Fundamenta Informaticae XXI (2001) 1001–1038
DOI 10.3233/FI-2016-0000
IOS Press

Causal-Consistent Replay Reversible Semantics for Message Passing Concurrent Programs*

Ivan Lanese
Focus Team, University of Bologna/INRIA
Mura Anteo Zamboni, 7, Bologna, Italy
ivan.lanese@unibo.it

Adrián Palacios
MiST, VRAIN, Universitat Politècnica de València
Camino de Vera, S/N, 46022 Valencia, Spain
apalacios@dsic.upv.es

Germán Vidal
MiST, VRAIN, Universitat Politècnica de València
Camino de Vera, S/N, 46022 Valencia, Spain
gvidal@dsic.upv.es

Abstract. Causal-consistent reversible debugging is an innovative technique for debugging concurrent systems. It allows one to go back in the execution focusing on the actions that most likely caused a visible misbehavior. When such an action is selected, the debugger undoes it, including all *and only* its consequences. This operation is called a causal-consistent rollback. In this way,

Address for correspondence: Germán Vidal, DSIC, Universitat Politècnica de València, Camino de Vera S/N, 46022 Valencia, Spain. Email: gvidal@dsic.upv.es.

*This work has been partially supported by the COST Action IC1405 on Reversible Computation - extending horizons of computing. The first author has also been partially supported by INdAM – GNCS 2020 project *Sistemi Reversibili Concorrenti: dai Modelli ai Linguaggi*. The first and third authors have been also partially supported by French ANR project DCore ANR-18-CE25-0007. The second and third authors have been also partially supported by the EU (FEDER) and the Spanish MCI/AEI under grants TIN2016-76843-C4-1-R and PID2019-104735RB-C41, and by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust).

the user can avoid being distracted by the actions of other, unrelated processes. In this work, we introduce its dual notion: causal-consistent *replay*. We allow the user to record an execution of a running program and, in contrast to traditional replay debuggers, to reproduce a visible misbehavior inside the debugger including all *and only* its causes. Furthermore, we present a unified framework that combines both causal-consistent replay and causal-consistent rollback. Although most of the ideas that we present are rather general, we focus on a popular functional and concurrent programming language based on message passing: Erlang.

Keywords: concurrency, logging, causal-consistent, debugging, reversible computing

1. Introduction

Debugging is a main activity in software development. According to a 2014 study [1], the cost of debugging faulty software amounts to $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 to current computing platforms, such as the Internet or the Cloud, and concurrency is a must to overcome the advent of the power wall [3]. Debugging concurrent and distributed software is clearly more difficult than debugging sequential code [4]. Furthermore, misbehaviors may depend, e.g., on the execution speed of the different processes, showing up only in some (sometimes rare) cases.

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. Moreover, even when the misbehavior shows up, it is still difficult to locate the bug causing it: the bug might be in a process different from the one showing a faulty behavior, so one needs to trace back the execution from the visible misbehavior to the bug, jumping from process to process following causal links (determined for instance by message exchanges). This last problem has been tackled by so called *causal-consistent reversible debugging* [5], where one has the possibility of selecting a direct cause of a visible misbehavior and undo it, including all *and only* its consequences. This operation is called a *causal-consistent rollback* [6, 7]. In most cases, iterating this operation leads to the bug.

Existing techniques and tools for causal-consistent reversible debugging do not provide a way to record the execution of a program in its actual environment, thus there are no guarantees that the faulty execution will be replayed in the debugger (i.e., it is up to the programmer to follow the right forward steps, a challenging task for concurrent programs). A first contribution of this paper is a theoretical framework for such a record and replay, which of course guarantees that misbehaviors are reproduced during replay. Our approach to replay, which we call *(controlled) causal-consistent replay*, 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 record and replay debuggers do), 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 only 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 causal-consistent replay is new.

Causal-consistent replay can be considered the dual of causal-consistent rollback. A second contribution of the paper is the integration of the two techniques, which provides a complete setting to explore a concurrent computation back and forth, always concentrating on the actions of interest and following causality links. Notably, replay works well with rollback: it can be used not only to bootstrap debugging activities by replaying the misbehavior as in standard debugging, but also during the debugging process. Indeed, it is not unusual to go "too far" backward (i.e., beyond the bug) in the execution, and replay allows the programmer to go forward again with the guarantee to replay the same execution (or a causally equivalent one) up to the given misbehavior. This was not possible in the standard framework of causal-consistent reversible debugging.

We develop the results of the present paper in the context of a (first-order) functional and concurrent language based on message passing. Therefore, our results can be directly applied to a language like Erlang [8]. We chose such a language since Erlang is used in high-visibility projects (such as some versions of the Facebook chat [9]), and it is particularly well-suited to program concurrent and distributed applications. In particular, it provides native support for concurrency without the need to rely on external libraries. Nevertheless, the theory of causal-consistent replay can be adapted to any concurrent or distributed language based on message passing. Indeed, causal-consistent reversibility has been studied for different foundational calculi as well as for the languages μOz [10], μKlaim [11] and Erlang [12], and the dual notion of causal-consistent replay is equally generic.

In this paper, we also introduce a novel semantics for an Erlang-like language which is more abstract than the ones in the literature [13, 14, 12], yet concrete enough to show misbehaviors and to look for the bugs causing them. In particular, it allows us to significantly improve the notion of concurrency as well as the formalization of reversibility and rollback for this language. Since these improvements are quite technical, we will contrast them with the approaches in the literature after having described them (see Sections 2.3 and 4.3).

The structure of this paper is as follows. First, we introduce the syntax and semantics of our language (Sect. 2), which corresponds to the functional and concurrent subset of Core Erlang [15]. We then present the generation of a log associated to a computation that can be used to drive the causal-consistent replay of this computation (Sect. 3). Both replay and rollback computations are formalized with a nondeterministic (uncontrolled) semantics (Sect. 4). We prove a number of properties for our semantics, e.g., that the replay of an execution contains the same (mis)behaviors as the original one (Theorem 4.22). We also prove the so called causal consistency of the reversible semantics (Theorem 4.17), a key result in the field of causal-consistent reversibility. We then present a *controlled* version of the semantics that is driven by a particular replay or rollback request (Sect. 5). Here, we also prove the soundness (Theorem 5.1) and minimality (Theorem 5.6) of this controlled semantics. Finally, we discuss some related work (Sect. 6) and conclude (Sect. 7).

This paper is a revised and extended version of [16], where causal-consistent replay has been introduced. However, [16] concentrates on replay and mentions reversibility only briefly, while the present paper integrates the notion of causal-consistent replay with the reversible semantics in [12]. Also, the present paper includes full proofs of the results. Additional material which is not central for the discussion can be found in a companion technical report [17].

$$
\begin{array}{rcl}
program & ::= & fun_1 \ \ldots \ fun_n \\
fun & ::= & fname = \mathsf{fun}\ (X_1, \ldots, X_n) \to expr \\
fname & ::= & Atom/Integer \\
lit & ::= & Atom \mid Integer \mid Float \mid [\,] \\
expr & ::= & Var \mid lit \mid fname \mid [expr_1|expr_2] \mid \{expr_1, \ldots, expr_n\} \\
 & \mid & \mathsf{call}\ expr\ (expr_1, \ldots, expr_n) \mid \mathsf{apply}\ expr\ (expr_1, \ldots, expr_n) \\
 & \mid & \mathsf{case}\ expr\ \mathsf{of}\ clause_1; \ldots; clause_m\ \mathsf{end} \\
 & \mid & \mathsf{let}\ Var = expr_1\ \mathsf{in}\ expr_2 \mid \mathsf{receive}\ clause_1; \ldots; clause_n\ \mathsf{end} \\
 & \mid & \mathsf{spawn}(expr, [expr_1, \ldots, expr_n]) \mid expr_1\,!\,expr_2 \mid \mathsf{self}() \\
clause & ::= & pat\ \mathsf{when}\ expr_1 \to expr_2 \\
pat & ::= & Var \mid lit \mid [pat_1|pat_2] \mid \{pat_1, \ldots, pat_n\}
\end{array}
$$

Figure 1. Language syntax rules

2. The Language

In this section, we present the considered language: a first-order functional and concurrent programming language based on message passing that mainly follows the actor model.

2.1. Language Syntax

The syntax of the language is shown in Figure 1. A program is a sequence of function definitions, where each function name f/n (atom/arity) has an associated definition $\mathsf{fun}\ (X_1, \ldots, X_n) \to e$, where $X_1, \ldots, X_n$ are (distinct) fresh variables and are the only variables that may occur free in e.[1] 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 (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 $\{e_1, \ldots, e_n\}$),[2] 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 a let binding of the form $\mathsf{let}\ X = expr_1\ \mathsf{in}\ expr_2$.

In this language, we distinguish expressions, patterns, and values. 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, i.e., they are *ground* (without variables) patterns. Expressions are ranged over by $e, e', e_1, \ldots$, patterns by $pat, pat', pat_1, \ldots$ and values by $v, v', v_1, \ldots$ 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 $\mathcal{D}om(\theta) = \{X \in Var \mid$

[1] Here, we consider that variable X is *bound* in $\mathsf{let}\ X = e_1\ \mathsf{in}\ e_2$; also, all pattern variables occurring in case or receive expressions and all parameters in function definitions are *bound* too; all other variables are *free*.

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

$X \neq \theta(X)\}$ is its domain. Substitutions are usually denoted by (finite) sets of bindings like, e.g., $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$. 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 $X \in Var$. We follow a postfix notation for substitution application: given an expression e and a substitution σ the application $\sigma(e)$ is denoted by $e\sigma$.

In a case expression "case e of pat_1 when $e_1 \to e'_1$; ...; pat_n when $e_n \to e'_n$ end", we first evaluate e to a value, say v; then, we find (if it exists) the first clause pat_i when $e_i \to e'_i$ such that v matches pat_i, i.e., such that there exists a substitution σ for the variables of pat_i with $v = pat_i\sigma$, 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 local (FIFO) queues of the processes until they are consumed. 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).

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

We consider the following functions with side-effects (where self and spawn are built-ins in the Erlang programming language): 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 $\overline{pat_n}$ when $e_n \to e'_n$ end" traverses the process' queue until one message matches a clause in the receive statement; i.e., it should find the *first* message v in the process' queue (if any) such that case v of $\overline{pat_n}$ when $e_n \to e'_n$ end can be reduced to some expression e''; then, the receive expression evaluates to the same expression to which the above case expression would be evaluated, 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. As in case expressions, patterns of receive statements can only contain fresh variables.

Our language models the functional and concurrent subset of Core Erlang [15], the intermediate representation used during the compilation of Erlang programs. Therefore, our developments can be directly applied to Erlang.

Example 2.1. 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.

```
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
```

Figure 2. A simple client/server program

2.2. A High-Level Semantics

In this section, we present an (asynchronous) operational semantics for our language. Following [13], we introduce a *global mailbox* (there called "ether") to guarantee that our semantics generates all admissible message interleavings. In contrast to previous semantics [12, 14, 13], though, our semantics abstracts away from processes' queues. We will see in Sections 2.3 and 4.3 that this decision simplifies both the semantics and the notion of concurrency, while still modeling the same potential computations.

Definition 2.2. (process)
A process is denoted by a configuration of the form $\langle p, \theta, e\rangle$, where p is the pid of the process, θ is an environment (a substitution of values for variables), and e is an expression to be evaluated.

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

Definition 2.3. (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 [18, Section 10.8].[3] 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 using FIFO queues serves only to select those computations satisfying the constraint. Nevertheless, if our approach to record and replay is applied to a computation satisfying the above constraint (e.g., because the program is run

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

in a standard environment for Erlang), then our replay computation will also satisfy it and, thus, no spurious computations are introduced.[4]

A system is now defined as follows:

Definition 2.4. (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 (thanks to the fact that " $\mid$ " is associative and commutative).

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 (typically a function application that starts the execution).

In [12], Γ was used to store messages after they were sent and before they were *delivered* to the local queue of the target process, i.e., Γ represented an abstraction of the *network*. Here, we go one step further and also abstract away the process' queues, so now Γ stores all the messages that have been sent and have not yet been *consumed* by a receive statement. In other words, Γ now denotes the union of the network and the original process' queues. The new formulation allows for a more efficient tracing, since it would be difficult to trace message delivery, yet it is precise enough for the purpose of this paper. In fact, all behaviors that can be obtained with the semantics in [12] can also be obtained with the present semantics and vice versa, hence the abstraction is correct. Since the relation with the semantics in [12] is not key to understand the present paper, we refer the interested reader to the companion technical report [17].

Following the style in [12], 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). We also use this information to define a notion of concurrency for our language (see Section 2.3). For the moment, 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 [12] 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:

- A one-step reduction of an expression without side-effects (e.g., a function application or a case expression) 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 complete the step by actually adding the triple $(p, p', \{v, \ell\})$ to Γ (p is the process performing

[4] An alternative definition ensuring the order of messages between any two given processes can be found in [14].

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

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

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

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

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

Figure 3. Logging semantics

the send). Observe that the message is *tagged* with some fresh (unique) identifier ℓ. These tags will allow us to track messages and avoid confusion when several messages have the same value. Moreover, they will become essential to uniquely identify every message, so that the effects of sending and/or receiving a particular message can be undone (these tags are similar to the timestamps used in [19]).

- 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 (i.e., a sort of *future* that we deal with as a variable for simplicity) 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 process executing the $\mathsf{self}()$ expression.

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

$$
\begin{aligned}
&\{\,\};\ \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{aligned}
$$

Figure 4. Faulty derivation with the client/server of Example 2.1

Example 2.5. Consider the program of Example 2.1 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) computation from this system is shown in Fig. 4 (the selected expression at each step is underlined). In the figure, 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.

Roughly speaking, the computation is faulty since 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.

2.3. Concurrency

In order to define a causal-consistent (reversible) semantics we need not only an interleaving semantics such as the one we just presented, but also a notion of concurrency (or, equivalently, the opposite notion of conflict). 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 concurrency, 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 $\mathsf{init}(d) = s_1$ and $\mathsf{final}(d) = s_2$. Two derivations, d_1 and d_2, are *composable* if $\mathsf{final}(d_1) = \mathsf{init}(d_2)$. In this case, we let $d_1; d_2$ denote their composition with $d_1; d_2 = (s_1 \hookrightarrow s_2 \hookrightarrow \cdots \hookrightarrow s_n \hookrightarrow s_{n+1} \hookrightarrow \cdots \hookrightarrow s_m)$ if $d_1 = (s_1 \hookrightarrow s_2 \hookrightarrow \cdots \hookrightarrow s_n)$ and

$d_2 = (s_n \hookrightarrow s_{n+1} \hookrightarrow \cdots \hookrightarrow s_m)$. Two derivations, d_1 and d_2, are said *coinitial* if $\mathsf{init}(d_1) = \mathsf{init}(d_2)$, and *cofinal* if $\mathsf{final}(d_1) = \mathsf{final}(d_2)$.

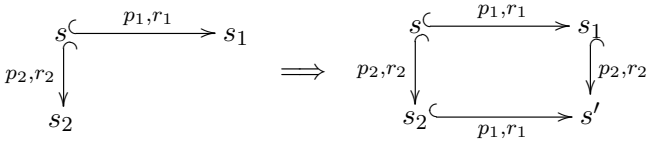
For simplicity, in the following, we consider derivations up to renaming of bound variables, i.e., we say that derivations d_1 and d_2 are equal if they are identical except for (possibly) a renaming of bound variables.

Under this assumption, the semantics is *almost* deterministic, i.e., the main sources of non-determinism are the selection of a process p (where any fair strategy would be fine, but we leave it unspecified in this work) and the selection of the message to be retrieved in rule $Receive$ when more than one message targets the selected process p. There is also some non-determinism in the choice of the fresh identifier ℓ for messages in rule *Send* and in the choice of the pid p' of the new process in rule *Spawn*. However, identifiers are just a technical mean to distinguish messages, hence we can safely consider them up to renaming. We also consider pids up to renaming, since in general their value is not relevant, and this simplifies the notion of concurrency (otherwise all applications of rule *Spawn* would be in conflict due to the selection of the pid). 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 $s_1 = s_2$.

Definition 2.6. (concurrent transitions)
Given two different coinitial transitions, $t_1 = (s \hookrightarrow_{p_1,r_1} s_1)$ and $t_2 = (s \hookrightarrow_{p_2,r_2} s_2)$, we say that they are *in conflict* if they consider the same process, i.e., $p_1 = p_2$, and the applied rules are both $Receive$, i.e., $r_1 = \mathsf{rec}(\ell_1)$ and $r_2 = \mathsf{rec}(\ell_2)$ for some ℓ_1, ℓ_2 with $\ell_1 \neq \ell_2$. Two different coinitial transitions are *concurrent* if they are not in conflict.

Now, we prove a key result for our notion of concurrency. For simplicity, we consider in the following a fixed (implicit) program in the technical results.

Lemma 2.7. (square lemma)
Given two coinitial concurrent transitions $t_1 = (s \hookrightarrow_{p_1,r_1} s_1)$ and $t_2 = (s \hookrightarrow_{p_2,r_2} s_2)$, there exist two cofinal transitions $t_2/t_1 = (s_1 \hookrightarrow_{p_2,r_2} s')$ and $t_1/t_2 = (s_2 \hookrightarrow_{p_1,r_1} s')$. Graphically,



Proof:
If $p_1 \neq p_2$ then, by applying rule r_2 to p_1 in s_1 and rule r_1 to p_2 in s_2, we get two transitions t_1/t_2 and t_2/t_1 which are cofinal. If $p_1 = p_2$, since the transitions are concurrent, at least one of the applied rules must be different from $Receive$. This case is not possible, though, since the semantics is deterministic in this case and, thus, there can be only one reduction issuing from s. □

Our notion of concurrency is less restrictive than the one in [12], which also considers actions to deliver messages to the local queues of the processes. Notably, the choice in [12] introduces some (unnecessary) conflicts, e.g., between delivering a message to a process and receiving a (different) message. Despite that, the two semantics model essentially the same derivations [17].

2.4. Independence

We now instantiate to our setting the well-known *happened-before* relation [20], and the related notion of *independent* transitions:[5]

Definition 2.8. (happened-before, independence)
Given a derivation d and two transitions $t_1 = (s_1 \hookrightarrow_{p_1,r_1} s_1')$ and $t_2 = (s_2 \hookrightarrow_{p_2,r_2} s_2')$ in d, 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 $t_2 \not\rightsquigarrow t_1$.

An interesting property of our semantics is that consecutive independent transitions can be switched without changing the final state:

Lemma 2.9. (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 exist 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.

Proof:
By definition, we have that $p_1 \neq p_2$, so that the transitions belong to different processes. Then, the two transitions can be trivially switched except when $r_1 = \mathsf{spawn}(p)$ and $p_2 = p$, for some pid p, and when $r_1 = \mathsf{send}(\ell)$ and $r_2 = \mathsf{rec}(\ell)$ for some tag ℓ. These cases, though, are not possible since the transitions are independent, thus the claim holds. Note that we can assume that the same fresh pids or message tags are used in the switched transitions since these values are still fresh. □

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

Definition 2.10. (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.

We note that our notion of causally equivalent derivations can be seen as an instance of the *trace equivalence* in [21].

[5]Here, we use the term *independent* instead of *concurrent*, used in [20], to avoid confusion with the notion in Definition 2.6, which is the typical meaning of the term concurrent in the literature of causal-consistent reversibility.

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 a debugger. Actually, we do not want to replay necessarily the exact same execution, but any 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, one should instrument the program so that its execution in the actual environment (e.g., the Erlang/OTP environment) produces a collection of sequences of logged events (one sequence per process). In the following, though, we exploit the logging semantics and, in particular, we compute the log from 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., [19]) 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 present in the model considered in [19] 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. Clearly, their detailed treatment would be needed in an implementation of a debugger for full Erlang. However, we consider that this issue is orthogonal to our approach, and handling it would complicate our semantics without necessity. As a consequence, we leave this discussion for a more practical work that considers the implementation of this semantics.

In the following, (ordered) sequences are denoted by $w = (r_1, r_2, \ldots, r_n)$, $n \geq 1$, where $()$ denotes the empty sequence. Given sequences w_1 and w_2, we denote their concatenation by $w_1{+}w_2$; when w_1 just contains one element, i.e., $w_1 = (r)$, we write $r{+}w_2$ instead of $(r){+}w_2$ for simplicity.

Definition 3.1. (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 process p in d*, in symbols

$\mathcal{L}(d, p)$, is inductively defined as follows:

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

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 $\mathcal{L}(d,p)$. Trivially, $\mathcal{L}(d,p)$ can be obtained from $\mathcal{L}(d)$, i.e., $\mathcal{L}(d,p) = \omega$ if $(p,\omega) \in \mathcal{L}(d)$ and $\mathcal{L}(d,p) = ()$ otherwise.

Example 3.2. Consider the derivation shown in Example 2.5, here referred to as d. If we run it under the logging semantics, we get the following logs:

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

Clearly, given a finite derivation d, the associated log $\mathcal{L}(d)$ is finite too. However, the opposite is not true: we might have a finite log associated to an infinite derivation (e.g., by applying infinitely many times rule Seq).

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 that showed some 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 3.3. (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. There are several ways of ensuring that d_1 and d_2 include the same amount of transitions. By restricting to fully-logged derivations, we include the minimal amount of transitions needed to produce the observed log.

We first introduce an easy result, needed for the proof of Theorem 3.6.

Lemma 3.4. 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, $\mathcal{L}(t_1; t_2) = \mathcal{L}(t_{2\langle\langle t_1}; t_{1\rangle\rangle t_2})$, where $t_{2\langle\langle t_1} = (s_1 \hookrightarrow_{p_2,r_2} s_4)$ and $t_{1\rangle\rangle t_2} = (s_4 \hookrightarrow_{p_1,r_1} s_3)$.

Proof:
By definition of independence, $p_1 \neq p_2$. Thus the two transitions contribute to different logs. The thesis follows since the log is a function of the label, and the labels of t_1 and t_2 coincide with the labels of $t_{1\rangle\rangle t_2}$ and $t_{2\langle\langle t_1}$, respectively. □

We now present a lemma capturing the fact that, if the log is fixed, the behavior of each process is also fixed (but not the interleavings among them).

Lemma 3.5. (local determinism)
Let d_1, d_2 be coinitial fully-logged derivations with $\mathcal{L}(d_1) = \mathcal{L}(d_2)$. Then, for each pid p occurring in d_1, d_2, we have $S^1_p = S^2_p$, where S^1_p (resp. S^2_p) is the ordered sequence of configurations $\langle p, \theta, e\rangle$ occurring in d_1 (resp. d_2), with consecutive equal elements collapsed.

Proof:
We prove the claim by induction on the length n of the derivation d_1. Since the base case is trivial, let us consider the case $n > 0$. Since d_1 and d_2 are coinitial, we have $\mathsf{init}(d_1) = \mathsf{init}(d_2)$. Let $s_0 = \mathsf{init}(d_1)$. Let $t_1 = (s_0 \hookrightarrow_{p_1,r_1} s_1)$ be the first transition in d_1 for some process p_1 and label r_1. Since $\mathcal{L}(d_1) = \mathcal{L}(d_2)$, we have $\mathcal{L}(d_1, p_1) = \mathcal{L}(d_2, p_1)$. Let $t_2 = (s_2 \hookrightarrow_{p_1,r_1} s_3)$ be the first transition for process p_1 in d_2. It is easy to see that t_2 must be independent w.r.t. all previous transitions: no transition can happen-before t_2 in d_2 since, then, t_1 would not be applicable to s_0 (note that d_1 and d_2 are coinitial). Therefore, by the switching lemma (Lemma 2.9), there exists a new derivation d_3 which is obtained from d_2 by switching t_2 with all previous transitions. It is easy to see that, for each pid p occurring in d_2, d_3, we have $S^2_p = S^3_p$, where S^2_p (resp. S^3_p) is the ordered sequence of configurations $\langle p, \theta, e\rangle$ occurring in d_2 (resp. d_3), with consecutive equal elements collapsed. Therefore, now we have $s_0 \hookrightarrow_{p_1,r_1} s'_1$ as the first transition in d_3 and, thus, $s_1 = s'_1$. Trivially, we have $\mathcal{L}(d_2) = \mathcal{L}(d_3)$ and, thus, the claim follows by applying the inductive hypothesis. □

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

Proof:
The "if" direction follows by induction on the number of switches using Lemma 3.4.

Let us now consider the "only if" direction. We prove the claim by contradiction. Assume that $\mathcal{L}(d_1) = \mathcal{L}(d_2)$ but $d_1 \not\approx d_2$. Let us swap independent transitions in d_2 so to match the longest possible prefix of d_1. Then, we have $d_1 = d_c; t_1; d'_1$ and $d_2 \approx d''_2 = d_c; t_2; d'_2$, where d_c is a common prefix (which might be empty), and we cannot swap independent transitions in $t_2; d'_2$ so to match t_1. Since $\mathcal{L}(d_1) = \mathcal{L}(d_2)$, we have $\mathcal{L}(d_1) = \mathcal{L}(d''_2)$ from the "if" direction. Moreover, since d_c is common to both derivations, we trivially have $\mathcal{L}(t_1; d'_1) = \mathcal{L}(t_2; d'_2)$. Let t_1 be labeled with p and r. By Lemma 3.5, there exists a transition t'_1 in $t_2; d'_2$ labeled with p and r too. Since $t_2 \neq t'_1$, we have $t_2 \rightsquigarrow t'_1$. Here, we assume that t'_1 has been moved as far to the left of the derivation as possible, so we have a full chain of causally-dependent transitions starting from t_2 (if there would be an independent transition in between it could have been moved after t'_1). Hence, $t_2; d'_2$ cannot be reordered so that the

first transition matches t_1. Let us call t'_2 the closest transition to t'_1 such that $t'_2 \rightsquigarrow t'_1$. According to Definition 2.8, we have the following possibilities:

- Transition t'_2 is performed by process p too. This is not possible since, thanks to Lemma 3.5, the first transition performed by p is the same in both derivations, and we assumed it to be t'_1.
- Transition t'_2 spawns process p. This is not possible since we assumed that t_1 was performed by p. Hence, process p should have existed before.
- Transition t'_2 sends a message received by t'_1. Here, we get a contradiction since $\mathcal{L}(t_1; d'_1) = \mathcal{L}(t_2; d'_2)$ and t_1 must reduce a receive statement taking the same message as t'_1, so t_2 cannot send such a message since the message should have already existed.

In all cases, we reach a contradiction and, thus, the claim follows. □

4. A Causal-Consistent Replay Reversible Semantics

In this section, we introduce an *uncontrolled* replay reversible semantics. It takes a program and the log of a given derivation, and allows us to go both forward and backward along any causally equivalent derivation. This semantics constitutes the kernel of our debugging framework. Following [22], the term uncontrolled indicates that the semantics specifies how to go back and forward, but there is no policy to select the applicable rule (when more than one is enabled) nor whether forward moves should be preferred over backward moves or vice versa. Uncontrolled semantics are suitable to fix the basis of a reversible computational model, yet they are not immediately useful in practice, since they do not provide facilities to decide which actions to replay/undo. For this reason, in Section 5 we build on top of this semantics a *controlled* one, where the selection of the actions to replay/undo is driven by the queries from the user of a debugger. This allows the user to focus on particular actions and processes (e.g., sending a given message, spawning a given process, etc).

In the following, processes have the form $\langle p, \omega, h, \theta, e\rangle$, with ω a *log* and h a *history*. Histories are needed to enable reversibility: without a history, forward transitions may lose information and, thus, it would not be possible to recover the predecessor of a given state. In this context, a history h records the intermediate states of a process using terms headed by constructors seq, send, rec, spawn, and self, and whose arguments are the information required to (deterministically) undo the step (following a typical Landauer embedding [23]). We will characterize later on (see Theorem 4.17 and the discussion preceding it) the suitable amount of history information to be stored.

In the following, we introduce two transition relations: $\rightharpoonup$ and $\leftharpoondown$. The former, $\rightharpoonup$, is similar to the logging semantics $\hookrightarrow$ (Figure 3) but it is now driven by the considered log. In contrast, the latter, $\leftharpoondown$, proceeds in the backward direction, "undoing" actions step by step. We refer to $\rightharpoonup$ and $\leftharpoondown$ as the (uncontrolled) *forward* and *backward* semantics, respectively. We denote their union $\rightharpoonup \cup \leftharpoondown$ by $\rightleftharpoons$.

4.1. Uncontrolled Forward Semantics

The uncontrolled forward semantics is shown in Figure 5. Even if not necessary from a practical point of view, labels of the forward semantics contain the same information in the labels of the logging

(*Seq*)

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

(*Send*)

$$\frac{\theta, e \xrightarrow{\mathsf{send}(p',v)} \theta', e'}{\begin{array}{l}\Gamma; \langle p, \mathsf{send}(\ell){+}\omega, h, \theta, e\rangle \mid \Pi \rightharpoonup_{p,\mathsf{send}(\ell),\{\mathsf{s},\ell^{\Uparrow}\}} \Gamma \cup \{(p, p', \{v, \ell\})\}; \\ \qquad\qquad \langle p, \omega, \mathsf{send}(\theta, e, p', \{v, \ell\}){+}h, \theta', e'\rangle \mid \Pi\end{array}}$$

(*Receive*)

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

(*Spawn*)

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

(*Self*)

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

Figure 5. Uncontrolled forward semantics

semantics. We do it for technical reasons, so that results such as Lemma 4.13 (see below) can be more easily formalized. 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*, where logs are added to new processes).

The rules for expressions are the same as in the logging semantics (an advantage of the modular design). The forward semantics is similar to the logging semantics, except for two main differences. First, some parameters are fixed by logs: the fresh message identifier in rule *Send*, the message received in rule *Receive*, and the fresh pid in rule *Spawn*. Second, we build a history with a sequence of items which allows us to go backwards to any point in the computation. The history items are headed by the applied rule (*Receive* is shortened to rec), and contain the current environment and expression, as well as some rule-specific information. In particular, rule *Send* records the target pid and the message, rule *Receive* the sender pid and the message, and rule *Spawn* the pid of the new process. We could optimize the information stored in these terms following [24, 25, 26], but this is orthogonal to our purpose in this paper.

Example 4.1. Consider the logs of Example 3.2. Then, we have, e.g., the forward derivation in Fig. 6. For simplicity, in the histories we only show events send, rec and spawn and, moreover, we skip the

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

Figure 6. Uncontrolled forward derivation with the traces of Example 3.2

$$(\overline{Seq})\quad \Gamma;\langle p,\omega,\mathsf{seq}(\theta,e)+h,\theta',e'\rangle \mid \Pi \leftharpoondown_{p,\mathsf{seq},\{\mathsf{s}\}\cup\mathcal{V}} \Gamma;\langle p,\omega,h,\theta,e\rangle \mid \Pi \quad \text{where } \mathcal{V}=\mathcal{D}om(\theta')\backslash\mathcal{D}om(\theta)$$

$$(\overline{Send})\quad \Gamma\cup\{(p,p',\{v,\ell\})\};\langle p,\omega,\mathsf{send}(\theta,e,p',\{v,\ell\})+h,\theta',e'\rangle \mid \Pi \leftharpoondown_{p,\mathsf{send}(\ell),\{\mathsf{s},\ell^{\Uparrow}\}} \Gamma;\langle p,\mathsf{send}(\ell)+\omega,h,\theta,e\rangle \mid \Pi$$

$$(\overline{Receive})\quad \Gamma;\langle p,\omega,\mathsf{rec}(\theta,e,p',\{v,\ell\})+h,\theta',e'\rangle \mid \Pi \leftharpoondown_{p,\mathsf{rec}(\ell),\{\mathsf{s},\ell^{\Downarrow}\}\cup\mathcal{V}} \Gamma\cup\{(p',p,\{v,\ell\})\};\langle p,\mathsf{rec}(\ell)+\omega,h,\theta,e\rangle \mid \Pi \quad \text{where } \mathcal{V}=\mathcal{D}om(\theta')\backslash\mathcal{D}om(\theta)$$

$$(\overline{Spawn})\quad \Gamma;\langle p,\omega,\mathsf{spawn}(\theta,e,p')+h,\theta',e'\rangle \mid \langle p',\omega',(),id,e''\rangle \mid \Pi \leftharpoondown_{p,\mathsf{spawn}(p'),\{\mathsf{s},\mathsf{sp}_{p'}\}} \Gamma;\langle p,\mathsf{spawn}(p')+\omega,h,\theta,e\rangle \mid \Pi$$

$$(\overline{Self})\quad \Gamma;\langle p,\omega,\mathsf{self}(\theta,e)+h,\theta',e'\rangle \mid \Pi \leftharpoondown_{p,\mathsf{self},\{\mathsf{s}\}} \Gamma;\langle p,\omega,h,\theta,e\rangle \mid \Pi$$

Figure 7. Uncontrolled backward semantics

first two arguments (the environment and the expression). The actions performed by each process are the same as in the original derivation in Example 2.5, 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 is ℓ_2, so message ℓ_3 cannot be received. Taking message ℓ_3 would lead to a derivation which is not causally equivalent to the original one.

4.2. Uncontrolled Backward Semantics

Fig. 7 shows the rules of the (uncontrolled) backward semantics. All rules restore the environment and the expression of the process, and rules $\overline{Send}$, $\overline{Receive}$ and $\overline{Spawn}$ additionally restore its stored log. Let us briefly discuss a few particular situations:

- Rule $\overline{Send}$ only applies if the message sent is in the global mailbox. If, instead, the message has already been received, then one should first apply backward steps to the receiver until, eventually, rule $\overline{Receive}$ puts the message back into the global mailbox, enabling rule $\overline{Send}$. In the next section, we will introduce a strategy that achieves this effect in a controlled manner.

- A similar situation occurs in rule $\overline{Spawn}$. Given a process p with a history item $\mathsf{spawn}(\theta,e,p')$, rule $\overline{Spawn}$ cannot be applied until the history of process p' is empty. Therefore, one should first apply a number of backward steps to p' in order to be able to undo the spawn item.[6]

Example 4.2. Consider the last system in the replay derivation of Example 4.1 and assume that we want to undo the actions of process c up to the sending of the message (tagged with ℓ_2) that produced the error. Such a derivation could be performed, e.g., as shown in Fig. 8 (the history item selected

[6] We note that there is no need to require that no message targeting the process p' (which would become an *orphan* message) is in the global mailbox: in order to send such a message the pid p' is needed, hence the sending of the message depends on the spawn and, thus, it must be undone beforehand.

$$\{(\mathrm{p},\mathrm{s},\{\{\mathrm{c},40\},\ell_3\})\};\ \langle \mathrm{c},(),(\mathsf{send}(\_,\_,\mathrm{s},\{2,\ell_2\}),\mathsf{send}(\_,\_,\mathrm{p},\{\{\mathrm{s},\{\mathrm{c},40\}\},\ell_1\}),$$
$$\mathsf{spawn}(\_,\_,\mathrm{p}),\mathsf{spawn}(\_,\_,\mathrm{s})),\_,\mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s},(),(\underline{\mathsf{rec}(\_,\_,\mathrm{c},\{2,\ell_2\})}),\_,\mathrm{error}\rangle$$
$$\mid \langle \mathrm{p},(),(\mathsf{send}(\_,\_,\mathrm{s},\{\{\mathrm{c},40\},\ell_3\}),\mathsf{rec}(\_,\_,\mathrm{c},\{\{\mathrm{s},\{\mathrm{c},40\}\},\ell_1\})),\_,\mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle$$
$$\leftharpoondown \{(\mathrm{p},\mathrm{s},\{\{\mathrm{c},40\},\ell_3\}),(\mathrm{c},\mathrm{s},\{2,\ell_2\})\};\ \langle \mathrm{c},(),(\underline{\mathsf{send}(\_,\_,\mathrm{s},\{2,\ell_2\})},\mathsf{send}(\_,\_,\mathrm{p},\{\{\mathrm{s},\{\mathrm{c},40\}\},\ell_1\}),$$
$$\mathsf{spawn}(\_,\_,\mathrm{p}),\mathsf{spawn}(\_,\_,\mathrm{s})),\_,\mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s},(\mathsf{rec}(\ell_2)),(),\_,\mathsf{receive}\ \ldots\rangle$$
$$\mid \langle \mathrm{p},(),(\mathsf{send}(\_,\_,\mathrm{s},\{\{\mathrm{c},40\},\ell_3\}),\mathsf{rec}(\_,\_,\mathrm{c},\{\{\mathrm{s},\{\mathrm{c},40\}\},\ell_1\})),\_,\mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle$$
$$\leftharpoondown \{(\mathrm{p},\mathrm{s},\{\{\mathrm{c},40\},\ell_3\})\};\ \langle \mathrm{c},(\mathsf{send}(\ell_2)),(\mathsf{send}(\_,\_,\mathrm{p},\{\{\mathrm{s},\{\mathrm{c},40\}\},\ell_1\}),\mathsf{spawn}(\_,\_,\mathrm{p}),$$
$$\mathsf{spawn}(\_,\_,\mathrm{s})),\_,\mathsf{let}\ Y=\mathrm{s}\,!\,2\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s},(\mathsf{rec}(\ell_2)),(),\_,\mathsf{receive}\ \ldots\rangle$$
$$\mid \langle \mathrm{p},(),(\mathsf{send}(\_,\_,\mathrm{s},\{\{\mathrm{c},40\},\ell_3\}),\mathsf{rec}(\_,\_,\mathrm{c},\{\{\mathrm{s},\{\mathrm{c},40\}\},\ell_1\})),\_,\mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle$$

Figure 8. Uncontrolled backward derivation

to be undone is underlined now). Note that process s must first undo the receiving of the message in order for the rule $\overline{Send}$ to be applicable to process c. Once the backward derivation is completed, one could inspect the current system and see what the problem is: there is no guarantee that the message tagged with ℓ_3 will be received before the message tagged with ℓ_2.

4.3. Basic Properties of the Replay Reversible Semantics

In this section we show that the uncontrolled semantics (involving both forward and backward transitions) is consistent and we relate it to the logging semantics. We need the following auxiliary functions:

Definition 4.3. Let $d = (s_1 \hookrightarrow^* s_2)$ be a derivation under the logging semantics, with[7]

$$s_1 = \Gamma; \langle p_1, \theta_1, e_1\rangle \mid \ldots \mid \langle p_n, \theta_n, e_n\rangle$$

The system corresponding to s_1 in the reversible 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 $s = \Gamma; \langle p_1, \omega_1, h_1, \theta_1, e_1\rangle \mid \ldots \mid \langle p_n, \omega_n, h_n, \theta_n, e_n\rangle$ in the reversible semantics, we let $del(s)$ be the system obtained from s by removing both logs and histories, i.e., $del(s) = \Gamma; \langle p_1, \theta_1, e_1\rangle \mid \ldots \mid \langle p_n, \theta_n, e_n\rangle$. It is extended to derivations in the obvious way: given a derivation d of the form $s_1 \hookrightarrow \ldots \hookrightarrow s_n$, we let $del(d)$ be $del(s_1) \hookrightarrow \ldots \hookrightarrow del(s_n)$.

Essentially, function $addLog$ equips each process with its log and an empty history.

In the following, we consider that the notion of log (cf. Definition 3.1) is extended to the *forward* semantics in the obvious way. We also extend the definitions of functions init and final from

[7] We note that we consider a derivation starting from a system which is not necessarily an initial system since a debugger might consider checkpoints that save the state of a computation periodically, so that one wants to replay the derivation from the last checkpoint to the faulty state.

Section 2.2 to reversible derivations, as well as the notions of composable, coinitial and cofinal derivations. Furthermore, we now call a system s' *initial* under the reversible 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 reversible semantics:

Definition 4.4. (fully-logged reversible derivation)
A derivation d under the replay reversible semantics is *fully-logged* if, for each process p in $\mathsf{final}(d)$, the log is empty and the last element in the history (if any) corresponds to a logged transition (i.e., one which is not labeled with seq nor self).

Note that, in addition to Definition 3.3, we now require that processes *consume* all their logs.

We relate the uncontrolled forward and backward semantics using the well-known loop lemma (see, e.g., [27, Lemma 6] in the context of the process calculus CCS), stating that each forward (resp. backward) transition can be undone by a backward (resp. forward) transition. We need to restrict the attention to systems reachable from the execution of a program:

Definition 4.5. (reachable systems)
A system s is *reachable* if there exists an initial system s_0 such that $s_0 \rightleftharpoons^* s$.

Since this restriction is needed for other results as well, and since only reachable systems are of interest (non-reachable systems are ill-formed), in the following we assume that all the systems are reachable. We can now show the loop lemma.

Lemma 4.6. (loop lemma)
For every pair of systems, s_1 and s_2, we have $s_1 \rightharpoonup_{p,r} s_2$ iff $s_2 \rightharpoonup_{p,r} s_1$.

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

The loop lemma ensures that each transition t has an *inverse*, that we denote by $\overline{t}$. More precisely, given a transition t, we let $\overline{t} = (s' \leftharpoonup_{p,r} s)$ if $t = (s \rightharpoonup_{p,r} s')$ and $\overline{t} = (s' \rightharpoonup_{p,r} s)$ if $t = (s \leftharpoonup_{p,r} s')$. This notation is naturally extended to derivations. We let ϵ_s denote the zero-step derivation $s \rightleftharpoons^* s$.

A nice property of our reversible semantics is that the key notions of concurrency (Definition 2.6) and independence (Definition 2.8) for the logging semantics are now subsumed by the following notion of concurrency for the reversible semantics (see Lemma 4.10 and 4.11 below):

Definition 4.7. (Concurrent transitions)
Given two different coinitial transitions, $t_1 = (s \rightleftharpoons_{p_1,r_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2} s_2)$, they are *in conflict* if at least one of the following conditions holds:

1. *both transitions are forward*, they consider the same process, i.e., $p_1 = p_2$, and the applied rules are both $Receive$, i.e., $r_1 = \mathsf{rec}(\ell_1)$ and $r_2 = \mathsf{rec}(\ell_2)$ for some ℓ_1, ℓ_2 with $\ell_1 \neq \ell_2$;

2. one is a *forward* transition that applies to a process p, say $p_1 = p$, and the other one is a *backward* transition that undoes the spawning of p, i.e., $r_2 = \mathsf{spawn}(p)$;

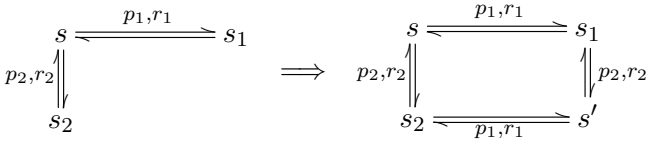
3. one is a *forward* transition where a process p_1 receives a message with identifier ℓ, i.e., $r_1 = \mathsf{rec}(\ell)$, and the other one is a *backward* transition that undoes the sending of the same message, i.e., $r_2 = \mathsf{send}(\ell)$;

4. one is a *forward* transition and the other one is a *backward* transition such that $p_1 = p_2$.

Two different coinitial transitions are *concurrent* if they are not in conflict. Note that two coinitial backward transitions are always concurrent.

Consequently, the following lemma subsumes both Lemma 2.7 (square lemma) and Lemma 2.9 (switching lemma) from the logging semantics:

Lemma 4.8. (square lemma)
Given two coinitial concurrent transitions $t_1 = (s \rightleftharpoons_{p_1,r_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2} s_2)$, there exist two cofinal transitions $t_2/t_1 = (s_1 \rightleftharpoons_{p_2,r_2} s')$ and $t_1/t_2 = (s_2 \rightleftharpoons_{p_1,r_1} s')$. Graphically,



Proof:
We distinguish the following cases depending on the applied rules.

- Two forward transitions. This case is perfectly analogous to the proof of Lemma 2.7.

- One forward transition and one backward transition. In this case, the proof is similar to the proof of Lemma 2.9.

- Two backward transitions. If $p_1 \neq p_2$, the claim follows trivially. The case where they apply to the same process, i.e., $p_1 = p_2$, is not possible since the backward semantics is deterministic once fixed the selected process: the top element in the history determines the rule to apply (as well as the message to be received in rule $\overline{Receive}$).

□

Nevertheless, we explicitly state a switching lemma for reversible transitions which is an easy consequence of the square lemma above and the loop lemma (Lemma 4.6):

Lemma 4.9. (switching lemma)
Given two composable transitions of the form $t_1 = (s_1 \rightleftharpoons_{p_1,r_1} s_2)$ and $t_2 = (s_2 \rightleftharpoons_{p_2,r_2} s_3)$ such that $\overline{t_1}$ and t_2 are concurrent, there exist a system s_4 and two composable transitions $t_{2\langle\langle t_1} = (s_1 \rightleftharpoons_{p_2,r_2} s_4)$ and $t_{1\rangle\rangle t_2} = (s_4 \rightleftharpoons_{p_1,r_1} s_3)$.

Proof:
First, using the loop lemma (Lemma 4.6), we have $\overline{t_1} = (s_2 \rightleftharpoons_{p_1,r_1} s_1)$. Now, since $\overline{t_1}$ and t_2 are concurrent, by applying the square lemma (Lemma 4.8) to $\overline{t_1} = (s_2 \rightleftharpoons_{p_1,r_1} s_1)$ and $t_2 = (s_2 \rightleftharpoons_{p_2,r_2} s_3)$, there exists a system s_4 such that $\overline{t_1}_{\rangle\rangle t_2} = \overline{t_1}/t_2 = (s_3 \rightleftharpoons_{p_1,r_1} s_4)$ and $t_{2\langle\langle t_1} = t_2/\overline{t_1} = (s_1 \rightleftharpoons_{p_2,r_2} s_4)$. Using the loop lemma (Lemma 4.6) again, we have $t_{1\rangle\rangle t_2} = t_1/t_2 = (s_4 \rightleftharpoons_{p_1,r_1} s_3)$, which concludes the proof. □

Finally, let us formally prove that the definition of concurrent transitions in the replay reversible semantics subsumes both the definition of concurrent transitions and of independent transitions in the logging semantics.

Lemma 4.10. Let t_1 and t_2 be forward transitions under the replay reversible semantics. Then t_1 and t_2 are concurrent iff $\mathsf{del}(t_1)$ and $\mathsf{del}(t_2)$ are concurrent under the logging semantics.

Proof:
Trivial, since the case of the definition of concurrency for two forward actions under the replay reversible semantics (case 1 in Definition 4.7) coincides with the definition of concurrency for the logging semantics (Definition 2.6). □

Lemma 4.11. Let t_1 and t_2 be independent transitions under the replay reversible semantics. Then $\mathsf{del}(t_1)$ and $\mathsf{del}(t_2)$ are independent under the logging semantics.

Proof:
Trivial, since the case of the definition of concurrency for a forward and a backward action under the replay reversible semantics (cases 2-4 in Definition 4.7) coincides with the definition of independence for the logging semantics. □

4.4. Causal Consistency of the Replay Reversible Semantics

In this section, we prove a number of properties that guarantee that our replay reversible semantics is causal-consistent, an essential result in the literature of reversible computation (see [27] for a detailed discussion).

We first extend the notion of *causal equivalence* from the logging semantics (cf. Definition 2.10) to consider both backward and forward reversible transitions (we use the same symbol to denote this relation since it subsumes the old notion, see below).

Definition 4.12. (causal equivalence)
Causal equivalence, in symbols $\approx$, is the least equivalence relation between transitions closed under composition that obeys the following rules:

$$t_1; t_2/t_1 \approx t_2; t_1/t_2 \qquad\qquad t; \overline{t} \approx \epsilon_{\mathsf{init}(t)}$$

Causal equivalence amounts to say that those derivations that only differ in swaps of concurrent transitions or the removal of consecutive inverse transitions are equivalent. Observe that any of the notations $t_1; t_2/t_1$ and $t_2; t_1/t_2$ requires t_1 and t_2 to be concurrent.

Causal equivalence for reversible derivations subsumes the corresponding notion for the logging semantics in the following sense. Consider two forward composable transitions t_1 and t_2 under the reversible replay semantics. According to the notion of causal equivalence in Definition 2.10, they can be switched if t_1 and t_2 are independent. However, if t_1 and t_2 are independent, we have that $\overline{t_1}$ and t_2 are concurrent (cases (2)-(4) in Definition 4.7). Therefore, by Lemma 4.9, they can be switched and, thus, the old notion is just a particular case of the new causal equivalence relation.

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. First, the forward semantics is a conservative extension of the logging semantics in the following sense:

Lemma 4.13. Let d be a fully-logged derivation under the logging semantics. Then, there exists a finite fully-logged derivation d' under the forward semantics such that $\mathsf{init}(d') = addLog(\mathcal{L}(d), \mathsf{init}(d))$, $\mathsf{del}(d') = d$ and $\mathcal{L}(d) = \mathcal{L}(d')$.

Proof:
The claim is trivial from the rules of the replay semantics, since each item consumed from the log of a process generates exactly the same item in the process' log. The fact that in $\mathsf{final}(d')$ the log of each process is empty ensures that the whole the log is re-created. Finiteness of d' is ensured since we assume all logging derivations to be finite and derivations d and d' have the same length. □

The replay and the original computation are causally equivalent:

Theorem 4.14. Let d be a fully-logged derivation under the logging semantics. Let d' be any finite fully-logged derivation under the forward semantics such that $\mathsf{init}(d') = addLog(\mathcal{L}(d), \mathsf{init}(d))$. Then $d \approx del(d')$.

Proof:
First, a consequence of Lemma 4.13 is that a derivation under the replay semantics for a given global log produces, for each process p, the corresponding log again, i.e., for each p we have $\mathcal{L}(d', p) = \mathcal{L}(d, p)$. The logs of d' and $del(d')$ are the same: $\mathcal{L}(d', p) = \mathcal{L}(del(d'), p)$. Furthermore, if d' is a finite derivation under the replay semantics, $del(d')$ is a derivation under the logging semantics. Therefore, we have $\mathcal{L}(d) = \mathcal{L}(del(d'))$. The claim follows by Theorem 3.6. □

We will generalize Lemma 4.13 above to include backward and forward transitions in Lemma 4.20. We prove such a result exploiting the theory of causal-consistency, first developed in [27] in the context of the process calculus CCS. We present below its main result, namely the causal consistency theorem, adapted to our setting. The causal consistency theorem is also interesting in itself, since it guarantees that the amount of history information in our history is correct w.r.t. the chosen notion of concurrency.

Intuitively, causal consistency states that two coinitial derivations reach the same final state if and only if they are causally equivalent. On the one hand, it means that causally equivalent derivations lead to the same final state and, in particular, produce the same history information, hence it is not possible to distinguish such derivations looking at their final states (hence, also their possible evolutions coincide). In particular, swapping two concurrent transitions or doing and undoing a given transition has

no impact on the final state. On the other hand, derivations differing in any other way are distinguishable by looking at their final state, e.g., the final state keeps track of any past nondeterministic choice. In other terms, causal consistency states that the amount of history information stored is enough to distinguish computations which are not causally equivalent, but no more.

We prove now two results needed for the proof of the causal consistency theorem.

Lemma 4.15. (rearranging lemma)
Given systems s, s', if $d = (s \rightleftharpoons^* s')$, then there exists a system s'' such that $d' = (s \leftharpoonup^* s'' \rightharpoonup^* s')$ and $d \approx d'$. Furthermore, d' is not longer than d.

Proof:
The proof is by lexicographic induction on the length of d and on the number of steps from the earliest pair of transitions in d of the form $s_1 \rightharpoonup s_2 \leftharpoonup s_3$ to s'. If there is no such pair we are done. If $s_1 = s_3$, then $s_1 \rightharpoonup s_2 = \overline{(s_2 \leftharpoonup s_3)}$. Indeed, if $s_1 \rightharpoonup s_2$ adds an item to the history of some process then $s_2 \leftharpoonup s_3$ should remove the same item. Then, we can remove these two transitions and the claim follows by induction since the resulting derivation is shorter and $(s_1 \rightharpoonup s_2 \leftharpoonup s_3) \approx \epsilon_{s_1}$. Otherwise, we apply Lemma 4.9 commuting $s_2 \leftharpoonup s_3$ with all forward transitions preceding it in d (note that a forward transition t followed by a backward transition t' can always be switched since $\overline{t}$ and t' are then both backward transitions and, thus, concurrent). If one such transition is its inverse, then we reason as above. Otherwise, we obtain a new derivation $d' \approx d$ which has the same length of d, and where the distance between the earliest pair of transitions in d' of the form $s'_1 \rightharpoonup s'_2 \leftharpoonup s'_3$ and s' has decreased. The claim follows then by the inductive hypothesis. □

Lemma 4.16. (shortening lemma)
Let d_1 and d_2 be coinitial and cofinal derivations, such that d_2 is a forward derivation while d_1 contains at least one backward transition. Then, there exists a forward derivation d'_1 of length strictly less than that of d_1 such that $d'_1 \approx d_1$.

Proof:
We prove this lemma by induction on the length of d_1. By the rearranging lemma (Lemma 4.15) there exist a backward derivation d and a forward derivation d' such that $d_1 \approx d; d'$. Furthermore, $d; d'$ is not longer than d_1. Let $s_1 \leftharpoonup_{p_1,r_1} s_2 \rightharpoonup_{p_2,r_2} s_3$ be the only two successive transitions in $d; d'$ with opposite direction. We will show below that there is in d' a transition t which is the inverse of $s_1 \leftharpoonup_{p_1,r_1} s_2$. Moreover, we can swap t with all the transitions between t and $s_1 \leftharpoonup_{p_1,r_1} s_2$, in order to obtain a derivation in which $s_1 \leftharpoonup_{p_1,r_1} s_2$ and t are adjacent.[8] To do so we apply Lemma 4.9, since for all transitions t' in between, we have that $\overline{t'}$ and t are concurrent (this is proved below too). When $s_1 \leftharpoonup_{p_1,r_1} s_2$ and t are adjacent we can remove both of them using $\approx$. The resulting derivation is strictly shorter, thus the claim follows by inductive hypothesis.

Let us now prove the results used above. Thanks to the loop lemma (Lemma 4.6) we have the derivations above iff we have two forward derivations which are coinitial (with s_2 as initial state) and cofinal: $\overline{d}; d_2$ and d'. Since the first transition of $\overline{d}; d_2$, $\overline{(s_1 \leftharpoonup_{p_1,\overline{r_1}} s_2)}$, adds some item k_1 to the history

[8] More precisely, the transition is not t, but a transition that applies the same rule to the same process and producing the same history item, but possibly applied to a different system.

of p_1 and such an item is never removed (since the derivation is forward), then the same item k_1 has to be added also by a transition in d', otherwise the two derivations cannot be cofinal. The earliest transition in d' adding item k_1 is exactly t.

Let us now justify that for each transition t' before t in d' we have that $\overline{t'}$ and t are concurrent. First, t' is a forward transition and it should be applied to a process which is different from p_1, otherwise the item k_1 would be added by transition t in the wrong position in the history of p_1. We consider the following cases:

- If t' applies rule $Spawn$ to create a process p, then t should not apply to process p since the process p_1 to which t applies already existed before t'. Therefore, $\overline{t'}$ and t are concurrent.
- If t' applies rule $Send$ to send a message to some process p, then t cannot receive the same message since the received messages necessarily existed before (after the corresponding $\overline{Receive}$ has been performed). Thus $\overline{t'}$ and t are concurrent.
- If t' applies some other rule, then t' and t are clearly concurrent.

□

We can now state and prove the causal consistency of our reversible semantics.

Theorem 4.17. (causal consistency)
Let d_1 and d_2 be coinitial derivations. Then, $d_1 \approx d_2$ iff d_1 and d_2 are cofinal.

Proof:
By definition of $\approx$, if $d_1 \approx d_2$, then they are coinitial and cofinal, so this direction of the theorem is verified.

Now, we have to prove that, if d_1 and d_2 are coinitial and cofinal, then $d_1 \approx d_2$. By the rearranging lemma (Lemma 4.15), we know that the two derivations can be written as the composition of a backward derivation, followed by a forward derivation, so we assume that d_1 and d_2 have this form. The claim is proved by lexicographic induction on the sum of the lengths of d_1 and d_2, and on the distance between the end of d_1 and the earliest pair of transitions t_1 in d_1 and t_2 in d_2 which are not equal. If all such transitions are equal, we are done. Otherwise, we have to consider three cases depending on the directions of the two transitions:

1. Consider that t_1 is a forward transition and t_2 is a backward one. Let us assume that $d_1 = d; t_1; d'$ and $d_2 = d; t_2; d''$. Here, we know that $t_1; d'$ is a forward derivation, so we can apply the shortening lemma (Lemma 4.16) to the derivations $t_1; d'$ and $t_2; d''$ (since d_1 and d_2 are coinitial and cofinal, so are $t_1; d'$ and $t_2; d''$), and we have that $t_2; d''$ has a strictly shorter forward derivation which is causally equivalent, and so the same is true for d_2. The claim then follows by induction.

2. Consider now that both t_1 and t_2 are forward transitions. By assumption, the two transitions must be different. Let us assume first that they are not concurrent. Therefore, they should be applied to the same process and both rules are $Receive$. In this case we get a contradiction to the fact that d_1 and d_2 are cofinal since both derivations are forward and, thus, we would end up

with systems where some process has a different history item in each derivation. Therefore, we can assume that t_1 and t_2 are concurrent transitions.

Now, let t_1' be the transition in d_2 creating the same history item as t_1. Then, we have to prove that t_1' can be switched back with all previous forward transitions. This holds since no previous forward transition can add any history item to the same process, since otherwise the two derivations could not be cofinal. Hence the previous forward transitions are applied to different processes. The only possible source of conflict would be rule $Spawn$ and rule $Receive$, but this could not happen since, in this case, t_1 could not happen.

Therefore, we can repeatedly apply the switching lemma (Lemma 4.9) to have a derivation causally equivalent to d_2 where t_2 and t_1' are consecutive. The same reasoning can be applied in d_1, so we end up with consecutive transitions t_1 and t_2'. Finally, we can apply the switching lemma once more to $t_1; t_2'$ so that the first pair of different transitions is now closer to the end of the derivation. Hence the claim follows by inductive hypothesis.

3. Finally, consider that both t_1 and t_2 are backward transitions. By definition, we have that t_1 and t_2 are concurrent. Here, we have that t_1 and t_2 cannot remove the same history item. Let k_1 be the history item removed by t_1. Since d_1 and d_2 are cofinal, either there is another transition in d_1 that puts k_1 back in the history or there is a transition t_1' in d_2 removing the same history item k_1. In the first case, $\overline{t_1}$ should be concurrent to all the backward transitions following it but the ones that remove history items from the history of the same process. All the transitions of this kind have to be undone by corresponding forward transitions (since they are not possible in d_2). Consider the last such transition: we can use the switching lemma (Lemma 4.9) to make it the last backward transition. Similarly, the forward transition undoing it should be concurrent to all the previous forward transitions (the reason is the same as in the previous case). Thus, we can use the switching lemma again to make it the first forward transition. Finally, we can apply the simplification rule $t; \overline{t} \approx \epsilon_{\mathsf{init}(t)}$ to remove the two transitions, thus shortening the derivation. In the second case (there is a transition t_1' in d_2 removing the same history item k_1), one can argue as in case (2) above. The claim then follows by inductive hypothesis.

□

4.5. Usefulness for Debugging

In this section, we show that our replay reversible semantics is useful as a basis for designing a debugging tool. In particular, we prove that a (faulty) behavior occurs in the logged derivation iff the replay derivation also exhibits the same *faulty* behavior, hence replay is correct and complete. Notably, correctness and completeness do not depend on the scheduling.

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.

Before proving the correctness and completeness of our replay semantics, we need a few results and definitions.

First, we extend local determinism (Lemma 3.5) to the forward semantics:

Lemma 4.18. (local determinism for the forward semantics)
Let d_1, d_2 be coinitial fully-logged derivations under the forward semantics. Then, for each pid p occurring in d_1, d_2, we have $S_p^1 = S_p^2$, where S_p^1 (resp. S_p^2) is the ordered sequence of configurations $\langle p, \omega, h, \theta, e\rangle$ occurring in d_1 (resp. d_2), with consecutive equal elements collapsed.

Proof:
First, since d_1 and d_2 are coinitial, we have that Γ contains the same messages in both derivations and that the initial configuration for each process p existing since the beginning is the same in both derivations. Now, observe that the forward semantics is deterministic but for the selection of the process to be reduced. Hence, since d_1 and d_2 are coinitial, the same sequence of configurations must be produced in both d_1 and d_2 for each process p. Note that if a process p is created during the derivation, then it is created in both derivations with the same initial configuration. Furthermore, since d_1 and d_2 are fully logged, we know that the transitions of each process stop in a logged transition. For each process p there is a unique configuration where the log is empty and the last element in the history is a logged transition, namely the configuration just after the last logged action has been performed (if the log is empty since the beginning this is the initial configuration). Hence, $S_p^1 = S_p^2$ for each process p. Note that, given the deterministic sequence of transitions for each process, messages with the same identifier sent in d_1 and d_2 also have the same value. □

As a corollary, we have the following result for the forward semantics:

Corollary 4.19. Let d_1, d_2 be coinitial fully-logged derivations under the forward semantics. Then d_1 and d_2 are cofinal.

Proof:
Let us consider the final states. First the two final states include the same processes, since the same processes were present at the beginning of the derivations, and the same spawn operations have been performed since logs have been completely consumed. Thanks to Lemma 4.18 each process is in the same configuration in the two derivations. Also, the two global mailboxes contain the same messages, since they contained the same messages in the initial state, and the same send and receive operations have been performed. The thesis follows. □

Now, we can show that any reachable state in the reversible semantics corresponds to a state reachable in the logging semantics:

Lemma 4.20. Let d, d' be fully-logged derivations under the logging and the replay reversible semantics, respectively. Let $\mathsf{init}(d') = addLog(\mathcal{L}(d), \mathsf{init}(d))$. Then there exists a finite forward computation $d'' \approx d'$ such that $d = del(d'')$.

Proof:
Thanks to Lemma 4.13 there exists a finite forward derivation d'' with $\mathsf{init}(d'') = addLog(\mathcal{L}(d), \mathsf{init}(d))$ and $\mathsf{del}(d'') = d$. We have that d' and d'' are coinitial. By Corollary 4.19 they are also cofinal. Then, the result follows from the causal consistency theorem (Theorem 4.17). □

Note that we cannot directly state $d \approx \mathsf{del}(d')$ since in this case $\approx$ would be at the level of logging derivations, hence it would only deal with forward moves.

Now, we can extend local determinism to the reversible semantics. The extension uses the notion of embedding between ordered sequences. We say that S_1 can be embedded in S_2 iff there is a strictly monotone function f from positions of S_1 to positions of S_2 such that for each position i of S_1 we have $S_1[i] = S_2[f(i)]$, where $S_1[i]$ denotes the i-th element in S_1.

Lemma 4.21. (local determinism for the replay reversible semantics)
Let d_1, d_2 be coinitial fully-logged derivations under the replay reversible semantics, with d_1 forward. Then, for each pid p occurring in d_1, d_2, we have that S^1_p can be embedded in S^2_p, where S^1_p (resp. S^2_p) is the ordered sequence of configurations $\langle p, \omega, h, \theta, e\rangle$ occurring in d_1 (resp. d_2), with consecutive equal elements collapsed.

Proof:
The proof is by induction on the number of backward transitions in d_2. If also d_2 is forward, then S^1_p and S^2_p coincide thanks to Lemma 4.18, as desired (the identity is an embedding). Let $\overline{t} = (s \leftharpoondown_{p,r} s')$ be the first backward transition in d_2. Since all transitions performed by p are in conflict, $\overline{t}$ must recover the previous configuration of p. Since all previous transitions are forward, $\overline{t}$ is independent from them and we can switch $\overline{t}$ with them using the switching lemma (Lemma 4.9) until it becomes adjacent to t. We can then simplify t and $\overline{t}$. The resulting derivation $\hat{d}_2$ has one less backward transition, is fully-logged and is coinitial with d_1. Furthermore, all the sequences are unchanged, but for the one for p, that we denote with $\hat{S}^2_p$. $\hat{S}^2_p$ can be trivially embedded in S^2_p. By inductive hypothesis S^1_p can be embedded in $\hat{S}^2_p$, hence the thesis follows by transitivity of embedding. □

We can now prove correctness and completeness of replay.

Theorem 4.22. (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 $\mathsf{init}(d') = addLog(\mathcal{L}(d), \mathsf{init}(d))$. 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 $del(\Gamma'; \Pi')$;

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'.

Proof:
From Lemma 4.20 there exists a forward derivation $d'' \approx d'$ such that $\mathsf{del}(d'') = d$. Trivially, the thesis holds for d''. Item (1) follows by applying local determinism (Lemma 4.21) to derivations d'' and d'.

Concerning item (2), every message in Γ should either be in $\mathsf{init}(d)$ or must be sent by a transition of d. In the first case, the claim follows since $del(\mathsf{init}(d')) = \mathsf{init}(d)$. In the second case, there should be a send action in d producing it, and the thesis follows from item (1), observing that since the derivation d' is fully-logged the send action is replayed. □

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, including going back and forward: 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 messages 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. Hence, such misbehaviors are not relevant in our context.

5. Controlled Replay/Rollback Semantics

In this section, we introduce a controlled version of the replay reversible semantics. The semantics in the previous section allows one to replay a given derivation, both forward and backward, 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 9 a semantics where the user can specify which actions (s)he wants to replay or undo, and the semantics takes care of replaying them or undoing them. Replaying an action requires to replay all *and only* its causes, while undoing an action requires to undo all *and only* its consequences. Notably, the bug causing a misbehavior causes the action showing the misbehavior.

Here, we consider that, given a system s, we want to start a replay (resp. rollback) until a particular action ψ is performed (resp. undone) on a given process p. We denote such a replay (resp. rollback) request with $\llparenthesis s \rrparenthesis_{(\{p,\psi\})}$ (resp. $\rceil\!\rceil s \lceil\!\lceil_{(\{p,\psi\})}$). In general, the subscript of $\llparenthesis\ \rrparenthesis$ (resp. $\rceil\!\rceil\ \lceil\!\lceil$) represents a sequence of requests that can be seen as a stack where the first element is the most recent request. In this paper, we consider the following rollback/replay requests:

- $\{p, \mathsf{s}\}$: one step backward/forward of process p;[9]
- $\{p, \ell^{\Uparrow}\}$: a backward/forward derivation of process p up to the sending of the message tagged with ℓ;

[9]The extension to n steps is straightforward. We omit it for simplicity.

REPLAY RULES:

$$\frac{\Gamma;\Pi \rightharpoonup_{p,r,\Psi'} \Gamma';\Pi' \quad \wedge \quad \psi \in \Psi'}{[\![\Gamma;\Pi]\!]_{\{p,\psi\}+\Psi} \rightsquigarrow [\![\Gamma';\Pi']\!]_{\Psi}} \qquad \frac{\Gamma;\Pi \rightharpoonup_{p,r,\Psi'} \Gamma';\Pi' \quad \wedge \quad \psi \notin \Psi'}{[\![\Gamma;\Pi]\!]_{\{p,\psi\}+\Psi} \rightsquigarrow [\![\Gamma';\Pi']\!]_{\{p,\psi\}+\Psi}}$$

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

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

ROLLBACK RULES:

$$\frac{\Gamma;\Pi \leftharpoondown_{p,r,\Psi'} \Gamma';\Pi' \quad \wedge \quad \psi \in \Psi'}{\lceil\!\lceil\Gamma;\Pi\rceil\!\rceil_{\{p,\psi\}+\Psi} \rightsquigarrow \lceil\!\lceil\Gamma';\Pi'\rceil\!\rceil_{\Psi}} \qquad \frac{\Gamma;\Pi \leftharpoondown_{p,r,\Psi'} \Gamma';\Pi' \quad \wedge \quad \psi \notin \Psi'}{\lceil\!\lceil\Gamma;\Pi\rceil\!\rceil_{\{p,\psi\}+\Psi} \rightsquigarrow \lceil\!\lceil\Gamma';\Pi'\rceil\!\rceil_{\{p,\psi\}+\Psi}}$$

$$\frac{\Gamma;\langle p,\omega,\mathsf{send}(\theta,e,p',\{v,\ell\})+h,\theta',e'\rangle \mid \Pi \not\leftharpoondown_{p,r,\Psi'}}{\begin{array}{l}\lceil\!\lceil\Gamma;\langle p,\omega,\mathsf{send}(\theta,e,p',\{v,\ell\})+h,\theta',e'\rangle \mid \Pi\rceil\!\rceil_{\{p,\psi\}+\Psi} \\ \quad \rightsquigarrow \lceil\!\lceil\Gamma;\langle p,\omega,\mathsf{send}(\theta,e,p',\{v,\ell\})+h,\theta',e'\rangle \mid \Pi\rceil\!\rceil_{(\{p',\ell^{\Downarrow}\},\{p,\psi\})+\Psi}\end{array}}$$

$$\frac{\Gamma;\langle p,\omega,\mathsf{spawn}(\theta,e,p')+h,\theta',e'\rangle \mid \Pi \not\leftharpoondown_{p,r,\Psi'}}{\begin{array}{l}\lceil\!\lceil\Gamma;\langle p,\omega,\mathsf{spawn}(\theta,e,p')+h,\theta',e'\rangle \mid \Pi\rceil\!\rceil_{\{p,\psi\}+\Psi} \\ \quad \rightsquigarrow \lceil\!\lceil\Gamma;\langle p,\omega,\mathsf{spawn}(\theta,e,p')+h,\theta',e'\rangle \mid \Pi\rceil\!\rceil_{(\{p',\mathsf{sp}\},\{p,\psi\})+\Psi}\end{array}}$$

$$\frac{}{\lceil\!\lceil\Gamma;\langle p,\omega,(),\theta',e'\rangle \mid \Pi\rceil\!\rceil_{\{p,\mathsf{sp}\}+\Psi} \rightsquigarrow \lceil\!\lceil\Gamma;\langle p,\omega,(),\theta',e'\rangle \mid \Pi\rceil\!\rceil_{\Psi}}$$

Figure 9. Controlled replay/rollback semantics

- $\{p,\ell^{\Downarrow}\}$: a backward/forward derivation of process p up to the reception of the message tagged with ℓ;
- $\{p,\mathsf{sp}_{p'}\}$: a backward/forward derivation of process p up to the spawning of the process with pid p'.
- $\{p,\mathsf{sp}\}$: a backward derivation of process p up to the point immediately after its creation;
- $\{p,X\}$: a backward derivation of process p up to the introduction of variable X.

We do not include the variable creations as 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. This is where the third element labeling the relations of the reversible semantics in Figures 5 and 7 comes into play. This third element is a set with the requests that are satisfied in the corresponding step.

Let us explain the rules of the controlled replay/rollback semantics in Fig. 9. Here, we assume that the computation always starts with a single request. We have the following possibilities:

- 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 of requests (first rule of both replay and rollback rules).

- If the desired process p can perform a step, but the step does not satisfy the request ψ, we update the system but keep the request in the stack (second rule of both replay and rollback rules).
- 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. For the replay semantics, 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 $\mathcal{L}(d)$.[10]

  For the rollback semantics, we have three rules: one to add a request to undo the receiving of a message whose sending we want to undo, one to undo the actions of a given process whose spawning we want to undo, and a final one to check that a process has reached its initial state (with an empty history), and the request $\{p, \mathsf{sp}\}$ can be removed. In this last case, the process p will actually be removed from the system when a request of the form $\{p', sp_p\}$ is on top of the stack.

The relation $\rightsquigarrow$ can be seen as a controlled version of the uncontrolled replay reversible 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. In order to formalize this claim we need some notation. Notions for derivations and transitions are easily extended to controlled derivations. We also need a notion of projection from controlled systems to uncontrolled systems:

$$uctrl(\llbracket \Gamma; \Pi \rrbracket_\Psi) = \Gamma; \Pi \qquad uctrl(\lceil\!\lceil \Gamma; \Pi \rceil\!\rceil_\Psi) = \Gamma; \Pi$$

The notion of projection trivially extends to derivations.

Theorem 5.1. (Soundness)
For each controlled derivation d, $uctrl(d)$ is an uncontrolled derivation.

Proof:
Trivial by inspection of the controlled rules, noting that each controlled rule either executes an uncontrolled step, or does some bookkeeping which is removed by function $uctrl$. □

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 $s = addLog(\mathcal{L}(d), \mathsf{init}(d))$ 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 4.22).

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

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

[10] The implementation of functions $sender$ and $parent$ requires some care: without additional information they need to explore the full log. However, explicit representation of the functions can be easily computed during logging, or by a single preprocessing pass on the log. Hence, we have a tradeof among time of replay, size of the log, and time of other phases.

Definition 5.2. A controlled system $c = [\![s]\!]_{(\{p,\psi\})}$ (resp. $c = \lceil\!\lceil s \rceil\!\rceil_{(\{p,\psi\})}$) is well initialized iff there exist a derivation d under the logging semantics, a system $s_0 = addLog(\mathcal{L}(d), \mathsf{init}(d))$, an uncontrolled derivation $s_0 \rightleftharpoons^* s$, and an uncontrolled forward (resp. backward) derivation from s satisfying $\{p, \psi\}$. A controlled derivation d is well initialized iff $\mathsf{init}(d)$ is well initialized.

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. For rollback requests the check can be done by inspecting the history.

We can now show that controlled derivations are finite.

Lemma 5.3. Let d be a well-initialized controlled derivation. Then d is finite.

Proof:
First, note that $uctrl(d)$ is finite. Indeed, for rollback request, the length is bounded by the total length of histories. For replay requests, we can always extend $uctrl(d)$ to a fully-logged derivation. From Lemma 4.13 there exists a derivation d' such that $\mathsf{del}(d')$ is the original logging derivation, hence d' is finite. Note that $uctrl(d)$ and d' are coinitial, hence by applying twice Theorem 4.14 and using transitivity we get $uctrl(d) \approx d'$. Since both derivations are forward, they can only differ for swaps of concurrent actions, hence they have the same length, as desired.

In addition to lifting the uncontrolled steps, the controlled semantics also takes some administrative steps. If we show that between each pair of uncontrolled steps there is a finite amount of administrative steps then the thesis follows. Let us consider the replay semantics. Administrative steps correspond to ask to send a message and ask to spawn a process. These are bound, respectively, by the number of messages and the number of processes in the log. Let us now consider the rollback semantics. The last rule can be applied only a finite number of times since it removes one rollback request. We also have rules asking to rollback a process to the beginning and to undo a receive, but they are bound, respectively, by the number of processes and the number of messages. The thesis follows. □

We can now show that all the uncontrolled transitions executed as a consequence of a replay/rollback request depend on the action that needs to be replayed/undone.

Theorem 5.4. For each well-initialized controlled system $c = \lceil\!\lceil \Gamma; \Pi \rceil\!\rceil_{(\{p,\psi\})}$ (resp. $c = [\![\Gamma; \Pi]\!]_{(\{p,\psi\})}$), consider a maximal derivation d with $\mathsf{init}(d) = c$. Let us call t the last transition in $uctrl(d)$. We have that t satisfies $\{p, \psi\}$, and for each transition t' in $uctrl(d)$, $t' \rightsquigarrow t$ (resp. $t \rightsquigarrow t'$).

Proof:
In both the cases, we can prove that t satisfies the request $\{p, \psi\}$ by inspection of the rules, since a derivation only terminates when the request at the bottom of the stack is removed, and this is always the original request $\{p, \psi\}$. In order to show this we need to show that the controlled semantics never gets stuck otherwise, namely that if the uncontrolled semantics gets stuck a new request is generated. This can be shown by contrasting uncontrolled and controlled rules.

For the second part of the thesis, let us consider the replay semantics. We will show two invariants of the derivation. First, consider transitions t_1 and t_2 satisfying two replay requests $\{p_1, \psi_1\}$ and

$\{p_2, \psi_2\}$ on the stack, such that $\{p_1, \psi_1\}$ is on top of $\{p_2, \psi_2\}$. Then $t_1 \rightsquigarrow t_2$. Second if t_1 satisfies the request on top of the stack, and transition t_3 is performed, then $t_3 \rightsquigarrow t_1$. Both the invariants can be proved by inspection of the rules. The thesis then follows by transitivity of $\rightsquigarrow$.

The case of the rollback semantics is dual to the one above. □

We conclude this section by showing that a controlled derivation causes an uncontrolled derivation satisfying the given request which is minimal. We first need some confluence results.

Proposition 5.5. (Confluence)
Let s be a system in the uncontrolled replay semantics. If $s \rightleftharpoons^* s_1$ and $s \rightleftharpoons^* s_2$ then (backward confluence) there exists s_3 such that $s_1 \leftharpoondown^* s_3$ and $s_2 \leftharpoondown^* s_3$ and (forward confluence) there exists s_4 such that $s_1 \rightharpoondown^* s_4$ and $s_2 \rightharpoondown^* s_4$.

Proof:
Let s_0 be the system obtained from s by undoing all actions. Consider the derivation $s_0 \rightharpoondown^* s \rightleftharpoons^* s_1$, where the first part exists from the loop lemma (Lemma 4.6). From the rearranging lemma (Lemma 4.15) we have $s_0 \leftharpoondown^* \rightharpoondown^* s_1$. Since there is no possible backward transition from s_0 we have $s_0 \rightharpoondown^* s_1$. Similarly, we get $s_0 \rightharpoondown^* s_2$. Backward confluence follows from the loop lemma.

For forward confluence, extend $s \rightleftharpoons^* s_1$ and $s \rightleftharpoons^* s_2$ by forward actions to get fully-logged derivations d_1 and d_2. Derivations $\overline{s_1 \leftharpoondown^* s_3}; d_1$ and $\overline{s_2 \leftharpoondown^* s_3}; d_2$ are coinitial and fully-logged, hence from Corollary 4.19 they are also cofinal. Forward confluence follows. □

Theorem 5.6. (Minimality)
Let d be a well-initialized controlled derivation such as $\mathsf{init}(d) = \llbracket s \rrbracket_{(\{p,\psi\})}$ or $\mathsf{init}(d) = \lceil\!\lceil s \rceil\!\rceil_{(\{p,\psi\})}$. Derivation $uctrl(d)$ has minimal length among all uncontrolled derivations d' with $\mathsf{init}(d') = s$ including at least one transition satisfying the request $\{p, \psi\}$.

Proof:
The proofs for the two cases are dual.

Take an uncontrolled derivation d' satisfying the premises. By definition $uctrl(d)$ and d' are coinitial. We can assume that there is in d' a unique transition satisfying the request, and that it is the last transition in d'. Let us focus on the second case. For forward (resp. backward) derivations, by forward (resp. backward) confluence (Prop. 5.5) we can extend the derivations to cofinal derivations $d'; d''$ and $uctrl(d); d'''$ with d'' and d''' forward (resp. backward). Thanks to the shortening lemma (Lemma 4.16) we can assume $d'; d''$ to be forward (resp. backward) too. By causal consistency the two derivations are causally equivalent, and since they are forward (resp. backward) they differ only for swaps of concurrent actions. Note also that for each request there is a unique action satisfying it (the step of a process after/before a given one is unique, and other requests are determined by process identifiers, message identifiers and variables), hence there is a sequence of swaps of independent transitions transforming $d'; d''$ into $uctrl(d); d'''$. Assume towards a contradiction $length(d') < length(uctrl(d))$. Then t in d' must be swapped with some of the transitions preceding (resp. following) t in $uctrl(d)$, but this is impossible thanks to Theorem 5.4. □

6. Related Work

As mentioned in the Introduction, this paper is a revised and extended version of [16], where causal-consistent replay has been introduced. However, [16] concentrates on replay and mentions reversibility only briefly, while the present paper integrates the notion of causal-consistent replay with the reversible semantics in [12].

Our replay mechanism is strongly related (indeed dual) to causal-consistent rollback, and its instance on debugging, namely causal-consistent reversible debugging. Causal-consistent reversible debugging has been introduced in [5] in the context of the *toy* language μOz, and, beyond this, it has only been used so far in the CauDEr [7, 28] debugger for Erlang. Causal-consistent rollback has also been studied in the context of the process calculus HOπ [6] and the coordination language Klaim [11]. We refer to [5] for a description of the relations between causal-consistent debugging and other forms of reversible debugging.

Beyond CauDEr, the only reversible debugger for actor systems we are aware of is Actoverse [29], for Akka-based applications. It provides many relevant features complementary to ours, such as a partial-order graphical representation of message exchanges that would nicely match our causal-consistent approach. On the other side, Actoverse has several limitations. For instance, its facilities to replay misbehaviors, such as message-oriented breakpoints to force specific message interleavings and support for session replay, are more limited than our causal-consistent replay. Furthermore, it allows one to explore only some states of the computation, such as the ones corresponding to message sending and receiving.

Another interesting related work is [30], where an approach to record and replay for actor languages is introduced. 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 [30] in order to produce more compact logs.

At the semantic level, the work closer to ours is the reversible semantics for Erlang in [12]. However, both our logging semantics and our replay reversible semantics do not consider queues in the processes nor a rule $Sched$ to deliver messages from Γ to the local queue of a process. This might seem a minor change, but it has a significant effect: the notion of concurrency is much more natural, as explained in Section 2.3, since we do not need to consider that moving a message from Γ to a local queue and receiving a message are in conflict. This conflict was artificially introduced in [12] because of the design of the semantics therein. Moreover, while the reversible semantics in [12] basically models a reversible interpreter for the language, our replay reversible semantics is driven by the log of an actual execution. Finally, our controlled semantics, built on top of the uncontrolled reversible semantics, is much simpler and elegant than the low-level controlled semantics in [12] which, anyway, is based on undoing the actions of an execution up to a given checkpoint (rollback requests were later introduced in [7]).

None of the works above treats the possibility of a causal-consistent replay and, as far as we know, such notion has never been explored in the (huge) literature on replay. For instance, no reference to it appears in a recent survey [31]. According to the terminology in the survey, our approach is classified 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).

Main distinctive traits of our work, beyond considering causal-consistent replay, are a formal characterization of the approach at the semantic level, and its instantiation to the Erlang language. A related work using the ordering-based scheme is [19]: it provides an interesting technique based on race detection to avoid logging all message exchanges, that we may try to integrate in our approach in the future (though it considers only systems with a fixed number of processes). A content-based work is [32] 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 [33], since it also deals with concurrent systems. Both the 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.

Techniques similar to the ones we used for rollback can also be used to define and implement safe sessions, where a computation that fails can automatically be re-executed, thus managing transient faults. Such an approach has been used, e.g., in Stabilizers [34] and Transactors [35].

7. Conclusions and Future Work

In this work, we have introduced the notion of causal-consistent replay, which is dual to the recent notion of causal-consistent reversibility. Our framework considers a functional and concurrent programming language based on message passing. Indeed, our language is close to Erlang, thus providing an excellent background for the development of a causal-consistent replay debugger for this language. Nevertheless, the basic ideas are applicable to other concurrent languages and calculi based on message passing. In principle, it 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). In our framework, the actual execution of a program produces some logs that can be used to replay this particular execution, or a causally equivalent one, back and forth. Moreover, we have proved that the same misbehaviors appear in the logged derivation and in all causally equivalent replays. Therefore, the user can focus on the actions and processes of interest, and still be able to find the bug. For this purpose, we have introduced a replay/rollback semantics controlled by the user requests which is sound and minimal. Thus, it constitutes an excellent basis for the implementation of a causal-consistent replay debugger.

We have undertaken the development of a proof-of-concept implementation of a replay debugging tool for Erlang, which is based on the developments presented in this work. In particular, we are working on an extension of our debugger CauDEr [7, 28] to accept both a source program and a log of a computation. Furthermore, we will incorporate all replay and rollback requests introduced in Section 5.

References

[1] Undo Software. Increasing software development productivity with reversible debugging, 2014. URL `https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf`.

[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] Sutter H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 2005. **30**(3).

[4] Huang J, Zhang C. Debugging Concurrent Software: Advances and Challenges. *J. Comput. Sci. Technol.*, 2016. **31**(5):861–868.

[5] Giachino E, Lanese I, Mezzina CA. Causal-Consistent Reversible Debugging. In: Gnesi S, Rensink A (eds.), Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE 2014), volume 8411 of *Lecture Notes in Computer Science*. Springer, 2014 pp. 370–384.

[6] Lanese I, Mezzina CA, 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), volume 6901 of *Lecture Notes in Computer Science*. Springer, 2011 pp. 297–311.

[7] Lanese I, Nishida N, Palacios A, Vidal G. CauDEr: A Causal-Consistent Reversible Debugger for Erlang (system description). In: Gallagher JP, Sulzmann M (eds.), Proceedings of the 14th International Symposium on Functional and Logic Programming (FLOPS'18), volume 10818 of *Lecture Notes in Computer Science*. Springer, 2018 pp. 247–263.

[8] Cesarini F, Thompson S. Erlang Programming - A Concurrent Approach to Software Development. O'Reilly, 2009. ISBN 978-0-596-51818-9.

[9] Letuchy E. Erlang at Facebook. `http://www.erlang-factory.com/conference/SFBayAreaErlangFactory2009/speakers/EugeneLetuchy`, 2009.

[10] Lienhardt M, Lanese I, Mezzina CA, Stefani JB. A Reversible Abstract Machine and Its Space Overhead. In: Giese H, Rosu G (eds.), Proceedings of the Joint 14th IFIP WG International Conference on Formal Techniques for Distributed Systems (FMOODS 2012) and the 32nd IFIP WG 6.1 International Conference (FORTE 2012), volume 7273 of *Lecture Notes in Computer Science*. Springer, 2012 pp. 1–17.

[11] Giachino E, Lanese I, Mezzina CA, Tiezzi F. Causal-consistent rollback in a tuple-based language. *J. Log. Algebr. Meth. Program.*, 2017. **88**:99–120.

[12] Lanese I, Nishida N, Palacios A, Vidal G. A Theory of Reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming*, 2018. **100**:71–97.

[13] Svensson H, Fredlund LA, Earle CB. A unified semantics for future Erlang. In: 9th ACM SIGPLAN workshop on Erlang. ACM, 2010 pp. 23–32.

[14] 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), volume 10184 of *Lecture Notes in Computer Science*. Springer, 2017 pp. 259–274.

[15] Carlsson R, Gustavsson B, Johansson E, Lindgren T, Nyström SO, 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`.

[16] Lanese I, Palacios A, Vidal G. Causal-Consistent Replay Debugging for Message Passing Programs. In: Pérez JA, Yoshida N (eds.), Proceedings of the 39th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2019), volume 11535 of *Lecture Notes in Computer Science*. Springer, 2019 pp. 167–184.

[17] Lanese I, Palacios A, Vidal G. Causal-Consistent Replay Reversible Semantics for Message Passing Concurrent Programs. Technical report, DSIC, Universitat Politècnica de València, 2019. URL `http://personales.upv.es/~gvidal/german/fi/paper.pdf`.

[18] Frequently Asked Questions about Erlang. Available at `http://erlang.org/faq/academic.html`, 2018.

[19] Netzer RH, Miller BP. Optimal tracing and replay for debugging message-passing parallel programs. *The Journal of Supercomputing*, 1995. **8**(4):371–388.

[20] Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 1978. **21**(7):558–565.

[21] Mazurkiewicz AW. 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, volume 255 of *Lecture Notes in Computer Science*. Springer, 1987 pp. 279–324.

[22] Lanese I, Mezzina CA, Tiezzi F. Causal-Consistent Reversibility. *Bulletin of the EATCS*, 2014. **114**.

[23] Landauer R. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 1961. **5**:183–191.

[24] Matsuda K, Hu Z, Nakano K, Hamana M, Takeichi M. Bidirectionalization transformation based on automatic derivation of view complement functions. In: Hinze R, Ramsey N (eds.), Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007. ACM, 2007 pp. 47–58.

[25] Nishida N, Palacios A, Vidal G. Reversible Term Rewriting. In: Kesner D, Pientka B (eds.), Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016), volume 52 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016 pp. 28:1–28:18.

[26] Thomsen MK, Axelsen HB. Interpretation and programming of the reversible functional language RFUN. In: Proc. of the 27th International Symposium on Implementation and Application of Functional Languages (IFL 2015). ACM, 2016 pp. 8:1 – 8:13.

[27] Danos V, Krivine J. Reversible Communicating Systems. In: CONCUR, volume 3170 of *LNCS*. Springer, 2004 pp. 292–307.

[28] Lanese I, Nishida N, Palacios A, Vidal G. CauDEr website. URL: `https://github.com/mistupv/cauder`, 2018.

[29] Shibanai K, Watanabe T. Actoverse: A Reversible Debugger for Actors. In: AGERE. ACM, 2017 pp. 50–57.

[30] Aumayr D, Marr S, Béra C, Boix EG, 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). ACM, 2018 pp. 15:1–15:14.

[31] Chen Y, Zhang S, Guo Q, Li L, Wu R, Chen T. Deterministic Replay: A Survey. *ACM Comput. Surv.*, 2015. **48**(2):17:1–17:47.

[32] Maruyama M, Tsumura T, Nakashima H. Parallel Program Debugging based on Data-Replay. In: Zheng SQ (ed.), Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005). IASTED/ACTA Press, 2005 pp. 151–156.

[33] Perera R, Garg D, Cheney J. Causally Consistent Dynamic Slicing. In: Desharnais J, Jagadeesan R (eds.), CONCUR, volume 59 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016 pp. 18:1–18:15.

[34] Ziarek L, Schatz P, Jagannathan S. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In: Reppy JH, Lawall JL (eds.), Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006. ACM, 2006 pp. 136–147.

[35] Field J, Varela CA. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In: Palsberg J, Abadi M (eds.), Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005). ACM, 2005 pp. 195–208.