CauDEr: A Causal-Consistent Reversible Debugger for Erlang

Chapter · January 2018

DOI: 10.1007/978-3-319-90686-7_16

4 authors, including:

Ivan Lanese
University of Bologna
132 PUBLICATIONS **1,810** CITATIONS
SEE PROFILE

Adrian Palacios
Universitat Politècnica de València
11 PUBLICATIONS **54** CITATIONS
SEE PROFILE

German Vidal
Universitat Politècnica de València
157 PUBLICATIONS **1,283** CITATIONS
SEE PROFILE

CauDEr: A Causal-Consistent Reversible Debugger for Erlang*

Ivan Lanese[1], Naoki Nishida[2], Adrián Palacios[3,**], and Germán Vidal[3]

[1] Focus Team, University of Bologna/INRIA
ivan.lanese@gmail.com
[2] Graduate School of Informatics, Nagoya University
nishida@i.nagoya-u.ac.jp
[3] MiST, DSIC, Universitat Politècnica de València
{apalacios, gvidal}@dsic.upv.es

Abstract. Programming languages based on the actor model, such as Erlang, avoid some concurrency bugs by design. However, other concurrency bugs, such as message order violations and livelocks, can still show up in programs. These hard-to-find bugs can be more easily detected by using causal-consistent reversible debugging, a debugging technique that allows one to traverse a computation both forward and backward. Most notably, causal consistency implies that, when going backward, an action can only be undone provided that its consequences, if any, have been undone beforehand. To the best of our knowledge, we present the first causal-consistent reversible debugger for Erlang, which may help programmers to detect and fix various kinds of bugs, including message order violations and livelocks.

1 Introduction

Over the last years, concurrent programming has become a common practice. However, it is also a difficult and error-prone activity, since concurrency enables faulty behaviours, such as *deadlocks* and *livelocks*, which are hard to avoid, detect and fix. One of the reasons for these difficulties is that these behaviours may show up only in some extremely rare circumstances (e.g., for some unusual scheduling).

A recent analysis [16] reveals that most of the approaches to software validation and debugging in message-passing concurrent languages like Erlang are based on some form of static analysis (e.g., Dialyzer [15], McErlang [6], Soter [5])

* This work has been partially supported by MINECO/AEI/FEDER (EU) under grant TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic), by COST Action IC1405 on Reversible Computation - extending horizons of computing, and by JSPS KAKENHI Grant Number JP17H01722.

** Partially supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores* and *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D*, MINECO (SEIDI), under FPI grants BES-2014-069749 and EEBB-I-16-11469.

or testing (e.g., QuickCheck [3], PropEr [18], Concuerror [10], CutEr [9]). However, these techniques are helpful only to find some specific categories of problems. On the other hand, traditional debuggers (like the one included in the OTP Erlang distribution) are sometimes not particularly useful when an unusual interleaving brings up an error, since recompiling the program for debugging may give rise to a completely different execution behaviour. In this setting, *causal-consistent reversible debugging* [7] may be useful to complement the previous approaches. Here, one can run a program in the debugger in a controlled manner. If something (potentially) incorrect shows up, the user can stop the forward computation and go backwards—in a causal-consistent way—to look for the origin of the problem. In this context, we say that a backward step is *causal consistent* [4,12] if an action cannot be undone until all the actions that depend on it have already been undone. Causal-consistent reversibility is particularly relevant for debugging because it allows us to undo the actions of a given process in a stepwise manner while ignoring the actions of the remaining processes, unless they are causally related. In a traditional reversible debugger, one can only go backwards in exactly the reverse order of the forward execution, which makes focusing on undoing the actions of a given process much more difficult, since they can be interleaved with completely unrelated actions from other processes.

The main contributions of this paper are the following. We have designed and implemented CauDEr, a publicly available software tool for causal-consistent reversible debugging of (a subset of) Erlang programs. The tool builds upon some recent developments on the causal-consistent reversible semantics of Erlang [17,13], though we also introduce (in Section 3) a new rollback semantics which is especially tailored for reversible debugging. In this semantics, one can for instance run a program backwards up to the sending of a particular message, the creation of a given process, or the introduction of a binding for some variable. We present our tool and illustrate its use for finding bugs that would be difficult to deal with using the previously available tools (Section 4). We use a concurrent implementation of the dining philosophers problem as a running example. CauDEr is publicly available from `https://github.com/mistupv/cauder`.

2 The Language

Erlang is a message passing concurrent and distributed functional programming language. We define our technique for (a subset of) Core Erlang [2], which is used as an intermediate representation during the compilation of Erlang programs. In this section, we describe the syntax and semantics of the subset of Core Erlang we are interested in.

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

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

Fig. 1. Language syntax rules

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

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

In a case expression "$\mathsf{case}\ e\ \mathsf{of}\ \mathit{pat}_1\ \mathsf{when}\ e_1 \to e'_1;\ \ldots;\ \mathit{pat}_n\ \mathsf{when}\ e_n \to e'_n\ \ \mathsf{end}$", we first evaluate e to a value, say v; then, we find (if it exists) the first clause $\mathit{pat}_i\ \mathsf{when}\ e_i \to e'_i$ such that v matches pat_i (i.e., there exists a substitution σ for the variables of pat_i such that $v = \mathit{pat}_i\sigma$) and $e_i\sigma$—the *guard*—reduces to *true*; then, the case expression reduces to $e'_i\sigma$. Note that guards can only contain calls to built-in functions (typically, arithmetic and relational operators).

Concurrent features. In this work, we consider that a *system* is a pool of processes that can only interact through message sending and receiving (i.e., there is no shared memory). Each process has an associated *pid* (process identifier), which is unique in a system. Here, pids are ordinary values. Formally, a process is denoted by a tuple $\langle p, (\theta, e), q\rangle$ where p is the pid of the process, (θ, e) is the control—which consists of an environment (a substitution) and an expression to be evaluated—and q is the process' mailbox, a FIFO queue with the sequence of messages that have been sent to the process.

A running *system*, which we denote by $\Gamma; \Pi$, is composed by Γ, the *global mailbox*, which is a multiset of pairs of the form $(target\_process\_pid, message)$, and Π, which is a pool of processes. Π is denoted by an expression of the form

$$\langle p_1, (\theta_1, e_1), q_1 \rangle \mid \cdots \mid \langle p_n, (\theta_n, e_n), q_n \rangle$$

Here, " $\mid$ " denotes an associative and commutative operator. We typically denote a system by an expression of the form $\Gamma; \langle p, (\theta, e), q \rangle \mid \Pi$ to point out that $\langle p, (\theta, e), q \rangle$ is an arbitrary process of the pool. Intuitively, Γ stores messages after they are sent, and before they are inserted in the target mailbox. Here, Γ (which is similar to the "ether" in [21]) is an artificial device used in our semantics to guarantee that all admissible message interleavings can be modelled.

In the following, we denote by $\overline{o_n}$ a sequence of syntactic objects $o_1, \ldots, o_n$ for some n.

The functions with side effects are $\mathsf{self}()$, "!", spawn, and $\mathsf{receive}$. The expression $\mathsf{self}()$ returns the pid of a process, while $p!v$ sends a message v to the process with pid p. New processes are spawned with a call of the form $\mathsf{spawn}(a/n, [\overline{v_n}])$, so that the new process begins with the evaluation of $\mathsf{apply}\ a/n\ (\overline{v_n})$. Finally, an expression "$\mathsf{receive}\ \overline{pat_n\ \mathsf{when}\ e_n \to e'_n}\ \ \mathsf{end}$" traverses the messages in the process' queue until one of them matches a branch in the receive statement; i.e., it should find the *first* message v in the process' queue (if any) such that $\mathsf{case}\ v\ \mathsf{of}\ pat_1\ \mathsf{when}\ e_1 \to e'_1; \ldots; pat_n\ \mathsf{when}\ e_n \to e'_n\ \mathsf{end}$ can be reduced; then, the receive expression evaluates to the same expression to which the above case expression would be evaluated, with the additional side effect of deleting the message v from the process' queue. If there is no matching message in the queue, the process *suspends* its execution until a matching message arrives.

Figure 2 shows an Erlang program implementing a simple client-server scheme with one server and two clients (a), as well as its translation into Core Erlang (b), where $\_C$, $\_X$ and $\_Y$ are anonymous variables introduced during the translation process to represent sequences of actions using let expressions. The execution starts with a call to function main/0. It first spawns two processes that execute functions server/0 and client/1, respectively, and then calls to function client/1 too. Client requests have the form $\{P, \mathrm{req}\}$, where P is the pid of the client. The server receives the message, returns a message ack to the client, and calls to function server/0 again in an endless loop. After processing the two requests, the server will suspend waiting for another request.

Following [13], the semantics of the language is defined in a modular way, so that the labelled transition relation $\xrightarrow{\ell}$ models the evaluation of *expressions* and $\hookrightarrow$ models the reduction of *systems*. Relation $\xrightarrow{\ell}$ follows a typical call-by-value semantics for side-effect free expressions;[4] in this case, reduction steps are labelled with τ. For the remaining functions, the expression rules cannot complete the reduction of an expression since some information is not *locally* available. In these cases, the steps are labelled with the information needed

[4] Because of lack of space, we are not presenting the rules of $\xrightarrow{\ell}$ here, but refer the interested reader to [13].

```
main() ->
  S = spawn(server/0, []),
  spawn(client/1, [S]),
  client(S).
server() ->
  receive
    {P, req} ->
      P ! ack,
      server()
  end.
client(S) ->
  S ! {self(), req},
  receive
    ack -> ok
  end.
```

(a) Erlang

$$
\begin{array}{rl}
\mathrm{main}/0 = \mathsf{fun}\ () \to & \mathsf{let}\ S = \mathsf{spawn}(\mathrm{server}/0, [\,]) \\
& \mathsf{in\ let}\ \_C = \mathsf{spawn}(\mathrm{client}/0, [S]) \\
& \mathsf{in\ apply}\ \mathrm{client}/1\ (S) \\
\mathrm{server}/0 = \mathsf{fun}\ () \to & \mathsf{receive} \\
& \quad \{P, \mathrm{req}\} \to \\
& \quad\ \mathsf{let}\ \_X = P\,!\,\mathrm{ack} \\
& \quad\ \mathsf{in\ apply}\ \mathrm{server}/0\ () \\
& \mathsf{end} \\
\mathrm{client}/1 = \mathsf{fun}\ (S) \to & \mathsf{let}\ \_Y = S\,!\,\{\mathsf{self}(), \mathrm{req}\} \\
& \mathsf{in\ receive} \\
& \quad \mathrm{ack} \to \mathrm{ok} \\
& \mathsf{end}
\end{array}
$$

(b) Core Erlang

Fig. 2. A simple client server

to complete the reduction within the system rules of Figure 3. For sending a message, an expression $p'' \,!\, v$ is reduced to v with the side-effect of (eventually) storing the message v in the mailbox of process p''. The associated label is thus $\mathsf{send}(p'', v)$ so that rule *Send* can complete the step by adding the pair (p'', v) to the global mailbox Γ.

The remaining functions, receive, spawn and self, are reduced to a fresh distinguished symbol κ (a sort of *future*) in the expression rules, since the value cannot be determined locally. Therefore, in these cases, the labels also include κ. Then, the system rules of Figure 3 will bind κ to its correct value: the selected expression in rule *Receive* and a pid in rules *Spawn* and *Self*.

To be more precise, for a receive statement, the label has the form $\mathsf{rec}(\kappa, \overline{cl_n})$ where $\overline{cl_n}$ are the clauses of the receive statement. In rule *Receive*, the auxiliary function matchrec is used to find the first message in the queue that matches a clause, then returning a triple with the matching substitution θ_i, the selected branch e_i and the selected message v. Here, $q \backslash\backslash v$ denotes a new queue that results from q by removing the oldest occurrence of message v.

For a spawn, the label has the form $\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])$, where a/n and $[\overline{v_n}]$ are the arguments of spawn. Rule *Spawn* then adds a new process with a fresh pid p' initialised with the application $\mathsf{apply}\ a/n\ (v_1, \ldots, v_n)$ and an empty queue.

For a self, only κ is needed in the label. Rule *Self* then proceeds in the obvious way by binding κ to the pid of the process.

The rules presented so far allow one to store messages in the global mailbox, but not to deliver them. This is the task of the scheduler, which is modelled by rule *Sched*. This rule nondeterministically chooses a pair (p, v) in the global mailbox Γ and delivers the message v to the target process p. Note also that Γ is a multiset, so we use "$\cup$" as multiset union.

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

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

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

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

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

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

Fig. 3. Standard semantics: system rules

3 Causal-Consistent Reversible Debugging

In this section, we present a causal-consistent reversible semantics for the considered language. The semantics is based on the reversible semantics for Erlang introduced in [17,13]. In particular, [13] presents an *uncontrolled* reversible semantics, which is highly non-deterministic, and a *controlled* semantics that performs a backward computation up to a given *checkpoint* in a mostly deterministic way. Here, we build on the uncontrolled semantics, and define a new controlled semantics which is more appropriate as a basis for a causal-consistent reversible debugger than the one in [13].

First, following [13], we introduce an instrumented version of the standard semantics. For this purpose, we exploit a typical Landauer's embedding [11] and include a "history" h in the states. In contrast to the standard semantics, messages now include a unique identifier (i.e., a timestamp λ). These identifiers are required to avoid mixing different messages with the same value (and possibly also with the same sender and/or receiver). More details can be found in [13].

The transition rules of the forward reversible semantics can be found in Figure 4. They are an easy—and conservative—extension of the semantics in Figure 3 by adding histories to processes. In the histories, we use terms headed by constructors τ, check, send, rec, spawn, and self to record the steps performed by the forward semantics. Note that the auxiliary function matchrec now deals with messages of the form $\{v, \lambda\}$, trivially extending the original function in the standard semantics by ignoring λ when computing the first matching message.

Rollback Debugging Semantics. Now, we introduce a novel *rollback semantics* to undo the actions of a given process. Here, processes in "rollback" mode are annotated using $\lfloor\ \rfloor_\Psi$, where Ψ is a set with the requested rollbacks. In par-

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

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

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

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

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

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

Fig. 4. Forward reversible semantics

ticular, we consider the following rollbacks to undo the actions of a given process in a causal-consistent way:

- s: one backward step;
- $\lambda^{\Uparrow}$: a backward derivation up to the sending of a message labelled with λ;
- $\lambda^{\Downarrow}$: a backward derivation up to the delivery of a message labelled with λ;
- λ^{rec}: a backward derivation up to the receive of a message labelled with λ;
- sp_p: a backward derivation up to the spawning of the process with pid p;
- sp: a backward derivation up to the creation of the annotated process;
- X: a backward derivation up to the introduction of variable X.

In the following, in order to simplify the reduction rules, we consider that our semantics satisfies the following *structural equivalence*:

$$(SC1)\ \Gamma; \lfloor \langle p, h, (\theta, e), q\rangle \rfloor_{\emptyset} \mid \Pi \equiv \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$
$$(SC2)\ \Gamma; \lfloor \langle p, [\,], (\theta, e), [\,]\rangle \rfloor_{\Psi} \mid \Pi \equiv \Gamma; \langle p, [\,], (\theta, e), [\,]\rangle \mid \Pi$$

Therefore, when the set of rollbacks is empty or the process is back to its initial state, we consider that the required rollback has been completed.

Our rollback debugging semantics is modelled with the reduction relation $\leftharpoondown$, defined by the rules in Figure 5. Here, we assume that $\Psi \neq \emptyset$ (but Ψ' might be empty). Let us briefly explain the rules of the rollback semantics:

- Some actions can be directly undone. This is the case dealt with by rules $\overline{Seq}$, $\overline{Send1}$, $\overline{Receive}$, $\overline{Spawn1}$, $\overline{Self}$, and $\overline{Sched}$. In every rule, we remove the corresponding rollback request from Ψ. In particular, all of them remove s (since a causal-consistent step has been performed). Rule $\overline{Seq}$ additionally removes the variables whose bindings were introduced in the last step; rule

$$(\overline{Seq}) \quad \Gamma; \lfloor\langle p, \tau(\theta,e):h, (\theta',e'), q\rangle\rfloor_\Psi \mid \Pi \rightharpoonup \Gamma; \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_{\Psi\setminus(\{\mathsf{s}\}\cup\mathcal{V})} \mid \Pi \quad \text{where } \mathcal{V} = \mathcal{D}om(\theta')\setminus\mathcal{D}om(\theta)$$

$$(\overline{Send1}) \quad \Gamma \cup \{(p',\{v,\lambda\})\}; \lfloor\langle p, \mathsf{send}(\theta,e,p',\{v,\lambda\}):h, (\theta',e'), q\rangle\rfloor_\Psi \mid \Pi \rightharpoonup \Gamma; \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_{\Psi\setminus\{\mathsf{s},\lambda^{\Uparrow}\}} \mid \Pi$$

$$(\overline{Send2}) \quad \begin{array}{r}\Gamma; \lfloor\langle p, \mathsf{send}(\theta,e,p',\{v,\lambda\}):h, (\theta',e'), q\rangle\rfloor_\Psi \mid \lfloor\langle p', h', (\theta'',e''), q'\rangle\rfloor_{\Psi'} \mid \Pi \\ \rightharpoonup \Gamma; \lfloor\langle p, \mathsf{send}(\theta,e,p',\{v,\lambda\}):h, (\theta',e'), q\rangle\rfloor_\Psi \mid \lfloor\langle p', h', (\theta'',e''), q'\rangle\rfloor_{\Psi'\cup\{\lambda^{\Downarrow}\}} \mid \Pi \\ \text{if } (p',\{v,\lambda\}) \text{ does not occur in } \Gamma \text{ and } \lambda^{\Downarrow} \notin \Psi'\end{array}$$

$$(\overline{Receive}) \quad \Gamma; \lfloor\langle p, \mathsf{rec}(\theta,e,\{v,\lambda\},q):h, (\theta',e'), q\backslash\backslash\{v,\lambda\}\rangle\rfloor_\Psi \mid \Pi \rightharpoonup \Gamma; \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_{\Psi\setminus\{\mathsf{s},\lambda^{\mathsf{rec}}\}} \mid \Pi$$

$$(\overline{Spawn1}) \quad \Gamma; \lfloor\langle p, \mathsf{spawn}(\theta,e,p''):h, (\theta',e'), q\rangle\rfloor_\Psi \mid \lfloor\langle p'', [\,], (\theta'',e''), [\,]\rangle\rfloor_{\Psi'} \mid \Pi \rightharpoonup \Gamma; \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_{\Psi\setminus\{\mathsf{s},\mathsf{sp}_{p''}\}} \mid \Pi$$

$$(\overline{Spawn2}) \quad \begin{array}{r}\Gamma; \lfloor\langle p, \mathsf{spawn}(\theta,e,p''):h, (\theta,e), q\rangle\rfloor_\Psi \mid \lfloor\langle p'', h'', (\theta'',e''), q''\rangle\rfloor_{\Psi'} \mid \Pi \\ \rightharpoonup \Gamma; \lfloor\langle p, \mathsf{spawn}(\theta,e,p''):h, (\theta,e), q\rangle\rfloor_\Psi \mid \lfloor\langle p'', h'', (\theta'',e''), q''\rangle\rfloor_{\Psi'\cup\{\mathsf{sp}\}} \mid \Pi \\ \text{if } h'' \neq [\,] \vee q'' \neq [\,] \text{ and } \mathsf{sp} \notin \Psi'\end{array}$$

$$(\overline{Self}) \quad \Gamma; \lfloor\langle p, \mathsf{self}(\theta,e):h, (\theta',e'), q\rangle\rfloor_\Psi \mid \Pi \rightharpoonup \Gamma; \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_{\Psi\setminus\{\mathsf{s}\}} \mid \Pi$$

$$(\overline{Sched}) \quad \begin{array}{r}\Gamma; \lfloor\langle p, h, (\theta,e), \{v,\lambda\}:q\rangle\rfloor_\Psi \mid \Pi \rightharpoonup \Gamma \cup (p,\{v,\lambda\}); \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_{\Psi\setminus\{\mathsf{s},\lambda^{\Downarrow}\}} \mid \Pi \\ \text{if the topmost } \mathsf{rec}(\ldots) \text{ item in } h \text{ (if any) has the} \\ \text{form } \mathsf{rec}(\theta',e',\{v',\lambda'\},q') \text{ with } q'\backslash\backslash\{v',\lambda'\} \neq \{v,\lambda\}:q\end{array}$$

Fig. 5. Rollback debugging semantics

$\overline{Send1}$ removes $\lambda^{\Uparrow}$ (representing the sending of the message with identifier λ); rule $\overline{Receive}$ removes λ^{rec} (representing the receiving of the message with identifier λ); rule $\overline{Spawn1}$ removes $\mathsf{sp}_{p''}$ (representing the spawning of the process with pid p''); and rule $\overline{Sched}$ removes $\lambda^{\Downarrow}$ (representing the delivery of the message with identifier λ). Note also that rule $\overline{Sched}$ requires a side condition to avoid the (incorrect) commutation of rules $\overline{Receive}$ and $\overline{Sched}$ (see [13] for more details on this issue).

– Other actions require some dependencies to be undone first. This is the case of rules $\overline{Send2}$ and $\overline{Spawn2}$. In the first case, rule $\overline{Send2}$ applies in order to "propagate" the rollback mode to the receiver of the message, so that rules $\overline{Sched}$ and $\overline{Send1}$ can be eventually applied. In the second case, rule $\overline{Spawn2}$ applies to propagate the rollback mode to process p'' so that, eventually, rule $\overline{Spawn1}$ can be applied. Observe that the rollback sp introduced by the rule $\overline{Spawn2}$ does not need to be removed from Ψ since the complete process is deleted from Π in rule $\overline{Spawn1}$.

The correctness of the new rollback semantics can be shown following a similar scheme as in [13] for proving the correctness of the rollback semantics for checkpoints.

We now introduce an operator that performs a causal-consistent backward derivation and is parameterised by a system, a pid and a set of rollback requests:

$$\mathsf{rb}(\Gamma; \langle p, h, (\theta, e), q\rangle \,|\, \Pi, p, \Psi) = \Gamma'; \Pi' \;\text{ if }\; \Gamma; \lfloor \langle p, h, (\theta, e), q\rangle \rfloor_{\Psi} \,|\, \Pi \leftharpoonup^{*} \Gamma'; \Pi' \not\leftharpoonup$$

The operator adds a set of rollback requests to a given process[5] and then performs as many steps as possible using the rollback debugging semantics.

By using the above parametric operator, we can easily define several rollback operators that are useful for debugging. Our first operator, $\mathsf{rollback}(\Gamma; \Pi, p)$, just performs a causal-consistent backward step for process p:

$$\mathsf{rollback}(\Gamma; \Pi, p) = \mathsf{rb}(\Gamma; \Pi, p, \{\mathsf{s}\})$$

Notice that this may trigger the execution of any number of backward steps in other processes in order to first undo the consequences, if any, of the step in p.

This operator can easily be extended to an arbitrary number of steps:

$$\mathsf{rollback}(\Gamma; \Pi, p, n) = \begin{cases} \Gamma; \Pi & \text{if } n = 0 \\ \mathsf{rollback}(\Gamma'; \Pi', p, n-1) & \text{if } n > 0 \text{ and} \\ & \mathsf{rollback}(\Gamma; \Pi, p) = \Gamma'; \Pi' \end{cases}$$

Also, we might be interested in going backward until a relevant action is undone. For instance, we introduce below operators that go backward up to, respectively, the sending of a message with a particular identifier λ, the receiving of a message with a particular identifier λ, and the spawning of a process with pid p':

$$\begin{aligned} \mathsf{rollback}(\Gamma; \Pi, p, \lambda^{\Uparrow}) &= \mathsf{rb}(\Gamma; \Pi, p, \{\lambda^{\Uparrow}\}) \\ \mathsf{rollback}(\Gamma; \Pi, p, \lambda^{\mathsf{rec}}) &= \mathsf{rb}(\Gamma; \Pi, p, \{\lambda^{\mathsf{rec}}\}) \\ \mathsf{rollback}(\Gamma; \Pi, p, \mathsf{sp}_{p'}) &= \mathsf{rb}(\Gamma; \Pi, p, \{\mathsf{sp}_{p'}\}) \end{aligned}$$

Note that p is a parameter of the three operators, but it could also be automatically computed (from λ in the first two rules, from p' in the last one) by inspecting the histories of the processes in Π. This is actually what CauDEr does.

Finally, we consider an operator that performs backward steps up to the introduction of a binding for a given variable:

$$\mathsf{rollback}(\Gamma; \Pi, p, X) = \mathsf{rb}(\Gamma; \Pi, p, \{X\})$$

Here, p cannot be computed automatically from X, since variables are local and, hence, variable X may occur in several processes; thus, p is needed to uniquely identify the process of interest.[6]

4 CauDEr: A Causal-Consistent Reversible Debugger

The CauDEr implementation is conveniently bundled together with a graphical user interface to facilitate the interaction of users with the reversible debugger.

[5] Actually, in this work, we only consider a single rollback request at a time, so Ψ is always a singleton. Nevertheless, our formalisation considers that Ψ is a set for notational convenience and, also, in order to accept multiple rollbacks in the future.

[6] Actually, in CauDEr, uniqueness of variable names is enforced via renaming.

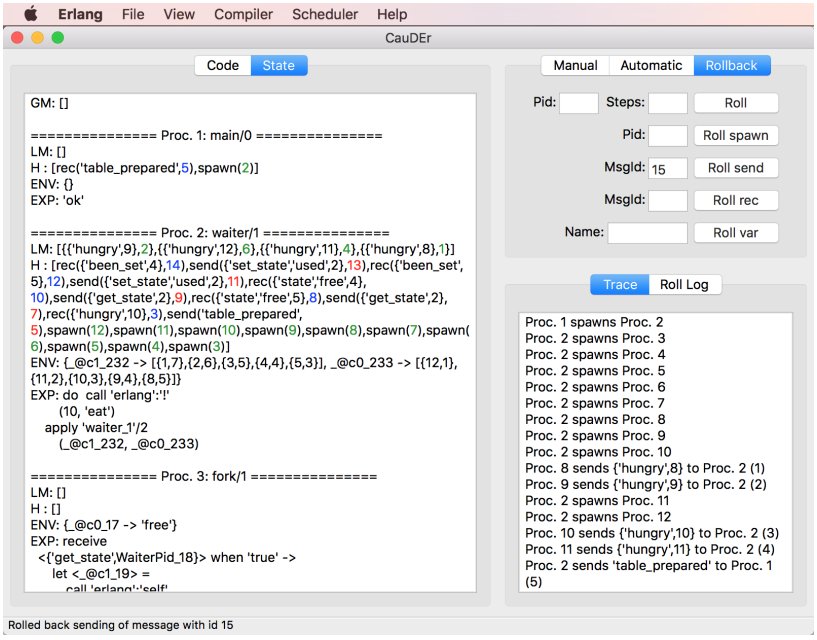

Fig. 6. CauDEr screenshot

CauDEr works as follows: when it is started, the first step is to select an Erlang source file. The selected source file is then translated into Core Erlang, and the resulting code is shown in the Code tab. Then, the user can choose any of the functions from the module and write the arguments that she wants to evaluate the function with. An initial system state, with an empty global mailbox and a single process performing the specified function application, appears in the State tab when the user presses the START button. Now, the user can explore possible program executions both forward and backward, according to three different modes, corresponding to the three tabs on the top right of the window in Figure 6. In the Manual mode, the user selects a process or message identifier, and buttons corresponding to forward and backward enabled reductions for the chosen process/message are available. Note that a backward reduction is *enabled* only if the action has no causal dependencies that need to be undone (single backward reductions correspond to applications of rules $\overline{Seq}$, $\overline{Send1}$, $\overline{Receive}$, $\overline{Spawn1}$, $\overline{Self}$, and $\overline{Sched}$ in Figure 5, see the uncontrolled reversible semantics in [13] for more details). In the Automatic mode one can decide the direction (forward or backward) and the number of steps to be performed. Actual steps are selected by a suitable scheduler. Currently, two (random) schedulers are available, one of which gives priority to processes w.r.t. the scheduling of messages (as in the "normalisation" strategy described in [13]), while the other has a uniform distribution. None of these schedulers mimics the Erlang/OTP scheduler. Indeed, it would be very hard to replicate this behaviour, as it depends on many parameters (threads, workload, etc). However, this is not necessary, since we are

only interested in reproducing the errors that occur in actual executions, and we discuss in future work how to obtain this without the need of mimicking the Erlang/OTP scheduler. The Automatic tab also includes a Normalize button, that executes all enabled actions but message schedulings. The last tab, Rollback, implements the rollback operators described in Section 3.

While exploring the execution, two tabs are updated to provide information on the system and its execution. The State tab describes the current system, including the global mailbox GM, and, for each process, the following components: the local mailbox LM, the history H, the environment ENV, and the expression under evaluation EXP. Identifiers of messages are highlighted in colour. This tab can be configured to hide any component of the process representation. Also, we consider two levels of abstraction for both histories and environments: for histories, we can either show all the actions or just the concurrent actions (send, receive and spawn); for environments, we can either show all variable bindings (called the *full* environment) or only the bindings for those variables occurring in the current expression (called the *relevant* environment).

The Trace tab gives a linearised description of the concurrent actions performed in the system, namely sends and receives of messages, and spawns of processes. This is aimed at giving a global picture of the system evolution, to highlight anomalies that might be caused by bugs.

A further tab is available, Roll Log, which is updated in case of rollbacks. It shows which actions have been actually undone upon a rollback request. This tab allows one to understand the causal dependencies of the target process of the rollback request, frequently highlighting undesired or missing dependencies directly caused by bugs.

The release version (v1.0) of CauDEr is fully written in Erlang, and it is publicly available from `https://github.com/mistupv/cauder` under the MIT license. The only requirement to build the application is to have Erlang/OTP installed and built with wxWidgets. The repository also includes some documentation and a few examples to easily test the application.

4.1 The CauDEr Workflow

A typical debugging session with CauDEr proceeds as follows. First, the user may run the program some steps forward using the Automatic mode in order to exercise the code. After each sequence of forward steps, she looks at the program output (which is not on the CauDEr window, but in the console where CauDEr has been launched) and possibly at the State and Trace tabs to check for abnormal behaviours. The State tab helps to identify these behaviours within a single process, while the Trace tab highlights anomalies in the global behaviour.

If the user identifies an unexpected action, she can undo it by using any (or a combination) of the available rollback commands. The Roll Log tab provides information on the causal-consistent rollbacks performed (in some cases, this log is enough to highlight the bug). From there, the user typically switches to the Manual mode in order to precisely control the doing or undoing of actions

in a specific state. This may involve performing other rollbacks to reach previous states. Our experience says that inspecting the full environment during the Manual exploration is quite helpful to locate bugs caused by sequential code.

4.2 Finding Concurrency Bugs with CauDEr

We use as a running example to illustrate the use of our debugger the well-known problem of dining philosophers. Here, we have a process for each philosopher and for each fork. We avoid implementations that are known to deadlock by using an arbitrator process, the waiter, that acts as an intermediary between philosophers and forks. In particular, if a philosopher wants to eat, he asks the waiter to get the forks. The waiter checks whether both forks are free or not. In the first case, he asks the forks to become used, and sends a message eat to the philosopher. Otherwise he sends a message think to the philosopher. When a philosopher is done eating, he sends a message eaten to the waiter, who in turn will release (i.e., set to free) the corresponding forks. The full Erlang code of the (correct) example, `dining.erl`, is available from `https://github.com/mistupv/dining-philos` .

Message order violation scenario. Here, we consider the buggy version of the program that can be found in file `dining_simple_bug.erl` of the above repository. In this example, running the program forward using the Automatic mode for about 600 steps is enough to discern something wrong. In particular, the user notices in the output that some philosophers are told to think when they should be told to eat, even at the beginning of the execution. Since the bug appears so early, it is probably a local bug, hence the user first focuses on the State tab. When the user considers the waiter process, she sees in the history an unexpected sequence of concurrent events of the following form (shown in reverse chronological order):

```
... ,send('think',10),rec('free',9),send({'get_state',2},8),
rec({'hungry',12},6),send({'get_state',2},7),rec({'hungry',9},2), ...
```

Here, the waiter has requested the state of a fork with send({'get_state',2},7), where 2 is the process id of the waiter itself and 7 the message id. Unexpectedly, the waiter has received a message hungry as a reply, instead of a message free or used. To get more insight on this, the user decides to rollback the receive of {'hungry',12}, which has 6 as message id. As a result, the rollback gets the system back to a state where send({'get_state',2},7) is the last concurrent event for the waiter process. Finally, the user switches to the Manual mode and notices that the next available action for the waiter process is to receive the message {'hungry',12} in the receive construct from the ask_state function. Function ask_state is called by the waiter process when it receives a hungry request from a philosopher (to get the state of the two forks). Obviously, a further message hungry should not be received here. The user easily realises then that the pattern in the receive is too general (in fact, it acts as a catch-all clause) and, as a result, the receive is matching also messages from other forks and even philosophers. Indeed, after

sending the message get_state to a fork, the programmer assumed that the next incoming message will be the state of the fork. However, the function is being evaluated in the context of the waiter process, where many other messages could arrive, e.g., messages hungry or eaten from philosophers.

It would not be easy to find the same bug using a standard debugger. Indeed, one would need to find where the wrong message hungry is sent, and put there a breakpoint. However, in many cases, no scheduling error will occur, hence many attempts would be needed. With a standard reversible debugger (like Actoverse [19]) one could look for the point where the wrong message is received, but it would be difficult to stop the execution at the exact message. Watch points do not help much, since all such messages are equal, but only some of them are received in the wrong receive operation. Indeed, in this example, the CauDEr facility of rollbacking a specific message receiving, coupled with the addition of unique identifiers to messages, is a key in ensuring the success of the debugging session.

Livelock scenario. Now, we consider the buggy version of the dining philosophers that can be found in file `dining_bug.erl` of our repository. In this case, the output of the program shows that, after executing some 2000 steps with the Automatic mode, some philosophers are always told to think, while others are always told to eat. In contrast to the previous example, this bug becomes visible only late in the execution, possibly only after some particular pattern of message exchanges has taken place (this is why it is harder to debug). In order to analyse the message exchanges the user should focus on the Trace tab first. By carefully examining it, the user realises that, in some cases, after receiving a message eaten from a philosopher, the waiter sends the two messages {'set_state','free',2} to release the forks to the same fork:

Proc. 2 receives {'eaten',10} (28)
Proc. 2 sends {'set_state','free',2} to Proc. 5 (57)
Proc. 5 receives {'set_state','free',2} (57)
Proc. 5 sends {'been_set',5} to Proc. 2 (58)
Proc. 2 receives {'been_set',5} (58)
Proc. 2 sends {'set_state','free',2} to Proc. 5 (59)
Proc. 5 receives {'set_state','free',2} (59)
Proc. 5 sends {'been_set',5} to Proc. 2 (60)
Proc. 2 receives {'been_set',5} (60)

Then, the user rollbacks the sending of the last message from the waiter process (the one with message id 59) and chooses to show the full environment (a clever decision). Surprisingly, the computed values for LeftForkId and RightForkId are equal. She decides to rollback also the sending of message with id 57, but she cannot see anything wrong there, so the computed value for RightForkId must be wrong. Now the user focuses on the corresponding line on the code, and she notices that the operands of the modulo operator have been swapped, which is the source of the erroneous behaviour.

This kind of livelocks are typically hard to find with other debugging tools. For instance, Concuerror [10] requires a finite computation, which is not the case in this scenario where the involved processes keep doing actions all the time but no global progress is achieved (i.e., some philosophers never eat).

5 Related Work

Causal-consistent debugging has been introduced by CaReDeb [7], in the context of language μOz. The present paper improves on CaReDeb in many directions. First, μOz is only a toy language where no realistic programs can be written (e.g., it supports only integers and a few arithmetic operations). Second, μOz is not distributed, since messages are atomically moved from the sender to a message queue, and from the queue to the target process. This makes its causality model, hence the definition of a causal-consistent reversible semantics, much simpler. Third, in [7] the precise semantics of debugging operators is not fully specified. Finally, the implementation described in [7] is just a proof-of-concept.

More in general, our work is in the research thread of causal-consistent reversibility (see [12] for a survey), first introduced in [4] in the context of process calculus CCS. Most of the works in this area are indeed on process calculi, but for the work on μOz already discussed (the theory was introduced in [14]) and a line of work on the coordination language μklaim [8]. However, μklaim is a toy language too. Hence, we are the first ones to consider a mainstream programming language. A first approach to the definition of a causal-consistent semantics of Erlang was presented in [17], and extended in [13]. While we based CauDEr on the uncontrolled semantics therein (and on its proof-of-concept implementation), we provided in the present paper an updated controlled semantics more suitable for debugging, and a mature implementation with a complete interface and many facilities for debugging. Moreover, our tool is able to deal with a larger subset of the language, mainly in terms of built-in functions and data structures.

While CaReDeb is the only other causal-consistent debugger we are aware of, two other reversible debuggers for actor systems exist. Actoverse [19] deals with Akka-based applications. It provides many relevant features which are complementary to ours. These include a partial-order graphical representation of message exchanges that would nicely match our causal-consistent approach, message-oriented breakpoints that allow one to force specific interleavings in message schedulings, and facilities for session replay to ensure bugs reappear when executing forward again. In contrast, Actoverse provides less facilities for state inspection and management than us (e.g., it has nothing similar to our Roll var command). Also, the paper does not include any theoretical framework defining the behaviour of the debugger. EDD is a declarative debugger for Erlang (see [1] for a version dealing with sequential Erlang). EDD tracks the concurrent actions of an execution and allows the user to select any of them to start the questions. Declarative debugging is essentially orthogonal to our approach.

Causeway [20] is not a full-fledged debugger but a post-mortem trace analyser, i.e., it performs no execution, but just explores a trace of a run. It con-

centrates on message passing aspects, e.g., it does not allow one to explore the state of single processes (states are not in the logs analysed by Causeway). On the contrary it provides nice mechanisms to abstract and filter different kinds of communications, allowing the user to decide at each stage of the debugging process which messages are of interest. These mechanisms would be an interesting addition for CauDEr.

6 Discussion

In this work, we have presented the design of CauDEr, a causal-consistent reversible debugger for Erlang. It is based on the reversible semantics introduced in [17,13], though we have introduced in this paper a new rollback semantics which is especially appropriate for debugging Erlang programs. We have shown in the paper that some bugs can be more easily located using our new tool, thus filling a gap in the collection of debugging tools for Erlang.

Currently, our debugger may run a program either forward or backward (in the latter case, in a causal-consistent way). After a backward computation that undoes some steps, we can resume the forward computation, though there are no guarantees that we will reproduce the previous forward steps. Some debuggers (so-called omniscient or back-in-time debuggers) allow us to move both forward and backward along a *particular* execution. As a future work, we plan to define a similar approach but ensuring that once we resume a forward computation, we can follow the same previous forward steps *or some other causal-consistent steps*. Such an approach might be useful, e.g., to determine which processes depend on a particular computation step and, thus, ease the location of a bug.

Another interesting line of future work involves the possibility of capturing a faulty behaviour during execution in the standard environment, and then replaying it in the debugger. For instance, we could instrument source programs so that their execution in a standard environment writes a log in a file. Then, when the program ends up with an error, we could use this log as an input to the debugger in order to explore this particular faulty behaviour (as postmortem debuggers do). This approach can be applied even if the standard environment is distributed and there is no common notion of time, since causal-consistent reversibility relies only on a notion of causality.

For the same reason we could also develop a fully distributed debugger, where each process is equipped with debugging facilities, and a central console allows us to coordinate them. This would strongly improve scalability, since most of the computational effort (running and backtracking programs) would be distributed. However, this step requires a semantics without any synchronous interaction (e.g., rules $\overline{Send2}$ and $\overline{Spawn2}$ would need to be replaced by a more complex asynchronous protocol).

Acknowledgements

The authors gratefully acknowledge the anonymous referees for their useful comments and suggestions.

References

1. Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S.: EDD: A declarative debugger for sequential Erlang programs. In: TACAS. LNCS, vol. 8413, pp. 581–586. Springer (2014)
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. Claessen, K., et al.: Finding race conditions in Erlang with QuickCheck and PULSE. In: ICFP. pp. 149–160. ACM (2009)
4. Danos, V., Krivine, J.: Reversible communicating systems. In: CONCUR. LNCS, vol. 3170, pp. 292–307. Springer (2004)
5. D'Osualdo, E., Kochems, J., Ong, C.L.: Automatic Verification of Erlang-Style Concurrency. In: SAS. LNCS, vol. 7935, pp. 454–476. Springer (2013)
6. Fredlund, L.A., Svensson, H.: McErlang: a model checker for a distributed functional programming language. In: ICFP. pp. 125–136. ACM (2007)
7. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: FASE. LNCS, vol. 8411, pp. 370–384. Springer (2014)
8. Giachino, E., Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent rollback in a tuple-based language. J. Log. Algebr. Meth. Program. 88, 99–120 (2017)
9. Giantsios, A., Papaspyrou, N., Sagonas, K.: Concolic testing for functional languages. In: PPDP. pp. 137–148. ACM (2015)
10. Gotovos, A., Christakis, M., Sagonas, K.: Test-driven development of concurrent programs using Concuerror. In: 10th ACM SIGPLAN workshop on Erlang. pp. 51–61. ACM (2011)
11. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5, 183–191 (1961)
12. Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent reversibility. Bulletin of the EATCS 114 (2014)
13. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang (2017), submitted for publication. Available from http://users.dsic.upv.es/~gvidal/lnpv17/paper.pdf
14. Lienhardt, M., Lanese, I., Mezzina, C.A., Stefani, J.: A reversible abstract machine and its space overhead. In: FMOODS/FORTE. LNCS, vol. 7273, pp. 1–17. Springer (2012)
15. Lindahl, T., Sagonas, K.: Practical type inference based on success typings. In: PPDP. pp. 167–178. ACM Press (2006)
16. Lopez, C.T., Marr, S., Mössenböck, H., Boix, E.G.: A study of concurrency bugs and advanced development support for actor-based programs. CoRR abs/1706.07372 (2017)
17. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: LOPSTR'16. LNCS, vol. 10184, pp. 259–274. Springer (2017)
18. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: 10th ACM SIGPLAN workshop on Erlang. pp. 39–50. ACM (2011)
19. Shibanai, K., Watanabe, T.: Actoverse: A reversible debugger for actors. In: AGERE. pp. 50–57. ACM (2017)
20. Stanley, T., Close, T., Miller, M.S.: Causeway: a message-oriented distributed debugger. Tech. rep., HPL-2009-78 (2009), available from http://www.hpl.hp.com/techreports/2009/HPL-2009-78.html
21. Svensson, H., Fredlund, L.A., Earle, C.B.: A unified semantics for future Erlang. In: 9th ACM SIGPLAN workshop on Erlang. pp. 23–32. ACM (2010)