Causal-Consistent Replay Debugging for Message Passing Programs[⋆]

Ivan Lanese[1], Adrián Palacios[2], and Germán Vidal[2]

[1] Focus Team, University of Bologna/INRIA
`ivan.lanese@gmail.com`
[2] MiST, DSIC, Universitat Politècnica de València
{`apalacios`, `gvidal`}`@dsic.upv.es`

Abstract. Debugging of concurrent systems is a tedious and error-prone activity. A main issue is that there is no guarantee that a bug that appears in the original computation is replayed inside the debugger. This problem is usually tackled by so-called replay debugging, which allows the user to record a program execution and replay it inside the debugger. In this paper, we present a novel technique for replay debugging that we call *controlled causal-consistent replay.* Controlled causal-consistent replay allows the user to record a program execution and, in contrast to traditional replay debuggers, to reproduce a visible misbehavior inside the debugger including all *and only* its causes. In this way, the user is not distracted by the actions of other, unrelated processes.

1 Introduction

Debugging is a main activity in software development. According to a 2014 study [24], the cost of debugging is $312 billions annually. Another recent study [2] estimates that the time spent in debugging is 49.9% of the total programming time. The situation is not likely to improve in the near future, given the increasing demand of concurrent and distributed software. Indeed, distribution is inherent in current computing platforms, such as the Internet or the Cloud, and concurrency is a must to overcome the advent of the power wall [25]. Debugging concurrent and distributed software is clearly more difficult than debugging sequential code [9]. Furthermore, misbehaviors may depend, e.g., on the execution speed of the different processes, showing up only in some (sometimes rare) cases.

[⋆] This work has been partially supported by the EU (FEDER) and the *Spanish Ministerio de Ciencia, Innovación y Universidades*/AEI (MICINN) under grant TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grants PROMETEO-II/2015/013 (SmartLogic) and Prometeo/2019/098 (DeepTrust), and by the COST Action IC1405 on Reversible Computation - extending horizons of computing. The first author has been also partially supported by French ANR project DCore ANR-18-CE25-0007. The second author has been also supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores* (MICINN) under FPI grant BES-2014-069749.

A particularly unfortunate situation is when a program exhibits a misbehavior in its usual execution environment, but it runs smoothly when re-executed in the debugger. This problem is usually tackled by so-called replay debugging, which allows the user to record a program execution and replay it inside the debugger. However, in concurrent programs, part of the execution may not be relevant: some processes may not have interacted with the one showing a misbehavior, or may have interacted with it only at the very beginning of their execution, hence most of their execution is not relevant for the debugging session. Having to replay all these behaviors is both time and resource consuming as well as distracting for the user.

Our main contribution in this paper is a novel technique for replay debugging that we call *controlled causal-consistent replay.* It extends the techniques in the literature as follows: given a log of a (typically faulty) concurrent execution, we do not replay exactly the same execution step by step (as traditional replay debuggers), but we allow the user to select any action in the log (e.g., one showing a misbehavior) and to replay the execution up to this action, including all *and only* its causes. This allows one to focus on those processes where (s)he thinks the bug(s) might be, disregarding the actual interleaving of processes. To the best of our knowledge, the notion of controlled causal-consistent replay is new.

We fully formalize causal-consistent replay for (a subset of) a realistic functional and concurrent programming language based on message-passing: Erlang. Moreover, we prove relevant properties, e,g., that misbehaviors in the original computation are always replayed, and that we guarantee minimal replay of observable behaviors. This is in contrast with most approaches to replay in the literature, that, beyond considering different languages, are either fully experimental (like, e.g., [18,19,27,1]), or present limited theoretical results, as in [21,8,10].

Causal-consistent replay can be seen as the dual of causal-consistent rollback, a technique for reversible computing which allows one to select an action in a computation and undo it, including all *and only* its consequences. Indeed, the two techniques integrate well, giving rise to a framework to explore back and forward a given concurrent computation, always concentrating on the actions of interest and avoiding unrelated actions. By lack of space, we will only present causal-consistent replay in this paper. More details, including the integration with causal-consistent rollback, proofs of technical results, and a description of an implemented reversible replay debugger for Erlang [16] that follows the ideas in this paper, can be found in an accompanying technical report [17]. While not technically needed, printing the paper in color may help the understanding.

2 The Language

We present below the considered language: a first-order functional and concurrent language based on message passing that mainly follows the actor model.

Language Syntax. The syntax of the language is in Figure 1. A program is a sequence of function definitions, where each function name f/n (atom/arity) has

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{program} & ::= & \mathit{fun}_1\ \ldots\ \mathit{fun}_n & \mathit{fun} & ::= & \mathit{fname} = \mathsf{fun}\ (X_1,\ldots,X_n) \rightarrow \mathit{expr}\\
\mathit{fname} & ::= & \mathit{Atom}/\mathit{Integer} & \mathit{lit} & ::= & \mathit{Atom} \mid \mathit{Integer} \mid \mathit{Float} \mid [\,]\\
\mathit{expr} & ::= & \multicolumn{4}{l}{\mathit{Var} \mid \mathit{lit} \mid \mathit{fname} \mid [\mathit{expr}_1|\mathit{expr}_2] \mid \{\mathit{expr}_1,\ldots,\mathit{expr}_n\}}\\
 & \mid & \multicolumn{4}{l}{\mathsf{call}\ \mathit{expr}\ (\mathit{expr}_1,\ldots,\mathit{expr}_n) \mid \mathsf{apply}\ \mathit{expr}\ (\mathit{expr}_1,\ldots,\mathit{expr}_n)}\\
 & \mid & \multicolumn{4}{l}{\mathsf{case}\ \mathit{expr}\ \mathsf{of}\ \mathit{clause}_1;\ldots;\mathit{clause}_m\ \mathsf{end}}\\
 & \mid & \multicolumn{4}{l}{\mathsf{let}\ \mathit{Var} = \mathit{expr}_1\ \mathsf{in}\ \mathit{expr}_2 \mid \mathsf{receive}\ \mathit{clause}_1;\ldots;\mathit{clause}_n\ \mathsf{end}}\\
 & \mid & \multicolumn{4}{l}{\mathsf{spawn}(\mathit{expr},[\mathit{expr}_1,\ldots,\mathit{expr}_n]) \mid \mathit{expr}_1\ !\ \mathit{expr}_2 \mid \mathsf{self}()}\\
\mathit{clause} & ::= & \mathit{pat}\ \mathsf{when}\ \mathit{expr}_1 \rightarrow \mathit{expr}_2 & \mathit{pat} & ::= & \mathit{Var} \mid \mathit{lit} \mid [\mathit{pat}_1|\mathit{pat}_2] \mid \{\mathit{pat}_1,\ldots,\mathit{pat}_n\}
\end{array}
$$

Fig. 1: Language syntax rules

an associated definition fun $(X_1,\ldots,X_n) \rightarrow e$, where $X_1,\ldots,X_n$ are (distinct) fresh variables and are the only variables that may occur free in e. The body of a function is an *expression*, which can include variables, literals, function names, lists (using Prolog-like notation: $[\,]$ is the empty list and $[e_1|e_2]$ is a list with head e_1 and tail e_2), tuples (denoted by $\{e_1,\ldots,e_n\}$),[3] calls to built-in functions (mainly arithmetic and relational operators), function applications, case expressions, let bindings, receive expressions, spawn (for creating new processes), "!" (for sending a message), and self. As is common practice, we assume that X is a fresh variable in let $X = \mathit{expr}_1$ in expr_2.

In this language, we distinguish expressions, patterns, and values, ranged over respectively by $e, e', e_1, \ldots$, by $\mathit{pat}, \mathit{pat}', \mathit{pat}_1, \ldots$ and by $v, v', v_1, \ldots$. In contrast to expressions, *patterns* are built from variables, literals, lists, and tuples. Patterns can only contain fresh variables. Finally, *values* are built from literals, lists, and tuples. Atoms (i.e., constants with a name) are written in roman letters, while variables start with an uppercase letter. A *substitution* θ is a mapping from variables to expressions, and $\mathcal{D}om(\theta)$ is its domain. Substitutions are usually denoted by (finite) sets of bindings like, e.g., $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$. 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 $X \in \mathit{Var}$. Substitution application $\sigma(e)$ is also denoted by $e\sigma$.

In a case expression "case e of pat_1 when $e_1 \rightarrow e'_1$; $\ldots$; pat_n when $e_n \rightarrow e'_n$ end", we first evaluate e to a value, say v; then, we find (if it exists) the first clause pat_i when $e_i \rightarrow e'_i$ such that v matches pat_i, i.e., such that there exists a substitution σ for the variables of pat_i with $v = \mathit{pat}_i\sigma$, and $e_i\sigma$ (the *guard*) reduces to *true*; then, the case expression reduces to $e'_i\sigma$.

In our language, a running system is a pool of processes that can only interact through message sending and receiving (i.e., there is no shared memory). Received messages are stored in the queues of processes until they are consumed; namely, each process has one associated local (FIFO) queue. Each process is uniquely identified by its *pid* (process identifier). Message sending is asynchronous, while receive instructions block the execution of a process until an appropriate message reaches its local queue (see below).

[3] As in Erlang, the only data constructors in the language (besides literals) are the predefined functions for lists and tuples.

```
  main/0 = fun () → let S = spawn(server/0, [ ])
                    in let P = spawn(proxy/0, [ ]) in apply client/2 (P, S)

server/0 = fun () → receive
                        {C, N} → receive
                                     M → let X = C ! call + (N, M) in apply server/0 ()
                                 end;
                        E → error
                    end

 proxy/0 = fun () → receive {T, M} → let W = T ! M in apply proxy/0 () end

client/2 = fun (P, S) → let X = P ! {S, {self(), 40}} in let Y = S ! 2 in receive N → N end
```

Fig. 2: A simple client/server program

In the paper, $\overline{o_n}$ denotes a sequence of syntactic objects $o_1, \ldots, o_n$.

We consider the following functions with side-effects: self, "!", spawn, and receive. The expression self() returns the pid of a process, while $p\,!\,v$ sends a message v to the process with pid p, which will be eventually stored in p's local queue. New processes are spawned with a call of the form spawn$(a/n, [\overline{v_n}])$, so that the new process begins with the evaluation of apply a/n $(\overline{v_n})$. Finally, an expression "receive $\overline{pat_n \text{ when } e_n \to e'_n}$ end" should find the *first* message v in the process' queue (if any) such that case v of $\overline{pat_n \text{ when } e_n \to e'_n}$ end can be reduced to some expression e''; then, the receive expression evaluates to e'', with the side effect of deleting the message v from the process' queue. If there is no matching message, the process *suspends* until a matching message arrives.

Our language models a significant subset of Core Erlang [3], the intermediate representation used during the compilation of Erlang programs. Therefore, our developments can be directly applied to Erlang (as can be seen in the technical report [17], where the development of a practical debugger is described).

Example 1. The program in Figure 2 implements a simple client/server scheme with one server, one client and a proxy. The execution starts with a call to function main/0. It spawns the server and the proxy and finally calls function client/2. Both the server and the proxy then suspend waiting for messages. The client makes two requests $\{C, 40\}$ and 2, where C is the pid of client (obtained using self()). The second request goes directly to the server, but the first one is sent through the proxy (which simply resends the received messages), so the client actually sends $\{S, \{C, 40\}\}$, where S is the pid of the server. Here, we expect that the server first receives the message $\{C, 40\}$ and, then, 2, thus sending back 42 to the client C (and calling function server/0 again in an endless recursion). If the first message does not have the right structure, the catch-all clause "$E \to$ error" returns error and stops.

A High-Level Semantics. Now, we present an (asynchronous) operational semantics for our language. Following [26], we introduce a *global mailbox* (there called "ether") to guarantee that our semantics generates all admissible message interleavings. In contrast to previous semantics [15,22,26], our semantics abstracts away from processes' queues. We will see in Section 2 that this decision simplifies both the semantics and the notion of independence, while still modeling the same potential computations (see the technical report [17]).

Definition 1 (process). *A process is a configuration $\langle p, \theta, e\rangle$, where p is its pid, θ an environment (a substitution of values for variables), and e an expression.*

In order to define a *system* (roughly, a pool of processes interacting through message exchange), we first need the notion of global mailbox.

Definition 2 (global mailbox). *We define a global mailbox, Γ, as a multiset of triples of the form $(sender\_pid, target\_pid, message)$. Given a global mailbox Γ, we let $\Gamma \cup \{(p, p', v)\}$ denote a new mailbox also including the triple (p, p', v), where we use "$\cup$" as multiset union.*

In Erlang, the order of two messages sent directly from process p to process p' is kept if both are delivered; see [5, Section 10.8].[4] To enforce such a constraint, we could define a global mailbox as a collection of FIFO queues, one for each sender-receiver pair. In this work, however, we keep Γ a multiset. This solution is both simpler and more general since FIFO queues serve only to select those computations satisfying the constraint. Nevertheless, if our logging approach is applied to a computation satisfying the above constraint, then our replay computation will also satisfy it, thus replay does not introduce spurious computations.

Definition 3 (system). *A system is a pair $\Gamma; \Pi$, where Γ is a global mailbox and Π is a pool of processes, denoted as $\langle p_1, \theta_1, e_1\rangle \mid \cdots \mid \langle p_n, \theta_n, e_n\rangle$; here "$\mid$" represents an associative and commutative operator. We often denote a system as $\Gamma; \langle p, \theta, e\rangle \mid \Pi$ to point out that $\langle p, \theta, e\rangle$ is an arbitrary process of the pool.*

A system is initial *if it has the form $\{\,\}; \langle p, id, e\rangle$, where $\{\,\}$ is an empty global mailbox, p is a pid, id is the identity substitution, and e is an expression.*

Following the style in [22], the semantics of the language is defined in a modular way, so that the labeled transition relations $\rightarrow$ and $\hookrightarrow$ model the evaluation of *expressions* and the reduction of *systems*, respectively. Given an environment θ and an expression e, we denote by $\theta, e \xrightarrow{l} \theta', e'$ a one-step reduction labeled with l. The relation $\xrightarrow{l}$ follows a typical call-by-value semantics for side-effect free expressions; for expressions with side-effects, we label the reduction with the information needed to perform the side-effects within the system rules of Figure 3. We refer to the rules of Figure 3 as the *logging* semantics, since the relation is labeled with some basic information used to log the steps of a computation (see Section 3). For now, the reader can safely ignore these labels (actually, labels will be omitted when irrelevant). The topics of this work are orthogonal to the evaluation of expressions, thus we refer the reader to [17] for the formalization of the rules of $\xrightarrow{l}$. Let us now briefly describe the interaction between the reduction of expressions and the rules of the logging semantics:

[4] Current implementations only guarantee this restriction within the same node.

$$(\mathit{Seq})\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{seq}} \Gamma; \langle p, \theta', e'\rangle \mid \Pi}$$

$$(\mathit{Send})\quad \frac{\theta, e \xrightarrow{\mathsf{send}(p',v)} \theta', e' \text{ and } \ell \text{ is a fresh symbol}}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{send}(\ell)} \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \theta', e'\rangle \mid \Pi}$$

$$(\mathit{Receive})\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta', e' \text{ and } \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{rec}(\ell)} \Gamma; \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid \Pi}$$

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

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

Fig. 3: Logging semantics

- A one-step reduction of an expression without side-effects is labeled with τ. In this case, rule *Seq* in Fig. 3 is applied to update correspondingly the environment and expression of the considered process.
- An expression $p' \,!\, v$ is reduced to v, with label $\mathsf{send}(p', v)$, so that rule *Send* in Fig. 3 can add the triple $(p, p', \{v, \ell\})$ to Γ (p is the process performing the send). The message is *tagged* with some fresh (unique) identifier ℓ. These tags allow us to track messages and avoid confusion when several messages have the same value (these tags are similar to the timestamps used in [21]).
- The remaining functions, receive, spawn and self, pose an additional problem: their value cannot be computed locally. Therefore, they are reduced to a fresh distinguished symbol κ, which is then replaced by the appropriate value in the system rules. In particular, a receive statement receive $\overline{cl_n}$ end is reduced to κ with label $\mathsf{rec}(\kappa, \overline{cl_n})$. Then, rule *Receive* in Fig. 3 nondeterministically checks if there exists a triple $(p', p, \{v, \ell\})$ in the global mailbox that matches some clause in $\overline{cl_n}$; pattern matching is performed by the auxiliary function matchrec. If the matching succeeds, it returns the pair (θ_i, e_i) with the matching substitution θ_i and the expression in the selected branch e_i. Finally, κ is bound to the expression e_i within the derived expression e'.
- For a spawn, an expression $\mathsf{spawn}(a/n, [\overline{v_n}])$ is also reduced to κ with label $\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])$. Rule *Spawn* in Fig. 3 then adds a new process with a fresh pid p' initialized with an empty environment id and the application $\mathsf{apply}\ a/n\ (v_1, \ldots, v_n)$. Here, κ is bound to p', the pid of the spawned process.
- Finally, the expression $\mathsf{self}()$ is reduced to κ with label $\mathsf{self}(\kappa)$ so that rule *Self* in Fig. 3 can bind κ to the pid of the given process.

We often refer to reduction steps derived by the system rules as *actions* taken by the chosen process.

Example 2. Let us consider the program of Example 1 and the initial system $\{\,\}; \langle \mathrm{c}, id, \mathsf{apply}\ \mathrm{main}/0\ ()\rangle$, where c is the pid of the process. A possible (faulty)

$$\begin{array}{rl}
 & \{\,\};\ \langle \mathrm{c}, \_, \underline{\mathsf{apply}\ \mathrm{main}/0\ ()}\rangle \\
\hookrightarrow & \{\,\};\ \langle \mathrm{c}, \_, \mathsf{let}\ S = \underline{\mathsf{spawn}(\mathrm{server}/0, [\,])}\ \mathsf{in}\ \ldots\rangle \\
\hookrightarrow & \{\,\};\ \langle \mathrm{c}, \_, \mathsf{let}\ P = \underline{\mathsf{spawn}(\mathrm{proxy}/0, [\,])}\ \mathsf{in}\ \mathsf{apply}\ \mathrm{client}/2\ (P, \mathrm{s})\rangle \mid \langle \mathrm{s}, \_, \mathsf{apply}\ \mathrm{server}/0\ ()\rangle \\
\hookrightarrow & \{\,\};\ \langle \mathrm{c}, \_, \underline{\mathsf{apply}\ \mathrm{client}/2\ (\mathrm{p}, \mathrm{s})}\rangle \mid \langle \mathrm{s}, \_, \mathsf{apply}\ \mathrm{server}/0\ ()\rangle \mid \langle \mathrm{p}, \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\hookrightarrow & \{\,\};\ \langle \mathrm{c}, \_, \mathsf{let}\ X = \mathrm{p}\,!\,\{\mathrm{s}, \{\underline{\mathsf{self}()}, 40\}\}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{apply}\ \mathrm{server}/0\ ()\rangle \mid \langle \mathrm{p}, \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\hookrightarrow & \{\,\};\ \langle \mathrm{c}, \_, \mathsf{let}\ X = \mathrm{p}\,!\,\{\mathrm{s}, \{\mathrm{c}, 40\}\}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \underline{\mathsf{apply}\ \mathrm{server}/0\ ()}\rangle \mid \langle \mathrm{p}, \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\hookrightarrow & \{\,\};\ \langle \mathrm{c}, \_, \mathsf{let}\ X = \mathrm{p}\,!\,\{\mathrm{s}, \{\mathrm{c}, 40\}\}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, \_, \underline{\mathsf{apply}\ \mathrm{proxy}/0\ ()}\rangle \\
\hookrightarrow & \{\,\};\ \langle \mathrm{c}, \_, \mathsf{let}\ X = \underline{\mathrm{p}\,!\,\{\mathrm{s}, \{\mathrm{c}, 40\}\}}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, \_, \mathsf{receive}\ \ldots\rangle \\
\hookrightarrow & \{(\mathrm{c}, \mathrm{p}, \{\{\mathrm{s}, \{\mathrm{c}, 40\}\}, \ell_1\})\};\ \langle \mathrm{c}, \_, \mathsf{let}\ Y = \underline{\mathrm{s}\,!\,2}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, \_, \mathsf{receive}\ \ldots\rangle \\
\hookrightarrow & \{(\mathrm{c}, \mathrm{p}, \{\{\mathrm{s}, \{\mathrm{c}, 40\}\}, \ell_1\}), (\mathrm{c}, \mathrm{s}, \{2, \ell_2\})\};\ \langle \mathrm{c}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, \_, \underline{\mathsf{receive}\ \ldots}\rangle \\
\hookrightarrow & \{(\mathrm{c}, \mathrm{s}, \{2, \ell_2\})\};\ \langle \mathrm{c}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, \_, \underline{\mathsf{let}\ W = \mathrm{s}\,!\,\{\mathrm{c}, 40\}\ \mathsf{in} \ldots}\rangle \\
\hookrightarrow & \{(\mathrm{c}, \mathrm{s}, \{2, \ell_2\}), (\mathrm{p}, \mathrm{s}, \{\{\mathrm{c}, 40\}, \ell_3\})\};\ \langle \mathrm{c}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \underline{\mathsf{receive}\ \ldots}\rangle \mid \langle \mathrm{p}, \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\hookrightarrow & \{(\mathrm{p}, \mathrm{s}, \{\{\mathrm{c}, 40\}, \ell_3\})\};\ \langle \mathrm{c}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathrm{error}\rangle \mid \langle \mathrm{p}, \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle
\end{array}$$

Fig. 4: Faulty derivation with the client/server of Example 1

computation from this system is shown in Fig. 4 (the selected expression at each step is underlined).[5] Here, we ignore the labels of the relation $\hookrightarrow$. Moreover, we skip the steps that just bind variables and we do not show the bindings of variables but substitute them for their values for clarity.

Independence. In order to define a causal-consistent replay semantics we need not only an interleaving semantics such as the one we just presented, but also a notion of causality or, equivalently, the opposite notion of independence. To this end, we use the labels of the logging semantics (see Figure 3). These labels include the pid p of the process that performs the transition, the rule used to derive it and, in some cases, some additional information: a message tag ℓ in rules *Send* and *Receive*, and the pid p' of the spawned process in rule *Spawn*.

Before formalizing the notion of independence, we need to introduce some notation and terminology. Given systems s_0, s_n, we call $s_0 \hookrightarrow^* s_n$, which is a shorthand for $s_0 \hookrightarrow_{p_1,r_1} \ldots \hookrightarrow_{p_n,r_n} s_n$, $n \geq 0$, a *derivation*. One-step derivations are simply called *transitions*. We use $d, d', d_1, \ldots$ to denote derivations and $t, t', t_1, \ldots$ for transitions. Given a derivation $d = (s_1 \hookrightarrow^* s_2)$, we define $\mathsf{init}(d) = s_1$.Two derivations, d_1 and d_2, are said *coinitial* if $\mathsf{init}(d_1) = \mathsf{init}(d_2)$.

For simplicity, in the following, we consider derivations up to renaming of bound variables. Under this assumption, the semantics is *almost* deterministic, i.e., the main sources of non-determinism are the selection of a process p and of the message to be retrieved by p in rule *Receive*. Choices of the fresh identifier ℓ for messages and of the pid p' of new processes are also non-deterministic.

Note that each process can perform at most one transition for each label, i.e., $s \hookrightarrow_{p,r} s_1$ and $s \hookrightarrow_{p,r} s_2$ trivially implies $s_1 = s_2$.

[5] Roughly speaking, the problem comes from the fact that the messages reach the server in the wrong order. Note that this faulty derivation is possible even by considering Erlang's policy on the order of messages, since they follow a different path.

We now instantiate to our setting the well-known *happened-before* relation [11], and the related notion of *independent* transitions:[6]

Definition 4 (happened-before, independence). *Given transitions* $t_1 = (s_1 \hookrightarrow_{p_1,r_1} s'_1)$ *and* $t_2 = (s_2 \hookrightarrow_{p_2,r_2} s'_2)$*, we say that* t_1 *happened before* t_2*, in symbols* $t_1 \rightsquigarrow t_2$*, if one of the following conditions holds:*

- *they consider the same process, i.e.,* $p_1 = p_2$*, and* t_1 *comes before* t_2*;*
- t_1 *spawns a process* p*, i.e.,* $r_1 = \mathsf{spawn}(p)$*, and* t_2 *is performed by process* p*, i.e.,* $p_2 = p$*;*
- t_1 *sends a message* ℓ*, i.e.,* $r_1 = \mathsf{send}(\ell)$*, and* t_2 *receives the same message* ℓ*, i.e.,* $r_2 = \mathsf{rec}(\ell)$*.*

Furthermore, if $t_1 \rightsquigarrow t_2$ *and* $t_2 \rightsquigarrow t_3$*, then* $t_1 \rightsquigarrow t_3$ *(transitivity). Two transitions* t_1 *and* t_2 *are* independent *if* $t_1 \not\rightsquigarrow t_2$ *and* $t_2 \not\rightsquigarrow t_1$*.*

Switching consecutive independent transitions does not change the final state:

Lemma 1 (switching lemma). *Let* $t_1 = (s_1 \hookrightarrow_{p_1,r_1} s_2)$ *and* $t_2 = (s_2 \hookrightarrow_{p_2,r_2} s_3)$ *be consecutive independent transitions. Then, there are two consecutive transitions* $t_{2\langle\langle t_1} = (s_1 \hookrightarrow_{p_2,r_2} s_4)$ *and* $t_{1\rangle\rangle t_2} = (s_4 \hookrightarrow_{p_1,r_1} s_3)$ *for some system* s_4*.*

The happened-before relation gives rise to an equivalence relation equating all derivations that only differ in the switch of independent transitions. Formally,

Definition 5 (causally equivalent derivations). *Let* d_1 *and* d_2 *be derivations under the logging semantics. We say that* d_1 *and* d_2 *are* causally equivalent*, in symbols* $d_1 \approx d_2$*, if* d_1 *can be obtained from* d_2 *by a finite number of switches of pairs of consecutive independent transitions.*

Causal equivalence is an instance of the *trace equivalence* in [20].

3 Logging Computations.

In this section, we introduce a notion of *log* for a computation. Basically, we aim to analyze in a debugger a faulty behavior that occurs in some execution of a program. To this end, we need to extract from an actual execution enough information to replay it inside the debugger. Actually, we do not want to replay necessarily the exact same execution, but a causally equivalent one. In this way, the programmer can focus on some actions of a particular process, and actions of other processes are only performed if needed (formally, if they happened-before these actions). As we will see in the next section, this ensures that the considered misbehaviors will still be replayed.

In a practical implementation (see the technical report [17]), one should instrument the program so that its execution in the actual environment produces

[6] Here, we use the term *independent*, instead of *concurrent* as in [11], since the latter has a slightly different meaning in the literature of causal-consistency.

a collection of sequences of logged events (one sequence per process). In the following, though, we exploit the logging semantics and, in particular, part of the information provided by the labels. The two approaches are equivalent, but the chosen one allows us to formally prove a number of properties in a simpler way.

One could argue (as in, e.g., [21]) that logs should only store information about the receive events, since this is the only nondeterministic action (once a process is selected). However, this is not enough in our setting, where:

– We need to log the sending of a message since this is where messages are tagged, and we need to know its (unique) identifier to be able to relate the sending and receiving of each message.
– We also need to log the spawn events, since the generated pids are needed to relate an action to the process that performed it (spawn events are not considered in [21] and, thus, their set of processes is fixed).

We note that other nondeterministic events, such as input from the user or from external services, should also be logged in order to correctly replay executions involving them. One can deal with them by instrumenting the corresponding primitives to log the input values, and then use these values when replaying the execution. Essentially, they can be dealt with as the receive primitive. Hence, we do not present them in detail to keep the presentation as simple as possible.

In the following, (ordered) sequences are denoted by $w = (r_1, r_2, \ldots, r_n)$, $n \geq 1$, where $()$ denotes the empty sequence. Concatenation is denoted by $+$. We write $r{+}w$ instead of $(r){+}w$ for simplicity.

Definition 6 (log). *A* log *is a (finite) sequence of events $(r_1, r_2, \ldots)$ where each r_i is either $\mathsf{spawn}(p)$, $\mathsf{send}(\ell)$ or $\mathsf{rec}(\ell)$, with p a pid and ℓ a message identifier. Logs are ranged over by ω. Given a derivation $d = (s_0 \hookrightarrow_{p_1,r_1} s_1 \hookrightarrow_{p_2,r_2} \ldots \hookrightarrow_{p_n,r_n} s_n)$, $n \geq 0$, under the logging semantics, the* log of a pid p in d, *in symbols $\mathcal{L}(d,p)$, is inductively defined as follows:*

$$\mathcal{L}(d,p) = \begin{cases} () & \text{if } n = 0 \text{ or } p \text{ does not occur in } d \\ r_1{+}\mathcal{L}(s_1 \hookrightarrow^* s_n, p) & \text{if } n > 0,\ p_1 = p, \text{ and } r_1 \not\in \{\mathsf{seq}, \mathsf{self}\} \\ \mathcal{L}(s_1 \hookrightarrow^* s_n, p) & \text{otherwise} \end{cases}$$

The log of d, *written $\mathcal{L}(d)$, is defined as: $\mathcal{L}(d) = \{(p, \mathcal{L}(d,p)) \mid p \text{ occurs in } d\}$. We sometimes call $\mathcal{L}(d)$ the* global *log of d to avoid confusion with $\mathcal{L}(d,p)$. Note that $\mathcal{L}(d,p) = \omega$ if $(p,\omega) \in \mathcal{L}(d)$ and $\mathcal{L}(d,p) = ()$ otherwise.*

Example 3. Consider the derivation shown in Example 2, here referred to as d. If we run it under the logging semantics, we get the following logs:

$$\begin{array}{ll} \mathcal{L}(d,\mathrm{c}) = (\mathsf{spawn}(\mathrm{s}), \mathsf{spawn}(\mathrm{p}), \mathsf{send}(\ell_1), \mathsf{send}(\ell_2)) & \\ \mathcal{L}(d,\mathrm{s}) = (\mathsf{rec}(\ell_2)) & \mathcal{L}(d,\mathrm{p}) = (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)) \end{array}$$

In the following we only consider finite derivations under the logging semantics. This is reasonable in our context where the programmer wants to analyze in the debugger a finite (possibly incomplete) execution showing a faulty behavior.

An essential property of our semantics is that causally equivalent derivations have the same log, i.e., the log depends only on the equivalence class, not on the selection of the representative inside the class. The reverse implication, namely that (coinitial) derivations with the same global log are causally equivalent, holds provided that we establish the following convention on when to stop a derivation:

Definition 7 (fully-logged derivation). *A derivation d is* fully-logged *if, for each process p, its last transition $s_1 \hookrightarrow_{p,r} s_2$ in d (if any) is a* logged *transition, i.e., $r \notin \{\mathsf{seq}, \mathsf{self}\}$. In particular, if a process performs no logged transition, then it performs no transition at all.*

Restricting to fully-logged derivations is needed since only logged transitions contribute to logs. Otherwise, two derivations d_1 and d_2 could produce the same log, but differ simply because, e.g., d_1 performs more non-logged transitions than d_2. Restricting to fully-logged derivations, we include the minimal amount of transitions needed to produce the observed log.

Finally, we present a key result of our logging semantics. It states that two derivations are causally equivalent iff they produce the same log.

Theorem 1. *Let d_1, d_2 be coinitial fully-logged derivations. $\mathcal{L}(d_1) = \mathcal{L}(d_2)$ iff $d_1 \approx d_2$.*

4 A Causal-Consistent Replay Semantics

In this section, we introduce an *uncontrolled* replay semantics. It takes a program and the log of a given derivation, and allows us to replay any causally equivalent derivation. This semantics constitutes the kernel of our replay framework. The term uncontrolled indicates that the semantics specifies how to perform replay, but there is no policy to select the applicable rule when more than one is enabled. The uncontrolled semantics is suitable to set the basis of our replay mechanism, but does not allow one to focus on the causes of a given action. For this reason, in Section 5, we build on top of this semantics a *controlled* one, where the selection of actions is driven by the queries from the user.

In the following, we introduce a transition relation $\rightharpoonup$ to specify replay. Transition $\rightharpoonup$ is similar to the logging semantics $\hookrightarrow$ (Figure 3) but it is now driven by the considered log. Thus, processes have the form $\langle p, \omega, \theta, e\rangle$, with ω a log.

The uncontrolled causal-consistent replay semantics is shown in Figure 5. For technical reasons, labels of the replay semantics contain the same information as the labels of the logging semantics. Moreover, the labels now also include a set of replay *requests*. The reader can ignore these elements until the next section. For simplicity, we also consider that the log $\mathcal{L}(d, p)$ of each process p in the original derivation d is a fixed global parameter of the transition rules (see rule *Spawn*).

The rules for expressions are the same as in the logging semantics (an advantage of the modular design). The replay semantics is similar to the logging semantics, except that logs fix some parameters: the fresh message identifier in rule *Send*, the message received in rule *Receive*, and the fresh pid in rule *Spawn*.

$$(\mathit{Seq})\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, \omega, \theta, e\rangle \mid \Pi \rightharpoonup_{p,\mathsf{seq},\{\mathsf{s}\}} \Gamma; \langle p, \omega, \theta', e'\rangle \mid \Pi}$$

$$(\mathit{Send})\quad \frac{\theta, e \xrightarrow{\mathsf{send}(p',v)} \theta', e'}{\Gamma; \langle p, \mathsf{send}(\ell){+}\omega, \theta, e\rangle \mid \Pi \rightharpoonup_{p,\mathsf{send}(\ell),\{\mathsf{s},\ell\Uparrow\}} \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \omega, \theta', e'\rangle \mid \Pi}$$

$$(\mathit{Receive})\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta', e' \;\text{ and }\; \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\begin{array}{l}\Gamma \cup \{(p', p, \{v, \ell\})\}\langle p, \mathsf{rec}(\ell){+}\omega, \theta, e\rangle \mid \Pi \\ \qquad\qquad \rightharpoonup_{p,\mathsf{rec}(\ell),\{\mathsf{s},\ell\Downarrow\}} \Gamma; \langle p, \omega, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid \Pi\end{array}}$$

$$(\mathit{Spawn})\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa,a/n,[\overline{v_n}])} \theta', e' \quad\text{ and }\quad \omega' = \mathcal{L}(d, p')}{\begin{array}{r}\Gamma; \langle p, \mathsf{spawn}(p'){+}\omega, \theta, e\rangle \mid \Pi \rightharpoonup_{p,\mathsf{spawn}(p'),\{\mathsf{s},\mathsf{sp}_{p'}\}} \Gamma; \langle p, \omega, \theta', e'\{\kappa \mapsto p'\}\rangle \\ \mid \langle p', \omega', id, \mathsf{apply}\ a/n\ (\overline{v_n})\rangle \mid \Pi\end{array}}$$

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

Fig. 5: Uncontrolled replay semantics

Example 4. Consider the logs of Example 3. Then, we have, e.g., the replay derivation in Fig. 6. The actions performed by each process are the same as in the original derivation in Example 2, but the interleavings are slightly different. Moreover, after ten steps, the server is waiting for a message, the global mailbox contains a matching message but, in contrast to the logging semantics, receive cannot proceed since the message identifier in the log does not match (ℓ_2 vs ℓ_3).

Basic Properties of the Replay Semantics. Here, we show that the uncontrolled replay semantics is consistent and we relate it with the logging semantics. We need the following auxiliary functions:

Definition 8. *Let $d = (s_1 \hookrightarrow^* s_2)$ be a derivation under the logging semantics, with $s_1 = \Gamma; \langle p_1, \theta_1, e_1\rangle \mid \ldots \mid \langle p_n, \theta_n, e_n\rangle$. The system corresponding to s_1 in the replay semantics is defined as follows:*

$$addLog(\mathcal{L}(d), s_1) = \Gamma; \langle p_1, \mathcal{L}(d, p_1), \theta_1, e_1\rangle \mid \ldots \mid \langle p_n, \mathcal{L}(d, p_n), \theta_n, e_n\rangle$$

Conversely, given a system $s = \Gamma; \langle p_1, \omega_1, \theta_1, e_1\rangle \mid \ldots \mid \langle p_n, \omega_n, \theta_n, e_n\rangle$ in the replay semantics, we let $del(s)$ be the system obtained from s by removing logs, i.e., $del(s) = \Gamma; \langle p_1, \theta_1, e_1\rangle \mid \ldots \mid \langle p_n, \theta_n, e_n\rangle$, and similarly for derivations.

In the following, we extend the notions of log and coinitial derivations, as well as function init, to replay derivations in the obvious way. Furthermore, we now call a system s' *initial* under the replay semantics if there exists a derivation d under the logging semantics, and $s' = addLog(\mathcal{L}(d), \mathsf{init}(d))$.

We extend the notion of fully-logged derivations to our replay semantics:

Definition 9 (fully-logged replay derivation). *A derivation d under the replay semantics is* fully-logged *if, for each process p, the log is empty and its last transition (if any) is a logged transition.*

$$\begin{array}{ll}
& \{\,\};\ \langle \mathrm{c}, (\mathsf{spawn}(\mathrm{s}), \mathsf{spawn}(\mathrm{p}), \mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), _, \underline{\mathsf{apply}\ \mathrm{main}/0\ ()}\rangle \\
\rightharpoonup & \{\,\};\ \langle \mathrm{c}, (\mathsf{spawn}(\mathrm{s}), \mathsf{spawn}(\mathrm{p}), \mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), _, \mathsf{let}\ S = \underline{\mathsf{spawn}(\mathrm{server}/0, [\,])}\ \mathsf{in}\ \ldots\rangle \\
\rightharpoonup & \{\,\};\ \langle \mathrm{c}, (\mathsf{spawn}(\mathrm{p}), \mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), _, \mathsf{let}\ P = \mathsf{spawn}(\mathrm{proxy}/0, [\,])\ \mathsf{in} \\
& \quad \mathsf{apply}\ \mathrm{client}/2\ (P, \mathrm{s})\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), _, \underline{\mathsf{apply}\ \mathrm{server}/0\ ()}\rangle \\
\rightharpoonup & \{\,\};\ \langle \mathrm{c}, (\mathsf{spawn}(\mathrm{p}), \mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), _, \mathsf{let}\ P = \underline{\mathsf{spawn}(\mathrm{proxy}/0, [\,])}\ \mathsf{in} \\
& \quad \mathsf{apply}\ \mathrm{client}/2\ (P, \mathrm{s})\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), _, \mathsf{receive}\ \ldots\rangle \\
\rightharpoonup & \{\,\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), _, \underline{\mathsf{apply}\ \mathrm{client}/2\ (\mathrm{p}, \mathrm{s})}\rangle \\
& \quad \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), _, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)), _, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\rightharpoonup & \{\,\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), _, \mathsf{let}\ X = \mathrm{p}\,!\,\{\mathrm{s}, \{\underline{\mathsf{self}()}, 40\}\}\ \mathsf{in}\ \ldots\rangle \\
& \quad \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), _, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)), _, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\rightharpoonup & \{\,\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), _, \mathsf{let}\ X = \underline{\mathrm{p}\,!\,\{\mathrm{s}, \{\mathrm{c}, 40\}\}}\ \mathsf{in}\ \ldots\rangle \\
& \quad \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), _, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)), _, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\rightharpoonup & \{(\mathrm{c}, \mathrm{p}, \{\{\mathrm{s}, \{\mathrm{c}, 40\}\}, \ell_1\})\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_2)), _, \mathsf{let}\ Y = \mathrm{s}\,!\,2\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), _, \mathsf{receive}\ \ldots\rangle \\
& \quad \mid \langle \mathrm{p}, (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)), _, \underline{\mathsf{apply}\ \mathrm{proxy}/0\ ()}\rangle \\
\rightharpoonup & \{(\mathrm{c}, \mathrm{p}, \{\{\mathrm{s}, \{\mathrm{c}, 40\}\}, \ell_1\})\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_2)), _, \mathsf{let}\ Y = \mathrm{s}\,!\,2\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), _, \mathsf{receive}\ \ldots\rangle \\
& \quad \mid \langle \mathrm{p}, (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)), _, \underline{\mathsf{receive}\ \ldots}\rangle \\
\rightharpoonup & \{\,\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_2)), _, \mathsf{let}\ Y = \mathrm{s}\,!\,2\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), _, \mathsf{receive}\ \ldots\rangle \\
& \quad \mid \langle \mathrm{p}, (\mathsf{send}(\ell_3)), _, \mathsf{let}\ \underline{\mathrm{s}\,!\,\{\mathrm{c}, 40\}}\ \mathsf{in} \ldots\rangle \\
\rightharpoonup & \{(\mathrm{p}, \mathrm{s}, \{\{\mathrm{c}, 40\}, \ell_3\})\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_2)), _, \mathsf{let}\ Y = \underline{\mathrm{s}\,!\,2}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), _, \mathsf{receive}\ \ldots\rangle \\
& \quad \mid \langle \mathrm{p}, (), _, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\rightharpoonup & \{(\mathrm{p}, \mathrm{s}, \{\{\mathrm{c}, 40\}, \ell_3\}), (\mathrm{c}, \mathrm{s}, \{2, \ell_2\})\};\ \langle \mathrm{c}, (), _, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), _, \underline{\mathsf{receive}\ \ldots}\rangle \\
& \quad \mid \langle \mathrm{p}, (), _, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\rightharpoonup & \{(\mathrm{p}, \mathrm{s}, \{\{\mathrm{c}, 40\}, \ell_3\})\};\ \langle \mathrm{c}, (), _, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, (), _, \mathrm{error}\rangle \mid \langle \mathrm{p}, (), _, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle
\end{array}$$

Fig. 6: Uncontrolled replay derivation with the traces of Example 3

Note that, in addition to Definition 7, we now require that processes *consume* all their logs.

We will only consider systems reachable from the execution of a program:

Definition 10 (reachable systems). *A system s is* reachable *if there exists an initial system s_0 such that $s_0 \rightharpoonup^* s$.*

Since only reachable systems are of interest (non-reachable systems are ill-formed), in the following we assume that all systems are reachable.

Now, we can tackle the problem of proving that our replay semantics preserves causal equivalence, i.e., that the original and the replay derivations are always causally equivalent.

Theorem 2. *Let d be a fully-logged derivation under the logging semantics. Let d' be any finite fully-logged derivation under the replay semantics such that* $\mathsf{init}(d') = \mathit{addLog}(\mathcal{L}(d), \mathsf{init}(d))$. *Then $d \approx \mathit{del}(d')$.*

Usefulness for Debugging. Now, we show that our replay semantics is indeed useful as a basis for designing a debugging tool. In particular, we prove that a (faulty) behavior occurs in the logged derivation iff any replay derivation also exhibits the same *faulty* behavior, hence replay is correct and complete.

In order to formalize such a result we need to fix the notion of faulty behavior we are interested in. For us, a misbehavior is a wrong system, but since the system is possibly distributed, we concentrate on misbehaviors visible from a "local" observer. Given that our systems are composed of processes and messages in the global mailbox, we consider that a (local) misbehavior is either a wrong message in the global mailbox or a process with a wrong configuration.

Theorem 3 (Correctness and completeness). *Let d be a fully-logged derivation under the logging semantics. Let d' be any fully-logged derivation under the uncontrolled replay semantics such that* $\mathsf{init}(d') = \mathit{addLog}(\mathcal{L}(d), \mathsf{init}(d))$*. Then:*

1. *there is a system $\Gamma; \Pi$ in d with a configuration $\langle p, \theta, e\rangle$ in Π iff there is a system $\Gamma'; \Pi'$ in d' with a configuration $\langle p, \theta, e\rangle$ in $\mathit{del}(\Gamma'; \Pi')$;*
2. *there is a system $\Gamma; \Pi$ in d with a message $(p, p', \{v, \ell\})$ in Γ iff there is a system $\Gamma'; \Pi'$ in d' with a message $(p, p', \{v, \ell\})$ in Γ'.*

The result above is very strong: it ensures that a misbehavior occurring in a logged execution is replayed in *any* possible fully-logged derivation. This means that any scheduling policy is fine for replay. Furthermore, this remains true whatever actions the user takes: either the misbehavior is reached, or it remains in any possible forward computation.

One may wonder whether more general notions of misbehavior make sense. Above, we consider just "local" observations. One could ask for more than one local observation to be replayed. By applying the result above to multiple observations we get that all of them will be replayed, but, if they concern different processes or messages, we cannot ensure that they are replayed *at the same time or in the same order.* For instance, in the derivation of Figure 4, process c sends the message with identifier ℓ_2 before process p receives the message with identifier ℓ_1, while in the replay derivation of Figure 6 the two actions are executed in the opposite order. Only a *super user* able to see the whole system at once could see such a (mis)behavior, which are thus not relevant in our context.

5 Controlled Replay Semantics

In this section, we introduce a controlled version of the replay semantics. The semantics in the previous section allows one to replay a given derivation and be guaranteed to replay, sooner or later, any local misbehavior. In practice, though, one normally knows in which process p the misbehavior appears, and thus (s)he wants to focus on a process p or even on some of its actions. However, to correctly replay these actions, one also needs to replay the actions that happened before them. We present in Figure 7 a semantics where the user can specify which actions (s)he wants to replay, and the semantics takes care of replaying them. Replaying an action requires to replay all *and only* its causes. Notably, the bug causing a misbehavior causes the action showing the misbehavior.

Here, given a system s, we want to start a replay until a particular action ψ is performed on a given process p. We denote such a replay request with $\llbracket s \rrbracket_{(\{p,\psi\})}$.

$$\frac{\Gamma;\Pi \rightharpoonup_{p,r,\Psi'} \Gamma';\Pi' \quad\wedge\quad \psi \in \Psi'}{\lfloor\!\lfloor \Gamma;\Pi \rfloor\!\rfloor_{\{p,\psi\}+\Psi} \rightsquigarrow \lfloor\!\lfloor \Gamma';\Pi' \rfloor\!\rfloor_{\Psi}} \qquad \frac{\Gamma;\Pi \rightharpoonup_{p,r,\Psi'} \Gamma';\Pi' \quad\wedge\quad \psi \not\in \Psi'}{\lfloor\!\lfloor \Gamma;\Pi \rfloor\!\rfloor_{\{p,\psi\}+\Psi} \rightsquigarrow \lfloor\!\lfloor \Gamma';\Pi' \rfloor\!\rfloor_{\{p,\psi\}+\Psi}}$$

$$\frac{\Gamma;\langle p, \mathsf{rec}(\ell){+}\omega, \theta, e\rangle \mid \Pi \not\rightharpoonup_{p,r,\Psi'} \quad\wedge\quad \mathit{sender}(\ell) = p'}{\lfloor\!\lfloor \Gamma;\langle p, \mathsf{rec}(\ell){+}\omega, \theta, e\rangle \mid \Pi \rfloor\!\rfloor_{\{p,\psi\}+\Psi} \rightsquigarrow \lfloor\!\lfloor \Gamma;\langle p, \mathsf{rec}(\ell){+}\omega, \theta, e\rangle \mid \Pi \rfloor\!\rfloor_{(\{p',\ell^{\Uparrow}\},\{p,\psi\})+\Psi}}$$

$$\frac{\nexists p \text{ in } \Pi \quad\wedge\quad \mathit{parent}(p) = p'}{\lfloor\!\lfloor \Gamma;\Pi \rfloor\!\rfloor_{\{p,\psi\}+\Psi} \rightsquigarrow \lfloor\!\lfloor \Gamma;\Pi \rfloor\!\rfloor_{(\{p',\mathsf{sp}_p\},\{p,\psi\})+\Psi}}$$

Fig. 7: Controlled replay semantics

In general, the subscript of $\lfloor\!\lfloor\ \rfloor\!\rfloor$ is a stack of requests, where the first element is the most recent one. In this paper, we consider the following replay requests:

- $\{p, \mathsf{s}\}$: one step of process p (the extension to n steps is straightforward);
- $\{p, \ell^{\Uparrow}\}$: request for process p to send the message tagged with ℓ;
- $\{p, \ell^{\Downarrow}\}$: request for process p to receive the message tagged with ℓ;
- $\{p, \mathsf{sp}_{p'}\}$: request for process p to spawn the process p'.

Variable creations as not valid targets for replay requests, since variable names are not known before their creation (variable creations are not logged). The requests above are *satisfied* when a corresponding uncontrolled transition is performed. Indeed, the third element labeling the relations of the replay semantics in Figure 5 is the set of requests satisfied in the corresponding step.

Let us explain the rules of the controlled replay semantics in Fig. 7. Here, we assume that the computation always starts with a single request.

- If the desired process p can perform a step satisfying the request ψ on top of the stack, we do it and remove the request from the stack (first rule).
- If the desired process p can perform a step, but it does not satisfy the request ψ, we update the system but keep the request in the stack (second rule).
- If a step on the desired process p is not possible, then we track the dependencies and add a new request on top of the stack. We have two rules: one for adding a request to a process to send a message we want to receive and another one to spawn the process we want to replay if it does not exist. Here, we use the auxiliary functions *sender* and *parent* to identify, respectively, the sender of a message and the parent of a process. Both functions *sender* and *parent* are easily computable from the logs in $\mathcal{L}(d)$.

The relation $\rightsquigarrow$ can be seen as a controlled version of the uncontrolled replay semantics in the sense that each derivation of the controlled semantics corresponds to a derivation of the uncontrolled one, while the opposite is not generally true. Notions for derivations and transitions are easily extended to controlled derivations. We also need a notion of projection from controlled systems to uncontrolled systems: $\mathit{uctrl}(\lfloor\!\lfloor \Gamma;\Pi \rfloor\!\rfloor_{\Psi}) = \Gamma;\Pi$. The notion of projection trivially extends to derivations.

Theorem 4 (Soundness). *For each controlled derivation d, $\mathit{uctrl}(d)$ is an uncontrolled derivation.*

While simple, this result allows one to recover many relevant properties from the uncontrolled semantics. For instance, by using the controlled semantics, if starting from a system $s = \mathit{addLog}(\mathcal{L}(d), \mathsf{init}(d))$ for some logging derivation d we find a wrong message $(p, p', \{v, \ell\})$, then we know that the same message exists also in d (from Theorem 3).

Our controlled semantics is not only sound but also minimal: causal-consistent replay redoes the minimal amount of actions needed to satisfy the replay request.

Here, we need to restrict the attention to requests that ask to replay transitions which are in the future of the process.

Definition 11. *A controlled system $c = \lfloor\!\lfloor s \rfloor\!\rfloor_{(\{p,\psi\})}$ is well initialized iff there are a derivation d under the logging semantics, a system $s_0 = \mathit{addLog}(\mathcal{L}(d), \mathsf{init}(d))$, an uncontrolled derivation $s_0 \rightharpoonup^* s$, and an uncontrolled derivation from s satisfying $\{p, \psi\}$.*

The existence of a derivation satisfying the request can be efficiently checked. For replay requests $\{p, \mathsf{s}\}$ it is enough to check that process p can perform a step, for other replay requests it is enough to check the process log.

Theorem 5 (Minimality). *Let d be a controlled derivation such as $\mathsf{init}(d) = \lfloor\!\lfloor s \rfloor\!\rfloor_{(\{p,\psi\})}$ is well-initialized. Derivation $\mathit{uctrl}(d)$ has minimal length among all uncontrolled derivations d' with $\mathsf{init}(d') = s$ including at least one transition satisfying the request $\{p, \psi\}$.*

6 Related Work and Conclusion

In this work, we have introduced (controlled) causal-consistent replay. It is strongly related (indeed dual) to the notion of causal-consistent reversibility, and its instance on debugging, causal-consistent reversible debugging, introduced in [6] for the *toy* language μOz. Beyond this, it has only been used so far in the CauDEr [13,14] debugger for Erlang, which we took as a starting point for our prototype implementation (see [17]). Causal-consistent rollback has also been studied in the context of the process calculus HOπ [12] and the coordination language Klaim [7]. We refer to [6] for a description of the relations between causal-consistent debugging and other forms of reversible debugging.

The basic ideas in this paper are also applicable to other message-passing languages and calculi. In principle, the approach could also be applied to shared memory languages, yet it would require to log all interactions with shared memory (which may give rise, in principle, to an inefficient scheme).

An approach to record and replay for actor languages is introduced in [1]. While we concentrate on the theory, they focus on low-level issues: dealing with I/O, producing compact logs, etc. Actually, we could consider some of the ideas in [1] to produce more compact logs and thus reduce our instrumentation overhead.

At the semantic level, the work closest to ours is the reversible semantics for Erlang in [15]. However, all our semantics abstract away local queues in processes and their management. This makes the notion of independence much more natural, and it avoids some spurious conflicts between deliveries of different messages present in [15]. Moreover, our replay semantics is driven by the log of an actual execution, while the one in [15] is not. Finally, our controlled semantics, built on top of the uncontrolled reversible semantics, is much simpler than the low-level controlled semantics in [15] which, anyway, is based on undoing the actions of an execution up to a given checkpoint (rollback requests appeared later, in [13]).

None of the works above treats causal-consistent replay and, as far as we know, such notion has never been explored. For instance, no reference to it appears in a recent survey [4]. The survey classifies our approach as a message-passing multi-processor scheme (the approach is studied in a single-processor multi-process setting, but it makes no use of the single-processor assumption). It is in between content-based schemes (that record the content of the messages) and ordering-based schemes (that record the source of the messages), since it registers just unique identifiers for messages. This reduces the size of the log (content of long messages is not stored) w.r.t. content-based schemes, yet differently from ordering-based schemes it does not necessarily require to replay the system from a global checkpoint (but we do not yet consider checkpoints).

A related ordering-based scheme is [21]: it uses race detection to avoid logging all message exchanges, and we may try to integrate it in our approach in the future (though it considers only systems with a fixed number of processes). A content-based work is [19] for MPI programs, which does not replay calls to MPI functions, but just takes the values from the log. By applying this approach in our case, the state of Γ would not be replayed, and causal-consistent replay would not be possible since no relation between send and receive is kept.

Our work is also related to slicing, and in particular to [23], since it also deals with concurrent systems. Both approaches are based on causal consistency, but slicing considers the whole computation and extracts the fragment of it needed to explain a visible behavior, while we instrument the computation so to be able to go back and forward. Other differences include the considered languages—pi calculus vs Erlang—, the style of the semantics—labelled transitions vs reductions—, etc.

References

1. Aumayr, D., Marr, S., Béra, C., Boix, E.G., Mössenböck, H.: Efficient and deterministic record & replay for actor languages. In: Tilevich, E., Mössenböck, H. (eds.) Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang 2018). pp. 15:1–15:14. ACM (2018)
2. Britton, T., Jeng, L., Carver, G., Cheak, P., Katzenellenbogen, T.: Reversible debugging software – quantify the time and cost saved using reversible debuggers. `http://www.roguewave.com` (2012)
3. Carlsson, R., Gustavsson, B., Johansson, E., Lindgren, T., Nyström, S.O., Pettersson, M., Virding, R.: Core Erlang 1.0.3. Language specification (2004), available from URL: `https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf`

4. Chen, Y., Zhang, S., Guo, Q., Li, L., Wu, R., Chen, T.: Deterministic replay: A survey. ACM Comput. Surv. 48(2), 17:1–17:47 (2015)
5. Frequently Asked Questions about Erlang. Available at `http://erlang.org/faq/academic.html` (2018)
6. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: Gnesi, S., Rensink, A. (eds.) Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE 2014). Lecture Notes in Computer Science, vol. 8411, pp. 370–384. Springer (2014)
7. 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)
8. Huang, J., Liu, P., Zhang, C.: LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010. pp. 385–386. ACM (2010)
9. Huang, J., Zhang, C.: Debugging concurrent software: Advances and challenges. J. Comput. Sci. Technol. 31(5), 861–868 (2016)
10. Jiang, Y., Gu, T., Xu, C., Ma, X., Lu, J.: CARE: cache guided deterministic replay for concurrent java programs. In: 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014. pp. 457–467. ACM (2014)
11. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
12. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.: Controlling reversibility in higher-order pi. In: Katoen, J., König, B. (eds.) Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR 2011). Lecture Notes in Computer Science, vol. 6901, pp. 297–311. Springer (2011)
13. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: A Causal-Consistent Reversible Debugger for Erlang (system description). In: Gallagher, J.P., Sulzmann, M. (eds.) Proceedings of the 14th International Symposium on Functional and Logic Programming (FLOPS'18). Lecture Notes in Computer Science, vol. 10818, pp. 247–263. Springer (2018)
14. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr website. URL: `https://github.com/mistupv/cauder` (2018)
15. 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)
16. Lanese, I., Palacios, A., Vidal, G.: CauDEr, Causal-consistent Reversible Replay Debugger. Logger: `https://github.com/mistupv/tracer`, debugger: `https://github.com/mistupv/cauder/tree/replay`
17. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay debugging for message passing programs. Tech. rep., DSIC, Universitat Politècnica de València (2019), `http://personales.upv.es/gvidal/german/forte19tr/paper.pdf`
18. LeBlanc, T.J., Mellor-Crummey, J.M.: Debugging parallel programs with instant replay. IEEE Trans. Computers 36(4), 471–482 (1987)
19. Maruyama, M., Tsumura, T., Nakashima, H.: Parallel program debugging based on data-replay. In: Zheng, S.Q. (ed.) Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005). pp. 151–156. IASTED/ACTA Press (2005)
20. Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, 1986. Lecture Notes in Computer Science, vol. 255, pp. 279–324. Springer (1987)

21. Netzer, R.H., Miller, B.P.: Optimal tracing and replay for debugging message-passing parallel programs. The Journal of Supercomputing 8(4), 371–388 (1995)
22. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: Hermenegildo, M., López-García, P. (eds.) Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016). Lecture Notes in Computer Science, vol. 10184, pp. 259–274. Springer (2017)
23. Perera, R., Garg, D., Cheney, J.: Causally consistent dynamic slicing. In: Desharnais, J., Jagadeesan, R. (eds.) CONCUR. LIPIcs, vol. 59, pp. 18:1–18:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
24. Software, U.: Increasing software development productivity with reversible debugging (2014), `https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf`
25. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobbs Journal 30(3) (2005)
26. 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)
27. Veeraraghavan, K., Lee, D., Wester, B., Ouyang, J., Chen, P.M., Flinn, J., Narayanasamy, S.: Doubleplay: Parallelizing sequential logging and replay. ACM Trans. Comput. Syst. 30(1), 3:1–3:24 (2012)