rangle \mid \langle p', (id, \texttt{apply}^*\ fname\ [\overline{v_n}])\rangle \mid \Pi_1 \end{array}}$$

  where $mnf = \kappa\{\kappa \mapsto p'\} = p'$. The translation of the first system is

$$(\!|\Gamma_1; \langle p, (\theta_1, \texttt{spawn}(\mathit{fname}, [\overline{v_n}])), q_1\rangle|\Pi_1|\!) = \ll p, \langle \texttt{spawn}(\mathit{fname}, [\overline{v_n}]), \theta_1\rangle, l_1 \gg \cdot \Pi'$$

so from the translated system we can perform the following step:

$$\text{(PROC)}\ \frac{\text{(SPAWN)}\ \dfrac{}{\langle \texttt{spawn}(\mathit{fname}, [\overline{v_n}]), \theta_1\rangle \to (\langle p', \theta_1\rangle, [\,], \ll p', \langle e, \theta'\rangle, [\,] \gg)}}{\ll p, \langle \texttt{spawn}(\mathit{fname}, [\overline{v_n}]), \theta_1\rangle, l_1 \gg \cdot \Pi' \Rightarrow \ll p, \langle p', \theta_1\rangle, l_1 \gg \cdot \ll p', \langle e, \theta'\rangle, [\,] \gg \cdot \Pi'}$$

where $\mathit{fname} = \texttt{fun}(\overline{X_n}) \to e$, $\theta' = \{\overline{X_n \mapsto v_n}\}$ and p' is fresh. As the `apply` is marked with *, it is translated to the body of the *fname* with the substitution for the parameter passing, so:

$$(\!|\Gamma_1; \langle p, (\theta_1, \mathit{mnf}), q_1\rangle \mid \langle p', (\mathit{id}, \texttt{apply}^*\ \mathit{fname}\ [\overline{v_n}])\rangle \mid \Pi_1|\!) = \ll p, \langle p', \theta_1\rangle, l_1 \gg \cdot \ll p', \langle e, \theta'\rangle, [\,] \gg \cdot \Pi'$$

If the inner step uses rule (Let1) the reasoning is similar but creating a (LET_2) derivation finishing in (SPAWN).

- (Sched): This step cannot occur because the (Sched) rule does not modify the expressions in the system, so $e_1 = \mathit{mnf}$ hence violating the premise.

Inductive Step: $m > 1$. In this case we have a MOP $\hookrightarrow$-derivation on the same process p as follows:

$$\begin{array}{ll} & \Gamma_1; \langle p, (\theta_1, e_1), q_1\rangle|\Pi_1 \\ \overset{p}{\hookrightarrow} & \Gamma_2; \langle p, (\theta_2, e_2), q_2\rangle|\Pi_2 \\ \overset{p}{\hookrightarrow} & \Gamma_3; \langle p, (\theta_3, e_3), q_3\rangle|\Pi_3 \\ & \vdots \\ \overset{p}{\hookrightarrow} & \Gamma_m; \langle p, (\theta_m, e_m), q_m\rangle|\Pi_m \\ \overset{p}{\hookrightarrow} & \Gamma_{m+1}; \langle p, (\theta_{m+1}, \mathit{mnf}), q_{m+1}\rangle|\Pi_{m+1} \end{array}$$

where $e_1 \neq \mathit{mnf}$ and the sequence of steps evaluates e_1 to the next MNF (i.e., only e_1 and *mnf* can be MNF expressions in $\{\overline{e_i}\}$). Here there are two cases: $e_2 = \mathit{mnf}$ or $e_2 \neq \mathit{mnf}$.

If $e_2 = \mathit{mnf}$ then the rest of steps are (Sched), the only rule that does not modify the expression. In this case, following the same reasoning as in the base case we have that $(\!|\Gamma_1; \langle p, (\theta_1, e_1), q_1\rangle|\Pi_1|\!) \Rightarrow (\!|\Gamma_2; \langle p, (\theta_2, \mathit{mnf}), q_2\rangle|\Pi_2|\!)$ or $(\!|\Gamma_1; \langle p, (\theta_1, e_1), q_1\rangle|\Pi_1|\!) = (\!|\Gamma_2; \langle p, (\theta_2, \mathit{mnf}), q_2\rangle|\Pi_2|\!)$. The rest of steps are (Sched), so $(\!|\Gamma_2; \langle p, (\theta_2, \mathit{mnf}), q_2\rangle|\Pi_2|\!) = (\!|\Gamma_{m+1}; \langle p, (\theta_{m+1}, \mathit{mnf}), q_{m+1}\rangle|\Pi_{m+1}|\!)$. Therefore we have one of these situations:

- $(\!|\Gamma_1; \langle p, (\theta_1, e_1), q_1\rangle|\Pi_1|\!) \Rightarrow (\!|\Gamma_{m+1}; \langle p, (\theta_{m+1}, \mathit{mnf}), q_{m+1}\rangle|\Pi_{m+1}|\!)$, or
- $(\!|\Gamma_1; \langle p, (\theta_1, e_1), q_1\rangle|\Pi_1|\!) = (\!|\Gamma_{m+1}; \langle p, (\theta_{m+1}, \mathit{mnf}), q_{m+1}\rangle|\Pi_{m+1}|\!)$.

On the other hand, if $e_2 \neq \mathit{mnf}$ we can split the derivation into two: a first step followed by a derivation:

$$\begin{array}{lll} \Gamma_1; \langle p, (\theta_1, e_1), q_1\rangle|\Pi_1 & \overset{p}{\hookrightarrow} & \Gamma_2; \langle p, (\theta_2, e_2), q_2\rangle|\Pi_2 \\ & \overset{p}{\hookrightarrow}^+ & \Gamma_{m+1}; \langle p, (\theta_{m+1}, \mathit{mnf}), q_{m+1}\rangle|\Pi_{m+1} \end{array}$$

By the Induction Hypothesis we have that:

$$(\!|\Gamma_2; \langle p, (\theta_2, e_2), q_2\rangle|\Pi_2|\!) \Rightarrow (\!|\Gamma_{m+1}; \langle p, (\theta_{m+1}, \mathit{mnf}), q_{m+1}\rangle|\Pi_{m+1}|\!)$$

(note that the translations of both systems cannot be equal because $e_2 \neq \mathit{mnf}$). Regarding the first step, it may be possible that

$$(\!|\Gamma_1; \langle p, (\theta_1, e_1), q_1\rangle|\Pi_1|\!) = (\!|\Gamma_2; \langle p, (\theta_2, e_2), q_2\rangle|\Pi_2|\!)$$

if the first step is (Sched), which does not change the expression, or if it is a (Seq) step using (Apply2) to evaluate a newly created process (then $e_1 = \texttt{apply}^*\ \mathit{fname}(\overline{v_n})$). In both cases using the Induction Hypothesis we have that:

$$(\!|\Gamma_1; \langle p, (\theta_1, e_1), q_1\rangle|\Pi_1|\!) \Rightarrow (\!|\Gamma_{m+1}; \langle p, (\theta_{m+1}, \mathit{mnf}), q_{m+1}\rangle|\Pi_{m+1}|\!)$$

On the other hand, if we have

$$(\!|\Gamma_1; \langle p, (\theta_1, e_1), q_1\rangle|\Pi_1|\!) \neq (\!|\Gamma_2; \langle p, (\theta_2, e_2), q_2\rangle|\Pi_2|\!)$$

then we proceed by case distinction on the rule used in the first $\overset{p}{\hookrightarrow}$ step, omitting (Sched) that has been already considered:

- (Seq): Then we have a derivation

$$\text{(Seq)}\ \frac{\theta_1, e_1 \xrightarrow{\tau} \theta_2, e_2}{\Gamma_1; \langle p, (\theta_1, e_1), q_1\rangle|\Pi_1 \hookrightarrow \Gamma_1; \langle p, (\theta_2, e_2), q_1\rangle|\Pi_1}$$

and from the Induction Hypothesis we have

$$\text{(PROC)}\ \frac{\langle e_2, \theta_2\rangle \to (\langle mnf, \theta_{m+1}\rangle, l, \Pi) \qquad e_2 \text{ is not MNF}}{(\!|\Gamma_2; \langle p, (\theta_2, e_2), q_2)\rangle|\Pi_2|\!) \Rightarrow (\!|\Gamma_{m+1}; \langle p, (\theta_{m+1}, mnf), q_{m+1})\rangle|\Pi_{m+1}|\!)}$$

(note that this step must be (PROC) because e_2 cannot be a MNF because $e_2 \neq mnf$ and a (Seq) step cannot be applied to a MNF). Then by Lemma A.11 we have that $\langle e_1, \theta_1\rangle \to (\langle mnf, \theta_{m+1}\rangle, l, \Pi)$, so we can raise that derivation to a step

$$(\!|\Gamma_1; \langle p, (\theta_1, e_1), q_1)\rangle|\Pi_1|\!) \Rightarrow (\!|\Gamma_{m+1}; \langle p, (\theta_{m+1}, mnf), q_{m+1})\rangle|\Pi_{m+1}|\!)$$

Again, e_1 is not MNF.

- (Send): This step cannot use (Send3) directly, because in that case $e_2 = v$ which conflicts $e_2 \neq mnf$, so it will be inside a `let` expression. We will consider only one level of `let` expressions:

$$\text{(Send)}\ \frac{\text{(Let1)}\ \dfrac{\text{(Send3)}\ \dfrac{}{\theta_1, p''\,!\,v \xrightarrow{send(p'',v)} \theta_1, v}}{\theta_1, \texttt{let}\ X = p''\,!\,v\ \texttt{in}\ e' \xrightarrow{send(p'',v)} \theta_1, \texttt{let}\ X = v\ \texttt{in}\ e'}}{\begin{array}{l}\Gamma_1; \langle p, (\theta_1, \texttt{let}\ X = p''\,!\,v\ \texttt{in}\ e'), q_1\rangle|\Pi_1 \\ \hookrightarrow \Gamma_1 \cup (p'', v); \langle p, (\theta_1, \texttt{let}\ X = v\ \texttt{in}\ e'), q_1\rangle|\Pi_1\end{array}}$$

By the Induction Hypothesis we have a derivation

$$\begin{array}{l}(\!|\Gamma_1 \cup (p'', v); \langle p, (\theta_1, \texttt{let}\ X = v\ \texttt{in}\ e'), q_1\rangle|\Pi_1|\!) \\ \Rightarrow (\!|\Gamma_{m+1}; \langle p, (\theta_{m+1}, mnf), q_{m+1}\rangle|\Pi_{m+1}|\!)\end{array}$$

Because of the shape of the expression being evaluated, this step will use the rule (LET_1) since the expression linked to the variable is evaluated to a value:

$$\text{(PROC)}\ \frac{\text{(LET}_1\text{)}\ \dfrac{\langle v, \theta_1\rangle \to (\langle v, \theta_1\rangle, [\,], \emptyset) \qquad \langle e', \theta'\rangle \to (\langle mnf, \theta_{m+1}\rangle, l', \Pi')}{\langle \texttt{let}\ X = v\ \texttt{in}\ e', \theta_1\rangle \to (\langle \theta_{m+1}, mnf\rangle, l', \Pi')}}{\begin{array}{l}(\!|\Gamma_1 \cup (p'', v); \langle p, (\theta_1, \texttt{let}\ X = v\ \texttt{in}\ e'), q_1\rangle|\Pi_1|\!) \\ \Rightarrow (\!|\Gamma_{m+1}; \langle p, (\theta_{m+1}, mnf), q_{m+1}\rangle|\Pi_{m+1}|\!)\end{array}}$$

where $\theta' = \theta_1 \uplus \{X \mapsto v\}$. This derivation can be extended with an extra level of (BANG) to process the message submission:

$$\text{(PROC)}\ \frac{\text{(LET}_1\text{)}\ \dfrac{\text{(BANG)}\ \dfrac{\langle p'', \theta_1\rangle \to (\langle p'', \theta_1\rangle, [\,], \emptyset) \quad \langle v, \theta_1\rangle \to (\langle v, \theta_1\rangle, [\,], \emptyset)}{\langle p''!v, \theta_1\rangle \to (\langle v, \theta_1\rangle, [p''!v], \emptyset)} \qquad \langle e', \theta'\rangle \to (\langle mnf, \theta_{m+1}\rangle, l', \Pi')}{\langle \texttt{let}\ X = p''!v\ \texttt{in}\ e', \theta_1\rangle \to (\langle \theta_{m+1}, mnf\rangle, [p''!v] + l', \Pi')}}{\begin{array}{l}(\!|\Gamma_1; \langle p, (\theta_1, \texttt{let}\ X = p''!v\ \texttt{in}\ e'), q_1\rangle|\Pi_1|\!) \\ \Rightarrow (\!|\Gamma_{m+1}; \langle p, (\theta_{m+1}, mnf), q_{m+1}\rangle|\Pi_{m+1}|\!)\end{array}}$$

Clearly, the message added in the global mailbox of the $\hookrightarrow$-step is the one generated in the process outbox, so by Definition 2.2 it will be in Γ_{m+1}. If there are more levels of `let` expression the reasoning is similar.

- (Receive): The proof is similar to the previous case. From the step in the Induction Hypothesis we know there is a (PROC) derivation of the expression of the selected branch in the `receive` that starts in a state without the consumed message. We can raise this derivation to a (CONSUME_1), if the message was originated in a different process p', or (CONSUME_2), if the consumed message comes from the same process p. In both cases the selected branch is the same by Proposition A.5, and the consumed message is removed from the outbox.
- (Spawn): This case is similar to the (Send) case. This step cannot use (Spawn2) directly because $e_2 \neq mnf$, so it will be inside a `let` expression. Then the step will apply (Spawn) with (Let1) and (Spawn2) to reduce the expression $\texttt{let}\ X = \texttt{spawn}\ fname(\overline{v_n})\ \texttt{in}\ e'$ to $\texttt{let}\ X = p'\ \texttt{in}\ e'$, while creating a new process p' containing $(id, \texttt{apply}^*\ fname(\overline{v_n}))$. From the Induction Hypothesis we have a $\Rightarrow$-step evaluating $\texttt{let}\ X = p'\ \texttt{in}\ e'$ using rules (PROC) with (LET_1) because p' is MNF. This derivation can be extended applying the rule (SPAWN) to evaluate the expression $\texttt{let}\ X = \texttt{spawn}\ fname(\overline{v_n})\ \texttt{in}\ e'$, which creates a process $\ll p', \langle e, \{\overline{X_n \mapsto v_n}\}\rangle, [\,] \gg$ where $fname = \texttt{fun}(\overline{X_n}) \to e$. Recall that by Definition 2.2 this process is the translation of $\langle p', (id, \texttt{apply}^*\ fname(\overline{v_n})), [\,]\rangle$. As in the (Send) case, if there are more levels of `let` expression the reasoning is similar. □

Lemma A.13. *If there is a MOP trace of m grouped steps ($m > 0$)*

$$\Gamma_0; \Pi_0 \overset{p^1}{\hookrightarrow}{}^{+} \Gamma_1; \Pi_1 \overset{p^2}{\hookrightarrow}{}^{+} \ldots \overset{p^m}{\hookrightarrow}{}^{+} \Gamma_m; \Pi_m$$

where $\Gamma_m; \Pi_m$ is MNF and each group of steps contains steps on the same process p^i such that (a) they do not change the expression, or (b) they evaluate the expression to the next MNF; then $(\!|\Gamma_0; \Pi_0|\!) = (\!|\Gamma_m; \Pi_m|\!)$ or $(\!|\Gamma_0; \Pi_0|\!) \Rightarrow^{+} (\!|\Gamma_m; \Pi_m|\!)$.

Proof. By induction on the number of grouped steps $\Gamma_{i-1}; \Pi_{i-1} \overset{p^i}{\hookrightarrow}{}^{+} \Gamma_i; \Pi_i$.

Base Case: $m = 1$. If the steps do not change the expression then $(\!|\Gamma_0; \Pi_0|\!) = (\!|\Gamma_m; \Pi_m|\!)$ by Lemma A.10; otherwise $(\!|\Gamma_0; \Pi_0|\!) = (\!|\Gamma_m; \Pi_m|\!)$ or $(\!|\Gamma_0; \Pi_0|\!) \Rightarrow^{+} (\!|\Gamma_m; \Pi_m|\!)$ by Lemma A.12.
Inductive Step: $m > 1$. By the Induction Hypothesis we have that $(\!|\Gamma_1; \Pi_1|\!) = (\!|\Gamma_m; \Pi_m|\!)$ or $(\!|\Gamma_1; \Pi_1|\!) \Rightarrow^{+} (\!|\Gamma_m; \Pi_m|\!)$. If the first group of steps $\Gamma_0; \Pi_0 \overset{p^0}{\hookrightarrow}{}^{+} \Gamma_1; \Pi_1$ does not modify the expression then by Lemma A.10 we have that $(\!|\Gamma_0; \Pi_0|\!) = (\!|\Gamma_1; \Pi_1|\!)$. Otherwise by Lemma A.12 $(\!|\Gamma_0; \Pi_0|\!) = (\!|\Gamma_1; \Pi_1|\!)$ or $(\!|\Gamma_0; \Pi_0|\!) \Rightarrow (\!|\Gamma_1; \Pi_1|\!)$. Then there are 4 possibilities:

- $(\!|\Gamma_0; \Pi_0|\!) = (\!|\Gamma_1; \Pi_1|\!)$ and $(\!|\Gamma_1; \Pi_1|\!) = (\!|\Gamma_m; \Pi_m|\!)$. Trivially $(\!|\Gamma_0; \Pi_0|\!) = (\!|\Gamma_m; \Pi_m|\!)$
- $(\!|\Gamma_0; \Pi_0|\!) = (\!|\Gamma_1; \Pi_1|\!)$ and $(\!|\Gamma_1; \Pi_1|\!) \Rightarrow^{+} (\!|\Gamma_m; \Pi_m|\!)$. Trivially $(\!|\Gamma_0; \Pi_0|\!) \Rightarrow^{+} (\!|\Gamma_m; \Pi_m|\!)$
- $(\!|\Gamma_0; \Pi_0|\!) \Rightarrow (\!|\Gamma_1; \Pi_1|\!)$ and $(\!|\Gamma_1; \Pi_1|\!) = (\!|\Gamma_m; \Pi_m|\!)$. Then we can build a derivation

$$(\text{Tr})\ \frac{(\!|\Gamma_0; \Pi_0|\!) \Rightarrow (\!|\Gamma_m; \Pi_m|\!)}{(\!|\Gamma_0; \Pi_0|\!) \Rightarrow^{+} (\!|\Gamma_m; \Pi_m|\!)}$$

- $(\!|\Gamma_0; \Pi_0|\!) \Rightarrow (\!|\Gamma_1; \Pi_1|\!)$ and $(\!|\Gamma_1; \Pi_1|\!) \Rightarrow^{+} (\!|\Gamma_m; \Pi_m|\!)$. From the transitive step we have that

$$(\text{Tr})\ \frac{(\!|\Gamma_1; \Pi_1|\!) \Rightarrow \Pi'_1 \quad \Pi'_1 \Rightarrow \Pi'_2 \quad \ldots \quad \Pi'_{n-1} \Rightarrow (\!|\Gamma_m; \Pi_m|\!) \quad n \geq 1}{(\!|\Gamma_1; \Pi_1|\!) \Rightarrow^{+} (\!|\Gamma_m; \Pi_m|\!)}$$

Then we can build an extended derivation with an extra first step:

$$(\text{Tr})\ \frac{(\!|\Gamma_0; \Pi_0|\!) \Rightarrow (\!|\Gamma_1; \Pi_1|\!) \quad (\!|\Gamma_1; \Pi_1|\!) \Rightarrow \Pi'_1 \quad \Pi'_1 \Rightarrow \Pi'_2 \ldots \quad \Pi'_{n-1} \Rightarrow (\!|\Gamma_m; \Pi_m|\!) \quad n+1 \geq 1}{(\!|\Gamma_0; \Pi_0|\!) \Rightarrow^{+} (\!|\Gamma_m; \Pi_m|\!)}$$

□

Theorem 2.2 *(Completeness of $\Rightarrow^{+}$ w.r.t. $\hookrightarrow$). If $\Gamma; \Pi \hookrightarrow^{+}_{mop} \Gamma'; \Pi'$ and $\Gamma'; \Pi'$ is a MNF system then $(\!|\Gamma; \Pi|\!) \Rightarrow^{+} (\!|\Gamma'; \Pi'|\!)$ or $(\!|\Gamma; \Pi|\!) = (\!|\Gamma'; \Pi'|\!)$.*

Proof. By Proposition A.4 there is a MOP trace $\Gamma; \Pi \hookrightarrow^{+} \Gamma'; \Pi'$, that by Lemma A.8 can be converted to a *grouped MOP trace* $\Gamma_1; \Pi_1 \overset{p^1}{\hookrightarrow}{}^{+} \Gamma_2; \Pi_2 \overset{p^2}{\hookrightarrow}{}^{+} \ldots \overset{p^m}{\hookrightarrow}{}^{+} \Gamma_{m+1}; \Pi_{m+1}$ where each group of steps $\Gamma_i; \Pi_i \overset{p^i}{\hookrightarrow}{}^{+} \Gamma_{i+1}; \Pi_{i+1}$ contains steps on the same process p^i such that (a) they do not change the expression, or (b) they evaluate the expression to the next MNF. Then by Lemma A.13 we have that $(\!|\Gamma_0; \Pi_0|\!) = (\!|\Gamma_m; \Pi_m|\!)$ or $(\!|\Gamma_0; \Pi_0|\!) \Rightarrow^{+} (\!|\Gamma_m; \Pi_m|\!)$. □

References

[1] E.Y. Shapiro, Algorithmic Program Debugging, ACM Distinguished Dissertation, MIT Press, 1983.
[2] R. Caballero, A. Riesco, J. Silva, A survey of algorithmic debugging, ACM Comput. Surv. 50 (4) (2017) 60:1–60:35, https://doi.org/10.1145/3106740.
[3] D. Insa, J. Silva, Algorithmic debugging generalized, J. Log. Algebraic Methods Program. 97 (2018) 85–104, https://doi.org/10.1016/j.jlamp.2018.02.003.
[4] R. Caballero, E. Martin-Martin, A. Riesco, S. Tamarit, EDD: a declarative debugger for sequential Erlang programs, in: E. Ábrahám, K. Havelund (Eds.), 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014, in: LNCS, vol. 8413, Springer, 2014, pp. 581–586.
[5] R. Caballero, E. Martin-Martin, A. Riesco, S. Tamarit, A zoom-declarative debugger for sequential Erlang programs, Sci. Comput. Program. 110 (2015) 104–118, https://doi.org/10.1016/j.scico.2015.06.011.
[6] R. Caballero, E. Martin-Martin, A. Riesco, S. Tamarit, Declarative debugging of concurrent Erlang programs, J. Log. Algebraic Methods Program. 101 (2018) 22–41, https://doi.org/10.1016/j.jlamp.2018.07.005.
[7] I. Lanese, N. Nishida, A. Palacios, G. Vidal, A theory of reversibility for Erlang, J. Log. Algebraic Methods Program. 100 (2018) 71–97, https://doi.org/10.1016/j.jlamp.2018.06.004.
[8] R. Carlsson, An introduction to Core Erlang, in: Proceedings of the Erlang Workshop 2001, in connection with PLI 2001, 2001, pp. 5–18.
[9] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, R. Virding, Core Erlang 1.0.3 language specification, available at http://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf, November 2004.
[10] C. Flanagan, A. Sabry, B.F. Duba, M. Felleisen, The essence of compiling with continuations, in: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI '93, ACM, New York, NY, USA, 1993, pp. 237–247.
[11] L.-Å. Fredlund, A Framework for Reasoning About Erlang Code, Ph.D. thesis, The Royal Institute of Technology, Sweden, August 2001.

[12] F. Huch, Verification of Erlang programs using abstract interpretation and model checking, in: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, ICFP 1999, ACM, New York, NY, USA, 1999, pp. 261–272.
[13] A. Farrugia, A. Francalanza, Towards a Formalisation of Erlang Failure and Failure Detection (extended abstract), Tech. rep., University of Malta, wICT, 2012.
[14] G. Vidal, Towards Erlang verification by term rewriting, in: G. Gupta, R. Peña (Eds.), Proc. 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2013), Revised Selected Papers, in: LNCS, vol. 8901, Springer International Publishing, 2014, pp. 109–126.
[15] J. Armstrong, Making Reliable Distributed Systems in the Presence of Software Errors, Ph.D. thesis, The Royal Institute of Technology, Sweden, December 2003.
[16] M. Lang, Frequently asked questions about Erlang - 10.8 Is the order of message reception guaranteed?, http://erlang.org/faq/academic.html, 2009, accessed on February 18, 2019.
[17] H. Svensson, L.-A. Fredlund, Programming distributed erlang applications: pitfalls and recipes, in: Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop, ERLANG '07, ACM, New York, NY, USA, 2007, pp. 37–42.
[18] M. Fidelman, Question re. message delivery, http://erlang.org/pipermail/erlang-questions/2017-September/093563.html, Sep. 2017, accessed on February 18, 2019.
[19] H. Svensson, L.-Å. Fredlund, A more accurate semantics for distributed Erlang, in: Proceedings of the 2007 SIGPLAN Workshop on Erlang Workshop, Erlang 2007, ACM, New York, NY, USA, 2007, pp. 43–54.
[20] H. Svensson, L.-A. Fredlund, C. Benac Earle, A unified semantics for future erlang, in: Proceedings of the 9th ACM SIGPLAN Workshop on Erlang, Erlang '10, ACM, New York, NY, USA, 2010, pp. 23–32.
[21] I. Lanese, N. Nishida, A. Palacios, G. Vidal, CauDEr: a causal-consistent reversible debugger for Erlang, in: J.P. Gallagher, M. Sulzmann (Eds.), Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9–11, 2018, Proceedings, in: LNCS, vol. 10818, Springer, 2018, pp. 247–263.
[22] N. Nishida, A. Palacios, G. Vidal, A reversible semantics for Erlang, in: M.V. Hermenegildo, P. López-García (Eds.), Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6–8, 2016, in: LNCS, vol. 10184, Springer, 2016, pp. 259–274, Revised Selected Papers.
[23] H. Nilsson, How to look busy while being as lazy as ever: the implementation of a lazy functional debugger, J. Funct. Program. 11 (6) (2001) 629–671, https://doi.org/10.1017/S095679680100418X.
[24] R. Caballero, A. Riesco, A. Verdejo, N. Martí-Oliet, Simplifying questions in Maude declarative debugger by transforming proof trees, in: G. Vidal (Ed.), Proceedings of the 21st International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2011, in: LNCS, vol. 7225, Springer, 2012, pp. 73–89.
[25] D. Insa, J. Silva, A. Riesco, Speeding up algorithmic debugging using balanced execution trees, in: M. Veanes, L. Vigan (Eds.), Proceedings of the 7th International Conference on Tests and Proofs, TAP 2013, in: LNCS, vol. 7942, Springer, 2013, pp. 133–151.
[26] M. Papadakis, K. Sagonas, A PropEr integration of types and function specifications with property-based testing, in: Proceedings of the 2011 ACM SIGPLAN Erlang Workshop, ACM Press, 2011, pp. 39–50.
[27] K. Claessen, J. Hughes, QuickCheck: a lightweight tool for random testing of Haskell programs, in: ACM SIGPLAN Notices, ACM Press, 2000, pp. 268–279.
[28] A. Gotovos, M. Christakis, K. Sagonas, Test-driven development of concurrent programs using Concuerror, in: Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, Erlang '11, ACM, New York, NY, USA, 2011, pp. 51–61.
[29] M. Christakis, A. Gotovos, K. Sagonas, Systematic testing for detecting concurrency errors in Erlang programs, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 154–163.
[30] D. Insa, S. Pérez, J. Silva, S. Tamarit, Behaviour preservation across code versions in Erlang, Sci. Program. (2018), https://doi.org/10.1155/2018/9251762.