A Semantics of Core Erlang with Handling of Signals

Aurélie Kong Win Chang, Jerome Feret, Gregor Gössler

A Semantics of Core Erlang with Handling of Signals*

Aurélie Kong Win Chang
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble, France

Jérôme Feret
Département d'Informatique de l'ENS, ENS, CNRS, PSL University, Inria Paris, France

Gregor Gössler
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble, France

Abstract

We introduce a small step semantics for a subset of Core Erlang modeling its monitoring and signal systems. The goal of our semantics is to enable the construction of causal explanations for property violations, which will be the object of future work. As a first axis of reflection, we chose to study the impact of the order of messages on a faulty behavior. We present our semantics and discuss some of our design choices. This work is a part of a broader project on causal debugging of concurrent programs in Erlang.

1 Introduction

Debugging concurrent systems is a difficult task, for which causal explanations of observed faults are a precious help. More precisely, our work is motivated by the goal of explaining errors made by choices of the scheduler that lead to incorrect interleavings, also known as concurrency bugs. In this paper we formalize a semantics of Core Erlang that preserves the pertinent information for such a causal analysis.

Most importantly, we want to investigate the impact of the order of messages on a faulty behavior. We thus need a semantics modeling precisely how message handling, signals, and monitoring affect the outcome. None of the existing semantics presented all of these characteristics at the same time. However, when possible, we adapt some ideas from them, and use observations made on the implementation.

Our small step semantics focuses on the order of messages. It models signal handling and come monitoring behaviors, but presents simplifications: its spawn never fails, it cannot represent continuous time — hence we do not support timers —, and nodes, try-catch expressions, or the ability to dynamically update code are not modeled.

This work is a part of a broader project called DCore, whose objective is to develop a semantically well-founded form of concurrent debugging, which we call causal debugging, that aims to alleviate the deficiencies of current debugging techniques for large concurrent software systems.

DCore encompasses two main work directions. First, the design of a reversible execution engine that allows programmers to backtrack and replay a concurrent or distributed program execution, in a way that is both precise and efficient (only the exact threads involved by a return to a target anterior or posterior program state are impacted) [10, 11].

The second axis of DCore is the development of a causal analysis engine that allows programmers to analyze concurrent executions by asking questions of the form "what caused the violation of this program property?", and that allows for the precise and efficient investigation of past and

*This work has been funded by the ANR grant ANR-18-CE25-0007 DCore.

potential program executions. Our causal analysis will be based on the semantics presented in this paper.

After a short overview of existing semantics, we expose the result of our work and the choices we made. We conclude with a discussion about the differences of our semantics.

2 Related Work

Our semantics is by no means the first one for Core Erlang. [6] is the most recent official specification of Core Erlang we found, but whilst useful it is informal and partly obsolete. [14] proposed a small step semantics of a subset of Core Erlang, making some choices such as the order of evaluation of function arguments based on its implementation rather than the official documentation. [4] formalized a subset of Core Erlang with a big step semantics, following [14] in some aspects, and validated it with Coq. [5] introduced the concept of medium step semantics, which considers as a step not only assignments, but also some events of interactions between processes. However, all these semantics lack the signal handling and monitoring part. The only semantics we found that finely modeled signals [15] treats them as side effects of operations that are instantaneously treated, which implies a strong hypothesis on their order of reception.

3 Design Choices

Erlang offers numerous functionalities and shortcuts in order to help its users, but can ultimately be translated, via an official tool, to a language with less syntactic sugar: Core Erlang. We thus chose, for our future analysis, to use a subset of Core Erlang. Both languages are still evolving, hence we tried to base our choices on concrete elements. We are using the initial, though somewhat outdated, specification of Core Erlang [6], and the official documentation available on the Erlang website [1] as far as possible. When a choice we made directly refers to this documentation, we will provide the clickable reference in the form [ERM, page, section] (Erlang Reference Manual [2]) or [ERTS, page, section] (Erlang Run-Time System Application reference manual [3]).

When these sources lack vital information, we search in the official implementation. If there is a conflict between both, as it is the case in the evaluation order of a function call arguments, we follow the former. For now, we put the concept of node aside. In our model, an Erlang system consists of a set of processes which interact through signal exchanges.

4 Execution Model

Our first goal is to formalize the different behaviors emerging from the non-deterministic order of the messages treatment. Thus, our model represents how processes send and handle signals, how they evaluate instructions and how they manage some parts of the monitoring aspect of Erlang.

Each process has a unique identifier *pid*. The state of a process is defined by the state of its execution task, its outbox, and the set of its links with others processes. They evolve asynchronously, but are synchonized when the processes interact.

4.1 Execution Task

A task e executes instructions. It is represented as a tuple $S_e = (\mathcal{I}_e, \theta, \tau, \mathtt{m}, st, \mathtt{r})$ where $\mathcal{I}_e$ is the next expression being evaluated, θ maps free variables to their values (as bound by lets or pattern matching), while τ does the same for variables that are bound to recursive functions by letrec. In this model, modules are named sets of functions used during the initialization of τ. We will not focus on the rules around letrec, which can be found in appendix. We also store information dedicated to generate signals: $\mathtt{m}$ is the name of the current module, st stores the execution stack and $\mathtt{r}$ stores the reason of the end of the process if known. Notice that the only role of st is to store the information needed to build error messages.

4.2 Sent Signals

We followed [5] in their choice of an outbox per process, and no inbox. It is a good way to keep the order guarantees given by the language, allowing to represent the local knowledge of a process when it sends signals without making presuppositions about when the signal is actually received [ERM, processes, delivery-of-signals].

There are two kinds of signals.

Messages. Messages content can be any constant in Core Erlang. We tag them as `msg`, and store the identifier of the process they are sent to. They are read and consumed only when their recipient evaluates a `receive` expression.

Other signals. Other signals are automatically processed: as stated in [ERM, processes, signals], "there is nothing a process must do to handle the reception of signals, or can do to prevent it". We tag them with `not_msg` and the identifier of their recipient. Their content depends on the nature of the signal, associated with specific flags ranging over `link`, `unlink`, `unlinkAck`, `monitored_by`, `demonitor`, and `down`. Links between processes are created and removed thanks to signals whose content is tagged with the flags `link`, `unlink`, and `unlinkAck`. More precisely, a signal with content $(\texttt{link}, pid_{source}, pid_{target})$ is generated by a process pid_{source} to generate a link with the process pid_{target}. A signal with content $(\texttt{unlink}, pid_{source}, pid_{target})$ is generated to remove this link, while a signal with content $(\texttt{unlinkAck}, pid_{source}, pid_{target})$ is used if the link did not already exist.

Core Erlang has another kind of links with their own signals, namely, monitor links. They are created and removed thanks to signals whose content is tagged with the flags `link` and `demonitor`. These signals also contain information about the pid of the process that has created the monitor link. Furthermore, each monitor link has its own identifier. Link identifiers take the form of terms built with the constructor `#Ref` on a quadruple of integers typically standing for the identifier of the node, the identifier of the source process, the identifier of the target process, and a unique integer. This way, there may be two monitor links between the same pair of processes. The set of monitor link identifiers is denoted as Ref_L. A signal with the content $(\texttt{link}, pid_{source}, lid)$ is used by the source process pid_{source} to initiate a monitor link with id lid, whereas the signals of with the content $(SigDemonitor, pid_{source}, lid)$ are used to removed them.

When a process terminates, it sends signals describing the reason of its termination to the processes with which it shares links. The content of these signals is tagged with the flag `down`. A signal with content $(\texttt{down}, fl, pid_{ended}, lid, r)$ where fl either the flag $monitored\_by$ or $link$ according to the kind of the link to be removed, pid_{ended} is the pid of the process that has terminated its execution, lid is either the pid of the target process or the monitor link id according to the kind of links, and r is the reason of the process termination.

The set of signals that are not messages is denoted by $SigOther$. In our model, the process outbox stores its sent signals in their order of sending. The set of its possible states is $(\{\texttt{msg}, \texttt{not\_msg}\} \times (Id \uplus \mathbb{N}) \times \{Const \uplus SigOther\})^*$.

4.3 Links

One of the main caracteristics of Erlang is the ability to implement easily a whole monitoring system between processes. This is done by linking them in order to suitably react when one of them dies. The language proposes two types of linking: *link*, associated with a tag `link`, and *monitor*, associated with tags `monitored_by` and `monitoring`.

When two processes are bound with a *link*, if one of them dies, so does the other one. Only one *link* can exist for a pair of processes. We model that a process of identifer pid_1 has a *link* with the process of identifier pid_2 with the tuple $(\texttt{link}, pid_2, pid_1)$. On the other hand, *monitor* links are asymmetric: one process is monitoring the other one. When the monitored process dies, it sends to the monitoring one a signal that is transformed into a message. This message can then be handled with a `receive` expression. A pair of processes can have multiple *monitor* links, hence we identify them through a unique reference in the set Ref_L. When a process of identifier pid_2

$$
\begin{array}{rcl}
\mathit{Module} & ::= & \underline{\texttt{module}}\ \mathit{atom}\ [\mathit{Fname}_{i_1}, \ldots, \mathit{Fname}_{i_k}] \\
 & & \underline{\texttt{attributes}}\ [\mathit{atom}_1 = \mathit{Cst}_1, \ldots, \mathit{atom}_m = \mathit{Cst}_m] \\
 & & \mathit{Fdef}_1 \ldots \mathit{Fdef}_n \\
\mathit{Fdef} & ::= & \mathit{Fname} = \mathit{Fun} \\
\mathit{Fname} & ::= & \mathit{atom}/\mathit{integer} \\
\mathit{Cst}(c) & ::= & \mathit{Lit} \mid [\ \mathit{Cst}_1 \mid \mathit{Cst}_2\ ] \\
 & \mid & \{\ \mathit{Cst}_1\ , \ldots\ , \mathit{Cst}_n\ \} \mid \mathit{Fun} \\
\mathit{Val}(v) & ::= & \mathit{Cst} \mid < \mathit{Cst}_1, \ldots, \mathit{Cst}_k > \mid \mathit{EoP} \mid \mathit{ended} \\
\mathit{Lit} & ::= & \mathit{integer}|\mathit{float}|\mathit{atom}|\mathit{char}|\mathit{string}|\ [\ ] \\
\mathit{Fun} & ::= & \underline{\texttt{fun}}\ (\ \mathit{var}_1, \ldots, \mathit{var}_n\ )\ ->\ \mathit{Exprs} \\
\mathit{Exprs} & ::= & \mathit{Expr} \mid <\ \mathit{Expr}_1, \ldots, \mathit{Expr}_n\ > \\
\mathit{Expr} & ::= & \mathit{var} \mid \mathit{Fname} \mid \mathit{Lit} \mid \mathit{Fun} \\
 & \mid & [\ \mathit{Expr}_1, \ldots, \mathit{Expr}_n\ ] \mid \{\ \mathit{Expr}_1, \ldots, \mathit{Expr}_n\ \} \\
 & \mid & \underline{\texttt{let}}\ \mathit{Vars}\ =\ \mathit{Exprs}_1\ \underline{\texttt{in}}\ \mathit{Exprs}_2 \\
 & \mid & \underline{\texttt{do}}\ \mathit{Exprs}_1\ \mathit{Exprs}_2 \\
 & \mid & \underline{\texttt{letrec}}\ \mathit{Fdef}_1\ \ldots\ \mathit{Fdef}_n\ \underline{\texttt{in}}\ \mathit{Exprs} \\
 & \mid & \underline{\texttt{apply}}\ \mathit{Exprs}_0\ (\mathit{Exprs}_1,\ \ldots,\ \mathit{Exprs}_n\ ) \\
 & \mid & \underline{\texttt{call}}\ \mathit{Exprs}_1\ :\ \mathit{Exprs}_2\ (\ \mathit{Exprs}_3\ , \ldots,\ \mathit{Exprs}_{n+2}\ ) \\
 & \mid & \underline{\texttt{primop}}\ \mathit{atom}\ (\ \mathit{Exprs}_1,\ \ldots,\ \mathit{Exprs}_n\ ) \\
 & \mid & \underline{\texttt{case}}\ \mathit{Exprs}\ \underline{\texttt{of}}\ \mathit{Cls}_1\ \ldots\ \mathit{Cls}_n\ \underline{\texttt{end}} \\
 & \mid & \underline{\texttt{receive}}\ \mathit{Cls}_1\ \ldots\ \mathit{Cls}_n\ \underline{\texttt{after}}\ \mathit{Exprs}_1\ ->\ \mathit{Exprs}_2 \\
\mathit{Vars}(x) & ::= & \mathit{var} \mid <\ \mathit{var}_1, \ldots, \mathit{var}_n\ > \\
\mathit{Cls}(cl) & ::= & \mathit{Pats}\ \underline{\texttt{when}}\ \mathit{Exprs}_1\ ->\ \mathit{Exprs}_2 \\
\mathit{Pats} & ::= & \mathit{Pat} \mid <\ \mathit{Pat}_1, \ldots, \mathit{Pat}_n\ > \\
\mathit{Pat}(p) & ::= & \mathit{var} \mid \mathit{Lit} \mid \_ \mid [\ \mathit{Pat}_1 \mid \mathit{Pat}_2\ ] \mid \mathit{var}\ =\ \mathit{Pat} \\
 & \mid & \{\ \mathit{Pat}_1, \ldots, \mathit{Pat}_n\ \} \mid \mathit{Fun}\ \underline{\texttt{end}}
\end{array}
$$

Figure 1: Our subset of Core Erlang syntax (based on [14], expanded with [6]). When a category is followed by a string in parentheses, this string denotes an element of this category in the rest of the paper.

is monitored by a process of identifier pid_1 through a *monitor* link of identifier lid, it stores the tuple $(\texttt{monitored\_by}, pid_1, lid)$, and the monitoring process stores $(\texttt{monitoring}, pid_2, lid)$.

The state of the of links with other processes is a set $S_L \in (\mathcal{F}_L \times Id \times \{Ref_L, Id\})^*$.

The state of a process π_1, of identifier pid_1, is thus described as the tuple $S_p = (pid_1, S_e, S_{sigsent}, S_L)$. The state of a system is the set of the states of the processes constituting it.

5 Syntax

We are using a subset of Core Erlang described in Figure 1. It has one major difference with the Core Erlang code obtained from Erlang code with the official compiler: the `receive` instruction no longer exists since the OTP 23 update and is replaced with the code presented in appendix. Replacing this code with the old receive during parsing allows us to keep `receive` as a shorter and equivalent expression.

We added two elements to *Val*: *EoP* signals that the process is terminating, and *ended* signals a terminated one.

$$\textsc{Case\_ok}\ \frac{\mathrm{match\_cls}(p,[cl_1,\ldots,cl_n],\theta,\tau)=(\theta',e_i) \qquad \theta'\neq no\_match \qquad \Pi\cup\{(pid,(e_i,\theta_i,\tau,\mathtt{m},st\cdot(\underline{\mathtt{case}}\ p\ \underline{\mathtt{of}}\ cl_1\ \ldots cl_n\ \underline{\mathtt{end}},\theta,\mathtt{m}),\mathtt{r}),sig_{sent},L)\}\xrightarrow{aux}\Pi\cup\{(pid,(e',\theta',\tau',\mathtt{m}',st,\mathtt{r}'),sig_{sent}{}',L')}{\Pi\cup\{(pid,(\underline{\mathtt{case}}\ p\ \underline{\mathtt{of}}\ cl_1\ \ldots cl_n\ \underline{\mathtt{end}},\theta,\tau,\mathtt{m},st,\mathtt{r}),sig_{sent},L)\}\xrightarrow{aux}\Pi\cup\{(pid,(e',\theta,\tau,\mathtt{m}',st,\mathtt{r}'),sig_{sent}{}',L')\}}$$

$$\textsc{Case\_ko}\ \frac{\mathrm{match\_cls}(p,[cl_1,\ldots,cl_n],\theta,\tau)=no\_match}{\Pi\cup\{(pid,(\underline{\mathtt{case}}\ p\ \underline{\mathtt{of}}\ cl_1\ \ldots cl_n\ \underline{\mathtt{end}},\theta,\tau,\mathtt{m},st,\mathtt{r}),sig_{sent},L)\}\xrightarrow{aux}\Pi\cup\{(pid,(EoP,\theta,\tau,\mathtt{m},st,\mathrm{end\_reason}(\mathtt{r},\{(case\_clause,p),st\})),sig_{sent},L)\}}$$

Figure 2: Successful and failing `case`.

$$\textsc{Val}\ \frac{J=\{i\mid 1\leq i\leq n\wedge fl_i\in\{\mathtt{monitored\_by},\mathtt{link}\}\} \qquad M=\{(\mathtt{not\_msg},pid_j,(\mathtt{down},fl_j,lid_j,\mathrm{end\_reason}(\mathtt{r},normal)))\mid j\in J\} \qquad sigs\in\mathrm{permutations}(M) \qquad v\neq ended}{\begin{array}{l}\Pi\cup\{(pid,(v,\theta,\tau,\mathtt{m},st,\mathtt{r}),sig_{sent},\{(fl_i,pid_i,lid_i)|1\leq i\leq n\wedge lid_i\in\{Id,Ref_L\}\})\}\\ \xrightarrow{term}\Pi\cup\{(pid,(ended,\theta,\tau,\mathtt{m},st,\mathtt{r}),sig_{sent}\cdot sigs,\{(fl_i,pid_i,lid_i)|1\leq i\leq n\wedge lid_i\in\{Id,Ref_L\}\})\}\end{array}}$$

Figure 3: End of a process. The function permutations returns the set of permutations of its argument converted into a list.

6 Semantics

6.1 Transition Relations

A transition relation $\xrightarrow{term}$ models small steps of execution. This relation is defined by means of inference rules. The evaluation of expressions is performed step-wise and bottom-up while respecting the policy about evaluation order (see 6.5). Yet, the evaluation of sub-expressions offers fewer capabilities than the evaluation of expressions at the top level. For instance, when a process terminates, an expression *EoP* is generated (see for instance Fig. 2), propagated at the top level of the expression (the rule is omitted), and only then signals are sent to warn the other processes (see Fig. 3). No signal can be emitted while the *EoP* expression has not reached the top level. This ensures that these signals are sent only once.

For this purpose, an auxiliary transition relation $\xrightarrow{aux}$ is used. Auxiliary transitions are not proper transitions, but they may occur in the inference proofs of the proper transitions. They describe only what can be done when evaluating sub-expressions. They can then be propagated by means of an evaluation context (see 6.5) and promoted as proper transitions $\xrightarrow{term}$ (see Fig. 4).

$$\textsc{expr\_term}\ \frac{\Pi\cup\{(pid,(e_1,\theta,\tau,\mathtt{m},st\cdot(e_1,\theta,\mathtt{m}),\mathtt{r}),s_{sent},L)\}\xrightarrow{aux}\Pi'\cup\{(pid,(e_1',\theta',\tau',\mathtt{m}',st,\mathtt{r}'),s_{sent}',L')\}}{\Pi\cup\{(pid,(e_1,\theta,\tau,\mathtt{m},st,\mathtt{r}),s_{sent},L)\}\xrightarrow{term}\Pi'\cup\{(pid,(e_1',\theta,\tau',\mathtt{m}',st,\mathtt{r}'),s_{sent}',L')\}}$$

Figure 4: Promotion from $\xrightarrow{aux}$ to $\xrightarrow{term}$.

6.2 Pattern Matching

Numerous functionalities are based on pattern matching, represented here as the function match. Upon success, pattern matching binds new variables by unification. This is the purpose of the disjoint union operator $\sqcup$ that takes two arguments θ_1 and θ_2 that are either a map or the symbol *no_match*. $\theta_1 \sqcup \theta_2$ is defined as

$$\{d\mapsto v\mid d\in\mathsf{dom}\ \theta_1\wedge\theta_1(d)=v\ \vee\ d\in\mathsf{dom}\ \theta_2\wedge\theta_2(d)=v\}$$

when $\theta_1 \neq no\_match \wedge \theta_2 \neq no\_match \wedge \forall d \in \mathsf{dom}(\theta_1)\cap\mathsf{dom}(\theta_2) : \theta_1(d) = \theta_2(d)$, and as *no_match* otherwise.

The function match takes four arguments, an expression e, a pattern p, a map θ and a functions table τ, and returns either a map or the symbol *no_match*. The function $\mathrm{match}(e,p,\theta,\tau)$ is defined as:

- if $v \in var$ and v is not bound, $\mathrm{match}(v,p,\theta,\tau) = \emptyset$

- if $p =' \_'$, $\text{match}(e,' \_', \theta, \tau) = \emptyset$
- if $p \in Lit \backslash ' \_'$ or $p = \{\}$ or $p = [\,]$, $\text{match}(p, p, \theta, \tau) = \emptyset$
- if $p \in Fun$ with arity n_p and name $Fname_p$:
  - if $v \in Fun$ with arity n_v, name $Fname_v$ and $\tau(Fname_p, n_p) = \tau(Fname_v, n_v)$, $\text{match}(v, p, \theta, \tau) = \emptyset$
  - if $v \in var$ and $\tau(Fname_p, n_p) = \theta(v), \text{match}(v, p, \theta, \tau) = \emptyset$
- if $p \in var$:
  - if p is bound, $\text{match}(e, p, \theta, \tau) = \text{match}(e, \theta(p), \theta, \tau)$
  - if p is not bound, $v \in var, \theta(v) \in Fun$ with arity n and name $Fname_v$, $\text{match}(v, p, \theta, \tau) = [(p \mapsto \tau(Fname_v, n))]$
  - if p is not bound and $v \in Val \backslash Fun, \text{match}(v, p, \theta, \tau) = [(p \mapsto \theta(v))]$
- if p and v are respectively of the form $\{p_0, \ldots, p_n\}$ and $\{v_0, \ldots, v_n\}$, or $[p_0, \ldots, p_n]$ and $[v_0, \ldots, v_n]$, or $< p_0, \ldots, p_n >$ and $< v_0, \ldots, v_n >$, with $v_i \in Val$ and $p_i \in Pat$ for $0 \leq i \leq n$,

$$\text{match}(v, p, \theta, \tau) = \text{match}(v_0, p_0, \theta, \tau) \sqcup \ldots \sqcup \text{match}(v_n, p_n, \theta, \tau)$$

- if p is of the form $[p_0, \ldots, p_{n-1}|p_n]$ and v is of the form $[v_0, \ldots, v_m]$ with $2 \leq n < m$ and $v_i \in Val$, for $0 \leq i \leq m$:

$$\begin{aligned}&\text{match}([v_0, \ldots, v_m], [p_0, \ldots, p_{n-1}|p_n], \theta, \tau) = \\ &\quad \text{match}([v_0, \ldots, v_{n-1}], [p_0, \ldots, p_{n-1}], \theta, \tau) \sqcup [(p_n \mapsto [v_n, \ldots, v_m])]\end{aligned}$$

- if p is of the form $v_{alias} = p_0$ with $v_{alias} \in var$ and $p_0 \in Pat$,

$$\text{match}(v, < v_{alias} = p_0 >, \theta, \tau) = \text{match}(v, p_0, \theta, \tau) \sqcup \text{match}(v_{alias}, v, \theta, \tau)$$

- for all other cases, $\text{match}(v, p, \theta, \tau) = no\_match$. Notice that when a function is defined, Erlang generates a name for it if needed. Furthermore, in the case where e is an unbound variable, match should return a special value treated as $\emptyset$ and generating a warning, but as we are not modeling this behavior, we simply return $\emptyset$.

6.3 Variables and End Reason

Variable assignments are done through unification thanks to the function match. When a correspondance can be made, the context θ is updated accordingly. Variables are local and they cannot be reassigned before exiting their scope [ERM, expressions, variables]. When the unification fails, the process ends while explaining the reason of the failure, unless the process was already terminated for a previous reason. This is the purpose of the function $\text{end\_reason}(r, r_{new})$, which output the reason r whenever it is not equal to $\perp$, or the reason r_{new} otherwise.

Such behavior occurs in local binding and switch cases. For example, see Fig. 2, where match_cls is a function looking for the first clause, of a list of clauses, which matches with a given pattern. It then returns its expression and the unification context. Otherwise, it returns no_match.

A variable x already associated with a value is simply read from $\theta(x)$. This rule is omitted for the sake of brevity.

6.4 Calling a Function

Calls to external and built-in functions are idealized. Their code is replaced by black-boxes which abstract their behavior. An example of the call of a built-in function is in appendix.

The instructions `apply` and `call` are used to call the functions that are defined in the program, respectively in the current module, or in arbitrary any module. For instance, see the rule *Call* in Fig. 5 where $bodyFunc(Mname, Fname, n)$ takes three arguments, a value $Mname$, a value $Fname$ and an integer n, and returns an expression, which is the body of the function named $Fname$, of arity n, coming from the module named $Mname$.

Rules for the external functions and the instruction `apply` works similarly, they are omitted for the sake of brevity

$$\textsc{Call}\ \frac{\begin{array}{c} Mname \in Val \backslash BIM \\ \Pi \cup \{(pid, (\text{bodyFunc}(Mname, Fname, n), \theta[a_1 \mapsto v1, \ldots, a_n \mapsto v_n], \tau, st \cdot (\text{bodyFunc}(Mname, Fname, n), \\ \theta[a_1 \mapsto v1, \ldots, a_n \mapsto v_n], Mname), \mathtt{r}), s_{sent}, L)\} \xrightarrow{aux} \Pi' \cup \{(pid, (e', \theta', \tau', st, \mathtt{r}'), s'_{sent}, L')\} \end{array}}{\Pi \cup \{(pid, (\underline{\mathtt{call}}\ Mname : Fname(v_1, \ldots, v_n), \theta, \tau, \mathtt{m}, st, \mathtt{r}), s_{sent}, L)\} \xrightarrow{aux} \Pi' \cup \{(pid, (e', \theta, \tau, \mathtt{m}', st, \mathtt{r}'), s'_{sent}, L')\}}$$

Figure 5: Calling a function.

6.5 Evaluation Order

The evaluation order of expressions is specified as follows.

For sequential composition `do` e_1 e_2, the expression e_1 is evaluated before the expression e_2. As for the evaluation of subexpressions, we follow the documentation rather than, as other semantics such as [14], the current implementation. Indeed, the documentation insisted on the fact that the evaluation order of the arguments of a function is not deterministic [ERM, expressions, expression-evaluation]; historically, it already changed once [12]; and further changes are possible.

The evaluation order is formalized by the means of evaluation contexts. An evaluation context is an expression with a hole '$\cdot$'. Intuitively, the hole contains a subexpression that can be evaluated, whereas the rest of the context remains as it is until the sub-expression is fully evaluated. Evaluation contexts are defined by the following grammar:

$$\begin{array}{rcl} c & = & [es_{1.}] \mid \{es_{1.}\} \mid < es_{1.} > \\ & \mid & \underline{\mathtt{call}}\ Module : Fname(es_{1.}) \mid \underline{\mathtt{apply}}\ Fname(es_{1.}) \\ & \mid & \underline{\mathtt{primop}}\ atom(es_{1.}) \mid \underline{\mathtt{let}}\ var = es_{1.}\ \underline{\mathtt{in}}\ e_2 \\ & \mid & \underline{\mathtt{case}}\ es_{1.}\ \underline{\mathtt{of}}\ cl_1 \ldots cl_n\ \underline{\mathtt{end}} \mid \underline{\mathtt{do}}\ \cdot\ e_2 \end{array}$$

where $es_{1.}$ denotes a list of expressions, excluding EoP, separated with commas, where exactly one expression is replaced with a hole '$\cdot$' at an arbitrary position. This models the cases where the evaluation order is non deterministic, as in the `apply` expression, or imposed, as in the `do` expression.

Applying an evaluation context $c[\cdot]$ to a subexpression e consists in replacing the unique occurrence of the hole '$\cdot$' in the evaluation context $c[\cdot]$ with the expression e. This is denoted as $c[e]$.

The rule for the evaluation of subexpressions is then as shown in Fig. 6; $Subexpr_{ok}$ if the evaluation goes without terminating the process, and $Subexpr_{ko}$ otherwise, which follows the same principle as $case_{ko}$. The rule for sequential composition is omitted for the sake of brevity.

$$\textsc{Subexpr\_ok}\ \frac{e_{eval} \in Expr \backslash Val \wedge e'_{eval} \neq EoP \quad \Pi \cup \{(pid, (e_{eval}, \theta, \tau, \mathtt{m}, st \cdot (e_{eval}, \theta, \mathtt{m}), \mathtt{r}), s_{sent}, L)\} \xrightarrow{aux} \Pi' \cup \{(pid, (e'_{eval}, \theta', \tau', \mathtt{m}', st, \mathtt{r}'), s'_{sent}, L')\}}{\Pi \cup \{(pid, (c[e_{eval}], \theta, \tau, \mathtt{m}, st), s_{sent}, L)\} \xrightarrow{aux} \Pi' \cup \{(pid, (c[e'_{eval}], \theta, \tau', \mathtt{m}, st, \mathtt{r}'), s'_{sent}, L')\}}$$

Figure 6: Subexpression evaluation.

6.6 Signals

As said in the documentation, "signals are received asynchronously and automatically" [ERM, processes, receiving-signals]. The only guarantee about their reception order is "if an entity sends multiple signals to the same destination entity, the order is preserved" [ERM, processes, signal-delivery], and the documentation emphasizes the importance of not using any other kind of order determinism based on the current implementation, as it might change in the future [ERTS, Communication, implementation].

For the sake of brevity, we will designate the first received message from a process j, given an outbox sig_{sent}, as first_sig(sig_{sent}, j).

6.7 Sending and Receiving Messages

In our model, processes send messages asynchronously, and never fail in doing so, as described in rule SEND in Fig. 7.

Our `receive` instruction is simplified in that, if no received message match with its clauses, it waits instead of starting a timer. As for the signals, the first received message from a process j is written first_msg(sig_{sent}, j). A `receive` instruction when the message comes from another process is then evaluated as in the rule rcv_ij_{ok} Fig. 7.

Here, the first message matching with a clause is consumed and the corresponding expression is evaluated in the updated context.

$$\textsc{Send}\ \frac{pid_1 \in Pid \qquad v \in Val}{\Pi \cup \{(pid, (\underline{\texttt{call}}\ \texttt{'erlang':'!'}(pid_1, v), \theta, \tau, \mathtt{m}, st, \mathtt{r}), sig_{sent}, L)\} \xrightarrow{aux} \Pi \cup \{(pid, (v, \theta, \tau, erlang, st, \mathtt{r}), sig_{sent} \cdot (\mathtt{msg}, pid_1, v), L)\}}$$

$$\textsc{rcv\_ij\_ok}\ \frac{\begin{array}{c}\text{first\_msg}((s_{i,1}, \ldots, s_{i,n_i}), j) = k \in \mathbb{N} \qquad i \neq j \qquad \text{match\_cls}(get\_sig((s_{i,1}, \ldots, s_{i,n_i}), k), [cl_1, \ldots, cl_n], \theta_j, \tau_j) = (\theta'_j, e'_j) \\ \Pi \cup \{(pid_j, (e'_j, \theta'_j, \tau_j, \mathtt{m}_j, st_j \cdot (\underline{\texttt{receive}}\ cl_1\ \ldots\ cl_n, \theta_j, \mathtt{m}_j), \mathtt{r}_j), sig_{sent,j}, L_j); (pid_i, (e_i, \theta_i, \tau_i, \mathtt{m}_i, st_i, \mathtt{r}_i), (s_{i,1}, \ldots, s_{i,k-1}, s_{i,k+1}, \ldots, s_{i,n_i}), L_i)\} \\ \xrightarrow{aux} \Pi' \cup \{(pid_j, (e''_j, \theta''_j, \tau_j, \mathtt{m}'_j, st_j, \mathtt{r}'_j), sig'_{sent,j}, L'_j)\}\end{array}}{\begin{array}{l}\Pi \cup \{(pid_j, (\underline{\texttt{receive}}\ cl_1\ \ldots\ cl_n, \theta_j, \tau_j, \mathtt{m}_j, st_j, \mathtt{r}_j), sig_{sent,j}, L_j); (pid_i, (e_i, \theta_i, \tau_i, \mathtt{m}_i, st_i, \mathtt{r}_i), (s_{i,1}, \ldots, s_{i,k}, \ldots, s_{i,n_i}), L_i)\} \\ \xrightarrow{aux} \Pi' \cup \{(pid_j, (e''_j, \theta_j, \tau_j, \mathtt{m}'_j, st_j, \mathtt{r}'_j), sig'_{sent,j}, L'_j)\}\end{array}}$$

Figure 7: Send and receive.

6.8 Spawning a Process

Our spawn function, described in Fig. 8, is greatly simplified: we only model the case where it always succeeds, and we omit its signal exchange protocol. However, most of the mechanisms needed to describe it properly being already present here, it might be the object of future work. It is based on fresh_pid, a function returning an identifier not already used by one of the processes of the system.

$$\textsc{Rspawn}\ \frac{\text{fresh\_pid}(\Pi) = pid_{son} \qquad Mname, v_1, \ldots, v_n \in Val}{\begin{array}{l}\Pi \cup \{(pid_{father}, (\underline{\texttt{call}}\ \texttt{'erlang':'spawn'}(Mname, Fname, [v_1, \ldots, v_n]), \theta, \tau, \mathtt{m}), sig_{sent}, L)\} \\ \xrightarrow{aux} \Pi \cup \{(pid_{father}, (pid_{son}, \theta, \tau, \mathtt{m}), sig_{sent}, L); (pid_{son}, (\underline{\texttt{call}}\ Mname : Fname(v_1, \ldots, v_n), [\,], \tau_{stat}, Mname), [\,], [\,])\}\end{array}}$$

Figure 8: Spawn.

6.9 Process Monitoring

The creation of a link between processes is asynchronous. The initiating process creates it locally and sends to the other process a signal. Once this signal has been processed, the other side of the link is created. In the case of a *link*, when such a *link* already exists, nothing is done and the evaluation continues. In the case of a *monitor link*, a new identifier is allocated as shown in Fig. 9. The sent signal is then handled.

$$\textsc{monitor}\ \frac{\text{fresh\_lid}(L_1) = lid}{\Pi \cup \{(pid_1, (\underline{\texttt{call}}\ erlang : monitor(\texttt{process}, pid_2), \theta, \tau, \texttt{m}, st, \texttt{r}), sig_{sent}, L)\} \xrightarrow{aux} \Pi \cup \{(pid_1, (lid, \theta, \tau, erlang, \texttt{m}, st, \texttt{r}), sig_{sent} \cdot (\texttt{not\_msg}, pid_2, (\texttt{monitored\_by}, pid_1, lid)), L_1 \cup \{(\texttt{monitoring}, pid_2, lid)\})\}}$$

$$\textsc{sig\_monitored\_by}\ \frac{\text{first\_sig}((s_{i,1}, \ldots, s_{i,k} \ldots, s_{i,n_i}), j) = k \qquad s_{i,k} = (\texttt{not\_msg}, pid_j, (\texttt{monitored\_by}, pid_i, lid)) \qquad i \neq j \qquad e_j \notin \{EoP, ended\}}{\Pi \cup \{(pid_j, (e_j, \theta_j, \tau_j, \texttt{m}_j, st_j, \texttt{r}_j), sig_{sent,j}, L_j); (pid_i, (e_i, \theta_i, \tau_i, \texttt{m}_i, st_i, \texttt{r}_i), (s_{i,1}, \ldots, s_{i,k}, \ldots, s_{i,n_i}), L_i)\} \xrightarrow{aux} \Pi \cup \{(pid_j, (e_j, \theta_j, \tau_j, \texttt{m}_j, st_j, \texttt{r}_j), sig_{sent,j}, L_j \cup \{(\texttt{monitored\_by}, pid_i, lid)\}); (pid_i, (e_i, \theta_i, \tau_i, \texttt{m}_i, st_i, \texttt{r}_i), (s_{i,1}, \ldots, s_{i,k-1}, s_{i,k+1}, \ldots, s_{i,n_i}), L_i)\}}$$

Figure 9: Birth of a monitor link.

In the case of a *link*, if the linked process does not exist and when it is "cheap" to know it, instead of executing the operation asynchronously, the calling process directly ends [ERTS, erlang, link-1], which changes the emitted signals. As the documentation suggest that the definition of "cheap" can evolve, we left the function is_cheap parametric (cf. Fig. 10).

$$\textsc{link\_cheap\_ko}\ \frac{pid_2 \in Id \qquad pid_2 \notin \text{pid}(\Pi)) \qquad \text{is\_cheap}(exists, [pid_2]) = true \qquad r_1' = \text{end\_reason}(\texttt{r}_1, \{noproc, st_1\})}{\Pi \cup \{(pid_1, (\underline{\texttt{call}}\ erlang : link(pid_2), \theta_1, \tau_1, \texttt{m}_1, st_1, \texttt{r}_1), sig_{sent,1}, L_1)\} \xrightarrow{aux} \Pi \cup \{(pid_1, (EoP, \theta_1, \tau_1, erlang, st_1, r_1'), sig_{sent,1}, L_1)\}}$$

Figure 10: Failing link, cheap.

When the verification is costly, the link is created, and an exit signal with *noproc* as a reason is sent. In the current implementation of Erlang, a hidden managing process sends the message, but this is an implementation choice. We thus model this behavior by storing the signal in the outbox of the calling process.

The destruction of such links is asynchronous with the guarantee that, once their destruction completed, they have no effect anymore [ERTS, erlang, unlink-1]. Ending the calling process fails when the argument is of wrong type: Ref for *demonitor*, Pid for *unlink*. These mechanisms are similar, we omit their definitions in this paper.

6.10 Exit Signals

In our model, exit signals are flagged with the type of link which caused their sending. If the link does not exist in the receiving process, the signal is simply ignored. Otherwise, when it is a *link*, the process ends too, as shown in the rule SIG_EXIT_LINK in Fig. 11 and when it is a *monitor link*, the signal is deleted and a message is created, as described in the rule SIG_EXIT_MONITORED in Fig. 11, in which `process` is a a flag indicating that the exit signal comes from a monitor link established between two processes.

$$\textsc{sig\_exit\_link}\ \frac{\begin{array}{c}\text{first\_sig}((s_{i,1},\ldots,s_{i,k},\ldots,s_{i,n_i}),j)=k \qquad i\neq j \qquad s_{i,k}=(\texttt{not\_msg},pid_j,(\texttt{down},\texttt{link},pid_i,raisonTerm))\\ (\texttt{link},pid_i,pid_j)\in L_j \qquad e_j\notin\{EoP,ended\}\end{array}}{\begin{array}{l}\Pi\cup\{(pid_j,(e_j,\theta_j,\tau_j,\mathtt{m}_j,st_j,\mathtt{r}_j),sig_{sent,j},L_j);(pid_i,(e_i,\theta_i,\tau_i,\mathtt{m}_i,st_i,\mathtt{r}_i),(s_{i,1},\ldots,s_{i,k},\ldots,s_{i,n_i}),L_i)\}\xrightarrow{aux}\\ \Pi\cup\{(pid_j,(EoP,\theta_j,\tau_j,\mathtt{m}_j,st_j,\text{end\_reason}(\mathtt{r}_j,raisonTerm)),sig_{sent,j},L_j);(pid_i,(e_i,\theta_i,\tau_i,\mathtt{m}_i,st_i,\mathtt{r}_i),(s_{i,1},\ldots,s_{i,k-1},s_{i,k+1},\ldots,s_{i,n_i}),L_i)\}\end{array}}$$

$$\textsc{sig\_exit\_monitored}\ \frac{\begin{array}{c}i\neq j \qquad \text{first\_sig}(sig_{sent,i},j)=k \qquad sig_{sent,i}=(s_{i,1},\ldots,s_{i,k},\ldots,s_{i,n_i})\\ s_{i,k}=(\texttt{not\_msg},pid_j,(\texttt{down},\texttt{monitored\_by},Ref,raisonTerm)) \qquad (\texttt{monitoring},pid_i,ref)\in L_j \qquad e_j\notin\{EoP,ended\}\\ sig'_{sent,i}=(s_{i,1},\ldots,s_{i,k-1},s_{i,k+1},\ldots,s_{i,n_i})\cdot(\texttt{msg},pid_j,(\{'DOWN',ref,\texttt{process},pid_i,raisonTerm\}))\end{array}}{\begin{array}{l}\Pi\cup\{(pid_j,(e_j,\theta_j,\tau_j,\mathtt{m}_j,st_j,\mathtt{r}_j),sig_{sent,j},L_j);(pid_i,(e_i,\theta_i,\tau_i,\mathtt{m}_i,st_i,\mathtt{r}_i),sig_{sent,i},L_i)\}\xrightarrow{aux}\\ \Pi\cup\{(pid_j,(e_j,\theta_j,\tau_j,\mathtt{m}_j,st_j,\mathtt{r}_j),sig_{sent,j},L_j);(pid_i,(e_i,\theta_i,\tau_i,\mathtt{m}_i,st_i,\mathtt{r}_i),sig'_{sent,i},L_i)\}\end{array}}$$

Figure 11: Exit signals.

7 Discussion

The semantics we proposed here contrasts with previous small steps semantics in two ways. First, it describes in a more precise way the Core Erlang monitoring and signal system. Second, it is based on the official documentation, rather than the current implementation. We tried to develop a semantics that is close enough to previous formalizations so as to make it easier to establish links between both, which we hope will come handy for the next part of our works.

We are currently working on an implementation of this semantics in Maude [8], which will be available, with an updated and further documented semantics, at [7]. Our next step will be to leverage abstract interpretation in order to generate causal explanations of an observed faulty behavior. Although not based on exactly the same language as our work, [9] and [13] seem to be a good starting point.

References

[1] Ericsson AB. Erlang online documentation, March 2023. 2

[2] Ericsson AB. Erlang reference manual, March 2023. 2

[3] Ericsson AB. Erlang run-time system application (erts) reference manual, March 2023. 2

[4] P. Bereczky, D. Horpácsi, and S. J. Thompson. Machine-checked natural semantics for core erlang: Exceptions and side effects. In *Proceedings of the 19th ACM SIGPLAN International Workshop on Erlang*, Erlang 2020, page 1–13, New York, NY, USA, 2020. ACM Press. 2

[5] R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. A core erlang semantics for declarative debugging. *Journal of Logical and Algebraic Methods in Programming*, 107:1–37, 2019. 2, 3

[6] R. Carlsson, T. Lindgren, B. Gustavsson, S.-O. Nystrom, R. Virding, E. Johansson, and M. Pettersson. Core Erlang 1.0.3 language specification. page 31, 2004. 2, 4

[7] A. Kong Win Chang, J. Feret, and G. Gössler. A semantics for core erlang with handlings of signals. https://gitlab.inria.fr/dcore-pub/erlangsemantics.git, 2023. 10, 13

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002. Rewriting Logic and its Applications. 10

[9] E. D'Osualdo, J. Kochems, and C. H. L. Ong. Automatic verification of erlang-style concurrency. In F. Logozzo and M. Fähndrich, editors, *Static Analysis - SAS 2013*, volume 7935 of *LNCS*, pages 454–476. Springer, 2013. 10

[10] G. Fabbretti, I. Lanese, and J.-B. Stefani. Causal-consistent debugging of distributed erlang programs. In S. Yamashita and T. Yokoyama, editors, *Reversible Computation - RC 2021*, volume 12805 of *LNCS*, pages 79–95. Springer, 2021. 1

[11] G. Fabbretti, I. Lanese, and J.-B. Stefani. Generation of a reversible semantics for erlang in maude. In A. Riesco and M. Zhang, editors, *Formal Methods and Software Engineering - ICFEM 2022*, volume 13478 of *LNCS*, pages 106–122. Springer, 2022. 1

[12] Erlang Forums. Need help understanding some erlang code #13. https://erlangforums.com/t/need-help-understanding-some-erlang-code/1623/13, July 2022. Accessed: 2023-07-18, Archived at http://web.archive.org/web/20230718093229/https://erlangforums.com/t/need-help-understanding-some-erlang-code/1623/13. 7

[13] F. Huch. Verification of erlang programs using abstract interpretation and model checking. *SIGPLAN Not.*, 34(9):261–272, sep 1999. 10

[14] M. Neuhäußer and T. Noll. Abstraction and Model Checking of Core Erlang Programs in Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):147–163, July 2007. 2, 4, 7

[15] H. Svensson and L.-A. Fredlund. A more accurate semantics for distributed erlang. In *Proceedings of the 2007 SIGPLAN workshop on Erlang Workshop - Erlang '07*, Freiburg, Germany, 2007. ACM Press. 2

Appendix

A Receive

Syntax of the equivalent of a `receive` instruction, as generated from Erlang code after the OTP23 update. Blue expressions match with the elements of a `receive` before the update.

```
letrec 'recv$^0'/0 =
 fun() ->
  let <var1, var2> = primop 'recv_peck_message'()
   in case var1 of
    <'true'> when 'true' ->
   case var2 of
    Cls1 ... Clsn
    (<Other> when 'true' ->
      do primop 'recv_next'()
      (apply 'recv_$^0'/0())
    )
   end
  (<'false'> when 'true' ->
   let var3 = primop 'recv_wait_timeout'(Expr1)
   in case var3 of
      <'true'> when 'true' -> Expr2
      (<'false'> when 'true' -> (apply 'recv$^0'/0 ())
   end)
  end
in (apply 'recv$^0'/0 ())
```

B Evaluation context

We give as follows the full definition of evaluation contexts :

$$
\begin{array}{rcl}
c & = & [es_{1.}] \mid \{es_{1.}\} \mid < es_{1.} > \\
 & \mid & \underline{\texttt{call}}\ Module : Fname(es_{1.}) \\
 & \mid & \underline{\texttt{apply}}\ Fname(es_{1.}) \\
 & \mid & \underline{\texttt{primop}}\ atom(es_{1.}) \\
 & \mid & \underline{\texttt{let}}\ var = es_{1.}\ \underline{\texttt{in}}\ e_2 \\
 & \mid & \underline{\texttt{case}}\ es_{1.}\ \underline{\texttt{of}}\ cl_1 \ldots cl_n\ \underline{\texttt{end}} \\
 & \mid & \underline{\texttt{do}}\ \cdot\ e_2
\end{array}
$$

$$\textsc{CallBIM}\ \frac{Mname \in BIM \qquad \text{evalResFunc}(\Pi, (Mname, Fname, n), (\theta, \tau, st, \mathtt{r}, s_{sent}, L)) = (\Pi', (e', \theta', \tau', st', \mathtt{r}', s'_{sent}, L'))}{\Pi \cup \{(pid, (\underline{\mathtt{call}}\ Mname : Fname(v_1, \ldots, v_n), \theta, \tau, \mathtt{m}, st, \mathtt{r}), s_{sent}, L)\} \xrightarrow{aux} \Pi' \cup \{(pid, (e', \theta, \tau, Mname, st, \mathtt{r}'), s'_{sent}, L')\}}$$

Figure 12: CallBIM.

$$\textsc{Subexpr\_ko}\ \frac{e_{eval} \in Expr \backslash Val \qquad \Pi \cup \{(pid, (e_{eval}, \theta, \tau, \mathtt{m}, st \cdot (e_{eval}, \theta, \mathtt{m}), \mathtt{r}), s_{sent}, L)\} \xrightarrow{aux} \Pi' \cup \{(pid, (EoP, \theta', \tau', \mathtt{m}', st, \mathtt{r}'), s'_{sent}, L')\}}{\Pi \cup \{(pid, (c[e_{eval}], \theta, \tau, \mathtt{m}, st, \mathtt{r}), s_{sent}, L)\} \xrightarrow{aux} \Pi' \cup \{(pid, (EoP, \theta, \tau', \mathtt{m}, st, \mathtt{r}'), s'_{sent}, L')\}}$$

Figure 13: Failing subexpression evaluation.

The notation es stands for a list of expressions, excluding EoP and separated with commas. The notation $es_1.$ stands also for a list of expressions, excluding EoP and separated with commas, except at exactly one position which contains the symbol $\cdot$ instead :

$$\begin{array}{rcl} es & = & e \mid e, es \\ es_1. & = & \cdot \mid \cdot, es \mid e, es_1. \end{array}$$

C Failing subexpression

If the evaluation of a subexpression fails, it stops the execution of its process and updates $\mathtt{r}$, as seen in Fig. 13 The evaluation $c[e]$ of an expression e under an evaluation context c is defined inductively over the syntax of the evaluation context c as follows:

$$\begin{array}{lcl}
[es_1.][e] & = & [es_1.[e]], \\
\{es_1.\}[e] & = & \{es_1.[e]\}, \\
< es_1. > [e] & = & < es_1.[e] >, \\
(\underline{\mathtt{call}}\ Module : Fname(es_1.))[e] & = & \underline{\mathtt{call}}\ Module : Fname(es_1.[e]), \\
(\underline{\mathtt{apply}}\ Fname(es_1.))[e] & = & \underline{\mathtt{apply}}\ Fname(es_1.[e]), \\
(\underline{\mathtt{primop}}\ atom(es_1.))[e] & = & \underline{\mathtt{primop}}\ atom(es_1.[e]). \\
(\underline{\mathtt{let}}\ var = es_1.\ \underline{\mathtt{in}}\ e_2)[e] & = & \underline{\mathtt{let}}\ var = es_1.[e]\ \underline{\mathtt{in}}\ e_2 \\
(\underline{\mathtt{case}}\ es_1.\ \underline{\mathtt{of}}\ cl_1 \ldots cl_n\ \underline{\mathtt{end}})[e] & = & \underline{\mathtt{case}}\ es_1.[e]\ \underline{\mathtt{of}}\ cl_1 \ldots cl_n\ \underline{\mathtt{end}} \\
(\underline{\mathtt{do}}\ \cdot\ e_2)[e] & = & \underline{\mathtt{do}}\ e\ e_2 \\
(\cdot)[e] & = & e \\
(\cdot, es)[e] & = & e, es \\
(e', es_1.)[e] & = & e', es_1.[e]
\end{array}$$

D Call of a built-in function

Rule CallBIM in Fig. 12, where evalResFunc returns result of such black box evaluation in the present context.

E Failing subexpression

If the evaluation of a subexpression fails, it stops the execution of its process and updates $\mathtt{r}$, as seen in Fig. 13

$$\textsc{letrec\_1}\ \frac{}{\Pi \cup (pid, (\underline{\mathtt{letrec}}\ Fname_1 \backslash n_1 = Fun_1\ \ldots\ Fname_m \backslash n_m = Fun_m\ \underline{\mathtt{in}}\ Expr, \theta, \tau, \mathtt{m}, st, \mathtt{r}), sig_{sent}, L) \xrightarrow{aux} \Pi \cup (pid, (\underline{\mathtt{letrec}}\ Fname_2 \backslash n_2 = Fun_2\ \ldots\ Fname_m \backslash n_m = Fun_m\ \underline{\mathtt{in}}\ Expr, \theta, \tau[(\mathtt{m}, Fname_1 \backslash n_1, \mathtt{m}) \to Fun_1], \mathtt{m}, st, \mathtt{r}), sig_{sent}, L)}$$

$$\textsc{letrec\_2}\ \frac{}{\Pi \cup (pid, (\underline{\mathtt{letrec}}\ Fname \backslash n = Fun\ \underline{\mathtt{in}}\ Expr, \theta, \tau, \mathtt{m}, st, \mathtt{r}), sig_{sent}, L) \xrightarrow{aux} \Pi \cup (pid, (Expr, \theta, \tau[(\mathtt{m}, Fname \backslash n) \to Fun], \mathtt{m}, st, \mathtt{r}), sig_{sent}, L)}$$

Figure 14: Letrec rules.

F Recursive call and other rules

As we focused on presenting the signals handling and monitoring aspects of our semantics, for the sake of briefty, we did not include our whole semantics. This explains why, in our model, some terms are never modified by the rules of the paper. It is for example the case of the functions table τ, only modified by the `letrec` expression (see Fig. 14). We are currently working on an implementation of this semantics in Maude, which we will make public, along with the complete semantics[7].