Causal-Consistent Debugging of Distributed Erlang Programs - Technical Report[⋆]

Giovanni Fabbretti[1][0000−0003−3002−0697], Ivan Lanese[2][0000−0003−2527−9995], and Jean-Bernard Stefani[1][0000−0003−1373−7602]

[1] Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
[2] Focus Team, Univ. of Bologna, INRIA, 40137 Bologna, Italy

Abstract. Debugging concurrent programs is an interesting application of reversibility. It has been renewed with the recent proposal by Giachino et al. to base the operations of a concurrent debugger on a causal-consistent reversible semantics, and subsequent work on CauDEr, a causal-consistent debugger for the Erlang programming language. This paper extends CauDEr and the related theory with the support for distributed programs. Our extension allows one to debug programs in which processes can run on different nodes, and new nodes can be created at runtime. From the theoretical point of view, the primitives for distributed programming give rise to more complex causal structures than those arising from the concurrent fragment of Erlang handled in CauDEr, yet we show that the main results proved for CauDEr still hold. From the practical point of view, we show how to use our extension of CauDEr to find a non trivial bug in a simple way.

Keywords: Debugging · Actor model · Distributed computation · Reversible computing

1 Introduction

Debugging concurrent programs is an interesting application of reversibility. A reversible debugger allows one to explore a program execution by going forward – letting the program execute normally –, or backward – rolling back the program execution by undoing the effect of previously executed instructions. Several works have explored this idea in the past, see, e.g., the survey in [6], and reversible debugging is used in mainstream tools as well [20]. It is only recently, however, that the idea of a causal-consistent debugger has been proposed by Giachino et al. in [9]. The key idea in [9] was to base the debugger primitives on a causal-consistent reversible semantics for the target programming language. Causal consistency, introduced by Danos and Krivine in their seminal work on reversible

⋆ The work has been partially supported by French ANR project DCore ANR-18-CE25-0007. The second author has also been partially supported by INdAM – GNCS 2020 project *Sistemi Reversibili Concorrenti: dai Modelli ai Linguaggi*.

CCS [5], allows one, in reversing a concurrent execution, to undo any event provided that its consequences, if any, are undone first. On top of a causal-consistent semantics one can define a rollback operator [14] to undo an arbitrary past action. It provides a minimality guarantee, useful to explore concurrent programs which are prone to state explosion, in that only events in the causal future of a target one are undone, and not events that are causally independent but which may have been interleaved in the execution.

The CauDEr debugger [16,10,17] builds on these ideas and provides a reversible debugger for a core subset of the Erlang programming language [3]. Erlang is interesting for it mixes functional programming with a concurrency model inspired by actors [1], and has been largely applied since its initial uses by Ericsson[3], to build distributed infrastructures.

This paper presents an extension of CauDEr to take into account distribution primitives which are not part of the core subset of Erlang handled by CauDEr. Specifically, we additionally consider the three Erlang primitives called start, to create a new node for executing Erlang processes, node, to retrieve the identifier of the current node, and nodes, which allows the current process to obtain a list of all the currently active nodes in an Erlang system. We also extend the spawn primitive handled by CauDEr to take as additional parameter the node on which to create a new Erlang process.

Adding support for these primitives is non trivial for they introduce causal dependencies in Erlang programs that are different than those originating from the functional and concurrent fragment considered in CauDEr, which covers, beyond sequential constructs, only message passing and process creation on the current node. Indeed, the set of nodes acts as a shared set variable that can be read, checked for membership, and extended with new elements. Interestingly, the causal dependencies induced by this shared set cannot be faithfully represented in the general model for reversing languages introduced in [13], which allows for resources that can only be produced and consumed.

The contributions of the current work are therefore as follows: (i) we extend the reversible semantics for the core subset of the Erlang language used by CauDEr with the above distribution primitives; (ii) we present a rollback semantics that underlies primitives in our extended CauDEr debugger; (iii) we have implemented an extension of the CauDEr debugger that handles Erlang programs written in our distributed fragment of the language; (iv) we illustrate on an example how our CauDEr extension can be used in capturing subtle bugs in distributed Erlang programs. Due to space constraints, we do not detail in this paper our extended CauDEr implementation, but the code is publicly available in the dedicated GitHub repository [8].

The rest of this paper is organized as follows. Section 2 briefly recalls the reversible semantics on which CauDER is based [18]. Section 3 presents the Erlang distributed language fragment we consider in our CauDEr extension, its reversible semantics and the corresponding rollback semantics. Section 4 briefly

[3] erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus.html

$$
\begin{array}{rcl}
\mathit{module} & ::= & \mathit{fun}_1 \ \ldots \ \mathit{fun}_n \\
\mathit{fun} & ::= & \mathit{fname} = \mathsf{fun}\ (X_1, \ldots, X_n) \rightarrow \mathit{expr} \\
\mathit{fname} & ::= & \mathit{Atom}/\mathit{Integer} \\
\mathit{lit} & ::= & \mathit{Atom} \mid \mathit{Integer} \mid \mathit{Float} \mid [\,] \\
\mathit{expr} & ::= & \mathit{Var} \mid \mathit{lit} \mid \mathit{fname} \mid [\mathit{expr}_1|\mathit{expr}_2] \mid \{\mathit{expr}_1, \ldots, \mathit{expr}_n\} \\
& \mid & \mathsf{call}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \mid \mathsf{apply}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \\
& \mid & \mathsf{case}\ \mathit{expr}\ \mathsf{of}\ \mathit{clause}_1; \ldots; \mathit{clause}_m\ \mathsf{end} \\
& \mid & \mathsf{let}\ \mathit{Var} = \mathit{expr}_1\ \mathsf{in}\ \mathit{expr}_2 \mid \mathsf{receive}\ \mathit{clause}_1; \ldots; \mathit{clause}_n\ \mathsf{end} \\
& \mid & \mathsf{spawn}(\mathit{expr}, [\mathit{expr}_1, \ldots, \mathit{expr}_n]) \mid \mathit{expr}_1\ !\ \mathit{expr}_2 \mid \mathsf{self}() \\
\mathit{clause} & ::= & \mathit{pat}\ \mathsf{when}\ \mathit{expr}_1 \rightarrow \mathit{expr}_2 \\
\mathit{pat} & ::= & \mathit{Var} \mid \mathit{lit} \mid [\mathit{pat}_1|\mathit{pat}_2] \mid \{\mathit{pat}_1, \ldots, \mathit{pat}_n\}
\end{array}
$$

Fig. 1. Language syntax

describes our extension to CauDEr, and presents an example that illustrates bug finding in distributed Erlang programs with our extended CauDEr. Section 5 discusses related work and concludes the paper with hints for future work. Further technical details are available in the appendix.

2 Background

We recall here the main aspects of the language in [18], as needed to understand our extension. We refer the interested reader to [18] for further details.

2.1 The language syntax

Fig. 1 shows the language syntax. The language depicted is a fragment of Core Erlang [2], an intermediate step in Erlang compilation. A module is a collection of function definitions, a function is a mapping between the function name and the function expression. An expression can be a variable, a literal, a function name, a list, a tuple, a call to a built-in function, a function application, a case expression, or a let binding. An expression can also be a spawn, a send, a receive, or a self, which are built-in functions. Finally, we distinguish expressions, patterns and variables. Here, patterns are built from variables, tuples, lists and literals, while values are built from literals, tuples and lists, i.e., they are ground patterns. When we have a case e of ... expression we first evaluate e to a value, say v, then we search for a clause that matches v. When one is found, if the guard when $expr$ is satisfied then the case construct evaluates to the clause expression, otherwise the search continues with the next clause. The let $X =\ expr_1$ in $expr_2$ expression binds inside $expr_2$ the fresh variable X to the value to which $expr_1$ reduces.

As for the concurrent features, since Erlang implements the actor model, there is no shared memory. An Erlang system is a pool of processes that interact by exchanging messages. Each process is uniquely identified by its pid and has its own queue of incoming messages. Function spawn $(expr, [expr_1, \ldots, expr_n])$

evaluates to a fresh process pid p. As a side-effect, it creates a new process with pid p. Process p will apply the function to which $expr$ evaluates to the arguments to which the expressions $expr_1, \ldots, expr_n$ evaluate. As in [18], we assume that the only way to introduce a new pid is through the evaluation of a spawn. Then, $expr_1 \, ! \, expr_2$ allows a process to send a message to another one. Expression $expr_1$ must evaluate to a pid (identifying the receiver process) and $expr_2$ evaluates to the content of the message, say v. The whole function evaluates to v and, as a side-effect, the message will eventually be stored in the receiver queue. The counterpart of message sending is $\mathsf{receive}\ clause_1, \ldots, clause_n\ \mathsf{end}$. This construct traverses the queue of messages searching for the first message v that matches one of the n clauses. If no message is found then the process suspends. Finally, self evaluates to the current process pid.

2.2 The language semantics

This subsection provides key elements to understand the CauDEr semantics. We start with the definition of process.

Definition 1 (Process). *A* process *is denoted by a tuple $\langle p, \theta, e, q \rangle$, where p is the process' pid, θ is an environment, i.e. a map from variables to their actual value, e is the current expression to evaluate, and q is the queue of messages received by the process.*

Two operations are allowed on queues: $v : q$ denotes the addition of a new message on top of the queue and $q \backslash\!\!\backslash v$ denotes the queue q after removing v (note that v may not be the first message).

A (running) system can be seen as a pool of running processes.

Definition 2 (System). *A* system *is denoted by the tuple $\Gamma; \Pi$. The* global mailbox *Γ is a multiset of pairs of the form ($target\_process\_pid$, $message$), where a message is stored after being sent and before being scheduled to its receiver. Π is the* pool of processes*, denoted by an expression of the form*

$$\langle p_1, \theta_1, e_1, q_1 \rangle \mid \ldots \mid \langle p_n, \theta_n, e_n, q_n \rangle$$

where "|" is an associative and commutative operator. $\Gamma \cup \{(p, v)\}$, where $\cup$ is multiset union, is the global mailbox obtained by adding the pair (p, v) to Γ. We write $p \in \Gamma; \Pi$ when Π contains a process with pid p.

We highlight a process p in a system by writing $\Gamma; \langle p, \theta, e, q \rangle \,|\, \Pi$. The presence of the global mailbox Γ, which is similar to the "ether" in [23], allows one to simulate all the possible interleavings of messages. Indeed, in this semantics the order of the messages exchanged between two processes belonging to the same runtime may not be respected, differently from what happens in current Erlang implementations. See [23] for a discussion on this design choice.

The semantics in [18] is defined in a modular way, similarly to the one presented in [4], i.e., there is a semantics for the expression level and one for the system level. This approach simplifies the design of the reversible semantics since only the system one needs to be updated. The expression semantics is defined as a labelled transition relation of the form:

$$\{Env, Expr\} \times Label \times \{Env, Expr\}$$

where *Env* represents the environment, i.e., a substitution, and *Expr* denotes the expression, while *Label* is an element of the following set:

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

The semantics , described in Appendix A.1 due to space constraints, is a classical call-by-value semantics for a first order language. Label τ denotes the evaluation of a (sequential) expression without side-effects, like the evaluation of a case expression or a let binding. The remaining labels denote a side-effect associated to the rule execution or the request of some needed information. The system semantics will use the label to execute the associated side-effect or to provide the necessary information. More precisely, in label $\mathsf{send}(v_1, v_2)$, v_1 and v_2 represent the pid of the sender and the value of a message. In label $\mathsf{rec}(\kappa, \overline{cl_n})$, $\overline{cl_n}$ denotes the n clauses of a receive expression. Inside label $\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])$, a/n represents the function name, while $[\overline{v_n}]$ is the (possibly empty) list of arguments of the function. Where used, κ acts as a future: the expression evaluates to κ, then the corresponding system rule replaces it with its actual value.

For space reasons, we do not show here the system rules, which are available in Appendix A.2. We will however show in the next section how sample rules are extended to support reversibility.

2.3 A reversible semantics

The reversible semantics is composed by two relations: a *forward* relation $\rightharpoonup$ and a *backward* relation $\leftharpoondown$. The forward reversible semantics is a natural extension of the system semantics by using a typical *Landauer embedding* [12]. The idea underlying Landauer's work is that any formalism or programming language can be made reversible by adding the *history* of the computation at each state. Hence, this semantics at each step saves in an external device, called history, the previous state of the computation so that later on such a state can be restored. The backward semantics allows us to undo a step while ensuring causal consistency [5,15], indeed before undoing an action we must ensure that all its consequences have been undone.

In the reversible semantics each message exchanged must be uniquely identified in order to allow one to undo the sending of the "right" message, hence we denote messages with the tuple $\{\lambda, v\}$, where λ is the unique identifier and v the message body. See [18] for a discussion on this design choice.

Due to the Landauer embedding the notion of process is extended as follows.

Definition 3 (Process). *A process is denoted by a tuple $\langle p, h, \theta, e, q\rangle$, where h is the* history *of the process. The other elements are as in Def. 1. The expression*

$$(\mathit{Spawn})\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh identifier}}{\begin{array}{l}\Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, \theta', e'\{\kappa \mapsto p'\}, q\rangle \\ \qquad\qquad\qquad\qquad \mid \langle p', [\,], id, \mathsf{apply}\ a/n(\overline{v_n}), [\,]\rangle \mid \Pi\end{array}}$$

$$(\overline{\mathit{Spawn}})\quad \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, \theta', e', q\rangle \mid \langle p', [\,], id, e'', [\,]\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi$$

Fig. 2. An example of a rule belonging to the forward semantics and its counterpart.

$\mathsf{op}(\ldots) : h$ *denotes the history* h *with a new history item added on top. The generic history item* $\mathsf{op}(\ldots)$ *can span over the following set.*

$$\{\tau(\theta, e), \mathsf{send}(\theta, e, \{\lambda, v\}), \mathsf{rec}(\theta, e, \{\lambda, v\}, q), \mathsf{spawn}(\theta, e, p), \mathsf{self}(\theta, e)\}$$

Here, each history item carries the information needed to restore the previous state of the computation. For rules that do not cause causal dependencies (i.e., τ and self) it is enough to save θ and e. For the other rules we must carry additional information to check that every consequence has been undone before restoring the previous state. We refer to [18] for further details.

Fig. 2 shows a sample rule from the forward semantics (additions w.r.t. the standard system rule are highlighted in red) and its counterpart from the backward semantics. In the premises of the rule *Spawn* we can see the expression-level semantics in action, transitioning from the configuration (θ, e) to (θ', e') and the corresponding label that the forward semantics uses to determine the associated side-effect. When rule *Spawn* is applied the system transits in a new state where process p' is added to the pool of processes and the history of process p is enriched with the corresponding history item. Finally, the forward semantics takes care of updating the value of the future κ by substituting it with the pid p' of the new process.

The reverse rule, $\overline{\mathit{Spawn}}$, can be applied only when all the consequences of the spawn, namely every action performed by the spawned process p', have been undone. Such constraint is enforced by requiring the history of the spawned process to be empty. Since the last history item of p is the spawn, and thanks to the assumption that every new pid, except for the first process, is introduced by evaluating a spawn, we are sure that there are no pending messages for p'. Then, if the history is empty, we can remove the process p' from Π and we can restore p to the previous state.

3 Distributed Reversible Semantics for Erlang

In this section we discuss how the syntax and the reversible semantics introduced in the previous section have been updated to tackle the three distribution primitives start, node and nodes. Lastly, we extend the rollback operator introduced in [18,19], which allows one to undo an arbitrary past action together with all and only its consequences, to support distribution.

3.1 Distributed System Semantics

The updated syntax is like the one in Fig. 1, with the only difference that now *expr* can also be start(e), node() and nodes(), and spawn takes an extra argument that represents the node where the new process must be spawned.

Let us now briefly discuss the semantics of the new primitives. First, in function start, e must evaluate to a node identifier (also called a nid), which is an atom of the form 'name@host'. Then, the function, as a side-effect, starts a new node, provided that no node with the same identifier exists in the network, and evaluates to the node identifier in case of success or to an error in case of failure. Node identifiers, contrarily to pids which are always generated fresh, can be hardcoded, as it usually happens in Erlang. Also, function node evaluates to the local node identifier. Finally, function nodes evaluates to the list (possibly empty) of nodes to which the executing node is connected. A formalization of the intuition above can be found in [7]. Here, we assume that each node has an atomic view of the network, therefore we do not consider network partitioning.

Notions of process and system are updated to cope with the extension above.

Definition 4 (Process). *A process is denoted by a tuple $\langle nid, p, \theta, e, q\rangle$, where nid is an atom of the form name@host, called a* node identifier (nid)*, pointing to the node on which the process is running. For the other elements of the tuple the reader can refer to Def. 1.*

The updated definitions of node and network follow.

Definition 5 (Node and network). *A* node *is a pool of processes, identified by a nid. A* network*, denoted by Ω, is a set of nids. Hence, nids in a network should all be distinct.*

Now, we can proceed to give the formal definition of a distributed system.

Definition 6 (Distributed system). *A* distributed system *is a tuple $\Gamma; \Pi; \Omega$. The global mailbox Γ and the pool of running processes Π are as before (but processes now include a nid). Instead, Ω represents the set of nodes connected to the network. We will use $\cup$ to denote set union.*

3.2 Causality

To understand the following development, one needs not only the operational semantics informally discussed above, but also a notion of causality. Indeed, backward rules can undo an action only if all its causal consequences have been undone, and forward rules should store enough information to both decide whether this is the case and, if so, to restore the previous state.

Thus, to guide the reader, we discuss below the possible causal links among the distribution primitives (including spawn). About the functional and concurrent primitives, the only dependencies are that a message receive is a consequence of the scheduling of the same message to the target process, (see rule *Sched* in Appendix A.3) which is a consequence of its send[4].

Intuitively, there is a dependency between two consecutive actions if either they cannot be executed in the opposite order (e.g., a message cannot be scheduled before having been sent), or by executing them in the opposite order the result would change (e.g., by swapping a successful start and a nodes the result of the nodes would change).

Beyond the fact that later actions in the same process are a consequence of earlier actions, we have the following dependencies:

1. every action of process p depends on the (successful) spawn of p;
2. a (successful) spawn on node nid depends on the start of nid;
3. a (successful) start of node nid depends on previous failed spawns on the same node, if any (if we swap the order, the spawn will succeed);
4. a failed start of node nid depends on its (successful) start;
5. a nodes reading a set Ω depends on the start of all nids in Ω, if any (as discussed above).

3.3 Distributed forward reversible semantics

Fig. 3 shows the forward semantics of distribution primitives, which are described below. The other rules are as in the original work [18] but for the introduction of Ω , and can be found in Appendix A.4.

The forward semantics in [18] has just one rule for spawn, since it can never fail. Here, instead, a spawn can fail if the node fed as first argument is not part of Ω. Nonetheless, following the approach of Erlang, we always return a fresh pid, independently on whether the spawn has failed or not. Also, the history item created in both cases is the same. Indeed, thanks to uniqueness of pids, one can ascertain whether the spawn of p' has been successful or not just by checking whether there is a process with pid p' in the system: if there is, the spawn succeeded, otherwise it failed. Hence, the unique difference between rules *SpawnS* and *SpawnF* is that a new process is created only in rule *SpawnS*.

Similarly, two rules describe the start function: rule *StartS* for a successful start, which updates Ω by adding the new nid nid', and rule *StartF* for a start which fails because a node with the same nid already exists. Here, contrarily to the spawn case, the two rules create different history items. Indeed, if two or more processes had a same history item start(θ, e, nid), then it would not be possible to decide which one performed the start first (and, hence, succeeded).

Lastly, the *Nodes* rule aves, together with θ and e, the current value of Ω. This is needed to check dependencies on the start executions, as discussed in Section 3.2. The *Node* rule, since node is a sequential operation, just saves the environment and the current expression.

$$(\mathit{SpawnS})\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, nid', a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid} \quad nid' \in \Omega}{\begin{array}{c}\Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle nid, p, \mathsf{spawn}(\theta, e, nid', p') : h, \theta', e'\{\kappa \mapsto p'\}, q\rangle \\ \mid \langle nid', p', [\,], id, \mathsf{apply}\ a/n(\overline{v_n}), [\,]\rangle \mid \Pi; \Omega\end{array}}$$

$$(\mathit{SpawnF})\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, nid', a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid} \quad nid' \notin \Omega}{\Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle nid, p, \mathsf{spawn}(\theta, e, nid', p') : h, \theta', e'\{\kappa \mapsto p'\}, q\rangle \mid \Pi; \Omega}$$

$$(\mathit{StartS})\quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, nid')} \theta', e' \quad nid' \notin \Omega}{\begin{array}{l}\Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \rightharpoonup \\ \qquad \Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\{\kappa \mapsto nid'\}, q\rangle \mid \Pi; \{nid'\} \cup \Omega\end{array}}$$

$$(\mathit{StartF})\quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, nid')} \theta', e' \quad nid' \in \Omega \quad err \text{ represents the error}}{\Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{fail}, nid') : h, \theta', e'\{\kappa \mapsto err\}, q\rangle \mid \Pi; \Omega}$$

$$(\mathit{Node})\quad \frac{\theta, e \xrightarrow{\mathsf{node}(\kappa)} \theta', e'}{\Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle nid, p, \mathsf{node}(\theta, e) : h, \theta', e'\{\kappa \mapsto nid\}, q\rangle \mid \Pi; \Omega}$$

$$(\mathit{Nodes})\quad \frac{\theta, e \xrightarrow{\mathsf{nodes}(\kappa)} \theta', e'}{\begin{array}{l}\Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \rightharpoonup \\ \qquad \Gamma; \langle nid, p, \mathsf{nodes}(\theta, e, \Omega) : h, \theta', e'\{\kappa \mapsto list(\Omega \setminus \{nid\})\}, q\rangle \mid \Pi; \Omega\end{array}}$$

Fig. 3. Distributed forward reversible semantics

3.4 Distributed backward reversible semantics

Fig. 4 depicts the backward semantics of the distribution primitives, the other rules are collected in Appendix A.4. The semantics is defined in terms of the relation $\leftharpoondown_{p,r,\Psi}$, where:

- p represents the pid of the process performing the backward transition
- r describes which action has been undone
- Ψ lists the requests satisfied by the backward transition (the supported requests are listed in Section 3.5)

These labels will come into play later on, while defining the rollback semantics. We may drop them when not relevant.

As already discussed, to undo an action, we need to ensure that its consequences, if any, have been undone before. When consequences in other processes may exist, side conditions are used to check that they have already been undone.

Rule $\overline{SpawnS}$ is analogous to rule $\overline{Spawn}$ in Fig. 2. Rule $\overline{SpawnF}$ undoes a failed spawn. As discussed in Section 3.2, we first need to undo, if any, a start of a node with the target nid, otherwise the spawn will now succeed. To this end, we check that $nid' \notin \Omega$.

[4] For technical reasons the formalization provides an approximation of this notion.

$$(\overline{SpawnS})\quad \begin{array}{c}\Gamma; \langle nid, p, \mathsf{spawn}(\theta, e, nid', p') : h, \theta', e', q\rangle \mid \langle nid', p', [\,], id, e'', [\,]\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p, \mathsf{spawn}(p'), \{s, sp_{p'}\}} \Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega\end{array}$$

$$(\overline{SpawnF})\quad \begin{array}{c}\Gamma; \langle nid, p, \mathsf{spawn}(\theta, e, nid', p') : h, \theta', e', q\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p, \mathsf{spawn}(p'), \{s, sp_{p'}\}} \Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \\ \text{if } nid' \notin \Omega\end{array}$$

$$(\overline{StartS})\quad \begin{array}{c}\Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e', q\rangle \mid \Pi; \Omega \cup \{nid'\} \\ \leftharpoondown_{p, \mathsf{start}(nid'), \{s, st_{nid'}\}} \Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \\ \text{if } spawns(nid', \Pi) = [\,] \land reads(nid', \Pi) = [\,] \land failed\_starts(nid', \Pi) = [\,]\end{array}$$

$$(\overline{StartF})\quad \begin{array}{c}\Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{fail}, nid') : h, \theta', e', q\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p, \mathsf{start}(nid'), \{s\}} \Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega\end{array}$$

$$(\overline{Node})\quad \Gamma; \langle nid, p, \mathsf{node}(\theta, e) : h, \theta', e', q\rangle \mid \Pi; \Omega \leftharpoondown_{p, \mathsf{node}, \{s\}} \Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega$$

$$(\overline{Nodes})\quad \begin{array}{c}\Gamma; \langle nid, p, \mathsf{nodes}(\theta, e, \Omega') : h, \theta', e', q\rangle \mid \Pi; \Omega \leftharpoondown_{p, \mathsf{nodes}, \{s\}} \Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \\ \text{if } \Omega = \Omega'\end{array}$$

Fig. 4. Extended backward reversible semantics

Then, we have rule $\overline{StartS}$ to undo the (successful) creation of node nid'. Before applying it we need to ensure three conditions: (i) that no process is running on node nid'; (ii) that no nodes has read nid'; and (iii) that no other start of a node with identifier nid' failed. The conditions, discussed in Section 3.2, are checked by ensuring that the lists of pids computed by auxiliary functions $spawns$, $reads$ and $failed\_starts$ are empty. Indeed, they compute the list of pids of processes in Π that have performed, respectively, a spawn on nid', a nodes returning a set containing nid', and a failed start of a node with identifier nid. Condition (i) needs to be checked since nids are hardcoded, hence any process could perform a spawn on nid'. The check would be redundant if nids would be created fresh by the start function.

Rule $\overline{StartF}$ instead requires no side condition: start fails only if the node already exists, but this condition remains true afterwards, since we do not have primitives to stop a node. Rule $\overline{Node}$ has no dependency either.

To execute rule $\overline{Nodes}$ we must ensure that the value of Ω' in the history item and of Ω in the system are the same, as discussed in Section 3.2.

We now report a fundamental result of the reversible semantics. As most of our results, it holds for *reachable* systems, that is systems that can be obtained using the rules of the semantics from a single process with empty history.

Lemma 1 (Loop Lemma). *For every pair of reachable systems, s_1 and s_2, we have $s_1 \rightharpoonup s_2$ iff $s_2 \leftharpoondown s_1$.*

Proof. The proof that a forward transition can be undone follows by rule inspection. The other direction relies on the restriction to reachable systems: consider the process undoing the action. Since the system is reachable, restoring the memory item would put us back in a state where the undone action can be performed again (if the system would not be reachable the memory item would be arbitrary, hence there would not be such a guarantee), as desired. Again, this can be proved by rule inspection. □

Note that, as exemplified above, this result would fail if we allow one to undo an action before its consequences.

3.5 Distributed rollback semantics

Since undoing steps one by one may be tedious and unproductive for the developer, CauDEr provides a rollback operator, that allows the developer to undo several steps in an automatic manner, while maintaining causal consistency. We extend it to cope with distribution. Our definition takes inspiration from the formalization style used in [19], but it improves it and applies it to a system with explicit local queues for messages. Dealing with explicit local queues is not trivial. Indeed, without local queues, the receive primitive takes messages directly from Γ. With local queues we use a rule called *Sched* to move a message from Γ to the local queue of the target process, and the receive takes the message from the local queue. A main point is that the *Sched* action does not create an item in the history of the process receiving the message, and as a result it is concurrent to all other actions of the same process but receive. A formalization of rule *Sched* and of its inverse is in Appendix A.3 (Fig. 11 for the forward rule and Fig. 12 for the backward one). When during a rollback both a $\overline{Sched}$ and another backward transition are enabled at the same time one has to choose which one to undo, and selecting the wrong one may violate the property that only consequences of the target action are undone.

We denote a system in rollback mode by $\lceil\!\lceil \mathcal{S} \rceil\!\rceil_{\{p,\psi\}}$, where the subscript means that we wish to undo the action ψ performed by process p and every action which depends on it. More generally, the subscript of $\lceil\!\lceil\,\rceil\!\rceil$, often depicted with Ψ or Ψ' (where Ψ can be empty while Ψ' cannot), can be seen as a stack (with : as cons operator) of undo requests that need to be satisfied. Once the stack is empty, the system has reached the state desired by the user. We consider requests $\{p, \psi\}$, asking process p to undo a specific action, namely:

- $\{p, s\}$: a single step back;
- $\{p, \lambda^{\Downarrow}\}$: the receive of the message uniquely identified by λ;
- $\{p, \lambda^{\Uparrow}\}$: the send of the message uniquely identified by λ;
- $\{p, \lambda^{sched}\}$: the scheduling of the message uniquely identified by λ;
- $\{p, st_{nid}\}$: the successful start of node nid';
- $\{p, sp_{p'}\}$: the spawn of process p'.

The rollback semantics is defined in Fig. 5 in terms of the relation $\rightsquigarrow$, selecting which backward rule to apply and when. There are two categories of rules: (i) U-rules that perform a step back using the backward semantics; (ii) rule *Request* that pushes a new request on top of Ψ whenever it is not possible to undo an action since its consequences need to be undone before.

$$(U-Satisfy)\ \frac{\mathcal{S} \leftharpoonup_{p,r,\Psi'} \mathcal{S}' \ \wedge\ \psi \in \Psi'}{\llbracket \mathcal{S} \rrbracket_{\{p,\psi\}:\Psi} \rightsquigarrow \llbracket \mathcal{S}' \rrbracket_{\Psi}} \quad (U-Sched)\ \frac{\mathcal{S} \leftharpoonup_{p,r,\{s,\lambda'^{sched}\}} \mathcal{S}' \ \wedge \lambda'^{sched} \neq \lambda^{sched}}{\llbracket \mathcal{S} \rrbracket_{\{p,\lambda^{sched}\}:\Psi} \rightsquigarrow \llbracket \mathcal{S}' \rrbracket_{\{p,\lambda^{sched}\}:\Psi}}$$

$$(U-Unique)\ \frac{\mathcal{S} \leftharpoonup_{p,r,\Psi'} \mathcal{S}' \ \wedge\ \psi \notin \Psi' \ \wedge\ \forall r'', \Psi''\ \mathcal{S} \leftharpoonup_{p,r'',\Psi''} \mathcal{S}'' \Rightarrow \mathcal{S}' = \mathcal{S}''}{\llbracket \mathcal{S} \rrbracket_{\{p,\psi\}:\Psi} \rightsquigarrow \llbracket \mathcal{S}' \rrbracket_{\{p,\psi\}:\Psi}}$$

$$(U-Act)\ \frac{\mathcal{S} \leftharpoonup_{p,r,\Psi'} \mathcal{S}' \ \wedge\ \psi \notin \Psi' \ \wedge\ \lambda^{sched} \notin \Psi' \ \wedge\ \psi \neq \lambda^{sched}\ \forall \lambda \in \mathbb{N}}{\llbracket \mathcal{S} \rrbracket_{\{p,\psi\}:\Psi} \rightsquigarrow \llbracket \mathcal{S}' \rrbracket_{\{p,\psi\}:\Psi}}$$

$$(Request)\ \frac{\mathcal{S} = \Gamma; \langle nid, p, h, \theta, e, q \rangle \mid \Pi; \Omega \ \wedge\ \mathcal{S} \not\leftharpoonup_{p,r,\Psi'} \ \wedge \{p', \psi'\} = dep(\langle nid, p, h, \theta, e, q \rangle, \mathcal{S})}{\llbracket \mathcal{S} \rrbracket_{\{p,\psi\}:\Psi} \rightsquigarrow \llbracket \mathcal{S}' \rrbracket_{\{p',\psi'\}:\{p,\psi\}:\Psi}}$$

Fig. 5. Rollback semantics

$$\begin{array}{lll}
\mathsf{dep}(<\_,\_,\mathsf{send}(\_,\_,p',\{\lambda,v\}):h,\_,\_,\_>,\_;\_;\_) & = \{p', \lambda^{sched}\} & \\
\mathsf{dep}(<\_,\_,\mathsf{nodes}(\_,\_,\Omega):h,\_,\_,\_>,\_;\Pi;\{nid\}\cup\Omega') & = \{parent(nid,\Pi), st_{nid}\} & \text{if } nid \notin \Omega \\
\mathsf{dep}(<\_,\_,\mathsf{spawn}(\_,\_,\_,p'):h,\_,\_,\_>,\_;\Pi;\_) & = \{p', s\} & \text{if } p' \in \Pi \\
\mathsf{dep}(<\_,\_,\mathsf{spawn}(\_,\_,nid',\_):h,\_,\_,\_>,\_;\Pi;\_) & = \{parent(nid',\Pi), st_{nid'}\} & \text{if } p' \notin \Pi \\
\mathsf{dep}(<\_,\_,\mathsf{start}(\_,\_,\mathsf{succ},nid'):h,\_,\_,\_>,\_;\Pi;\_) & = \{fst(reads(nid',\Pi)), s\} & \text{if } reads(nid',\Pi) \neq [\,] \\
\mathsf{dep}(<\_,\_,\mathsf{start}(\_,\_,\mathsf{succ},nid'):h,\_,\_,\_>,\_;\Pi;\_) & = \{fst(spawns(nid',\Pi)), s\} & \text{if } spawns(nid',\Pi) \neq [\,] \\
\mathsf{dep}(<\_,\_,\mathsf{start}(\_,\_,\mathsf{succ},nid'):h,\_,\_,\_>,\_;\Pi;\_) & = \{fst(failed\_start(nid',\Pi)), s\} &
\end{array}$$

Fig. 6. Dependencies operator

Let us analyse the U-rules. During rollback, more than one backward rule could be applicable to the same process. In our setting, the only possibility is that one of the rules is a $\overline{Sched}$ and the other one is not. It is important to select which rule to apply, to ensure that only consequences of the target action are undone.

First, if an enabled transition satisfies our target, then it is executed and the corresponding request is removed (rule $U-Satisfy$). Intuitively, since two applications of rule *Sched* to the same process are always causally dependent, if the target action is an application of *Sched*, an enabled *Sched* is for sure one of its consequences, hence it needs to be undone (rule $U-Sched$). Dually, if the target is not a *Sched* and a non *Sched* is enabled, we do it (rule $U-Act$). If a unique rule is applicable, then it is selected (rule $U-Unique$).

Rule *Request* considers the case where no backward transition in the target process is enabled. This depends on some consequence on another process of the action on top of the history. Such a consequence needs to be undone before, hence the rule finds out using operator dep in Fig. 6 both the dependency and the target process and adds on top of Ψ the corresponding request.

Let us discuss operator dep. In the first case, a send cannot be undone since the sent message is not in the global mailbox, hence a request has to be made to the receiver p' of undoing the *Sched* of the message λ.

In case of multiple dependencies, we add them one by one. This happens, e.g., in case nodes, where we need to undo the start of all the nodes which are in

$\{nid'\} \cup \Omega'$ but not in Ω. Adding all the dependencies at once would make the treatment more complex, since by solving one of them we may solve others as well, and thus we would need an additional check to avoid starting a computation to undo a dependency which is no more there. Adding the dependencies one by one solves the problem, hence operator `dep` nondeterministically selects one of them. Notice also that the order in which dependencies are solved is not relevant.

In some cases (e.g., `send`) we find a precise target event, in others we use just s, that is a single step. In the latter case, a backward step is performed (and its consequences are undone), then the condition is re-checked and another backward step is required, until the correct step is undone. We could have computed more precise targets, but this would have required additional technicalities.

Function $parent(nid', \Pi)$, used in the definition of `dep`, returns the pid of the process that started nid' while function $fst(\cdot)$ returns the first element of a list.

An execution of the rollback operator corresponds to a backward derivation, while the opposite is generally false.

Theorem 1 (Soundness of rollback). *If $\llbracket \mathcal{S} \rrbracket_{\Psi'} \rightsquigarrow^* \llbracket \mathcal{S}' \rrbracket_{\Psi}$ then $\mathcal{S} \leftharpoondown^* \mathcal{S}'$ where * denotes reflexive and transitive closure.*

Proof. The rollback semantics is either executing backward steps using the backward semantics or executing administrative steps (i.e., pushing new requests on top of Ψ), which do not alter the state of the system. The thesis follow. □

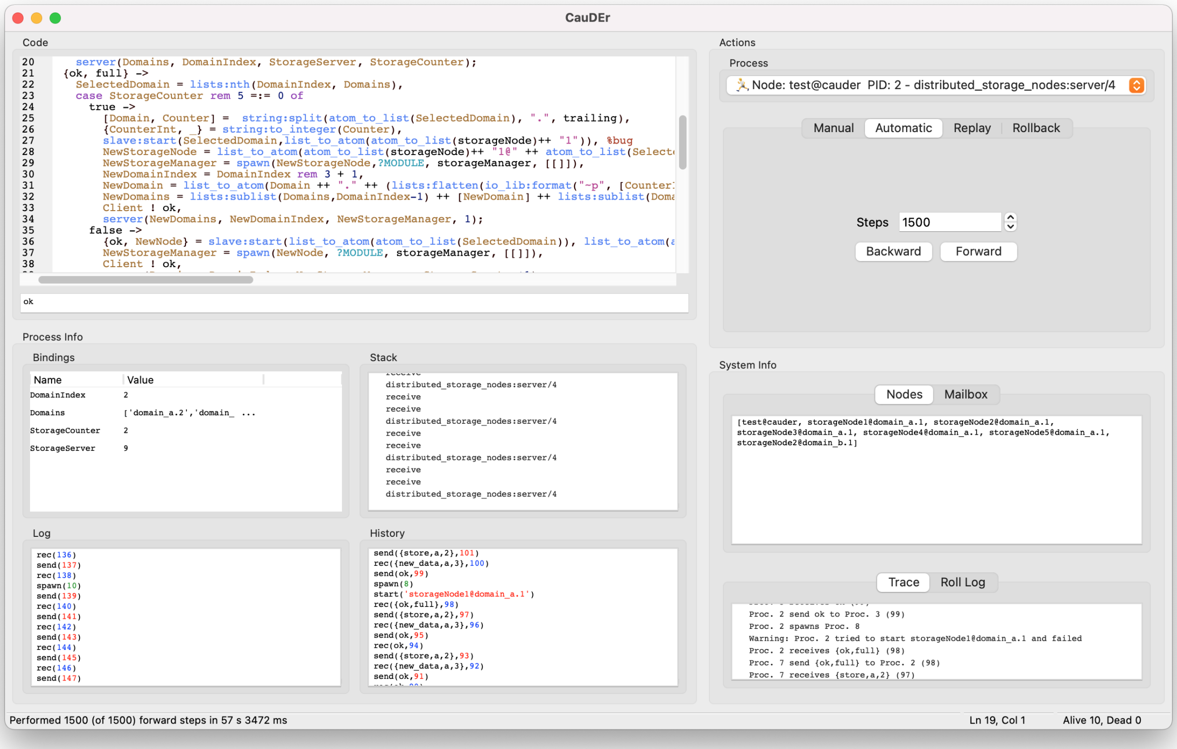
In addition, the rollback semantics generates the shortest computation satisfying the desired rollback request.

Theorem 2 (Minimality of rollback). *If $\llbracket \mathcal{S} \rrbracket_{\Psi} \rightsquigarrow^* \llbracket \mathcal{S}' \rrbracket_{\emptyset}$ then the backward steps occurring as first premises in the derivation of $\llbracket \mathcal{S} \rrbracket_{\Psi} \rightsquigarrow^* \llbracket \mathcal{S}' \rrbracket_{\emptyset}$ form the shortest computation from $\mathcal{S}$ satisfying Ψ derivable in the reversible semantics.*

A precise formalization and proof of this result is quite long, hence for space reasons we refer to [7, Theorem 3.2].

4 Distributed CauDEr

CauDEr [16,10,17] is the proof-of-concept debugger that we extended to support distribution following the semantics above. Notably, CauDEr works on Erlang, but primitives for distribution are the same in Core Erlang and in Erlang, hence our approach can be directly applied. CauDEr is written completely in Erlang and bundled up with a convenient graphical user interface to facilitate the interaction. The usual CauDEr workflow is the following. The user selects the Erlang source file, then CauDEr loads the program and shows the source code to the user. Then, the user can select the function that will act as entry point, specify its arguments, and the node identifier where the first process is running. The user can either perform single steps on some process (both forward and backward), or perform n steps in the chosen direction in an automatic manner (a scheduler decides which process will perform each step), or use the rollback operator.



```
    server(Domains, DomainIndex, StorageServer, StorageCounter);
  {ok, full} ->
    SelectedDomain = lists:nth(DomainIndex, Domains),
    case StorageCounter rem 5 =:= 0 of
      true ->
        [Domain, Counter] =  string:split(atom_to_list(SelectedDomain), ".", trailing),
        {CounterInt, _} = string:to_integer(Counter),
        slave:start(SelectedDomain,list_to_atom(atom_to_list(storageNode)++ "1")), %bug
        NewStorageNode = list_to_atom(atom_to_list(storageNode)++ "1@" ++ atom_to_list(Selecte
        NewStorageManager = spawn(NewStorageNode,?MODULE, storageManager, [[]]),
        NewDomainIndex = DomainIndex rem 3 + 1,
        NewDomain = list_to_atom(Domain ++ "." ++ (lists:flatten(io_lib:format("~p", [Counter
        NewDomains = lists:sublist(Domains,DomainIndex-1) ++ [NewDomain] ++ lists:sublist(Doma
        Client ! ok,
        server(NewDomains, NewDomainIndex, NewStorageManager, 1);
      false ->
        {ok, NewNode} = slave:start(list_to_atom(atom_to_list(SelectedDomain)), list_to_atom(a
        NewStorageManager = spawn(NewNode, ?MODULE, storageManager, [[]]),
        Client ! ok,
```

```
ok
```

Name	Value
DomainIndex	2
Domains	['domain_a.2','domain_ ...
StorageCounter	2
StorageServer	9

```
distributed_storage_nodes:server/4
receive
receive
distributed_storage_nodes:server/4
receive
receive
distributed_storage_nodes:server/4
receive
receive
distributed_storage_nodes:server/4
```

```
rec(136)
send(137)
rec(138)
spawn(10)
send(139)
rec(140)
send(141)
rec(142)
send(143)
rec(144)
send(145)
rec(146)
send(147)
```

```
send({store,a,2},101)
rec({new_data,a,3},100)
send(ok,99)
spawn(8)
start('storageNode1@domain_a.1')
rec({ok,full},98)
send({store,a,2},97)
rec({new_data,a,3},96)
send(ok,95)
rec(ok,94)
send({store,a,2},93)
rec({new_data,a,3},92)
send(ok,91)
```

```
[test@cauder, storageNode1@domain_a.1, storageNode2@domain_a.1,
storageNode3@domain_a.1, storageNode4@domain_a.1, storageNode5@domain_a.1,
storageNode2@domain_b.1]
```

```
Proc. 2 send ok to Proc. 3 (99)
Proc. 2 spawns Proc. 8
Warning: Proc. 2 tried to start storageNode1@domain_a.1 and failed
Proc. 2 receives {ok,full} (98)
Proc. 7 send {ok,full} to Proc. 2 (98)
Proc. 7 receives {store,a,2} (97)
```

```
Performed 1500 (of 1500) forward steps in 57 s 3472 ms        Ln 19, Col 1        Alive 10, Dead 0
```

Fig. 7. A screenshot of CauDEr.

The interface (see Fig. 7) is organized as follow: CauDEr shows the source code on the top left, the selected process' state and history (log is not considered in this paper) on the bottom left, and information on system structure and execution on the bottom right. Execution controls are on the top right.

We illustrate below how to use CauDEr to find a non-trivial bug.

Finding distributed bugs with CauDEr. Let us consider the following scenario. A client produces a stream of data and wants to store them in a distributed storage system. A server acts as a hub: it receives data from the client, forwards them to a storage node, receives a confirmation that the storage node has saved the data, and finally sends an acknowledgement to the client. Each storage node hosts one process only, acting as node manager, and has an id as part of its name, ranging from one to five. Each node manager is able to store at most m packets. Once the manager reaches the limit, it informs the server that its capacity has been reached. The server holds a list of domains and an index referring to one of them. Each domain is coupled with a counter, i.e., an integer, and each domain can host at most five storage nodes. Each time the server receives a notification from a node manager stating that the node maximum capacity has been reached, it proceeds as follows. If the id of the current storage manager is five it means that such domain has reached its capacity. Then, the server selects the next domain in the list, resets its counter and starts a new node (and a corresponding storage manager) on the new domain. If the id of the node is less than five then the server increases its counter and then starts a new node (and storage manager) on the same domain, using the value of the counter as new id. Each node should host at most one process.

Let us now consider the program distributed_storage_node.erl, available in the GitHub repository [8], which shows a wrong implementation of the program described above. In order to debug the program one has to load it and start the system. Then, it is sufficient to execute about 1500 steps forward to notice that something went wrong. Indeed, by checking the Trace box (Fig. 7) one can see a warning: a start has failed since a node with the same identifier already existed. Then, since no check is performed on the result of the start, the program spawns a new storage manager on a node with the same identifier as the one that failed to start. Hence, now two storage managers run on the same node.

To investigate why this happened one can roll back to the reception of the message $\{\mathsf{store}, \mathsf{full}\}$ right before the failed start. Note that it would not be easy to obtain the same result without reversibility: one would need to re-run the program, and, at least in principle, a different scheduling may lead to a different state where the error may not occur. After rolling back one can perform forward steps on the server in manual mode since the misbehavior happened there. After receiving the message, the server enters the case where the index of the storage manager is 5, which is correct because so far we have 5 storage nodes on the domain. Now, the server performs the start of the node (and of the storage manager) on the selected domain and only afterwards it selects the new domain, whereas it should have first selected a new domain and then proceeded to start a new storage node (and a new storage manager) there. This misbehavior has occurred because a few lines of code have been swapped.

5 Related work and conclusion

In this work we have presented an extension of CauDEr, a causal-consistent reversible debugger for Erlang, and the related theory. CauDEr has been first introduced in [16] (building on the theory in [18]) and then improved in [10] with a refined graphic interface and to work directly on Erlang instead of Core Erlang. We built our extension on top of this last version. CauDEr was able to deal with concurrent aspects of Erlang: our extension supports also some distribution primitives (start, node and nodes). We built the extension on top of the modular semantics for Erlang described in [18,10]. Monolithic approaches to the semantics of Erlang also exist [21], but the two-layer approach is more convenient for us since the reversible extension only affects the system layer.

Another work defining a formal semantics for distributed Erlang is [4]. There the emphasis is on ensuring the order of messages is respected in intra-node communications but not in inter-node communications (an aspect we do not consider). Similarly to us, they have rules to start new nodes and to perform remote spawns, although they do not consider the case where these rules fail.

In the context of CauDEr also replay has been studied [19]. In particular CauDEr supports causal-consistent replay, which allows one to replay the execution of the system up to a selected action, including *all and only* its *causes*. This can be seen as dual to rollback. Our extension currently does not support replay, we leave it for future work.

To the best of our knowledge causal-consistent debugging has been explored in a few settings only. The seminal paper [9] introduced causal-consistent debugging in the context of the toy language μOz. Closer to our work is Actoverse [22], a reversible debugger for the Akka actor model. Actoverse provides message-oriented breakpoints, which allow the user to stop when some conditions on messages are satisfied, rollback, state inspection, message timeline and session replay, which allows one to replay the execution of a program given the log of a computation, as well as the capacity to go back in the execution. While many of these features will be interesting for CauDEr, they currently do not support distribution.

Reversible debugging of concurrent programs has also been studied for imperative languages [11]. However, differently from us, they force undoing of actions in reverse order of execution, and they do not support distribution.

As future work it would be interesting to refine the semantics to deal with failures (node crashes, network partitions). Indeed, failures are unavoidable in practice, and we think reverse debugging in a faulty context could be of great help to the final user. Also, it would be good to extend CauDEr and the related theory to support additional features of the Erlang language, such as error handling, failure notification, and code hot-swapping. Finally, it would be good to experiment with more case studies to understand the practical impact of our tool.

References

1. Agha, G.A.: Actors: A Model of Concurrent Computation in Distributed Systems. The MIT Press (1986)
2. Carlsson, R., et al.: Core erlang 1.0.3. language specification (2004), URL: `https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf`
3. Cesarini, F., Thompson, S.: ERLANG Programming. O'Reilly Media, Inc. (2009)
4. Claessen, K., Svensson, H.: A semantics for distributed Erlang. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang. p. 78–87. ACM (2005)
5. Danos, V., Krivine, J.: Reversible communicating systems. In: CONCUR. LNCS, vol. 3170, pp. 292–307. Springer (2004)
6. Engblom, J.: A review of reverse debugging. In: Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference. pp. 1–6 (2012)
7. Fabbretti, G.: Causal-Consistent Debugging Of Distributed Erlang. Master's thesis, University of Bologna (2020), `https://amslaurea.unibo.it/22195/`

8. Fabbretti, G., Lanese, I.: Distributed CauDEr website. URL: `https://github.com/gfabbretti8/cauder-v2.git` (2021)
9. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: FASE. LNCS, vol. 8411, pp. 370–384. Springer (2014)
10. González-Abril, J.J., Vidal, G.: Causal-consistent reversible debugging: Improving CauDEr. In: PADL. LNCS, vol. 12548, pp. 145–160. Springer (2021)
11. Hoey, J., Ulidowski, I.: Reversible imperative parallel programs and debugging. In: RC. LNCS, vol. 11497, pp. 108–127. Springer (2019)
12. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development **5**(3), 183–191 (1961)
13. Lanese, I., Medic, D.: A general approach to derive uncontrolled reversible semantics. In: CONCUR. LIPIcs, vol. 171, pp. 33:1–33:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
14. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.: Controlling reversibility in higher-order pi. In: CONCUR. LNCS, vol. 6901, pp. 297–311. Springer (2011)
15. Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent reversibility. Bull. EATCS **114** (2014)
16. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: A causal-consistent reversible debugger for Erlang. In: FLOPS. LNCS, vol. 10818, pp. 247–263 (2018)
17. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr website. URL: `https://github.com/mistupv/cauder-v2` (2018)
18. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. Journal of Logical and Algebraic Methods in Programming **100**, 71 – 97 (2018)
19. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay reversible semantics for message passing concurrent programs. Fundam. Informaticae pp. 229–266 (2021)
20. McNellis, J., Mola, J., Sykes, K.: Time travel debugging: Root causing bugs in commercial scale software. CppCon talk, `https://www.youtube.com/watch?v=l1YJTg_A914` (2017)
21. R. Caballero, E. Martin-Martin, A.R., Tamarit, S.: Declarative debugging of concurrent Erlang programs. Journal of Logical and Algebraic Methods in Programming **101**, 22–41 (2018)
22. Shibanai, K., Watanabe, T.: Actoverse: A reversible debugger for actors. In: ACM SIGPLAN. p. 50–57 (2017)
23. Svensson, H., Fredlund, L.r., Benac Earle, C.: A unified semantics for future Erlang. In: Proceedings of the 9th ACM SIGPLAN Workshop on Erlang. p. 23–32 (2010)

$$\mathit{Var}\,\frac{}{\theta, X \xrightarrow{\tau} \theta, \theta(X)} \qquad \mathit{Tuple}\,\frac{\theta, e_i \xrightarrow{\ell} \theta', e_i'}{\theta, \{\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}\} \xrightarrow{\ell} \theta', \{\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}\}}$$

$$\mathit{List1}\,\frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, [e_1|e_2] \xrightarrow{\ell} \theta', [e_1'|e_2]} \qquad \mathit{List2}\,\frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, [v_1|e_2] \xrightarrow{\ell} \theta', [v_1|e_2']}$$

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

$$\mathit{Case1}\,\frac{\theta, e \xrightarrow{\ell} \theta', e'}{\theta, \mathsf{case}\ e\ \mathsf{of}\ cl_1; ...; cl_n\ \mathsf{end} \xrightarrow{\ell} \theta', \mathsf{case}\ e'\ \mathsf{of}\ cl_1; ...; cl_n\ \mathsf{end}}$$

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

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

$$\mathit{Call2}\,\frac{\mathsf{eval}(\mathsf{op}, v_1, ..., v_n) = v}{\theta, \mathsf{call}\ \mathsf{op}(v_1, ..., v_n) \xrightarrow{\tau} \theta, v}$$

$$\mathit{Apply1}\,\frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, ..., n\}}{\theta, \mathsf{apply}\ a/n(\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \xrightarrow{\ell} \theta', \mathsf{apply}\ a/n(\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}})}$$

$$\mathit{Apply2}\,\frac{\mu(a/n) = \mathsf{fun}(X_1, ..., X_n) \to e}{\theta, \mathsf{apply}\ a/n(v_1, ..., v_n) \xrightarrow{\tau} \theta \cup \{X_1 \mapsto v_1, ..., X_n \mapsto v_n\}, e}$$

Fig. 8. Standard semantics: evaluation of sequential expressions.

A Semantics

This section contains the various semantics that due to space reasons we could not fit in the paper. It is intended as a help for the reader providing auxiliary information locally but should not be considered a complete discussion, indeed, for the sake of brevity, we omit some details and we do not discuss some of the simplest scenarios. For a complete discussion about the concurrent aspects of the semantics we refer the reader to [18] and to [7] for an extensive discussion of the distributed aspects.

A.1 Expression level semantics

The expression level semantics is defined in terms of the labeled transition relation:

$$\{Env, Expr\} \times Label \times \{Env, Expr\}$$

where Env represents the domain of environments, $Expr$ represents the domain of expressions while $Label$ is an element of the following set:

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

$$\textit{Send1}\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1 \ !\ e_2 \xrightarrow{\ell} \theta', e_1' \ !\ e_2} \qquad \textit{Send2}\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, v_1 \ !\ e_2 \xrightarrow{\ell} \theta', v_1 \ !\ e_2'}$$

$$\textit{Send3}\ \frac{}{\theta, v_1 \ !\ v_2 \xrightarrow{\mathsf{send}(v_1,v_2)} \theta, v_2}$$

$$\textit{Receive}\ \frac{}{\theta, \mathsf{receive}\ cl_1; ...; cl_n\ \mathsf{end} \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta, \kappa}$$

$$\textit{Spawn1}\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, ..., n\}}{\theta, \mathsf{spawn}(a/n, [\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}]) \xrightarrow{\ell} \theta', \mathsf{spawn}(a/n, [\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}])}$$

$$\textit{Spawn2}\ \frac{}{\theta, \mathsf{spawn}(a/n, [\overline{v_n}]) \xrightarrow{\mathsf{spawn}(\kappa,a/n,[\overline{v_n}])} \theta, \kappa}$$

$$\textit{Self}\ \frac{}{\theta, \mathsf{self}() \xrightarrow{\mathsf{self}(\kappa)} \theta, \kappa}$$

Fig. 9. Concurrent semantics: evaluation of concurrent expressions.

We use ℓ to spawn over labels.

We divide the expression rules in two sets, the first one being the set of sequential expressions, depicted in Fig. 8, and the second one being the set of concurrent expressions, depicted in Fig. 9 (by abuse of notation we put self in the second set). As it is in Erlang, we consider that the order of evaluation of the arguments is fixed from left to right.

The sequential expressions define the behavior of some constructs of the language without side-effects, like the case construct, the let binding or the call of a function. Moreover, these rules define the evaluation of an expression inside the data structures of the language (lists and tuples). We label the evaluation of sequential expressions with τ since they can be considered 'silent' operations and we do not need to distinguish them in the system semantics.

A function in a program can be either defined by the user or built-in. In the former case we exploit rule *Apply*, where μ maps a function name a/n to its body, in the latter case we apply rule rule *Call*, where eval evaluates the function.

The only tricky situation in the sequential rules arises in rule $Case2$, where the auxiliary function match is used. Given an environment θ, a value v, and $\overline{cl_n}$ clauses the function match searches the first clause cl_i such that v matches pat_i and the guard holds.

Now, let us move to the concurrent expressions. Here, we gathered the rules that define the semantics of concurrent operations at the expression level. We can categorize these rules depending on whether we locally know or not to what they reduce. Rules $Send1, Send2, Send3$ and $Spawn1$ belong to the first category, indeed we do not need any of the information available at the system level to evaluate them. On the contrary, to evaluate rules $Receive, Spawn2$ and $Self$ we

$$\mathit{Seq}\,\frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, \theta, e, q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, \theta', e', q\rangle \mid \Pi}$$

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

$$\mathit{Receive}\,\frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q) = (\theta_i, e_i, v)}{\Gamma; \langle p, \theta, e, q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}, q\backslash\!\!\backslash v\rangle \mid \Pi}$$

$$\mathit{Spawn}\,\frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa,a/n,[\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, \theta, e, q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, \theta', e'\{\kappa \mapsto p'\}, q\rangle \mid \langle p', [\,], id, \mathsf{apply}\ a/n(\overline{v_n}, [\,])\rangle \mid \Pi}$$

$$\mathit{Self}\,\frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, \theta, e, q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, \theta', e'\{\kappa \mapsto p\}, q\rangle \mid \Pi}$$

$$\mathit{Sched}\,\frac{}{\Gamma \cup \{(p, v)\}; \langle p, \theta, e, q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, \theta, e, v : q\rangle \mid \Pi}$$

Fig. 10. Standard semantics: system rules.

need information available only at the system level. To tackle the problem, each of these expressions returns a fresh variable κ, which acts like a future, then the system level is in charge of binding κ to its proper value (e.g., the pid of the new process).

A.2 System level semantics

Fig. 10 depicts the rules of the system semantics, which is defined in terms of the relation $\hookrightarrow$. The system semantics defines the behavior of the system, here intended as the global mailbox Γ and the pool of processes Π. Each rule defines how the system can transit from one state to another in a forward manner.

We can see how the system semantics relies on the expression semantics for the evaluation of the concurrent expressions and then how it performs the corresponding action. The corresponding action can be the substitution of κ with its actual value, the application of the appropriate side-effect or both.

Let us consider the case of rule *Spawn*. In rule *Spawn* the system level relies on the expression level to evaluate the spawn expression, then it chooses a fresh identifier p as the pid of the new process, replaces κ with p and finally adds to Π the new process.

Now, let us discuss more in detail rule *Receive*, as it may not be of immediate understanding. The receive construct traverses the queue of messages, from the oldest to the newest, searching for the first message that matches one of the n clauses. If one is found then the receive evaluates to the expression relative to the matching clause, otherwise the process suspends itself. Here, the auxiliary function $\mathsf{matchrec}$, given an environment, a list of clauses, and a queue of mes-

$$(\mathit{Seq}) \qquad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \tau(\theta, e) : h, \theta', e', q\rangle \mid \Pi}$$

$$(\mathit{Send}) \qquad \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e' \quad \lambda \text{ is a fresh identifier}}{\Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi \rightharpoonup \Gamma \cup (p'', \{\lambda, v\}); \langle p, \mathsf{send}(\theta, e, p'', \{\lambda, v\}) : h, \theta', e', q\rangle \mid \Pi}$$

$$(\mathit{Receive}) \qquad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q = (\theta_i, e_i, \{\lambda, v\})}{\Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{rec}(\theta, e, \{\lambda, v\}, q) : h, \theta'\theta_i, e'\{\kappa \mapsto e_i\}, q \backslash\!\!\backslash \{\lambda, v\}\rangle \mid \Pi}$$

$$(\mathit{Spawn}) \qquad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh identifier}}{\begin{array}{r}\Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, \theta', e'\{\kappa \mapsto p'\}, q\rangle \\ \mid \langle p', [\,], id, \mathsf{apply}\ a/n(\overline{v_n}), [\,]\rangle \mid \Pi\end{array}}$$

$$(\mathit{Self}) \qquad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{self}(\theta, e) : h, \theta', e'\{\kappa \mapsto p\}, q\rangle \mid \Pi}$$

$$(\mathit{Sched}) \qquad \frac{}{\Gamma \cup \{(p, \{\lambda, v\})\}; \langle p, h, \theta, e, q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, h, \theta, e, \{\lambda, v\} : q\rangle \mid \Pi}$$

Fig. 11. Forward reversible semantics.

sages, searches for the first message v that matches one of the patterns. When one is found it returns the corresponding environment and expression alongside with the matched message v. Then, the system semantics updates the process' environment with the new variables introduced by the selected branch, replaces κ with the corresponding expression, and removes v from the process' queue.

Then, rule *Send* and rule *Sched* are the rules that respectively send and schedule a message. More precisely, we apply rule *Send* every time a process performs a send and, as a side-effect, we update the value of Γ by adding the pair (p'', m), where p'' is the pid of the intended receiver and m is the message. Whereas, by applying rule *Sched* nondeterministically we remove a message from Γ and we add it to its receiver queue. The fact that rule *Sched* chooses the pair nondeterministically allows to model every possible interleaving of the messages.

Finally, rule *Seq* denotes a silent operation without side-effects and *Self* the call of function self by process p.

A.3 A reversible semantics

The forward reversible semantics, depicted in Fig. 11 and defined by the relation $\rightharpoonup$, is the natural extension of the system semantics where each process has been endowed with a history of its previous configurations. More precisely, each time a process performs a forward step, the current configuration and possibly additional pieces of information are saved in the history, then the process transits in a new state.

$$(\overline{Seq}) \quad \Gamma; \langle p, \tau(\theta, e) : h, \theta', e', q\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi$$

$$(\overline{Send}) \quad \Gamma \cup \{(p'', \{\lambda, v\})\}; \langle p, \mathsf{send}(\theta, e, p'', \{\lambda, v\}) : h, \theta', e', q\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi$$

$$(\overline{Receive}) \quad \Gamma; \langle p, \mathsf{rec}(\theta, e, \{\lambda, v\}, q) : h, \theta', e', q \backslash\!\backslash \{\lambda, v\}\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi$$

$$(\overline{Spawn}) \quad \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, \theta', e', q\rangle \mid \langle p', [\,], id, e'', [\,]\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi$$

$$(\overline{Self}) \quad \Gamma; \langle p, \mathsf{self}(\theta, e) : h, \theta', e', q\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, \theta, e, q\rangle \mid \Pi$$

$$(\overline{Sched}) \quad \begin{array}{l} \Gamma; \langle p, h, \theta, e, \{\lambda, v\} : q\rangle \mid \Pi \leftharpoondown \Gamma \cup \{(p, \{\lambda, v\})\}; \langle p, h, \theta, e, q\rangle \mid \Pi \\ \qquad \text{if the topmost } \mathsf{rec}(...) \text{ item in h (if any) has the} \\ \qquad \text{form } \mathsf{rec}(\theta', e', \{\lambda', v'\}, q') \text{ with } q' \backslash\!\backslash \{\lambda', v'\} \neq \{\lambda, v\} : q \end{array}$$

Fig. 12. Backward reversible semantics.

The history serves two purposes: it is used to check that all the consequences of the action, if any, have been undone and to travel back in the process' computation.

The only difference, beyond the history, between the forward reversible semantics and the system semantics is that in the former each message is uniquely identified by an id, usually denoted by λ or λ'. The need to uniquely identify each message arise from the necessity to identify a precise point in the past of a process' computation.

Let us consider the following example to clarify such need.

Example 1. Let us consider three processes p_1, p_2 and p_3; p_1 sends a message v to p_2, then p_3 sends the same message v to p_2. Now, if we desire to undo the send of message v sent by p_1 we first need to undo the receive of p_2 and to do so we need to be able to precisely identify the moment in p_2's computation when it received the message. If the only information available in the process' history is the content of the message it would be impossible to determine who is the sender of two identical messages.

The problem described in the previous example does not arise if each message is paired with a unique identifier. Indeed, in that case it is sufficient to undo the computation of the receiver up to the receive of the message with the desired identifier. We refer to [18] for a more detailed discussion of the problem.

The uncontrolled backward semantics is depicted in Fig. 12 and defined in terms of the relation $\leftharpoondown$. The backward semantics restores previous states of a process' computation if all of the consequences of the target action have been undone. In some cases, e.g., rule $\overline{Seq}$, since the action does not have conquences we can simply restore the computation without performing any checks. In other cases, e.g., rule $\overline{Spawn}$, before undoing a step we need to perform additional checks.

$$(Seq)\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle nid, p, \tau(\theta, e) : h, \theta', e', q\rangle \mid \Pi; \Omega}$$

$$(Send)\quad \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e' \quad \lambda \text{ is a fresh identifier}}{\begin{array}{c}\Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \\ \rightharpoonup \Gamma \cup \{(p'', \{\lambda, v\})\}; \langle nid, p, \mathsf{send}(\theta, e, p'', \{\lambda, v\}) : h, \theta', e', q\rangle \mid \Pi; \Omega\end{array}}$$

$$(Receive)\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q) = (\theta_i, e_i, \{\lambda, v\})}{\begin{array}{c}\Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \\ \rightharpoonup \Gamma; \langle nid, p, \mathsf{rec}(\theta, e, \{\lambda, v\}, q) : h, \theta'\theta_i, e'\{\kappa \mapsto e_i\}, q \backslash\!\backslash \{\lambda, v\}\rangle \mid \Pi; \Omega\end{array}}$$

$$(Self)\quad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle nid, p, \mathsf{self}(\theta, e) : h, \theta', e'\{\kappa \mapsto p\}, q\rangle \mid \Pi; \Omega}$$

$$(Sched)\quad \frac{}{\Gamma \cup \{(p, \{\lambda, v\})\}; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle nid, p, h, \theta, e, \{\lambda, v\} : q\rangle \mid \Pi; \Omega}$$

Fig. 13. Missing rules of the extended forward reversible semantics

Rules $\overline{Receive}$ and $\overline{Sched}$ are the most complicated cases of the semantics, hence let us discuss them in detail. Transitions derived using rules $\overline{Sched}$ and $\overline{Receive}$ (applied to the same process) do not commute. In other words, to ensure causal consistency, we must undo them in reverse order of execution. The fact that two applications of rule $\overline{Sched}$ do not commute is ensured by the fact that we undo a $\overline{Sched}$ only when the target message is on top of the queue q, hence q is forcing the order. Applications of rule $\overline{Receive}$ do not commute because we can undo a *Receive* only when the corresponding item is on the top of the process history. Finally, applications of rules $\overline{Sched}$ and $\overline{Receive}$ do not commute thanks to the side condition of rule $\overline{Sched}$ and since we apply rule $\overline{Receive}$ only when the queue is the same as when the forward step has been performed.

The other rules are more straightforward. Rule $\overline{Send}$ can be applied when the message is in Γ because in this case we are sure that each of its consequences has been undone already. Rule $\overline{Spawn}$ can be applied when the child has an empty history. Finally, rules $\overline{Seq}$ and $\overline{Self}$ can always be applied since they never have consequences.

A.4 A distributed reversible semantics

Fig. 13 and Fig. 14 show the missing rules of the semantics depicted in Fig. 3 and Fig. 4. The behavior defined by the rules depicted in the two figures is the same as the one described in Section A.3.

$$(\overline{\mathit{Seq}}) \quad \Gamma; \langle nid, p, \tau(\theta, e) : h, \theta', e', q\rangle \mid \Pi; \Omega \leftharpoondown_{p,\mathsf{seq},\{s\}} \Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega$$

$$(\overline{\mathit{Send}}) \quad \begin{array}{c} \Gamma \cup \{(p'', \{\lambda, v\})\}; \langle nid, p, \mathsf{send}(\theta, e, p'', \{\lambda, v\}) : h, \theta', e', q\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p,\mathsf{send}(\lambda),\{s,\lambda\Uparrow\}} \Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \end{array}$$

$$(\overline{\mathit{Receive}}) \quad \begin{array}{c} \Gamma; \langle nid, p, \mathsf{rec}(\theta, e, \{\lambda, v\}, q) : h, \theta', e', q \setminus\!\!\setminus \{\lambda, v\}\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p,\mathsf{rec}(\lambda),\{s,\lambda\Downarrow\}} \Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \end{array}$$

$$(\overline{\mathit{Self}}) \quad \Gamma; \langle nid, p, \mathsf{self}(\theta, e) : h, \theta', e', q\rangle \mid \Pi; \Omega \leftharpoondown_{p,\mathsf{self},\{s\}} \Gamma; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega$$

$$(\overline{\mathit{Sched}}) \quad \begin{array}{c} \Gamma; \langle nid, p, h, \theta, e, \{\lambda, v\} : q\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p,\mathsf{sched}(\lambda),\{\lambda^{sched}\}} \Gamma \cup \{(p, \{\lambda, v\})\}; \langle nid, p, h, \theta, e, q\rangle \mid \Pi; \Omega \\ \text{if the topmost } \mathsf{rec}(...) \text{ item in h (if any) has the} \\ \text{form } \mathsf{rec}(\theta', e', \{\lambda', v'\}, q') \text{ with } q' \setminus\!\!\setminus \{\lambda', v'\} \neq \{\lambda, v\} : q \end{array}$$

Fig. 14. Missing rules of the extended backward reversible semantics