Journal of Logical and Algebraic Methods in Programming 100 (2018) 71–97

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

A theory of reversibility for Erlang ☆

Ivan Lanese [a], Naoki Nishida [b], Adrián Palacios [c], Germán Vidal [c,*]

[a] Focus Team, University of Bologna/INRIA, Mura Anteo Zamboni, 7, Bologna, Italy
[b] Graduate School of Informatics, Nagoya University, Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
[c] MiST, DSIC, Universitat Politècnica de València, Camino de Vera, s/n, 46022 Valencia, Spain

ARTICLE INFO

Article history:
Received 25 July 2017
Accepted 13 June 2018
Available online xxxx

Keywords:
Reversible computation
Actor model
Concurrency
Rollback recovery

ABSTRACT

In a reversible language, any forward computation can be undone by a finite sequence of backward steps. Reversible computing has been studied in the context of different programming languages and formalisms, where it has been used for testing and verification, among others. In this paper, we consider a subset of Erlang, a functional and concurrent programming language based on the actor model. We present a formal semantics for reversible computation in this language and prove its main properties, including its causal consistency. We also build on top of it a rollback operator that can be used to undo the actions of a process up to a given checkpoint.

1. Introduction

Let us consider that the operational semantics of a programming language is specified by a state transition relation R such that $R(s, s')$ holds if the state s' is reachable—in one step—from state s. Then, we say that a programming language (or formalism) is *reversible* if there exists a constructive algorithm that can be used to recover the predecessor state s from s'. In general, such a property does not hold for most programming languages and formalisms. We refer the interested reader to, e.g., [4,12,33,34] for a high level account of the principles of reversible computation.

The notion of *reversible computation* was first introduced in Landauer's seminal work [17] and, then, further improved by Bennett [3] in order to avoid the generation of "garbage" data. The idea underlying these works is that any programming language or formalism can be made reversible by adding the *history* of the computation to each state, which is usually called a *Landauer embedding*. Although carrying the history of a computation might seem infeasible because of its size, there are several successful proposals that are based on this idea. In particular, one can restrict the original language or apply a number of analysis in order to restrict the required information in the history as much as possible, as in, e.g., [24,26,31] in the context of a functional language.

☆ This work has been partially supported by MINECO/AEI/FEDER (EU) under grants TIN2013-44742-C4-1-R and TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic), by the COST Action IC1405 on Reversible Computation—extending horizons of computing, and by JSPS KAKENHI Grant Number JP17H01722. Adrián Palacios was partially supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores* and *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D*, MINECO (SEIDI), under FPI grants BES-2014-069749 and EEBB-I-16-11469. Ivan Lanese was partially supported by INdAM as a member of GNCS (Gruppo Nazionale per il Calcolo Scientifico). Part of this research was done while the third and fourth authors were visiting Nagoya and Bologna Universities; they gratefully acknowledge their hospitality. Finally, we thank Salvador Tamarit and the anonymous reviewers for their helpful suggestions and comments.

* Corresponding author.
E-mail addresses: ivan.lanese@gmail.com (I. Lanese), nishida@i.nagoya-u.ac.jp (N. Nishida), apalacios@dsic.upv.es (A. Palacios), gvidal@dsic.upv.es (G. Vidal).

https://doi.org/10.1016/j.jlamp.2018.06.004
2352-2208/

In this paper, we aim at introducing a form of reversibility in the context of a programming language that follows the actor model (concurrency based on message passing), a first-order subset of the concurrent and functional language Erlang [1]. Previous approaches have mainly considered reversibility in—mostly synchronous—concurrent calculi like CCS [9,10] and π-calculus [8]; a general framework for reversibility of algebraic process calculi [28], or the recent approach to reversible session-based π-calculus [32]. However, we can only find a few approaches that considered the reversibility of *asynchronous* calculi, e.g., Cardelli and Laneve's reversible structures [6], and reversible extensions of the concurrent functional language μ Oz [23], of a higher-order asynchronous π-calculus [19], and of the coordination language μ Klaim [15]. In the last two cases, a form of control of the backward execution using a rollback operator has also been studied [18,15]. In the case of μOz, reversibility has been exploited for debugging [14].

To the best of our knowledge, our work is the first one that considers reversibility in the context of the functional, concurrent, and distributed language Erlang. Here, given a running Erlang system consisting of a pool of interacting processes, possibly distributed in several computers, we aim at allowing a *single* process to undo its actions in a stepwise manner, including the interactions with other processes, following a rollback fashion. In this context, we must ensure *causal consistency* [9], i.e., an action cannot be undone until all the actions that depend on it have already been undone. E.g., if a process p1 spawns a process p2, we cannot undo the spawning of process p2 until all the actions performed by the process p2 are undone too. This is particularly challenging in an asynchronous and distributed setting, where ensuring causal consistency for backward computations is far from trivial.

In this paper, we consider a simple Erlang-like language that can be seen as a subset of *Core Erlang* [7]. We present the following contributions:

- First, we introduce an appropriate semantics for the language. In contrast to previous semantics like that in [5] which were monolithic, ours is modular, which simplifies the definition of a reversible extension. Here, we follow some of the ideas in [30], e.g., the use of a global mailbox (there called "ether"). There are also some differences though. In the semantics of [30], at the expression level, the semantics of a receive statement is, in principle, infinitely branching, since their formulation allows for an infinite number of possible queues and selected messages (see [13, page 53] for a detailed explanation). This source of nondeterminism is avoided in our semantics.
- We then introduce a reversible semantics that can go both forward and backward (basically, a Landauer embedding), in a nondeterministic fashion, called an *uncontrolled* reversible semantics according to the terminology in [20]. Here, we focus on the concurrent actions (namely, process spawning, message sending and receiving) and, thus, we do not define a reversible semantics for the functional component of the language; rather, we assume that the state of the process—the current expression and its environment—is stored in the history after each execution step. This approach could be improved following, e.g., the techniques presented in [24,26,31]. We state and formally prove several properties of the semantics and, particularly, its causal consistency.
- Finally, we add control to the reversible semantics by introducing a *rollback operator* that can be used to undo the actions of a given process until a given checkpoint—introduced by the programmer—is reached. In order to ensure causal consistency, the rollback action might be propagated to other, dependent processes.

This paper is an extended version of [27]. Compared to [27], we introduce an uncontrolled reversible semantics and prove a number of fundamental theoretical properties, including its causal consistency. The rollback semantics, originally introduced in [27], has been refined and improved (see Section 7 for more details).

The paper is organised as follows. The syntax and semantics of the considered language are presented in Sections 2 and 3, respectively. Our (uncontrolled) reversible semantics is then introduced in Section 4, while the rollback operator is defined in Section 5. A proof-of-concept implementation of the reversible semantics is described in Section 6. Finally, some related work is discussed in Section 7, and Section 8 concludes and points out some directions for future work.

2. Language syntax

In this section, we present the syntax of a first-order concurrent and distributed functional language that follows the actor model. Our language is equivalent to a subset of Core Erlang [7].

The syntax of the language can be found in Fig. 1. Here, a module is a sequence of function definitions, where each function name f/n (atom/arity) has an associated definition of the form $\mathsf{fun}\ (X_1, \ldots, X_n) \to e$. We consider that a program consists of a single module for simplicity. The body of a function is an *expression*, which can include variables, literals, function names, lists, tuples, calls to built-in functions—mainly arithmetic and relational operators—, function applications, case expressions, let bindings, and receive expressions; furthermore, we also include the functions spawn, "!" (for sending a message), and self() that are usually considered built-ins in the Erlang language. As is common practice, we assume that X is a fresh variable in a let binding of the form $\mathsf{let}\ X = expr_1\ \mathsf{in}\ expr_2$.

As shown by the syntax in Fig. 1, we only consider first-order expressions. Therefore, the first argument in applications and spawns is a function name (instead of an arbitrary expression or closure). Analogously, the first argument in calls is a built-in operation *Op*.

In this language, we distinguish expressions, patterns, and values. Here, *patterns* are built from variables, literals, lists, and tuples, while *values* are built from literals, lists, and tuples, i.e., they are *ground*—without variables—patterns. Expressions are

$$
\begin{array}{rcl}
module & ::= & \text{module } Atom = fun_1 \ \ldots \ fun_n \\
fun & ::= & fname = \text{fun } (Var_1, \ldots, Var_n) \rightarrow expr \\
fname & ::= & Atom/Integer \\
lit & ::= & Atom \mid Integer \mid Float \mid Pid \mid [\,] \\
expr & ::= & Var \mid lit \mid fname \mid [expr_1|expr_2] \mid \{expr_1, \ldots, expr_n\} \\
 & \mid & \text{call } Op\ (expr_1, \ldots, expr_n) \mid \text{apply } fname\ (expr_1, \ldots, expr_n) \\
 & \mid & \text{case } expr \text{ of } clause_1; \ldots; clause_m \text{ end} \\
 & \mid & \text{let } Var = expr_1 \text{ in } expr_2 \mid \text{receive } clause_1; \ldots; clause_n \text{ end} \\
 & \mid & \text{spawn}(fname, [expr_1, \ldots, expr_n]) \mid expr\,!\,expr \mid \text{self}() \\
clause & ::= & pat \text{ when } expr_1 \rightarrow expr_2 \\
pat & ::= & Var \mid lit \mid [pat_1|pat_2] \mid \{pat_1, \ldots, pat_n\}
\end{array}
$$

Fig. 1. Language syntax rules.

denoted by $e, e', e_1, e_2, \ldots$, patterns by pat, pat', pat_1, $pat_2, \ldots$ and values by $v, v', v_1, v_2, \ldots$ Atoms are typically denoted with roman letters, while variables start with an uppercase letter. As it is common practice, a *substitution* θ is a mapping from variables to expressions, and $\mathcal{D}om(\theta) = \{X \in Var \mid X \neq \theta(X)\}$ is its domain.[1] Substitutions are usually denoted by sets of bindings like, e.g., $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$. Substitutions are extended to morphisms from expressions to expressions in the natural way. The identity substitution is denoted by id. Composition of substitutions is denoted by juxtaposition, i.e., $\theta\theta'$ denotes a substitution θ'' such that $\theta''(X) = \theta'(\theta(X))$ for all $X \in Var$. Also, we denote by $\theta[X_1 \mapsto v_1, \ldots, X_n \mapsto v_n]$ the *update* of θ with the mapping $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$, i.e., it denotes a new substitution θ' such that $\theta'(X) = v_i$ if $X = X_i$, for some $i \in \{1, \ldots, n\}$, and $\theta'(X) = \theta(X)$ otherwise.

In a case expression "case e of pat_1 when $e_1 \rightarrow e'_1$; $\ldots$; pat_n when $e_n \rightarrow e'_n$ end", we first evaluate e to a value, say v; then, we should find (if any) the first clause pat_i when $e_i \rightarrow e'_i$ such that v matches pat_i (i.e., there exists a substitution σ for the variables of pat_i such that $v = pat_i\sigma$) and $e_i\sigma$—the *guard*—reduces to *true*; then, the case expression reduces to $e'_i\sigma$. Note that guards can only contain calls to built-in functions (typically, arithmetic and relational operators).

As for the concurrent features of the language, we consider that a *system* is a pool of processes that can only interact through message sending and receiving (i.e., there is no shared memory). Each process has an associated *pid* (process identifier), which is unique in a system. As in Erlang, we consider a specific type or domain Pid for pids. Furthermore, in this work, we assume that pids can only be introduced in a computation from the evaluation of functions spawn and self (see below). By abuse of notation, when no confusion can arise, we refer to a process with its pid.

An expression of the form spawn$(f/n, [e_1, \ldots, e_n])$ has, as a *side effect*, the creation of a new process, with a fresh pid p, initialised with the expression apply f/n $(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ are the evaluations of $e_1, \ldots, e_n$, respectively; the expression spawn$(f/n, [e_1, \ldots, e_n])$ itself evaluates to the new pid p. The function self() just returns the pid of the current process. An expression of the form $e_1\,!\,e_2$, where e_1 evaluates to a pid p and e_2 to a value v, also evaluates to the value v and, as a side effect, the value v—the *message*—will be stored in the queue or *mailbox* of process p at some point in the future.

Finally, an expression "receive pat_1 when $e_1 \rightarrow e'_1; \ldots;$ pat_n when $e_n \rightarrow e'_n$ end" traverses the messages in the process' queue until one of them matches a branch in the receive statement; i.e., it should find the *first* message v in the process' queue (if any) such that case v of pat_1 when $e_1 \rightarrow e'_1; \ldots;$ pat_n when $e_n \rightarrow e'_n$ end can be reduced; then, the receive expression evaluates to the same expression to which the above case expression would be evaluated, with the additional side effect of deleting the message v from the process' queue. If there is no matching message in the queue, the process suspends its execution until a matching message arrives.

Example 1. Consider the program shown in Fig. 2, where the symbol "_" is used to denote an *anonymous* variable, i.e., a variable whose name is not relevant. The computation starts with "apply main/0 ()." This creates a process, say p1. Then, p1 spawns two new processes, say p2 and p3, and then sends the message hello to process p3 and the message {p3, world} to process p2, which then resends world to p3. Note that we consider that variables $P2$ and $P3$ are bound to pids p2 and p3, respectively.

In our language, there is no guarantee regarding which message arrives first to p3, i.e., both interleavings (a) and (b) in Fig. 3 are possible (resulting in function target/0 returning either {hello, world} or {world, hello}). This is coherent with the semantics of Erlang, where the only guarantee is that if two messages are sent from process p to process p', and both are delivered, then the order of these messages is kept.[2]

3. The language semantics

In order to precisely set the framework for our proposal, in this section we formalise the semantics of the considered language.

[1] Since we consider an eager language, variables are bound to values.

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

```
main/0 = fun () → let P2 = spawn(echo/0, [ ])
                  in let P3 = spawn(target/0, [ ])
                  in let _ = P3 ! hello
                  in P2 ! {P3, world}
target/0 = fun () → receive
                      A → receive
                            B → {A, B}
                          end
                  end
echo/0 = fun () → receive
                      {P, M} → P ! M
                  end
```

Fig. 2. A simple concurrent program.

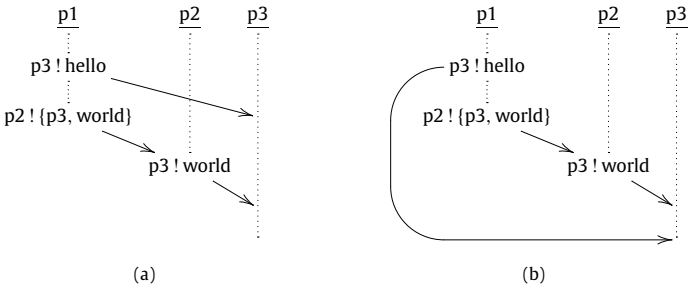


Fig. 3. Admissible interleavings in Example 1.

Definition 2 *(Process).* A process is denoted by a tuple $\langle p, (\theta, e), q\rangle$ where p is the pid of the process, (θ, e) is the control—which consists of an environment (a substitution) and an expression to be evaluated—and q is the process' mailbox, a FIFO queue with the sequence of messages that have been sent to the process.

We consider the following operations on local mailboxes. Given a message v and a local mailbox q, we let $v : q$ denote a new mailbox with message v on top of it (i.e., v is the newer message). We also denote with $q \setminus\setminus v$ a new queue that results from q by removing the oldest occurrence of message v (which is not necessarily the oldest message in the queue).

A running *system* can then be seen as a pool of processes, which we formally define as follows:

Definition 3 *(System).* A system is denoted by $\Gamma; \Pi$, where Γ, the *global mailbox*, is a multiset of pairs of the form $(target\_process\_pid, message)$, and Π is a pool of processes, denoted by an expression of the form

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

where "$\mid$" denotes an associative and commutative operator. Given a global mailbox Γ, we let $\Gamma \cup \{(p, v)\}$ denote a new mailbox also including the pair (p, v), where we use "$\cup$" as multiset union.

We often denote a system by an expression of the form $\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi$ to point out that $\langle p, (\theta, e), q\rangle$ is an arbitrary process of the pool (thanks to the fact that "$\mid$" is associative and commutative).

Intuitively, Γ stores messages after they are sent, and before they are inserted in the target mailbox, hence it models messages which are in the network. The use of Γ (which is similar to the "ether" in [30]) is needed to guarantee that all message interleavings admissible in an asynchronous communication model (where the order of messages is not preserved) can be generated by our semantics.

In the following, we denote by $\overline{o_n}$ a sequence of syntactic objects $o_1, \ldots, o_n$ for some n. We also write $\overline{o_{i,j}}$ for the sequence $o_i, \ldots, o_j$ when $i \leq j$ (and the empty sequence otherwise). We write $\overline{o}$ when the number of elements is not relevant.

The semantics is defined by means of two transition relations: $\longrightarrow$ for expressions and $\hookrightarrow$ for systems. Let us first consider the labelled transition relation

$$\longrightarrow : (Env, Exp) \times Label \times (Env, Exp)$$

where Env and Exp are the domains of environments (i.e., substitutions) and expressions, respectively, and $Label$ denotes an element of the set

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

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

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

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

$$(Case1)\ \frac{\theta, e \xrightarrow{\ell} \theta', e'}{\theta, \mathsf{case}\ e\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\ell} \theta', \mathsf{case}\ e'\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end}}$$

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

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

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

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

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

Fig. 4. Standard semantics: evaluation of sequential expressions.

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

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

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

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

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

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

Fig. 5. Standard semantics: evaluation of concurrent expressions.

whose meaning will be explained below. We use ℓ to range over labels. For clarity, we divide the transition rules of the semantics for expressions in two sets: rules for sequential expressions are depicted in Fig. 4, while rules for concurrent ones are in Fig. 5.[3] Note, however, that concurrent expressions can occur inside sequential expressions.

Most of the rules are self-explanatory. In the following, we only discuss some subtle or complex issues. In principle, the transitions are labelled either with τ (a sequential reduction without side effects) or with a label that identifies the reduction of a (possibly concurrent) action with some side-effects. Labels are used in the system rules (Fig. 6) to determine the associated side effects and/or the information to be retrieved.

As in Erlang, we consider that the order of evaluation of the arguments in a tuple, list, etc., is fixed from left to right.

[3] By abuse, we include the rule for self() together with the concurrent actions.

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

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

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

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

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

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

Fig. 6. Standard semantics: system rules.

For case evaluation, we assume an auxiliary function match which selects the first clause, $cl_i = (pat_i \text{ when } e'_i \to e_i)$, such that v matches pat_i, i.e., $v = \theta_i(pat_i)$, and the guard holds, i.e., $\theta\theta_i, e'_i \longrightarrow^* \theta', true$. As in Core Erlang, we assume that the patterns can only contain fresh variables (but guards might have bound variables, thus we pass the current environment θ to function match). Note that, for simplicity, we assume here that if the argument v matches no clause then the evaluation is blocked.[4]

Functions can either be defined in the program (in this case they are invoked by apply) or be a built-in (invoked by call). In the latter case, they are evaluated using the auxiliary function eval. In rule *Apply2*, we consider that the mapping μ stores all function definitions in the program, i.e., it maps every function name a/n to a copy of its definition 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. As for the applications, note that we only consider first-order functions. In order to extend our semantics to also consider higher-order functions, one should reduce the function name to a *closure* of the form $(\theta', \mathsf{fun}\ (X_1, \ldots, X_n) \to e)$. We skip this extension since it is orthogonal to our contribution.

Let us now consider the evaluation of concurrent expressions that produce some side effect (Fig. 5). Here, we can distinguish two kinds of rules. On the one hand, we have rules *Send1*, *Send2* and *Send3* for "!". In this case, we know *locally* what the expression should be reduced to (i.e., v_2 in rule *Send3*). For the remaining rules, this is not known locally and, thus, we return a fresh distinguished symbol, κ—by abuse, κ is dealt with as a variable—so that the system rules of Fig. 6 will eventually bind κ to its correct value[5]: the selected expression in rule *Receive* and a pid in rules *Spawn* and *Self*. In these cases, the label of the transition contains all the information needed by system rules to perform the evaluation at the system level, including the symbol κ. This *trick* allows us to keep the rules for expressions and systems separated (i.e., the semantics shown in Figs. 4 and 5 is mostly independent of the rules in Fig. 6), in contrast to other Erlang semantics, e.g., [5], where they are combined into a single transition relation.

Finally, we consider the system rules, which are depicted in Fig. 6. In most of the transition rules, we consider an arbitrary system of the form $\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi$, where Γ is the global mailbox and $\langle p, (\theta, e), q\rangle \mid \Pi$ is a pool of processes that contains at least one process $\langle p, (\theta, e), q\rangle$. Let us briefly describe the system rules.

Rule *Seq* just updates the control (θ, e) of the considered process when a sequential expression is reduced using the expression rules.

Rule *Send* adds the pair (p'', v) to the global mailbox Γ instead of adding it to the queue of process p''. This is necessary to ensure that all possible message interleavings are correctly modelled (as discussed in Example 1). Observe that e' is usually different from v since e may have different nested operators. E.g., if e has the form "case p ! v of {...}," then e' will be "case v of {...}" with label send(p, v).

In rule *Receive*, we use the auxiliary function matchrec to evaluate a receive expression. The main difference w.r.t. match is that matchrec also takes a queue q and returns the selected message v. More precisely, function matchrec scans the queue q looking for the *first* message v matching a pattern of the receive statement. Then, κ is bound to the expression in the selected clause, e_i, and the environment is extended with the matching substitution. If no message in the queue q matches any clause, then the rule is not applicable and the selected process cannot be reduced (i.e., it suspends). As in case expressions, we assume that the patterns can only contain fresh variables.

[4] This is not an issue in practice since, when an Erlang program is translated to the intermediate representation Core Erlang, a catch-all clause is added to every case expression in order to deal with pattern matching errors.

[5] Note that κ takes values on the domain $expr \cup Pid$, in contrast to ordinary variables that can only be bound to values.

$$
\begin{array}{lll}
 & \{\}; & \langle \mathrm{p1}, (id, \underline{\mathsf{apply\ main/0\ ()}}), [\,] \rangle \\
\hookrightarrow_{Seq} & \{\}; & \langle \mathrm{p1}, (id, \mathsf{let}\ P2 = \underline{\mathsf{spawn(echo/0, [\,])}}\ \mathsf{in}\ \ldots), [\,] \rangle \\
\hookrightarrow_{Spawn} & \{\}; & \langle \mathrm{p1}, (id, \mathsf{let}\ P2 = \mathrm{p2}\ \mathsf{in}\ \ldots), [\,] \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \underline{\mathsf{apply\ echo/0\ [\,]}}), [\,] \rangle \\
\hookrightarrow_{Seq} & \{\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}\}, \mathsf{let}\ P3 = \underline{\mathsf{spawn(target/0, [\,])}}\ \mathsf{in}\ \ldots), [\,] \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply\ echo/0\ [\,]}), [\,] \rangle \\
\hookrightarrow_{Spawn} & \{\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}\}, \underline{\mathsf{let}\ P3 = \mathrm{p3}\ \mathsf{in}\ \ldots}, [\,]) \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply\ echo/0\ [\,]}), [\,] \rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply\ target/0\ [\,]}), [\,] \rangle \\
\hookrightarrow_{Seq} & \{\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \mathsf{let}\ \_ = \underline{P3}\ !\ \mathsf{hello}\ \mathsf{in}\ \ldots, [\,]) \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply\ echo/0\ [\,]}), [\,] \rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply\ target/0\ [\,]}), [\,] \rangle \\
\hookrightarrow_{Seq} & \{\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \mathsf{let}\ \_ = \underline{\mathrm{p3}\ !\ \mathsf{hello}}\ \mathsf{in}\ \ldots, [\,]) \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply\ echo/0\ [\,]}), [\,] \rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply\ target/0\ [\,]}), [\,] \rangle \\
\hookrightarrow_{Send} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \underline{\mathsf{let}\ \_ = \mathsf{hello}\ \mathsf{in}\ \ldots}, [\,]) \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply\ echo/0\ [\,]}), [\,] \rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply\ target/0\ [\,]}), [\,] \rangle \\
\hookrightarrow_{Seq} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \underline{P2}\ !\ \{P3, \mathrm{world}\}, [\,]) \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply\ echo/0\ [\,]}), [\,] \rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply\ target/0\ [\,]}), [\,] \rangle \\
\hookrightarrow_{Seq} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \mathrm{p2}\ !\ \{\underline{P3}, \mathrm{world}\}, [\,]) \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply\ echo/0\ [\,]}), [\,] \rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply\ target/0\ [\,]}), [\,] \rangle \\
\hookrightarrow_{Seq} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \underline{\mathrm{p2}\ !\ \{\mathrm{p3}, \mathrm{world}\}}, [\,]) \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply\ echo/0\ [\,]}), [\,] \rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply\ target/0\ [\,]}), [\,] \rangle \\
\hookrightarrow_{Send} & \{m_1, m_2\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,]) \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \underline{\mathsf{apply\ echo/0\ [\,]}}), [\,] \rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply\ target/0\ [\,]}), [\,] \rangle \\
\hookrightarrow_{Seq} & \{m_1, m_2\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,]) \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{receive}\ \{P, M\} \to P\ !\ M\ \mathsf{end}), [\,] \rangle \\
 & & \mid \langle \mathrm{p3}, (id, \underline{\mathsf{apply\ target/0\ [\,]}}), [\,] \rangle \\
\hookrightarrow_{Seq} & \{m_1, m_2\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,]) \rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{receive}\ \{P, M\} \to P\ !\ M\ \mathsf{end}), [\,] \rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{receive}\ A \to \ldots\ \mathsf{end}), [\,] \rangle
\end{array}
$$

Fig. 7. A derivation from "apply main/0 ()", where $m_1 = (\mathrm{p3}, \mathrm{hello})$, $m_2 = (\mathrm{p2}, \{\mathrm{p3}, \mathrm{world}\})$, and $m_3 = (\mathrm{p3}, \mathrm{world})$ (part 1/2).

The rules presented so far allow one to store messages in the global mailbox, but not to remove messages from it. This is precisely the task of the scheduler, which is modelled by rule *Sched*. This rule nondeterministically chooses a pair (p, v) in the global mailbox Γ and delivers the message v to the target process p. Here, we deliberately ignore the restriction mentioned in Example 1: "the messages sent—directly—between two given processes arrive in the same order they were sent", since current implementations only guarantee it within the same node. In practice, ignoring this restriction amounts to consider that each process is potentially run in a different node. An alternative definition ensuring this restriction can be found in [27].

Example 4. Consider again the program shown in Example 1. Figs. 7 and 8 show a derivation from "apply main/0 ()" where the call to function target reduces to {world, hello}, as discussed in Example 1 (i.e., the interleaving shown in Fig. 2(b)). Processes' pids are denoted with p1, p2 and p3. For clarity, we label each transition step with the applied rule and underline the reduced expression.

3.1. Erlang concurrency

In order to define a causal-consistent reversible semantics for Erlang 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). While concurrency is a main feature of Erlang, as far as we know no formal definition of the concurrency model of Erlang exists in the literature. We propose below one such definition.

$\hookrightarrow_{Sched}$ $\{m_1\}$; $\langle$p1, ($\{P2 \mapsto$ p2, $P3 \mapsto$ p3$\}$, {p3, world}, [ ])$\rangle$
| $\langle$p2, (id, receive $\{P, M\} \to P\,!\,M$ end), [{p3, world}]$\rangle$
| $\langle$p3, (id, receive $A \to \ldots$ end), [ ]$\rangle$

$\hookrightarrow_{Receive}$ $\{m_1\}$; $\langle$p1, ($\{P2 \mapsto$ p2, $P3 \mapsto$ p3$\}$, {p3, world}, [ ])$\rangle$
| $\langle$p2, ($\{P \mapsto$ p3, $M \mapsto$ world$\}$, $\underline{P}\,!\,M$), [ ]$\rangle$
| $\langle$p3, (id, receive $A \to \ldots$ end), [ ]$\rangle$

$\hookrightarrow_{Seq}$ $\{m_1\}$; $\langle$p1, ($\{P2 \mapsto$ p2, $P3 \mapsto$ p3$\}$, {p3, world}, [ ])$\rangle$
| $\langle$p2, ($\{P \mapsto$ p3, $M \mapsto$ world$\}$, p3 $!\,\underline{M}$), [ ]$\rangle$
| $\langle$p3, (id, receive $A \to \ldots$ end), [ ]$\rangle$

$\hookrightarrow_{Seq}$ $\{m_1\}$; $\langle$p1, ($\{P2 \mapsto$ p2, $P3 \mapsto$ p3$\}$, {p3, world}, [ ])$\rangle$
| $\langle$p2, ($\{P \mapsto$ p3, $M \mapsto$ world$\}$, $\underline{\text{p3 ! world}}$), [ ]$\rangle$
| $\langle$p3, (id, receive $A \to \ldots$ end), [ ]$\rangle$

$\hookrightarrow_{Send}$ $\{m_1, m_3\}$; $\langle$p1, ($\{P2 \mapsto$ p2, $P3 \mapsto$ p3$\}$, {p3, world}, [ ])$\rangle$
| $\langle$p2, ($\{P \mapsto$ p3, $M \mapsto$ world$\}$, world), [ ]$\rangle$
| $\langle$p3, (id, receive $A \to \ldots$ end), [ ]$\rangle$

$\hookrightarrow_{Sched}$ $\{m_1\}$; $\langle$p1, ($\{P2 \mapsto$ p2, $P3 \mapsto$ p3$\}$, {p3, world}, [ ])$\rangle$
| $\langle$p2, ($\{P \mapsto$ p3, $M \mapsto$ world$\}$, world), [ ]$\rangle$
| $\langle$p3, (id, $\underline{\text{receive } A \to \ldots \text{ end}}$), [world]$\rangle$

$\hookrightarrow_{Receive}$ $\{m_1\}$; $\langle$p1, ($\{P2 \mapsto$ p2, $P3 \mapsto$ p3$\}$, {p3, world}, [ ])), [ ]$\rangle$
| $\langle$p2, ($\{P \mapsto$ p3, $M \mapsto$ world$\}$, world), [ ]$\rangle$
| $\langle$p3, ($\{A \mapsto$ world$\}$, receive $B \to \{A, B\}$ end), [ ]$\rangle$

$\hookrightarrow_{Sched}$ { }; $\langle$p1, ($\{P2 \mapsto$ p2, $P3 \mapsto$ p3$\}$, {p3, world}, [ ])), [ ]$\rangle$
| $\langle$p2, ($\{P \mapsto$ p3, $M \mapsto$ world$\}$, world), [ ]$\rangle$
| $\langle$p3, ($\{A \mapsto$ world$\}$, $\underline{\text{receive } B \to \{A, B\} \text{ end}}$), [hello]$\rangle$

$\hookrightarrow_{Receive}$ { }; $\langle$p1, ($\{P2 \mapsto$ p2, $P3 \mapsto$ p3$\}$, {p3, world}, [ ])), [ ]$\rangle$
| $\langle$p2, ($\{P \mapsto$ p3, $M \mapsto$ world$\}$, world), [ ]$\rangle$
| $\langle$p3, ($\{A \mapsto$ world, $B \mapsto$ hello$\}$, $\{\underline{A}, B\}$), [ ]$\rangle$

$\hookrightarrow_{Seq}$ { }; $\langle$p1, ($\{P2 \mapsto$ p2, $P3 \mapsto$ p3$\}$, {p3, world}, [ ])), [ ]$\rangle$
| $\langle$p2, ($\{P \mapsto$ p3, $M \mapsto$ world$\}$, world), [ ]$\rangle$
| $\langle$p3, ($\{A \mapsto$ world, $B \mapsto$ hello$\}$, {world, $\underline{B}$}), [ ]$\rangle$

$\hookrightarrow_{Seq}$ { }; $\langle$p1, ($\{P2 \mapsto$ p2, $P3 \mapsto$ p3$\}$, {p3, world}, [ ])), [ ]$\rangle$
| $\langle$p2, ($\{P \mapsto$ p3, $M \mapsto$ world$\}$, world), [ ]$\rangle$
| $\langle$p3, ($\{A \mapsto$ world, $B \mapsto$ hello$\}$, {world, hello}), [ ]$\rangle$

Fig. 8. A derivation from "apply main/0 ()", where $m_1 = (\text{p3}, \text{hello})$, $m_2 = (\text{p2}, \{\text{p3}, \text{world}\})$, and $m_3 = (\text{p3}, \text{world})$ (part 2/2).

Given systems s_1, s_2, we call $s_1 \hookrightarrow^* s_2$ 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. We label transitions as follows: $s_1 \hookrightarrow_{p,r} s_2$ where[6]

- p is the pid of the selected process in the transition or of the process to which a message is delivered (if the applied rule is *Sched*);
- r is the label of the applied transition rule.

We ignore some labels when they are clear from the context.

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 $\mathsf{final}(d_1) = \mathsf{final}(d_2)$.

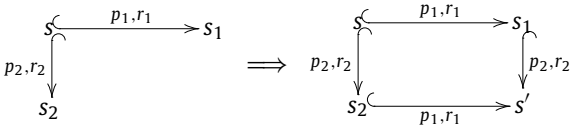
We let ϵ_s denote the zero-step derivation $s \hookrightarrow^* s$.

Definition 5 *(Concurrent transitions).* Given two 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 either $r_1 = r_2 = Sched$ or one transition applies rule *Sched* and the other transition applies rule *Receive*. Two coinitial transitions are *concurrent* if they are not in conflict.

We show below that our definition of concurrent transitions makes sense.

[6] Note that p, r in $\hookrightarrow_{p,r}$ are not parameters of the transition relation $\hookrightarrow$ but just labels with some information on the reduction step. This information becomes useful to formally define the notion of concurrent transitions.

Lemma 6 *(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. We have the following cases:

- Two transitions t_1 and t_2 where $r_1 \neq Sched$ and $r_2 \neq Sched$. Trivially, they apply to different processes, i.e., $p_1 \neq p_2$. Then, we can easily prove that by applying rule r_2 to p_1 in s_1 and rule r_1 to p_2 in s_2 we have two transitions t_1/t_2 and t_2/t_1 which are cofinal.
- One transition t_1 which applies rule $r_1 = Sched$ to deliver message v_1 to process $p_1 = \mathrm{p}$, and another transition which applies a rule r_2 different from *Sched*. All cases but $r_2 = Receive$ with $p_2 = \mathrm{p}$ are straightforward. This last case, though, cannot happen since transitions using rules *Sched* and *Receive* are not concurrent.
- Two transitions t_1 and t_2 with rules $r_1 = r_2 = Sched$ delivering messages v_1 and v_2, respectively. Since the transitions are concurrent, they should deliver the messages to different processes, i.e., $p_1 \neq p_2$. Therefore, we can see that delivering v_2 from s_1 and v_1 from s_2 we get two cofinal transitions. □

We remark here that other definitions of concurrent transitions are possible. Changing the concurrency model would require to change the stored information in the reversible semantics in order to preserve causal consistency. We have chosen the notion above since it is reasonably simple to define and to work with, and captures most of the pairs of coinitial transitions that satisfy the Square lemma.

4. A reversible semantics for erlang

In this section, we introduce a reversible—uncontrolled—semantics for the considered language. Thanks to the modular design of the concrete semantics, the transition rules for the language expressions need not be changed in order to define the reversible semantics.

To be precise, in this section we introduce two transition relations: $\rightharpoonup$ and $\leftharpoonup$. The first relation, $\rightharpoonup$, is a conservative extension of the standard semantics $\hookrightarrow$ (Fig. 6) to also include some additional information in the states, following a typical Landauer embedding. We refer to $\rightharpoonup$ as the *forward* reversible semantics (or simply the forward semantics). In contrast, the second relation, $\leftharpoonup$, proceeds in the backward direction, "undoing" actions step by step. We refer to $\leftharpoonup$ as the backward (reversible) semantics. We denote the union $\rightharpoonup \cup \leftharpoonup$ by $\rightleftharpoons$.

In the next section, we will introduce a rollback operator that starts a reversible computation for a process. In order to avoid undoing all actions until the beginning of the process, we will also let the programmer introduce *checkpoints*. Syntactically, they are denoted with the built-in function check, which takes an identifier `t` as an argument. Such identifiers are supposed to be unique in the program. Given an expression, *expr*, we can introduce a checkpoint by replacing *expr* with "let $X = \mathsf{check}(\mathtt{t})$ in *expr*". A call of the form $\mathsf{check}(\mathtt{t})$ just returns `t` (see below). In the following, we consider that the rules to evaluate the language expressions (Figs. 4 and 5) are extended with the following rule:

$$(Check)\ \frac{}{\theta, \mathsf{check}(\mathtt{t}) \xrightarrow{\mathsf{check}(\mathtt{t})} \theta, \mathtt{t}}$$

In this section, we will mostly ignore checkpoints, but they will become relevant in the next section.

The most significant novelty in the forward semantics is that messages now include a unique identifier (e.g., a timestamp λ). Let us illustrate with some examples why we introduce these identifiers. Consider first diagram (a) in Fig. 9, where two different processes, p1 and p3, send the same message v to process p2. In order to undo the action p2 ! v in process p3, one needs to first undo all actions of p2 up to $\boxed{t_1}$ (to ensure causal consistency). However, currently, messages only store information about the target process and the value sent, therefore it is not possible to know whether it is safe to stop undoing actions at $\boxed{t_1}$ or at $\boxed{t_2}$. Actually, the situations in diagrams (a) and (b) are not distinguishable. In this case, it would suffice to add the pid of the sender to every message in order to avoid the confusion. However, this is not always sufficient. Consider now diagram (c). Here, a process p1 sends two identical messages to another process p2 (which is not unusual, say an "ack" after receiving a request). In this case, in order to undo the first action p2 ! v of process p1 one needs to undo all actions of process p2 up to $\boxed{t_1}$. However, we cannot distinguish $\boxed{t_1}$ from $\boxed{t_2}$ unless some additional information is taken into account (and considering triples of the form (*source_process_pid*, *target_process_pid*, *message*) would not help). Therefore, one needs to introduce some unique identifier in order to precisely distinguish case (c) from case (d).

Of course, we could have a less precise semantics where just the message, v, is observable. However, that would make the backward semantics unpredictable (e.g., we could often undo the "wrong" message delivery). Also, defining the corresponding notion of *conflicting* transitions (see Definition 12 below) would be challenging, since one would like to have only

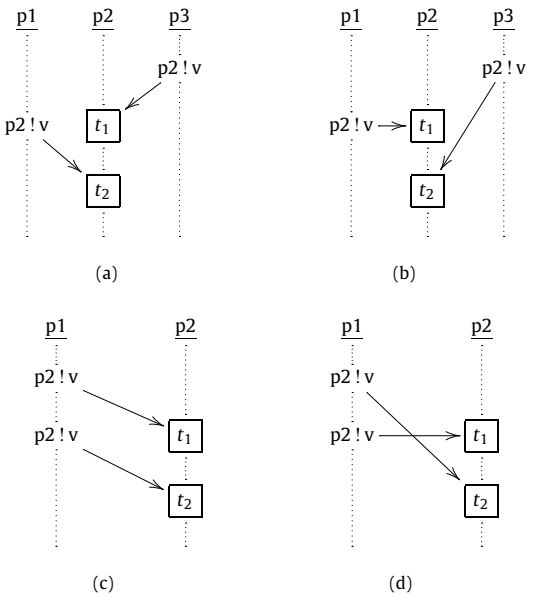


Fig. 9. Interleavings and the need for unique identifiers for messages.

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

$$(Check)\quad \frac{\theta, e \xrightarrow{\mathsf{check}(\mathsf{t})} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{check}(\theta, e, \mathsf{t}){:}h, (\theta', e'), q\rangle \mid \Pi}$$

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

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

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

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

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

Fig. 10. Forward reversible semantics.

a conflict between the sending of a message v and the "last" delivery of the same message v, which would be very tricky. Therefore, in this paper, we prefer to assume that messages can be uniquely distinguished.

The transition rules of the forward reversible semantics can be found in Fig. 10. Processes now include a memory (or *history*) h that records the intermediate states of a process, and messages have an associated unique identifier. In the memory, we use terms headed by constructors τ, check, send, rec, spawn, and self to record the steps performed by the forward semantics. Note that we could optimise the information stored in these terms by following a strategy similar to that in [24,26,31] for the reversibility of functional expressions, but this is orthogonal to our purpose in this paper, so we focus mainly on the concurrent actions. Note also that the auxiliary function matchrec now deals with messages of the form $\{v, \lambda\}$, which is a trivial extension of the original function in the standard semantics by just ignoring λ when computing the first matching message.

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

$$(\overline{Check}) \quad \Gamma; \langle p, \mathsf{check}(\theta, e, \mathtt{t}){:}h, (\theta', e'), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$

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

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

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

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

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

Fig. 11. Backward reversible semantics.

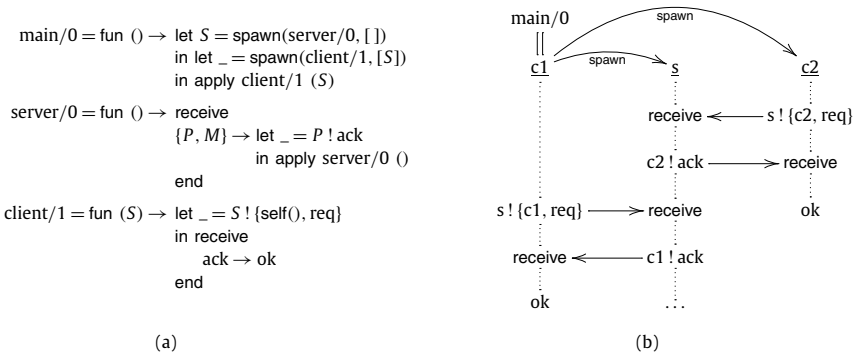


Fig. 12. A simple client-server.

Example 7. Let us consider the program shown in Fig. 12(a), together with the execution trace sketched in Fig. 12(b). Fig. 13 shows a high level account of the corresponding derivation under the forward semantics. For clarity, we consider the following conventions:

- Processes client1, client2 and server are denoted with c1, c2 and s, respectively.
- In the processes, we do not show the current environment. Moreover, we use the notation $C[e]$ to denote that e is the redex to be reduced next and $C[\,]$ is an arbitrary (possibly empty) context. We also underline the selected redex when there are more than one (e.g., a redex in each process).
- In the histories, some arguments are denoted by "_" since they are not relevant in the current derivation.
- Finally, we only show the steps performed with rules *Spawn*, *Send*, *Receive* and *Sched*; the transition relation is labelled with the applied rule.

We now prove that the forward semantics $\rightharpoonup$ is a conservative extension of the standard semantics $\hookrightarrow$.

In order to state the result, we let $\mathsf{del}(s)$ denote the system that results from s by removing the histories of the processes; formally, $\mathsf{del}(\Gamma; \Pi) = \Gamma; \mathsf{del}'(\Pi)$, where

$$\begin{array}{lcl} \mathsf{del}'(\langle p, h, (\theta, e), q\rangle) & = & \langle p, (\theta, e), q\rangle \\ \mathsf{del}'(\langle p, h, (\theta, e), q\rangle \mid \Pi) & = & \langle p, (\theta, e), q\rangle \mid \mathsf{del}'(\Pi) \end{array}$$

where we assume that Π is not empty.

We can now state the conservative extension result.

Theorem 8. *Let s_1 be a system of the reversible semantics without occurrences of "*check*" and $s_1' = \mathsf{del}(s_1)$ a system of the standard semantics. Then, $s_1' \hookrightarrow^* s_2'$ iff $s_1 \rightharpoonup^* s_2$ and $\mathsf{del}(s_2) = s_2'$.*

Proof. The proof is straightforward since the transition rules of the forward semantics in Fig. 10 are just annotated versions of the corresponding rules in Fig. 6. The only tricky point is noticing that the introduction of unique identifiers for messages does not change the behaviour of rule *Receive* since function matchrec always returns the oldest occurrence (in terms of position in the queue) of the selected message. □

$$
\begin{array}{lll}
 & \{\}; & \langle \mathrm{c1}, [\,], (id, C[\mathsf{apply}\ \mathrm{main/0}\ ()]), [\,]\rangle \\
\rightarrow^{*} & \{\}; & \langle \mathrm{c1}, [\,], (\_, C[\mathsf{spawn}(\mathrm{server/0}, [\,])]), [\,]\rangle \\
\rightarrow_{Spawn} & \{\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\underline{\mathsf{spawn}(\mathrm{client/1}, [\mathrm{s}])}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\,], (\_, C[\mathsf{receive}\ \{P, M\} \rightarrow \ldots]), [\,]\rangle \\
\rightarrow_{Spawn} & \{\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\,], (\_, C[\mathsf{receive}\ \{P, M\} \rightarrow \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\,], (\_, C[\underline{\mathrm{s} \,!\, \{\mathrm{c2}, \mathrm{req}\}}]), [\,]\rangle \\
\rightarrow_{Send} & \{(\mathrm{s}, m_1)\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\,], (\_, C[\mathsf{receive}\ \{P, M\} \rightarrow \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, C[\mathsf{receive}\ \mathrm{ack} \rightarrow \mathrm{ok}]), [\,]\rangle \\
\rightarrow_{Sched} & \{\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\,], (\_, C[\underline{\mathsf{receive}\ \{P, M\} \rightarrow \ldots}]), [m_1]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, C[\mathsf{receive}\ \mathrm{ack} \rightarrow \mathrm{ok}]), [\,]\rangle \\
\rightarrow_{Receive} & \{\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{rec}(\_, \_, m_1, [m_1])], (\_, C[\underline{\mathrm{c2} \,!\, \mathrm{ack}}]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, C[\mathsf{receive}\ \mathrm{ack} \rightarrow \mathrm{ok}]), [\,]\rangle \\
\rightarrow_{Send} & \{(\mathrm{c2}, m_2)\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_, \_, \mathrm{c2}, m_2), \mathsf{rec}(\_, \_, m_1, [m_1])], (\_, C[\mathsf{receive}\ \{P, M\} \rightarrow \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, C[\mathsf{receive}\ \mathrm{ack} \rightarrow \mathrm{ok}]), [\,]\rangle \\
\rightarrow_{Sched} & \{\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_, \_, \mathrm{c2}, m_2), \mathsf{rec}(\_, \_, m_1, [m_1])], (\_, C[\mathsf{receive}\ \{P, M\} \rightarrow \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, C[\underline{\mathsf{receive}\ \mathrm{ack} \rightarrow \mathrm{ok}}]), [m_2]\rangle \\
\rightarrow_{Receive} & \{\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\underline{\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_, \_, \mathrm{c2}, m_2), \mathsf{rec}(\_, \_, m_1, [m_1])], (\_, C[\mathsf{receive}\ \{P, M\} \rightarrow \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_, \_, m_2, [m_2]), \mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightarrow_{Send} & \{(\mathrm{s}, m_3)\}; & \langle \mathrm{c1}, [\mathsf{send}(\_, \_, \mathrm{s}, m_3), \mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\mathsf{receive}\ \mathrm{ack} \rightarrow \mathrm{ok}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_, \_, \mathrm{c2}, m_2), \mathsf{rec}(\_, \_, m_1, [m_1])], (\_, C[\mathsf{receive}\ \{P, M\} \rightarrow \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_, \_, m_2, [m_2]), \mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightarrow_{Sched} & \{\}; & \langle \mathrm{c1}, [\mathsf{send}(\_, \_, \mathrm{s}, m_3), \mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\mathsf{receive}\ \mathrm{ack} \rightarrow \mathrm{ok}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_, \_, \mathrm{c2}, m_2), \mathsf{rec}(\_, \_, m_1, [m_1])], (\_, C[\underline{\mathsf{receive}\ \{P, M\} \rightarrow \ldots}]), [m_3]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_, \_, m_2, [m_2]), \mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightarrow_{Receive} & \{\}; & \langle \mathrm{c1}, [\mathsf{send}(\_, \_, \mathrm{s}, m_3), \mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\mathsf{receive}\ \mathrm{ack} \rightarrow \mathrm{ok}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{rec}(\_, \_, m_3, [m_3]), \mathsf{send}(\_, \_, \mathrm{c2}, m_2), \mathsf{rec}(\_, \_, m_1, [m_1])], (\_, C[\underline{\mathrm{c1} \,!\, \mathrm{ack}}]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_, \_, m_2, [m_2]), \mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightarrow_{Send} & \{(\mathrm{c1}, m_4)\}; & \langle \mathrm{c1}, [\mathsf{send}(\_, \_, \mathrm{s}, m_3), \mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\mathsf{receive}\ \mathrm{ack} \rightarrow \mathrm{ok}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_, \_, \mathrm{c1}, m_4), \mathsf{rec}(\_, \_, m_3, [m_3]), \mathsf{send}(\_, \_, \mathrm{c2}, m_2), \mathsf{rec}(\_, \_, m_1, [m_1])], \\
 & & \ (\_, C[\mathsf{receive}\ \{P, M\} \rightarrow \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_, \_, m_2, [m_2]), \mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightarrow_{Sched} & \{\}; & \langle \mathrm{c1}, [\mathsf{send}(\_, \_, \mathrm{s}, m_3), \mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, C[\underline{\mathsf{receive}\ \mathrm{ack} \rightarrow \mathrm{ok}}]), [m_4]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_, \_, \mathrm{c1}, m_4), \mathsf{rec}(\_, \_, m_3, [m_3]), \mathsf{send}(\_, \_, \mathrm{c2}, m_2), \mathsf{rec}(\_, \_, m_1, [m_1])], \\
 & & \ (\_, C[\mathsf{receive}\ \{P, M\} \rightarrow \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_, \_, m_2, [m_2]), \mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightarrow_{Receive} & \{\}; & \langle \mathrm{c1}, [\mathsf{rec}(\_, \_, m_4, [m_4]), \mathsf{send}(\_, \_, \mathrm{s}, m_3), \mathsf{spawn}(\_, \_, \mathrm{c2}), \mathsf{spawn}(\_, \_, \mathrm{s})], (\_, \mathrm{ok}), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_, \_, \mathrm{c1}, m_4), \mathsf{rec}(\_, \_, m_3, [m_3]), \mathsf{send}(\_, \_, \mathrm{c2}, m_2), \mathsf{rec}(\_, \_, m_1, [m_1])], \\
 & & \ (\_, C[\mathsf{receive}\ \{P, M\} \rightarrow \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_, \_, m_2, [m_2]), \mathsf{send}(\_, \_, \mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle
\end{array}
$$

Fig. 13. A derivation under the forward semantics, with $m_1 = \{\{\mathrm{c2}, \mathrm{req}\}, 1\}$, $m_2 = \{\mathrm{ack}, 2\}$, $m_3 = \{\{\mathrm{c1}, \mathrm{req}\}, 3\}$, and $m_4 = \{\mathrm{ack}, 4\}$.

The transition rules of the backward semantics are shown in Fig. 11. In general, all rules restore the control (and, if it applies, also the queue) of the process. Nevertheless, let us briefly discuss a few particular situations:

- First, observe that rule $\overline{Send}$ can only be applied when the message sent is in the global mailbox. If this is not the case (i.e., the message has been delivered using rule *Sched*), then we should first apply backward steps to the receiver process until, eventually, the application of rule $\overline{Sched}$ puts the message back into the global mailbox and rule $\overline{Send}$ becomes applicable. This is required to ensure causal consistency. In the next section, we will introduce a particular strategy that achieves this effect in a controlled manner.
- A similar situation occurs with 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 and the queue of process p' are both empty. Therefore, one should first apply a number of backward steps to process p' in order to be able to undo the spawn item. 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.
- Observe to that rule $\overline{Receive}$ can only be applied when the queue of the process is exactly the same queue that was obtained after applying the corresponding (forward) *Receive* step. This is necessary in order to ensure that the restored queue is indeed the right one (note that adding the message to an arbitrary queue would not work since we do not know the "right" position for the message).
- In principle, there is some degree of freedom in the application of rule $\overline{Sched}$ since it does not interfere with the remaining rules, except for $\overline{Receive}$ and other applications of $\overline{Sched}$. Therefore, the application of rule $\overline{Sched}$ can be switched with the application of any other backward rule except for $\overline{Receive}$ or another $\overline{Sched}$. The fact that two $\overline{Sched}$ (involving the same process) do not commute is ensured since $\overline{Sched}$ always applies to the most recent message of a queue. The fact that a $\overline{Sched}$ and a $\overline{Receive}$ do not commute is ensured since the side condition of $\overline{Sched}$ checks that there is no $\mathsf{rec}(\ldots)$ item in the history of the process that can be used to apply rule $\overline{Receive}$ with the current queue. Hence, their applicability conditions do not overlap.

Example 9. Consider again the program shown in Fig. 12. By starting from the last system in the forward derivation shown in Fig. 13, we may construct the backward derivation shown in Fig. 14. Observe that it does not strictly follow the inverse order of the derivation shown in Fig. 13. Actually, a derivation that undoes the steps in the precise inverse order exists, but it is not the only possibility. We will characterise later on (see Corollary 22) which orders are allowed and which are not. In Fig. 14, besides following the same conventions of Example 7, for clarity, we underline the selected history item to be undone or the element in the queue to be removed (when the applied rule is $\overline{Sched}$).

4.1. Properties of the uncontrolled reversible semantics

In the following, we prove several properties of our reversible semantics, including its *causal consistency*, an essential property for reversible concurrent calculi [9].

Given systems s_1, s_2, we call $s_1 \rightharpoonup^* s_2$ a *forward* derivation and $s_2 \leftharpoonup^* s_1$ a *backward* derivation. A derivation potentially including both forward and backward steps is denoted by $s_1 \rightleftharpoons^* s_2$. We label transitions as follows: $s_1 \rightleftharpoons_{p,r,k} s_2$ where

- p, r are the pid of the selected process and the label of the applied rule, respectively, as in Section 3.1,
- k is a history item if the applied rule was different from $\overline{Sched}$ and $Sched$, and
- $k = \mathsf{sched}(\{v, \lambda\})$ when the applied rule was $Sched$ or $\overline{Sched}$, where $\{v, \lambda\}$ is the message delivered or put back into Γ. Note that this information is available when applying the rule.

We ignore some labels when they are clear from the context.

We extend the definitions of functions init and final from Section 3.1 to reversible derivations in the natural way. The notions of composable, coinitial and cofinal derivations are extended also in a straightforward manner.

Given a rule label r, we let $\overline{r}$ denote its reverse version, i.e., if $r = Send$ then $\overline{r} = \overline{Send}$ and vice versa (if $r = \overline{Send}$ then $\overline{r} = Send$). Also, given a transition t, we let $\overline{t} = (s' \leftharpoondown_{p,\overline{r},k} s)$ if $t = (s \rightharpoondown_{p,r,k} s')$ and $\overline{t} = (s' \rightharpoondown_{p,r,k} s)$ if $t = (s \leftharpoondown_{p,\overline{r},k} s')$. We say that $\overline{t}$ is the *inverse* of t. This notation is naturally extended to derivations. We let ϵ_s denote the zero-step derivation $s \rightleftharpoons^* s$.

In the following we restrict the attention to systems reachable from the execution of a program:

Definition 10 *(Reachable systems).* A system is *initial* if it is composed by a single process, and this process has an empty history and an empty queue; furthermore the global mailbox is empty. A system s is *reachable* if there exists an initial system s_0 and a derivation $s_0 \rightleftharpoons^* s$ using the rules corresponding to a given program.

Moreover, for simplicity, we also consider an implicit, fixed program in the technical results, that is we fix the function μ in the semantics of expressions.

The next lemma proves that every forward (resp. backward) transition can be undone by a backward (resp. forward) transition.

Lemma 11 *(Loop lemma). For every pair of reachable systems, s_1 and s_2, we have $s_1 \rightharpoondown_{p,r,k} s_2$ iff $s_2 \leftharpoondown_{p,\overline{r},k} s_1$.*

Proof. The proof is by case analysis on the applied rule. We discuss below the most interesting cases.

- Rule *Sched*: notice that the queue of a process is changed only by rule *Receive* (which removes messages) and *Sched* (which adds messages). Since, after the last *Receive* at least one message has been added, then the side condition of rule $\overline{Sched}$ is always verified.

$$
\begin{array}{lll}
 & \{\,\}; & \langle \mathsf{c1}, [\mathsf{rec}(\_,\_,m_4,[m_4]), \mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_,\mathsf{ok}), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\underline{\mathsf{rec}(\_,\_,m_2,[m_2])}, \mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_,\mathsf{ok}), [\,]\rangle \\
\leftarrow_{\overline{Receive}} & \{\,\}; & \langle \mathsf{c1}, [\underline{\mathsf{rec}(\_,\_,m_4,[m_4])}, \mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_,\mathsf{ok}), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [m_2]\rangle \\
\leftarrow_{\overline{Receive}} & \{\,\}; & \langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\underline{m_4}]\rangle \\
 & & \mid \langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [m_2]\rangle \\
\leftarrow_{\overline{Sched}} & \{(\mathsf{c1},m_4)\}; & \langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\underline{m_2}]\rangle \\
\leftarrow_{\overline{Sched}} & \{(\mathsf{c2},m_2),(\mathsf{c1},m_4)\}; & \langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\underline{\mathsf{send}(\_,\_,\mathsf{c1},m_4)}, \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
\leftarrow_{\overline{Send}} & \{(\mathsf{c2},m_2)\}; & \langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\underline{\mathsf{rec}(\_,\_,m_3,[m_3])}, \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{c1}\,!\,\mathsf{ack}]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
\leftarrow_{\overline{Receive}} & \{(\mathsf{c2},m_2)\}; & \langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\underline{m_3}]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
\leftarrow_{\overline{Sched}} & \{(\mathsf{s},m_3),(\mathsf{c2},m_2)\}; & \langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\underline{\mathsf{send}(\_,\_,\mathsf{c2},m_2)}, \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
\leftarrow_{\overline{Send}} & \{(\mathsf{s},m_3)\}; & \langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\underline{\mathsf{rec}(\_,\_,m_1,[m_1])}], (\_, C[\mathsf{c2}\,!\,\mathsf{ack}]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
\leftarrow_{\overline{Receive}} & \{(\mathsf{s},m_3)\}; & \langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\,], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\underline{m_1}]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
\leftarrow_{\overline{Sched}} & \{(\mathsf{s},m_1),(\mathsf{s},m_3)\}; & \langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\,], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\underline{\mathsf{send}(\_,\_,\mathsf{s},m_1)}], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
\leftarrow_{\overline{Send}} & \{(\mathsf{s},m_3)\}; & \langle \mathsf{c1}, [\underline{\mathsf{send}(\_,\_,\mathsf{s},m_3)}, \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{receive\ ack} \to \mathsf{ok}]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\,], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\,], (\_, C[\mathsf{s}\,!\,\{\mathsf{c2},\mathsf{req}\}]), [\,]\rangle \\
\leftarrow_{\overline{Send}} & \{\,\}; & \langle \mathsf{c1}, [\underline{\mathsf{spawn}(\_,\_,\mathsf{c2})}, \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{s}\,!\,\{\mathsf{c1},\mathsf{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\,], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\,], (\_, C[\mathsf{s}\,!\,\{\mathsf{c2},\mathsf{req}\}]), [\,]\rangle \\
\leftarrow_{\overline{Spawn}} & \{\,\}; & \langle \mathsf{c1}, [\underline{\mathsf{spawn}(\_,\_,\mathsf{s})}], (\_, C[\mathsf{spawn}(\mathsf{client}/1,[\mathsf{s}])]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\,], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
\leftarrow_{\overline{Spawn}} & \{\,\}; & \langle \mathsf{c1}, [\,], (\_, C[\mathsf{spawn}(\mathsf{server}/0,[\,])]), [\,]\rangle \\
\leftarrow^{*} & \{\,\}; & \langle \mathsf{c1}, [\,], (\_, C[\mathsf{apply\ main}/0\ ()]), [\,]\rangle
\end{array}
$$

Fig. 14. A derivation under the backward semantics, with $m_1 = \{\{\mathsf{c2},\mathsf{req}\},1\}$, $m_2 = \{\mathsf{ack},2\}$, $m_3 = \{\{\mathsf{c1},\mathsf{req}\},3\}$, and $m_4 = \{\mathsf{ack},4\}$.

- Rule $\overline{Seq}$: one has to check that the restored control (θ, e) can indeed perform a sequential step to (θ', e'). This always holds for reachable systems. An analogous check needs to be done for all backward rules. □

The following notion of concurrent transitions allows us to characterise which actions can be switched without changing the semantics of a computation. It extends the same notion from the standard semantics (cf. Definition 5) to the reversible semantics.

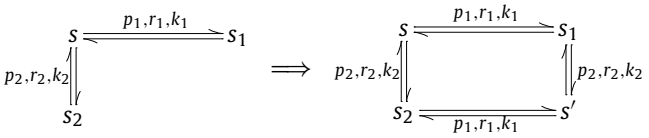
Definition 12 *(Concurrent transitions).* Given two coinitial transitions, $t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$, we say that they are *in conflict* if at least one of the following conditions holds:

- **both transitions are forward**, they consider the same process, i.e., $p_1 = p_2$, and either $r_1 = r_2 = \mathit{Sched}$ or one transition applies rule *Sched* and the other transition applies rule *Receive*.
- one is a **forward** transition that applies to a process p, say $p_1 = \mathrm{p}$, and the other one is a **backward** transition that undoes the creation of p, i.e., $p_2 = \mathrm{p}' \neq \mathrm{p}$, $r_2 = \overline{\mathit{Spawn}}$ and $k_2 = \mathsf{spawn}(\theta, e, \mathrm{p})$ for some control (θ, e);
- one is a **forward** transition that delivers a message $\{v, \lambda\}$ to a process p, say $p_1 = \mathrm{p}$, $r_1 = \mathit{Sched}$ and $k_1 = \mathsf{sched}(\{v, \lambda\})$, and the other one is a **backward** transition that undoes the sending $\{v, \lambda\}$ to p, i.e., $p_2 = \mathrm{p}'$ (note that $\mathrm{p} = \mathrm{p}'$ if the message is sent to its own sender), $r_2 = \overline{\mathit{Send}}$ and $k_2 = \mathsf{send}(\theta, e, \mathrm{p}, \{v, \lambda\})$ for some control (θ, e);
- one is a **forward** transition and the other one is a **backward** transition such that $p_1 = p_2$ and either i) both applied rules are different from both *Sched* and $\overline{\mathit{Sched}}$, i.e., $\{r_1, r_2\} \cap \{\mathit{Sched}, \overline{\mathit{Sched}}\} = \emptyset$; ii) one rule is *Sched* and the other one is $\overline{\mathit{Sched}}$; iii) one rule is *Sched* and the other one is $\overline{\mathit{Receive}}$; or iv) one rule is $\overline{\mathit{Sched}}$ and the other one is *Receive*.

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

The following lemma (the counterpart of Lemma 13 for the standard semantics) is a key result to prove the causal consistency of the semantics.

Lemma 13 *(Square lemma). Given two coinitial concurrent transitions $t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$, there exist two cofinal transitions $t_2/t_1 = (s_1 \rightleftharpoons_{p_2,r_2,k_2} s')$ and $t_1/t_2 = (s_2 \rightleftharpoons_{p_1,r_1,k_1} s')$. Graphically,*



Proof. We distinguish the following cases depending on the applied rules:
(1) Two forward transitions. Then, we have the following cases:

- Two transitions t_1 and t_2 where $r_1 \neq \mathit{Sched}$ and $r_2 \neq \mathit{Sched}$. Trivially, they apply to different processes, i.e., $p_1 \neq p_2$. Then, we can easily prove that by applying rule r_2 to p_1 in s_1 and rule r_1 to p_2 in s_2 we have two transitions t_1/t_2 and t_2/t_1 which produce the corresponding history items and are cofinal.
- One transition t_1 which applies rule $r_1 = \mathit{Sched}$ to deliver message $\{v_1, \lambda_1\}$ to process $p_1 = \mathrm{p}$, and another transition which applies a rule r_2 different from *Sched*. All cases but $r_2 = \mathit{Receive}$ with $p_2 = \mathrm{p}$ and $k_2 = \mathsf{rec}(\theta, e, \{v_2, \lambda_2\}, q)$ are straightforward. Note that $\lambda_1 \neq \lambda_2$ since these identifiers are unique. Here, by applying rule *Receive* to s_1 and rule *Sched* to s_2 we will end up with the same mailbox in p (since it is a FIFO queue). However, the history item $\mathsf{rec}(\theta, e, \{v_2, \lambda_2\}, q')$ will be necessarily different since $q \neq q'$ by the application of rule *Sched*. This situation, though, cannot happen since transitions using rules *Sched* and *Receive* are not concurrent.
- Two transitions t_1 and t_2 with rules $r_1 = r_2 = \mathit{Sched}$ delivering messages $\{v_1, \lambda_1\}$ and $\{v_2, \lambda_2\}$, respectively. Since the transitions are concurrent, they should deliver the messages to different processes, i.e., $p_1 \neq p_2$. Therefore, we can easily prove that delivering $\{v_2, \lambda_2\}$ from s_1 and $\{v_1, \lambda_1\}$ from s_2 we get two cofinal transitions.

(2) One forward transition and one backward transition. Then, we distinguish the following cases:

- If the two transitions apply to the same process, i.e., $p_1 = p_2$, then, since they are concurrent, we can only have $r_1 = \mathit{Sched}$ and a rule different from both $\overline{\mathit{Sched}}$ and $\overline{\mathit{Receive}}$, or $r_1 = \overline{\mathit{Sched}}$ and a rule different from both *Sched* and *Receive*. In these cases, the claim follows easily by a case distinction on the applied rules.
- Let us now consider that the transitions apply to different processes, i.e., $p_1 \neq p_2$, and the applied rules are different from $\mathit{Sched}, \overline{\mathit{Sched}}$. In this case, the claim follows easily except when one transition considers a process p and the other one undoes the spawning of the same process p. This case, however, is not allowed since the transitions are concurrent.
- Finally, let us consider that the transitions apply to different processes, i.e., $p_1 \neq p_2$, and that one transition applies rule *Sched* to deliver a message $\{v, \lambda\}$ from sender p to receiver p′, i.e., $p_1 = \mathrm{p}'$, $r_1 = \mathit{Sched}$ and $k_1 = \mathsf{sched}(\{v, \lambda\})$. In this case, the other transition should apply a rule r_2 different from $\overline{\mathit{Send}}$ with $k_2 = \mathsf{send}(\theta, e, \mathrm{p}', \{v, \lambda\})$ for some control (θ, e) since, otherwise, the transitions would not be concurrent. In any other case, one can easily prove that by applying r_2 to s_1 and *Sched* to s_2 we get two cofinal transitions.

(3) Two backward transitions. We distinguish the following cases:

- If the two transitions apply to different processes, the claim follows easily.
- Let us now consider that they apply to the same process, i.e., $p_1 = p_2$ and that the applied rules are different from $\overline{Sched}$. This case is not possible since, given a system, only one backward transition rule different from $\overline{Sched}$ is applicable (i.e., the one that corresponds to the last item in the history).
- Let us consider that both transitions apply to the same process and that both are applications of rule $\overline{Sched}$. This case is not possible since rule $\overline{Sched}$ can only take the newest message from the local queue of the process, and thus only one rule $\overline{Sched}$ can be applied to a given process.
- Finally, consider that both transitions apply to the same process and only one of them applies rule $\overline{Sched}$. In this case, the only non-trivial case is when the other applied rule is $\overline{Receive}$, since both change the local queue of the process. However, this case is not allowed by the backward semantics, since the conditions to apply rule $\overline{Sched}$ and rule $\overline{Receive}$ are non-overlapping. □

Corollary 14 *(Backward confluence). Given two backward derivations $s \leftharpoonup^* s_1$ and $s \leftharpoonup^* s_2$ there exist s_3 and two backward derivations $s_1 \leftharpoonup^* s_3$ and $s_2 \leftharpoonup^* s_3$.*

Proof. By iterating the square lemma (Lemma 13), noticing that backward transitions are always concurrent. This is a standard result for abstract relations (see, e.g., [2] and the original work by Rosen [29]), where confluence is implied by the *diamond property* (the square lemma in our work). □

The notion of concurrent transitions for the reversible semantics is a natural extension of the same notion for the standard semantics:

Lemma 15. *Let t_1 and t_2 be two forward coinitial transitions using the reversible semantics, and let t_1' and t_2' be their counterpart in the standard semantics obtained by removing the histories and the unique identifiers for messages. Then, t_1 and t_2 are concurrent iff t_1' and t_2' are.*

Proof. The proof is straightforward since Definition 5 and the first case of Definition 12 are perfectly analogous. □

The next result is used to switch the successive application of two transition rules. Let us note that previous proof schemes of causal consistency (e.g., [9]) did not include such a result, directly applying the square lemma instead. In our case, this would not be correct.

Lemma 16 *(Switching lemma). Given two composable transitions of the form $t_1 = (s_1 \rightleftharpoons_{p_1,r_1,k_1} s_2)$ and $t_2 = (s_2 \rightleftharpoons_{p_2,r_2,k_2} s_3)$ such that $\overline{t_1}$ and t_2 are concurrent, there exist a system s_4 and two composable transitions $t_1' = (s_1 \rightleftharpoons_{p_2,r_2,k_2} s_4)$ and $t_2' = (s_4 \rightleftharpoons_{p_1,r_1,k_1} s_3)$.*

Proof. First, using the loop lemma (Lemma 11), we have $\overline{t_1} = (s_2 \rightleftharpoons_{p_1,\overline{r_1},k_1} s_1)$. Now, since $\overline{t_1}$ and t_2 are concurrent, by applying the square lemma (Lemma 13) to $\overline{t_1} = (s_2 \rightleftharpoons_{p_1,\overline{r_1},k_1} s_1)$ and $t_2 = (s_2 \rightleftharpoons_{p_2,r_2,k_2} s_3)$, there exists a system s_4 such that $\overline{t_1'} = \overline{t_1}/t_2 = (s_3 \rightleftharpoons_{p_1,\overline{r_1},k_1} s_4)$ and $t_2' = t_2/\overline{t_1} = (s_1 \rightleftharpoons_{p_2,r_2,k_2} s_4)$. Using the loop lemma (Lemma 11) again, we have $t_1' = t_1/t_2 = (s_4 \rightleftharpoons_{p_1,r_1,k_1} s_3)$, which concludes the proof. □

Corollary 17. *Given two composable transitions $t_1 = (s_1 \rightharpoonup_{p_1,r_1,k_1} s_2)$ and $t_2 = (s_2 \leftharpoondown_{p_2,r_2,k_2} s_3)$, there exist a system s_4 and two composable transitions $t_1' = (s_1 \leftharpoondown_{p_2,r_2,k_2} s_4)$ and $t_2' = (s_4 \rightharpoonup_{p_1,r_1,k_1} s_3)$. Graphically,*

$$\begin{array}{ccc} s_1 & \xrightarrow{p_1,r_1,k_1} & s_2 \\ & & \downarrow{\scriptstyle p_2,r_2,k_2} \\ & & s_3 \end{array} \quad \Longrightarrow \quad \begin{array}{ccc} s_1 & \xrightarrow{p_1,r_1,k_1} & s_2 \\ \downarrow{\scriptstyle p_2,r_2,k_2} & & \downarrow{\scriptstyle p_2,r_2,k_2} \\ s_4 & \xrightarrow[p_1,r_1,k_1]{} & s_3 \end{array}$$

Proof. The corollary follows by applying the switching lemma (Lemma 16), noticing that two backward transitions are always concurrent. □

We now formally define the notion of causal equivalence between derivations, in symbols $\approx$, as the least equivalence relation between transitions closed under composition that obeys the following rules:

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

Causal equivalence amounts to say that those derivations that only differ for swaps of concurrent actions or the removal of successive inverse actions 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.

Lemma 18 *(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. Otherwise, $s_1 \rightharpoonup s_2$ is an application of rule *Sched* and $s_2 \leftharpoonup s_3$ should undo the scheduling of the same message. 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 Corollary 17 commuting $s_2 \leftharpoonup s_3$ with all forward transitions preceding it in d. 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. □

An interesting consequence of the rearranging lemma is the following result, which states that every system obtained by both forward and backward steps from an initial system, is also reachable by a forward-only derivation:

Corollary 19. *Let s be an initial system. For each derivation $s \rightleftharpoons^* s'$, there exists a forward derivation of the form $s \rightharpoonup^* s'$.*

The following auxiliary result is also needed for proving causal consistency.

Lemma 20 *(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 18) 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 $s_1 \leftharpoonup_{p_1,\overline{r_1},k_1} s_2 \rightharpoonup_{p_2,r_2,k_2} s_3$ 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 $s_1 \leftharpoonup_{p_1,\overline{r_1},k_1} s_2$. Moreover, we can swap t with all the transitions between t and $s_1 \leftharpoonup_{p_1,\overline{r_1},k_1} s_2$, in order to obtain a derivation in which $s_1 \leftharpoonup_{p_1,\overline{r_1},k_1} s_2$ and t are adjacent.[7] To do so we use the switching lemma (Lemma 16), since for all transitions t' in between, we have that $\overline{t'}$ and t are concurrent (this is proved below too). When $s_1 \leftharpoonup_{p_1,\overline{r_1},k_1} s_2$ and t are adjacent we can remove both of them using $\approx$. The resulting derivation is strictly shorter, thus the claim follows by the inductive hypothesis.

Let us now prove the results used above. Thanks to the loop lemma (Lemma 11) 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'. We first consider the case where $\overline{r_1} \neq \overline{Sched}$. Since the first transition of $\overline{d}; d_2$, $\overline{(s_1 \leftharpoonup_{p_1,\overline{r_1},k_1} s_2)}$, adds item k_1 to the history 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 deliver the same message since we know that t is not a *Sched* since it adds item k_1 to the history. Thus $\overline{t'}$ and t are concurrent.
- If t' applies some other rule, then t' and t are clearly concurrent.

Now, we consider the case $\overline{r_1} = \overline{Sched}$ with $k_1 = \mathsf{sched}(\{v, \lambda\})$, so that $\overline{(s_1 \leftharpoonup_{p_1,\overline{Sched},k_1} s_2)}$ adds a message $\{v, \lambda\}$ to the queue of p_1. We now distinguish two cases according to whether there is in $\overline{d}; d_2$ an application of rule *Receive* to p_1 or not:

- If the forward derivation $\overline{d}; d_2$ contains no application of rule *Receive* to p_1 then, in the final state, the queue of process p_1 contains the message. Hence, d' needs to contain a *Sched* for the same message. The earliest such *Sched* transition in d' 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. Consider the case where t' applies rule *Sched* to deliver a different message to the same process p_1. Since no *Receive* would be performed on p_1

[7] 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.

then the queue will stay different, and the two derivations could not be cofinal, hence this case can never happen. In all the other cases the two transitions are concurrent.

- If the forward derivation $\overline{d}; d_2$ contains at least an application of rule *Receive* to p_1, let us consider the first such application. This creates a history item k_2. In order for the two derivations to be cofinal, the same history item needs to be created in d'. The queue stored in k_2 has a suffix $\{v, \lambda\} : q$, hence also in d' the first *Sched* delivering a message to p_1 should deliver message $\{v, \lambda\}$. Since there are no other *Sched* nor *Receive* targeting p_1 then the *Sched* delivering message $\{v, \lambda\}$ to p_1 is concurrent to all previous transitions as desired. □

Finally, we can state and prove the causal consistency of our reversible semantics. Intuitively speaking, it states that two different derivations starting from the same initial state can reach the same final state if and only if they are causal consistent. On the one hand, it means that derivations which are causal consistent lead to the same final state, hence it is not possible to distinguish such derivations looking at their final states (as a consequence, 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 precisely what is needed to distinguish computations which are not causal consistent, and no more.

Theorem 21 *(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 18), 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 20) 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 either both rules are *Sched*, or one is *Sched* and the other one is *Receive*. In the first 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 either have a different queue in the process or different items $\mathsf{rec}(\ldots)$ in the history. In the second case, where we have one rule *Sched* and one *Receive*, the situation is similar. Therefore, we can assume that t_1 and t_2 are concurrent transitions.
   We have two cases, according to whether t_1 is an application of *Sched* or not. If it is not, 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 and thus we never have a conflict since the only possible sources of conflict would be rules *Spawn* and *Sched*, but this could not happen since, in this case, t_1 could not happen either.
   If t_1 is an application of *Sched* then we can find the transition t_1' in d_2 scheduling the same message (otherwise the two derivations could not be cofinal), and show that it can be switched with all the previous transitions. If the previous transition targets a different process then the only possible conflicts are with rules *Send* or *Spawn*, but in this case t_1 could not have been performed. If the previous transition targets the same process then the only possible conflicts are with rules *Sched* or *Receive*, but in this case the derivations could not be cofinal.
   Then, in all the cases, we can repeatedly apply the switching lemma (Lemma 16) 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 the 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. Let us consider first that the rules applied in the transitions are different from *Sched*. Then, 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 16) 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 the inductive hypothesis.

The case when at least one of the rules applied in the transitions is $\overline{Sched}$ follows by a similar reasoning by considering the respective queues instead of the histories. □

We now show that, as a corollary of previous results, a transition can be undone if and only if each of its consequences, if any, has been undone. Formally, a *consequence* of a forward transition t is a forward transition t' that can only happen after t has been performed (assuming t has not been undone in between). Hence t' cannot be switched with t. E.g., consuming a message from the queue of a process (using rule *Receive*) is a consequence of delivering this message (using rule *Sched*). Similarly, every action performed by a process is a consequence of spawning this process.

Corollary 22. *Let $d = (s_1 \rightleftharpoons \cdots \rightleftharpoons s_n \rightharpoonup s_{n+1} \rightleftharpoons \cdots \rightleftharpoons s_m)$ be a derivation, with $t = (s_n \rightharpoonup_{p,r,k} s_{n+1})$ a forward transition. Then, transition $\overline{t}$ can be applied to s_m, i.e., $s_m \leftharpoondown_{p,\overline{r},k} s_{m+1}$ iff each consequence of t in d, if any, has been undone in d.*

Proof. If each consequence t' of t in d has been undone in d then we can find $d' \approx d$ with no consequence of t, by moving each consequence t' and its undoing $\overline{t'}$ close to each other (they can be switched using the switching lemma (Lemma 16) with all the transitions in between, but for further consequences which can be removed beforehand) and then applying $t';\overline{t'} \approx \epsilon_{\mathsf{init}(t')}$. Then we can find $d'' \approx d'$ where t is the last transition, since t is concurrent to all subsequent transitions, hence we can apply the switching lemma (Lemma 16) again. The thesis then follows by applying the loop lemma (Lemma 11).

Assume now that transition $\overline{t}$ can be applied to s_m. Thanks to the rearranging lemma (Lemma 18) there is a derivation $d_b; d_f \approx d;\overline{t}$ where d_b is a backward derivation and d_f is a forward derivation. In order to transform $d;\overline{t}$ into $d_b; d_f$ we need to move $\overline{t}$ backward using the switching lemma (Lemma 16) until we find t. However, neither t nor $\overline{t}$ can be switched with the consequences of t, hence the only possibility is that all the consequences t' of t can be removed using $t';\overline{t'} \approx \epsilon_{\mathsf{init}(t')}$ as above. □

5. Rollback semantics

In this section, we introduce a (nondeterministic) "undo" operation which has some similarities to, e.g., the rollback operator of [18,14]. Here, processes in "rollback" mode are annotated using $\lfloor\ \rfloor_\Psi$, where Ψ is the set of requested rollbacks. A typical rollback refers to a checkpoint that the backward computation of the process has to go through before resuming its forward computation. To be precise, we distinguish the following types of rollbacks:

- $\#^{\mathtt{t}}_{\mathsf{ch}}$, where "ch" stands for "checkpoint": a rollback to undo the actions of a process until a checkpoint with identifier $\mathtt{t}$ is reached;
- $\#_{\mathsf{sp}}$, where "sp" stands for "spawn": a rollback to undo *all* the actions of a process, finally deleting it from the system;
- $\#^{\lambda}_{\mathsf{sch}}$, where "sch" stands for "sched": a rollback to undo the actions of a process until the delivery of a message $\{v, \lambda\}$ is undone.

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

$$(SC)\quad \Gamma;\lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\emptyset \mid \Pi \equiv \Gamma;\langle p, h, (\theta, e), q\rangle \mid \Pi$$

Note that only the first of the rollback types above targets a checkpoint. This kind of checkpoint is introduced nondeterministically by the rule below, where we denote by $\leftharpoondown$ the new reduction relation that models backward moves of the rollback semantics:

$$(\overline{Undo})\quad \Gamma;\lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi \leftharpoondown \Gamma;\lfloor\langle p, h, (\theta, e), q\rangle\rfloor_{\Psi\cup\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \mid \Pi$$
$$\text{if } \mathsf{check}(\theta', e', \mathtt{t}) \text{ occurs in } h, \text{ for some } \theta' \text{ and } e'$$

Only after this rule is applied steps can be undone, since default computation in the rollback semantics is forward.

The backward rules of the rollback semantics are shown in Fig. 15. Here, we assume that $\Psi \neq \emptyset$ (but Ψ' might be empty).

Note that, while rollbacks to checkpoints are generated nondeterministically by rule $\overline{Undo}$, the two other kinds of checkpoints are generated by the backward reduction rules in order to ensure causal consistency (in the sense of Corollary 22). This is clarified by the discussion below, where we briefly explain the main differences w.r.t. the uncontrolled backward semantics:

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

$$(\overline{Check}) \quad \Gamma; \lfloor\langle p, \mathsf{check}(\theta, e, \mathtt{t}){:}h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi \twoheadleftarrow \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_{\Psi\setminus\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \mid \Pi$$

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

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

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

$$(\overline{Spawn1}) \quad \begin{array}{c} \Gamma; \lfloor\langle p, \mathsf{spawn}(\theta, e, p''){:}h, (\theta', e'), q\rangle\rfloor_\Psi \mid \lfloor\langle [\,], p'', (\theta'', e''), [\,]\rangle\rfloor_{\Psi'} \mid \Pi \\ \twoheadleftarrow \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi \end{array}$$

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

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

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

Fig. 15. Rollback semantics: backward reduction rules.

- As in the uncontrolled semantics of Fig. 11, the sending of a message can be undone when the message is still in the global mailbox (rule $\overline{Send1}$). Otherwise, one may need to first apply rule $\overline{Send2}$ in order to "propagate" the rollback mode to the receiver of the message, so that rules $\overline{Sched}$ and $\overline{Send1}$ can be eventually applied.
- As for undoing the spawning of a process p'', rule $\overline{Spawn1}$ steadily applies when both the history and the queue of the spawned process p'' are empty, thus deleting both the history item in p and the process p''. Otherwise, we apply rule $\overline{Spawn2}$ to propagate the rollback mode to process p'' so that, eventually, rule $\overline{Spawn1}$ can be applied.
- Finally, observe that rule $\overline{Sched}$ requires the same side condition as in the uncontrolled semantics. This is needed in order to avoid the commutation of rules $\overline{Receive}$ and $\overline{Sched}$.

The rollback semantics is modelled by the relation $\looparrowright$, which is defined as the union of the forward reversible relation $\rightharpoonup$ (Fig. 10) and the backward relation $\twoheadleftarrow$ defined in Fig. 15. Note that, in contrast to the (uncontrolled) reversible semantics of Section 4, the rollback semantics given by the relation $\looparrowright$ has less nondeterministic choices: all computations run forward except when a rollback action demands some backward steps to recover a previous state of a process (which can be propagated to other processes in order to undo the spawning of a process or the sending of a message).

Note, however, that besides the introduction of rollbacks, there is still some nondeterminism in the backward rules of the rollback semantics: on the one hand, the selection of the process when there are several ongoing rollbacks is nondeterministic; also, in many cases, both rule $\overline{Sched}$ and another rule are applicable to the same process. The semantics could be made deterministic by using a particular strategy to select the processes (e.g., round robin) and applying rule $\overline{Sched}$ whenever possible (i.e., give to $\overline{Sched}$ a higher priority than to the remaining backward rules).

Example 23. Consider again the program shown in Fig. 12. Let us assume that function main/0 is now defined as follows:

$$\begin{array}{l} \mathsf{main}/0 = \mathsf{fun}\,() \rightarrow \ \mathsf{let}\ S = \mathsf{spawn}(\mathsf{server}/0, [\,]) \\ \qquad\qquad\quad\ \ \mathsf{in\ let}\ \_ = \mathsf{spawn}(\mathsf{client}/1, [S]) \\ \qquad\qquad\quad\ \ \mathsf{in\ let}\ X = \mathsf{check}(\mathtt{t}) \\ \qquad\qquad\quad\ \ \mathsf{in\ apply\ client}/1\ (S) \end{array}$$

so that a checkpoint has been introduced after spawning the two processes: the server (s) and one of the clients (c2). Then, by repeating the same forward derivation shown in Fig. 13 (with the additional step to evaluate the checkpoint), we get the following final system:

$$\begin{array}{l} \{\,\};\ \langle \mathsf{c1}, [\mathsf{rec}(\_, \_, m_4, [m_4]), \mathsf{send}(\_, \_, \mathsf{s}, m_3), \mathsf{check}(\_, \_, \mathtt{t}), \mathsf{spawn}(\_, \_, \mathsf{c2}), \\ \qquad\quad \mathsf{spawn}(\_, \_, \mathsf{s})], (\_, \mathsf{ok}), [\,]\rangle \\ \quad \mid \langle \mathsf{s}, [\mathsf{send}(\_, \_, \mathsf{c1}, m_4), \mathsf{rec}(\_, \_, m_3, [m_3]), \mathsf{send}(\_, \_, \mathsf{c2}, m_2), \\ \qquad\quad \mathsf{rec}(\_, \_, m_1, [m_1])], (\_, C[\mathsf{receive}\ \{P, M\} \rightarrow \ldots]), [\,]\rangle \\ \quad \mid \langle \mathsf{c2}, [\underline{\mathsf{rec}(\_, \_, m_2, [m_2])}, \mathsf{send}(\_, \_, \mathsf{s}, m_1)], (\_, \mathsf{ok}), [\,]\rangle \end{array}$$

$$
\begin{array}{lll}
 & \{\,\}; & \lfloor\langle \mathsf{c1}, [\underline{\mathsf{rec}(\_,\_,m_4,[m_4])}, \mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{check}(\_,\_,\mathsf{t}), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], \\
 & & \quad (\_, \mathsf{ok}), [\,]\rangle\rfloor_{\{\#^{\mathsf{t}}_{\mathsf{ch}}\}} \\
 & & \mid \langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & \quad (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, \mathsf{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{\mathit{Receive}}} & \{\,\}; & \lfloor\langle \mathsf{c1}, [\underline{\mathsf{send}(\_,\_,\mathsf{s},m_3)}, \mathsf{check}(\_,\_,\mathsf{t}), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], \\
 & & \quad (\_, C[\mathsf{receive}\ \mathsf{ack} \to \mathsf{ok}]), [m_4]\rangle\rfloor_{\{\#^{\mathsf{t}}_{\mathsf{ch}}\}} \\
 & & \mid \langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & \quad (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, \mathsf{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{\mathit{Send2}}} & \{\,\}; & \lfloor\langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{check}(\_,\_,\mathsf{t}), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], \\
 & & \quad (\_, C[\mathsf{receive}\ \mathsf{ack} \to \mathsf{ok}]), [m_4]\rangle\rfloor_{\{\#^{\mathsf{t}}_{\mathsf{ch}}\}} \\
 & & \mid \lfloor\langle \mathsf{s}, [\underline{\mathsf{send}(\_,\_,\mathsf{c1},m_4)}, \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & \quad (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle\rfloor_{\{\#^{3}_{\mathsf{sch}}\}} \\
 & & \mid \langle \mathsf{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, \mathsf{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{\mathit{Send2}}} & \{\,\}; & \lfloor\langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{check}(\_,\_,\mathsf{t}), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], \\
 & & \quad (\_, C[\mathsf{receive}\ \mathsf{ack} \to \mathsf{ok}]), [\underline{m_4}]\rangle\rfloor_{\{\#^{\mathsf{t}}_{\mathsf{ch}},\#^{4}_{\mathsf{sch}}\}} \\
 & & \mid \lfloor\langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & \quad (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle\rfloor_{\{\#^{3}_{\mathsf{sch}}\}} \\
 & & \mid \langle \mathsf{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, \mathsf{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{\mathit{Sched}}} & \{(\mathsf{c1},m_4)\}; & \lfloor\langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{check}(\_,\_,\mathsf{t}), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], \\
 & & \quad (\_, C[\mathsf{receive}\ \mathsf{ack} \to \mathsf{ok}]), [\,]\rangle\rfloor_{\{\#^{\mathsf{t}}_{\mathsf{ch}}\}} \\
 & & \mid \lfloor\langle \mathsf{s}, [\underline{\mathsf{send}(\_,\_,\mathsf{c1},m_4)}, \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & \quad (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle\rfloor_{\{\#^{3}_{\mathsf{sch}}\}} \\
 & & \mid \langle \mathsf{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, \mathsf{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{\mathit{Send1}}} & \{\,\}; & \lfloor\langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{check}(\_,\_,\mathsf{t}), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], \\
 & & \quad (\_, C[\mathsf{receive}\ \mathsf{ack} \to \mathsf{ok}]), [\,]\rangle\rfloor_{\{\#^{\mathsf{t}}_{\mathsf{ch}}\}} \\
 & & \mid \lfloor\langle \mathsf{s}, [\underline{\mathsf{rec}(\_,\_,m_3,[m_3])}, \mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{c1}\,!\,\mathsf{ack}]), [\,]\rangle\rfloor_{\{\#^{3}_{\mathsf{sch}}\}} \\
 & & \mid \langle \mathsf{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, \mathsf{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{\mathit{Receive}}} & \{\,\}; & \lfloor\langle \mathsf{c1}, [\mathsf{send}(\_,\_,\mathsf{s},m_3), \mathsf{check}(\_,\_,\mathsf{t}), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], \\
 & & \quad (\_, C[\mathsf{receive}\ \mathsf{ack} \to \mathsf{ok}]), [\,]\rangle\rfloor_{\{\#^{\mathsf{t}}_{\mathsf{ch}}\}} \\
 & & \mid \lfloor\langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\underline{m_3}]\rangle\rfloor_{\{\#^{3}_{\mathsf{sch}}\}} \\
 & & \mid \langle \mathsf{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, \mathsf{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{\mathit{Sched}}} & \{(\mathsf{s},m_3)\}; & \lfloor\langle \mathsf{c1}, [\underline{\mathsf{send}(\_,\_,\mathsf{s},m_3)}, \mathsf{check}(\_,\_,\mathsf{t}), \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], \\
 & & \quad (\_, C[\mathsf{receive}\ \mathsf{ack} \to \mathsf{ok}]), [\,]\rangle\rfloor_{\{\#^{\mathsf{t}}_{\mathsf{ch}}\}} \\
 & & \mid \langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, \mathsf{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{\mathit{Send1}}} & \{\,\}; & \lfloor\langle \mathsf{c1}, [\underline{\mathsf{check}(\_,\_,\mathsf{t})}, \mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{s}\,!\,\{\mathsf{c1},\mathsf{req}\}]), [\,]\rangle\rfloor_{\{\#^{\mathsf{t}}_{\mathsf{ch}}\}} \\
 & & \mid \langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, \mathsf{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{\mathit{Check}}} & \{\,\}; & \langle \mathsf{c1}, [\mathsf{spawn}(\_,\_,\mathsf{c2}), \mathsf{spawn}(\_,\_,\mathsf{s})], (\_, C[\mathsf{check}(\mathsf{t})]), [\,]\rangle \\
 & & \mid \langle \mathsf{s}, [\mathsf{send}(\_,\_,\mathsf{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathsf{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathsf{s},m_1)], (\_, \mathsf{ok}), [\,]\rangle
\end{array}
$$

Fig. 16. A derivation under the backward reduction rules of Fig. 15, with $m_1 = \{\{\mathsf{c2}, \mathsf{req}\}, 1\}$, $m_2 = \{\mathsf{ack}, 2\}$, $m_3 = \{\{\mathsf{c1}, \mathsf{req}\}, 3\}$, and $m_4 = \{\mathsf{ack}, 4\}$.

Fig. 16 shows the steps performed by the rollback semantics in order to undo the steps of process c1 until the checkpoint is reached. In Fig. 16 we follow the same conventions as in Examples 7 and 9. Observe that we could also use the relation "$\rightsquigarrow$" here in order to also perform some forward steps on process c2, as it would happen in practice.

We state below the soundness of the rollback semantics. In order to do it, we let $\mathsf{rolldel}(s)$ denote the system that results from s by removing ongoing rollbacks; formally, $\mathsf{rolldel}(\Gamma; \Pi) = \Gamma; \mathsf{rolldel}'(\Pi)$, with

$$
\begin{array}{lcl}
\mathsf{rolldel}'(\langle p, h, (\theta, e), q\rangle) & = & \langle p, h, (\theta, e), q\rangle \\
\mathsf{rolldel}'(\lfloor\langle p, h, (\theta, e), q\rangle\rfloor_{\Psi}) & = & \langle p, h, (\theta, e), q\rangle \\
\mathsf{rolldel}'(\langle p, h, (\theta, e), q\rangle \mid \Pi) & = & \langle p, h, (\theta, e), q\rangle \mid \mathsf{rolldel}'(\Pi) \\
\mathsf{rolldel}'(\lfloor\langle p, h, (\theta, e), q\rangle\rfloor_{\Psi} \mid \Pi) & = & \langle p, h, (\theta, e), q\rangle \mid \mathsf{rolldel}'(\Pi)
\end{array}
$$

where we assume that Π is not empty. We also extend the definition of initial and reachable systems to the rollback semantics.

Definition 24 *(Reachable systems under the rollback semantics).* A system is *initial* under the rollback semantics if it is composed by a single process with an empty set Ψ of active rollbacks; furthermore, the history, the queue and the global mailbox are empty too. A system s is *reachable* under the rollback semantics if there exist an initial system s_0 and a derivation $s_0 \rightarrow\!\!\!\!\!\!\rightarrow^* s$ using the rules corresponding to a given program.

Theorem 25 *(Soundness). Let s be a system reachable under the rollback semantics. If $s \rightarrow\!\!\!\!\!\!\rightarrow^* s'$, then* $\mathsf{rollDel}(s) \rightleftharpoons^* \mathsf{rollDel}(s')$.

Proof. For forward transitions the proof is trivial since the forward rules are the same in both semantics, and they apply only to processes which are not under rollback. For backward transitions the proof is by case analysis on the applied rule, noting that the effect of structural equivalence is removed by rollDel:

- Rule $\overline{Undo}$: the effect is removed by rollDel, hence an application of this rule corresponds to a zero-step derivation under the uncontrolled semantics;
- Rules $\overline{Seq}$, $\overline{Check}$, $\overline{Send1}$, $\overline{Receive}$, $\overline{Spawn1}$, $\overline{Self}$ and $\overline{Sched}$: they are matched, respectively, by rules $\overline{Seq}$, $\overline{Check}$, $\overline{Send}$, $\overline{Receive}$, $\overline{Spawn}$, $\overline{Self}$ and $\overline{Sched}$ of the uncontrolled semantics;
- Rules $\overline{Send2}$ and $\overline{Spawn2}$: the effect is removed by rollDel, hence an application of any of these rules corresponds to a zero-step derivation under the uncontrolled semantics. □

We can now show the completeness of the rollback semantics provided that the involved process is in rollback mode:

Lemma 26 *(Completeness in rollback mode). Let s be a reachable system. If $s \leftharpoondown s'$ then take any system s_r such that* $\mathsf{rollDel}(s_r) = s$ *and where the process that performed the transition $s \leftharpoondown s'$ is in rollback mode for a non-empty set of rollbacks. There exists s'_r such that $s_r \leftharpoondown\!\!\!\!\!\!\leftharpoondown s'_r$ and* $\mathsf{rollDel}(s'_r) = s'$.

Proof. The proof is by case analysis on the applied rule. Each step is matched by the homonymous rule, but for $\overline{Send}$ and $\overline{Spawn}$ which are matched by rules $\overline{Send1}$ and $\overline{Spawn1}$. □

The following result illustrates the usefulness of the rollback semantics:

Lemma 27. *Let us consider a forward derivation d of the form:*

$$\begin{array}{l}\Gamma; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \mid \Pi \\ \rightharpoonup \Gamma; \langle p, \mathsf{check}(\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e, \mathtt{t}) : h, (\theta, \mathsf{let}\ X = \mathtt{t}\ \mathsf{in}\ e), q\rangle \mid \Pi \\ \rightharpoonup^* \Gamma'; \langle p, h', (\theta', e'), q'\rangle \mid \Pi'\end{array}$$

Then, there is a backward derivation d' under the rollback semantics restoring process p:

$$\begin{array}{l}\Gamma'; \lfloor \langle p, h', (\theta', e'), q'\rangle \rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \mid \Pi' \\ \leftharpoondown^* \Gamma''; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \mid \Pi''\end{array}$$

Proof. Trivially (by Theorem 25) the forward derivation d can also be performed under the uncontrolled reversible semantics. Now, by applying the loop lemma (Lemma 11) to each step of d, we have a backward derivation $\overline{d}$ of the form:

$$\begin{array}{l}\Gamma'; \langle p, h', (\theta', e'), q'\rangle \mid \Pi' \\ \leftharpoondown^* \Gamma; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \mid \Pi\end{array}$$

Consider the relation $\leq$ on transitions of $\overline{d}$ defined as the reflexive and transitive closure of the following clauses:

- $t_1 \leq t_2$ if both t_1 and t_2 undo actions in the same process p', and the transition undone by t_2 is a direct consequence of the one undone by t_1;
- $t_1 \leq t_2$ if t_1 undoes a spawn of process p_2 and t_2 undoes the first transition of p_2;
- $t_1 \leq t_2$ if t_1 undoes the send of a message λ and t_2 undoes the scheduling of the same message.

Let us show that $\leq$ is a partial order. We only need to show that there are no cycles, but this follows from the fact that the total order given by $\overline{d}$ is compatible with $\leq$.

We also notice that any two transitions which are not related by $\leq$ can be swapped using the switching lemma (Lemma 16).

Then, there exists a derivation $\overline{d_r}; \overline{d_u}$ such that $\overline{d_r}$ contains all transitions t such that $t_l \leq t$ where t_l is the last transition in $\overline{d}$, and only them. Since $\overline{d_u}$ contains no transition on p we have that $\overline{d_r}$ is of the form:

$$\begin{array}{l}\Gamma'; \langle p, h', (\theta', e'), q'\rangle \mid \Pi' \\ \leftharpoondown^* \Gamma''; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \mid \Pi''\end{array}$$

Using again the switching lemma (Lemma 16) one can transform $\overline{d_r}$ into a derivation $\overline{d_r'}$ obtained using the following execution strategy, where initially the active process is p, the termination condition is "the checkpoint action $\mathtt{t}$ has been undone", and the stack is empty:

- transitions of the active process are undone if possible, until the termination condition holds; if there is an occurrence of the active process in the stack and the termination condition for this process is matched because of the current transition undo, remove such occurrence from the stack (this remove does not follow the usual FIFO strategy for stacks);
- if the termination condition holds, then pop a new active process from the stack, if there are no processes on the stack then terminate;
- if no transition is possible for the active process then one of the two following subconditions should hold:
  1. the active process needs to undo a spawn of a process which is not in the initial state: push the active process on the stack, and set the spawned process as new active process with termination condition "all actions have been undone";
  2. the active process needs to undo a send of a message λ which is not in the global mailbox: push the active process on the stack, and set the process to which message λ has been scheduled as new active process with termination condition "the scheduling of the message λ has been undone".

The switching lemma can be applied since this execution strategy is compatible with $\preceq$. Now we show that the same execution strategy can be performed using the rollback semantics. We only need to show that the active process is in rollback mode, then the thesis will follow from the completeness in rollback mode (Lemma 26). This can be shown by inspection of the execution strategy, considering the following invariant: the active process and all the processes on the stack are in rollback mode, and they have one checkpoint for each occurrence in the stack, plus one for the occurrence as active process. The invariant holds at the beginning since p has one checkpoint corresponding to its termination condition. When the termination condition holds, a checkpoint is removed by rule $\overline{Check}$, $\overline{Spawn1}$, or $\overline{Sched}$. When a new active process is selected, a new checkpoint is added by rule $\overline{Spawn2}$ or $\overline{Send2}$. □

One can notice that in the lemma above only the process containing the checkpoint is restored. We can restore the whole system to the original configuration only if we restrict the forward derivation to be a causal derivation, following the terminology in [10].

Definition 28. A forward derivation d is causal iff all the transitions are consequences of the first one.

Hence, we have the following corollary:

Corollary 29. *Let us consider a causal derivation d of the form:*

$$\begin{aligned}&\Gamma; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \mid \Pi\\ &\rightharpoonup \Gamma; \langle p, \mathsf{check}(\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e, \mathtt{t}) : h, (\theta, \mathsf{let}\ X = \mathtt{t}\ \mathsf{in}\ e), q\rangle \mid \Pi\\ &\rightharpoonup^* \Gamma'; \langle p, h', (\theta', e'), q'\rangle \mid \Pi'\end{aligned}$$

Then, there is a backward derivation d' under the rollback semantics restoring the system to the original configuration:

$$\begin{aligned}&\Gamma'; \lfloor \langle p, h', (\theta', e'), q'\rangle \rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \mid \Pi'\\ &\leftharpoondown^* \Gamma; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \mid \Pi\end{aligned}$$

Proof. The proof follows the same strategy as the one of Lemma 27, noticing that $\overline{d_u}$ is empty hence $\Gamma = \Gamma''$ and $\Pi = \Pi''$. □

While a derivation restoring the whole system exists, not all derivations do so. More in general, given a set of rollbacks, it is not the case that there is a unique system that is obtained by executing backward transitions as far as possible (without executing any $\overline{Undo}$). Indeed, the only nondeterminism is due to the fact that $\overline{Sched}$ can commute with other transitions, e.g., with $\overline{Check}$, which ends the rollback. If we establish a policy for *Sched* actions, and we use the dual policy for undoing them, then the result is unique. A sample policy could be that *Sched* steps are performed as late as possible, and dually undone as soon as possible. In such a setting we have the following result:

Lemma 30. *Let s be a reachable system. If $s \leftharpoondown s_1$ and $s \leftharpoondown s_2$, both transitions use the same policy for $\overline{Sched}$, and the rules are different from $\overline{Undo}$, then there exists a system s' such that $s_1 \leftharpoondown^* s'$ and $s_2 \leftharpoondown^* s'$.*

Proof. Let us consider the case where both transitions are applied to the same process p. In this case, only one backward rule is applicable and the claim follows trivially. Note that the only case where more than one backward rule would be applicable is when one of the rules is $\overline{Sched}$ and the other one is a different rule, but this case is excluded by the fact that we consider a fixed policy for $\overline{Sched}$ as mentioned above.

Consider now the case where each transition is applied to a different process, say p_1 and p_2, so that we have $s \rightharpoonup s_1$ and $s \rightharpoonup s_2$. By the soundness of the backward reduction rules of the rollback semantics (Theorem 25), we have $\mathsf{rolldel}(s) \leftharpoonup^* \mathsf{rolldel}(s_1)$ and $\mathsf{rolldel}(s) \leftharpoonup^* \mathsf{rolldel}(s_2)$. Note that each of the derivations above has either length 1 or 0. We just consider the case where they have both length 1, since the others are simpler. By the square lemma (Lemma 13), there exists a system s'' such that $\mathsf{rolldel}(s_1) \leftharpoonup s''$ and $\mathsf{rolldel}(s_2) \leftharpoonup s''$. Now, we show that processes p_2 and p_1 are still in rollback mode in s_1 and s_2, respectively. Here, the only case where the application of a backward rule to a process removes a rollback from a different process is *Spawn*. Consider, e.g., that the rule applied to process p_1 is *Spawn* and that the removed process is p_2. In this case, however, no backward rule could be applied to process p_2, so this case is not possible. Therefore, by applying the completeness of the rollback semantics, we have $s_1 \rightharpoonup s_1'$ and $s_2 \rightharpoonup s_2'$ with $\mathsf{rolldel}(s_1') = \mathsf{rolldel}(s_2') = s''$. The thesis follows by noticing that the rollbacks in s_1' and s_2' coincide (in both the cases they are the rollbacks in s minus the ones removed by the performed transitions, which are the same in both the cases) hence $s_1' = s_2' = s'$. □

The following result is an easy corollary of the previous lemma:

Corollary 31. *Let s be a reachable system. If $s \rightharpoonup^* s_1 \not\rightharpoonup$ and $s \rightharpoonup^* s_2 \not\rightharpoonup$, both derivations use the same policy for $\overline{Sched}$, and never use rule Undo, then $s_1 = s_2$.*

Proof. Analogously to the proof of Corollary 14, using standard results for confluence of abstract relations [2], we have that Lemma 30 implies that there exists a system s' such that $s_1 \rightharpoonup^* s'$ and $s_2 \rightharpoonup^* s'$. Moreover, since both s_1 and s_2 are irreducible, we have $s_1 = s_2$. □

6. Proof-of-concept implementation of the reversible semantics

We have developed a proof-of-concept implementation of the uncontrolled reversible semantics for Erlang that we presented in Section 3. This implementation is conveniently bundled together with a graphical user interface (we refer to this as "the application") in order to facilitate the interaction of users with the reversible semantics. However, the application has been developed in a modular way, so that it is possible to include the implementation of the reversible semantics in other projects (e.g., it has been included in the reversible debugger CauDEr [22,21]).

Let us recall that our semantics is defined for a language that is equivalent to Core Erlang [7], a much simpler language than Erlang. Not surprisingly, the implementation of our reversible semantics is defined for Core Erlang as well. Prior to its compilation, Erlang programs are translated to Core Erlang by the Erlang/OTP system, so that the resulting code is simplified. For instance, pattern matching can occur almost anywhere in an Erlang program, whereas in Core Erlang, pattern matching can only occur in case statements. Nevertheless, directly writing Core Erlang programs would not be comfortable for the user, since Core Erlang is only used as an intermediate language. Hence, our implementation considers the Core Erlang code translated from the Erlang program provided by the user.

The application works as follows: when it is started, the first step is to select an Erlang source file. The selected source file is then translated into Core Erlang, and the resulting code is shown in the code window. Then, the user can choose any of the functions from the module and write the arguments that she wants to evaluate the function with. An initial system state, with an empty global mailbox and a single process performing the specified function application, appears on the state window when the user presses the start button, as shown in Fig. 17. Now, the user is able to control the system state by selecting the rules from the reversible semantics that she wants to fire.

We have defined two different modes for controlling the reversible semantics. The first mode is a *manual* mode, where the user selects the rule to be fired for a particular process or message. Here, the user is in charge of "controlling" the reversible semantics, although this approach can rapidly become exhausting. The second mode is the *automatic* mode. Here, the user specifies a number of steps and chooses a direction (forward or backward), and the rules to be applied are selected at random—for the chosen direction—until the specified number of steps is reached or no more rules can be applied. Alternatively, the user can move the state forward up to a *normalised system*. To normalise a system, one must ignore the *Sched* rule and apply only the other rules. A normalised system is reached when no rule other than *Sched* can be fired. Hence, in a normalised system, either all processes are blocked (waiting for some message to arrive) or the system state is final. Normalising a system allows the user to perform all the reductions that do not depend on the network. Reductions depending on the network can then be performed one by one to understand their impact on the derivation.

The release version (v1.0) of the application is fully written in Erlang, and it is publicly available from https://github.com/mistupv/rev-erlang under the MIT license. Hence, the only requirement to build the application is to have Erlang/OTP installed. Besides, we have included some documentation and a few examples to easily test the application.

7. Related work

First, regarding the semantics of Erlang presented in Section 3, we have some similarities with both [5] and [30]. In contrast to [5], which presents a monolithic semantics, our relation is split into expression-level rules and system-level rules. This division eases the presentation of a reversible semantics, since it only affects the system-level rules. As for [30],

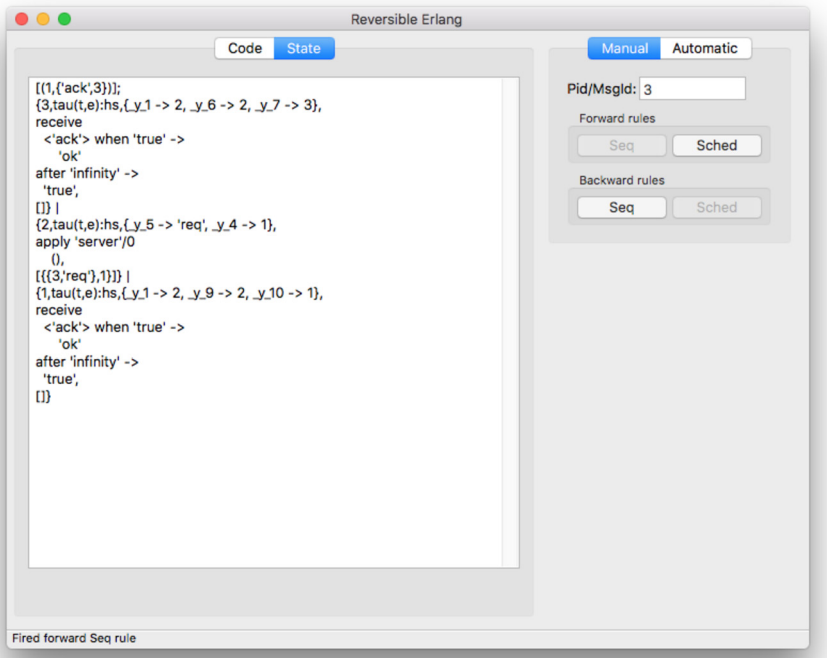


Fig. 17. Screenshot of the application.

we follow the idea of introducing a global mailbox (there called "ether") so that every message passing communication can be decomposed into two steps: sending and scheduling. Their semantics considers other features of Erlang (such as links or monitors) but does not present the semantics of expressions, as we do. Another difference lies in the fact that all *side effects* are asynchronous in [30] (e.g., the spawning of a process is asynchronous), a design decision that allows for a simpler semantics. In our case, spawning a process is dealt with in a synchronous manner, which is closer to the actual behaviour of Erlang. Finally, as mentioned in Section 3, we deliberately ignore the restriction that guarantees the order of messages for any pair of given processes. This may increase the number of possible interleavings, but we consider that it models better the behaviour of current Erlang implementations.

Regarding reversibility, the approach presented in this paper is in the line of work on causal-consistent reversibility [9,28] (see [20] for a survey). In particular, our work is closer to [9], since we also consider adding a *memory* (a history in our terminology) in order to make a computation reversible. Moreover, our proof of causal consistency mostly follows the proof scheme in [9]. In contrast, we consider a different concurrent language with asynchronous communication, while communication in [9] is synchronous. On the other hand, [28] does not introduce a memory but keeps the old actions marked with a "key". As pointed out in [28], process equivalence is easier to check than in [9] (where one would need to abstract away from the memories). Like [9], also [28] considers synchronous communication. Formalising the Erlang semantics using a labelled transition relation as in [9,28] (rather than a reduction semantics, as we do in this paper), and then defining a reversible extension would be an interesting and challenging approach for further research.

Nevertheless, as mentioned in the Introduction, the closest to our work is the debugging approach based on a rollback construct of [14,15,18,19,23], but it is defined in the context of a different language or formalism. Among the languages considered in the works above, the closest to ours is μOz [23,14]. A main difference is that μOz is not distributed: messages move atomically from the sender to a chosen queue, and from the queue to the receiver. Each of the two actions is performed by a specific process, hence naturally part of its history. In our case, the scheduling action is not directly performed by a process, and it is only potentially observed when the target process performs the receive action (but not necessarily observed, e.g., if the message does not match the patterns in the receive). The definition of the notions of conflict and concurrency in this setting is, as a consequence, much trickier than in μOz. This difficulty carries over to the definition of the history information that needs to be tracked, and to how this information is exploited in the reversible semantics (actually, this was one of the main difficulties we encountered during our work). Furthermore, in the case of μOz only the uncontrolled semantics has been fully formalised [23], while the controlled semantics and the corresponding results are only sketched [14].

Also, we share some similarities with the checkpointing technique for fault-tolerant distributed computing of [11,16], although the aim is different (they aim at defining a new language rather than extending an existing one).

On the other hand, [25] has very recently introduced a novel technique for recovery in Erlang based on session types. Although the approach is different, our rollback semantics could also be used for rollback recovery. In contrast to [25], that

only considers recovery of processes as a whole, our approach could be helpful to design a more fine grained recovery strategy.

Finally, as mentioned in the Introduction, this paper extends and improves [27] in different ways. Firstly, [27] only presents a rollback semantics. Here, we have introduced an uncontrolled reversible semantics and have proved a number of fundamental theoretical properties, including its causal consistency (no proofs of technical results are provided in [27]). Secondly, the reversible semantics in [27] does not consider messages' unique identifiers (λ), so that the problems mentioned in Section 4 are not avoided. Moreover, the process' histories also include items for the applications of rule *Sched*, which makes the underlying notion of concurrency unnecessarily restrictive. As for the rollback semantics of [27], besides the points mentioned above, it only considered one rollback for each process, while sets of rollbacks are accepted in this work. Consequently, we have now reduced the number of rules required to undo the sending of a message or to undo the introduction of a checkpoint, so that the rollback semantics is simpler. Furthermore, we have designed and developed a proof-of-concept implementation in this paper that allowed us to check the viability of the reversible semantics in practice.

8. Conclusion and future work

We have defined a reversible semantics for a first-order subset of Erlang that undoes the actions of a process step by step in a sequential way. To the best of our knowledge, this is the first attempt to define a reversible semantics for Erlang. In this work, we have first introduced an uncontrolled, reversible semantics, and have proved that it enjoys the usual properties (loop lemma, square lemma, and causal consistency). Then, we have introduced a controlled version of the backward semantics that can be used to model a rollback operator that undoes the actions of a process up to a given checkpoint. A proof-of-concept implementation shows that our approach is indeed viable in practice.

As future work, we consider the definition of mechanisms to control reversibility so that history information is stored only when needed to perform a rollback. This could be essential to extend Erlang with a new construct for *safe sessions*, where all the actions in a session can be undone if the session aborts. Such a construct could have a great potential to automate the fault-tolerance capabilities of the language Erlang.

References

[1] J. Armstrong, R. Virding, C. Wikström, M. Williams, Concurrent Programming in Erlang, 2nd edition, Prentice Hall, 1996.
[2] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.
[3] C. Bennett, Logical reversibility of computation, IBM J. Res. Dev. 17 (1973) 525–532.
[4] C. Bennett, Notes on the history of reversible computation, IBM J. Res. Dev. 44 (1) (2000) 270–278.
[5] R. Caballero, E. Martín-Martín, A. Riesco, S. Tamarit, A Declarative Debugger for Concurrent Erlang Programs (extended version), Technical Report SIC-15/13, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2013.
[6] L. Cardelli, C. Laneve, Reversible structures, in: F. Fages (Ed.), Proceedings of the 9th International Conference on Computational Methods in Systems Biology, CMSB 2011, ACM, 2011, pp. 131–140.
[7] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, R. Virding, Core erlang 1.0.3. language specification, Available from https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf, 2004.
[8] I. Cristescu, J. Krivine, D. Varacca, A compositional semantics for the reversible p-calculus, in: 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, IEEE Computer Society, 2013, pp. 388–397.
[9] V. Danos, J. Krivine, Reversible communicating systems, in: P. Gardner, N. Yoshida (Eds.), Proc. of the 15th International Conference on Concurrency Theory, CONCUR 2004, in: Lecture Notes in Computer Science, vol. 3170, Springer, 2004, pp. 292–307.
[10] V. Danos, J. Krivine, Transactions in RCCS, in: M. Abadi, L. de Alfaro (Eds.), Proc. of the 16th International Conference on Concurrency Theory, CONCUR 2005, in: Lecture Notes in Computer Science, vol. 3653, Springer, 2005, pp. 398–412.
[11] J. Field, C.A. Varela, Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments, in: J. Palsberg, M. Abadi (Eds.), Proceedings of the 32nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, POPL 2005, ACM, 2005, pp. 195–208.
[12] M.P. Frank, Introduction to reversible computing: motivation, progress, and challenges, in: N. Bagherzadeh, M. Valero, A. Ramírez (Eds.), Proceedings of the Second Conference on Computing Frontiers, ACM, 2005, pp. 385–390.
[13] L.-A. Fredlund, A Framework for Reasoning About Erlang Code, PhD Thesis, The Royal Institute of Technology, Sweeden, 2001.
[14] E. Giachino, I. Lanese, C.A. Mezzina, Causal-consistent reversible debugging, in: S. Gnesi, A. Rensink (Eds.), Proc. of the 17th International Conference on Fundamental Approaches to Software Engineering, FASE 2014, in: Lecture Notes in Computer Science, vol. 8411, Springer, 2014, pp. 370–384.
[15] E. Giachino, I. Lanese, C.A. Mezzina, F. Tiezzi, Causal-consistent reversibility in a tuple-based language, in: M. Daneshtalab, M. Aldinucci, V. Leppänen, J. Lilius, M. Brorsson (Eds.), Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, IEEE Computer Society, 2015, pp. 467–475.
[16] P. Kuang, J. Field, C.A. Varela, Fault tolerant distributed computing using asynchronous local checkpointing, in: E.G. Boix, P. Haller, A. Ricci, C. Varela (Eds.), Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control, AGERE! 2014, ACM, 2014, pp. 81–93.
[17] R. Landauer, Irreversibility and heat generation in the computing process, IBM J. Res. Dev. 5 (1961) 183–191.
[18] I. Lanese, C.A. Mezzina, A. Schmitt, J. Stefani, Controlling reversibility in higher-order pi, in: J. Katoen, B. König (Eds.), Proceedings of the 22nd International Conference on Concurrency Theory, CONCUR 2011, in: Lecture Notes in Computer Science, vol. 6901, Springer, 2011, pp. 297–311.
[19] I. Lanese, C.A. Mezzina, J. Stefani, Reversibility in the higher-order π-calculus, Theor. Comput. Sci. 625 (2016) 25–84.
[20] I. Lanese, C.A. Mezzina, F. Tiezzi, Causal-consistent reversibility, Bull. Eur. Assoc. Theor. Comput. Sci. 114 (2014).
[21] I. Lanese, N. Nishida, A. Palacios, G. Vidal, CauDEr website. URL: https://github.com/mistupv/cauder.
[22] I. Lanese, N. Nishida, A. Palacios, G. Vidal, CauDEr: a causal-consistent reversible debugger for Erlang, in: J.P. Gallagher, M. Sulzmann (Eds.), Proceedings of the 14th International Symposium on Functional and Logic Programming (FLOPS 2018), in: Lecture Notes in Computer Science, vol. 10818, Springer-Verlag, Berlin, 2018, pp. 247–263.

[23] M. Lienhardt, I. Lanese, C.A. Mezzina, J. Stefani, A reversible abstract machine and its space overhead, in: H. Giese, G. Rosu (Eds.), Proceedings of the Joint 14th IFIP WG Int'l Conf. on Formal Techniques for Distributed Systems (FMOODS 2012) and the 32nd IFIP WG 6.1 International Conference, FORTE 2012, in: Lecture Notes in Computer Science, vol. 7273, Springer, 2012, pp. 1–17.
[24] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, M. Takeichi, Bidirectionalization transformation based on automatic derivation of view complement functions, in: R. Hinze, N. Ramsey (Eds.), Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, ACM, 2007, pp. 47–58.
[25] R. Neykova, N. Yoshida, Let it recover: multiparty protocol-induced recovery, in: P. Wu, S. Hack (Eds.), Proceedings of the 26th International Conference on Compiler Construction, CC 2017, ACM, 2017, pp. 98–108.
[26] N. Nishida, A. Palacios, G. Vidal, Reversible term rewriting, in: D. Kesner, B. Pientka (Eds.), 1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, in: LIPIcs, vol. 52, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016, pages 28:1–28:18.
[27] N. Nishida, A. Palacios, G. Vidal, A reversible semantics for Erlang, in: M. Hermenegildo, P. López-García (Eds.), Proc. of the 26th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2016, in: LNCS, vol. 10184, Springer, 2017, pp. 259–274, Preliminary version available from https://arxiv.org/abs/1608.05521.
[28] I. Phillips, I. Ulidowski, Reversing algebraic process calculi, J. Log. Algebraic Program. 73 (1–2) (2007) 70–96.
[29] B.K. Rosen, Tree-manipulating systems and Church–Rosser theorems, J. ACM 20 (1) (1973) 160–187.
[30] H. Svensson, L.-A. Fredlund, C.B. Earle, A unified semantics for future Erlang, in: Proc. of the 9th ACM SIGPLAN Workshop on Erlang, ACM, 2010, pp. 23–32.
[31] M.K. Thomsen, H.B. Axelsen, 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, pages 8:1–8:13.
[32] F. Tiezzi, N. Yoshida, Reversible session-based pi-calculus, J. Log. Algebraic Methods Program. 84 (5) (2015) 684–707.
[33] T. Yokoyama, Reversible computation and reversible programming languages, Electron. Notes Theor. Comput. Sci. 253 (6) (2010) 71–81, Proc. of the Workshop on Reversible Computation (RC 2009).
[34] T. Yokoyama, H.B. Axelsen, R. Glück, Principles of a reversible programming language, in: A. Ramírez, G. Bilardi, M. Gschwind (Eds.), Proc. of the 5th Conference on Computing Frontiers, ACM, 2008, pp. 43–54.