From Reversible Computation to Checkpoint-Based Rollback Recovery for Message-Passing Concurrent Programs*

Germán Vidal

VRAIN, Universitat Politècnica de València
gvidal@dsic.upv.es

Abstract. The reliability of concurrent and distributed systems often depends on some well-known techniques for fault tolerance. One such technique is based on checkpointing and rollback recovery. Checkpointing involves processes to take snapshots of their current states regularly, so that a rollback recovery strategy is able to bring the system back to a previous consistent state whenever a failure occurs. In this paper, we consider a message-passing concurrent programming language and propose a novel rollback recovery strategy that is based on some explicit checkpointing operators and the use of a (partially) reversible semantics for rolling back the system.

Keywords: reversible computation, message-passing, concurrency, rollback recovery, checkpointing

1 Introduction

The reliability of concurrent and distributed systems often depends on some well-known techniques for fault tolerance. In this context, a popular approach is based on *checkpointing* and *rollback recovery* (see, e.g., the survey by Elnozahy *et al* [3]). Checkpointing requires each process to take a snapshot of its state at specific points in time. This state is stored in some stable memory so that it can be accessed in case of a failure. Then, if an unexpected error happens, a recovery strategy is responsible of rolling back the necessary processes to a previous checkpoint so that we recover a consistent state of the complete system and normal execution can be safely resumed.

In this paper, we consider the definition of a rollback recovery strategy based on three explicit operators: check, commit, and rollback. The first operator, check, is used to define a checkpoint, thus saving the current state of a process. The checkpoint is assigned a fresh identifier τ. Then, one can either *commit* the

* This work has been partially supported by grant PID2019-104735RB-C41 funded by MCIN/AEI/ 10.13039/501100011033, by French ANR project DCore ANR-18-CE25-0007, and by *Generalitat Valenciana* under grant CIPROM/2022/6 (FassLow).

computation performed so far (up to the checkpoint), $\mathsf{commit}(\tau)$, or roll back to the state immediately before the checkpoint, $\mathsf{rollback}(\tau)$.

There are several possible uses for these operators. For example, the functional and concurrent language Erlang [5] includes the usual try_catch expression, which in its simplest form is as follows: "`try` e `catch` $\_ : \_ \to e'$ `end`." Here, if the evaluation of expression e terminates with some value, then try_catch reduces to this value. Otherwise, if an exception is raised (no matter the exception in this example since "$\_ : \_$" catches all of them), the execution jumps to the catch statement and e' is evaluated instead. However, the actions performed during the incomplete evaluation of e are not undone, which may give rise to an inconsistent state of the system. Using the above operators, we could write down a safer version of the try_catch expression above as follows:

$$\texttt{try}\ T = \mathsf{check},\ X = e,\ \mathsf{commit}(T),\ X\ \texttt{catch}\ \_ : \_ \to \mathsf{rollback}(T),\ e'\ \texttt{end}$$

In this case, we first introduce a checkpoint which reduces to a fresh (unique) identifier, say τ, and saves the current state of the process as a side-effect; variable T is bound to τ. Then, if the evaluation of expression e completes successfully, we gather the computed value in variable X, which is returned after $\mathsf{commit}(\tau)$ removes the checkpoint.[1] Otherwise, if an exception is raised, the execution jumps to the catch statement and $\mathsf{rollback}(\tau)$ rolls back the process to the state saved by checkpoint τ (possibly also rolling back other processes in order to get a *causally consistent* state; see below).

Our approach to rollback recovery is based on the notion of *reversible computation* (see [1,13] and references therein). In principle, most programming languages are irreversible, in the sense that the execution of a statement cannot generally be undone. This is the case of Erlang, for instance. Nevertheless, in these languages, one can still define a so-called *Landauer embedding* [19] so that computations become reversible. Intuitively speaking, this operation amounts to defining an instrumented semantics where states carry over a *history* with past states. While this approach may seem impractical at first, there are several useful reversibilization techniques that are roughly based on this idea (typically including some optimization to reduce the amount of saved information, as in, e.g., [28,32]).

While the notion of reversible computation is quite natural in a sequential programming language, the extension to concurrent and distributed languages presents some additional challenges. Danos and Krivine [2] first introduced the notion of *causal consistency* in the context of a reversible calculus (Milner's CCS [30]). Essentially, in a causal consistent reversible setting, *an action cannot be undone until all the actions that causally depend on this action have been already undone*. E.g., we cannot undo the spawning of a process until all the actions of this process have been already undone; similarly, we cannot undo the

[1] Binding a temporal variable X to the evaluation of expression e is required so that the try_catch expression still reduces to the same value of the original try_catch expression; if we had just "$T = \mathsf{check},\ e,\ \mathsf{commit}(T)$" then this sequence would reduce to the value returned by commit in Erlang, thus changing the original semantics.

sending of a message until its reception (and the subsequent actions) have been undone. This notion of causality is closely related with Lamport's "happened before" relation [18]. In our work, we use a similar notion of causality to either propagate checkpoints and to perform causal consistent rollbacks.

Our main contributions are the following. First, we propose the use of three explicit operators for rollback recovery and provide their semantics. They can be used for defining a sort of *safe* transactions, as mentioned above, but not only. For instance, they could also be used as the basis of a reversible debugging scheme where only some computations of interest are reversible, thus reducing the run time overhead. Then, we define a rollback semantics for the extended language that may proceed both as the standard semantics (when no checkpoint is active) or as a reversible semantics (otherwise). Finally, we prove the soundness of our approach w.r.t. an *uncontrolled* reversible semantics.

2 A Message-Passing Concurrent Language

In this work, we consider a simple concurrent language that mainly follows the *actor model* [16]. Here, a running system consists of a number of processes (or actors) that can (dynamically) create new processes and can only interact through message sending and receiving (i.e., no shared memory). This is the case, e.g., of (a significant subset of) the functional and concurrent language Erlang [5].

In the following, we will ignore the sequential component of the language (e.g., a typical eager functional programming language in the case of Erlang) and will focus on its concurrent actions:

- *Process spawning.* A process may spawn new processes dynamically. Each process is identified by a *pid* (a shorthand for *p*rocess *id*entifier), which is unique in a running system.
- *Message sending.* A process can send a message to any other process as long as it knows the pid of the target process. This action is asynchronous.
- *Message reception.* Messages need not be immediately consumed by the target process; rather, they are stored in an associated mailbox until they are consumed (if any). We consider so-called *selective* receives, i.e., a process does not necessarily consume the messages in its mailbox in the same order they were delivered, since receive statements may impose additional constraints. When no message matches any constraint, the execution of the process is *blocked* until a matching message reaches its mailbox.

In the following, we let $s, s', \ldots$ denote *states*, typically including some environment, an expression (or statement) to be evaluated and, in some case, a stack. The structure of states is not relevant for the purpose of this paper, though.

A *process* configuration is denoted by a tuple of the form $\langle p, s \rangle$, where p is the pid of the process and s is its current state. Messages have the form (p, p', v) where p is the pid of the sender, p' that of the receiver, and v is the message value. A *system* is then denoted by a parallel composition of both processes and (*floating*) messages, as in [26,20] (instead of using a *global mailbox*, as in [23,25]).

$$(Seq) \quad \frac{s \xrightarrow{\mathsf{seq}} s'}{\langle p, s\rangle \rightarrowtail_{p,\mathsf{seq}} \langle p, s'\rangle}$$

$$(Send) \quad \frac{s \xrightarrow{\mathsf{send}(p',v)} s'}{\langle p, s\rangle \rightarrowtail_{p,\mathsf{send}} (p, p', v) \mid \langle p, s'\rangle}$$

$$(Receive) \quad \frac{s \xrightarrow{\mathsf{rec}(\kappa,cs)} s' \text{ and } \mathsf{matchrec}(cs, v) = cs_i}{(p', p, v) \mid \langle p, s\rangle \rightarrowtail_{p,\mathsf{rec}} \langle p, s'[\kappa \leftarrow cs_i]\rangle}$$

$$(Spawn) \quad \frac{s \xrightarrow{\mathsf{spawn}(\kappa,s_0)} s' \text{ and } p' \text{ is a fresh pid}}{\langle p, s\rangle \rightarrowtail_{p,\mathsf{spawn}(p')} \langle p, s'[\kappa \leftarrow p']\rangle \mid \langle p', s_0\rangle}$$

$$(Par) \quad \frac{S_1 \rightarrowtail_e S_1' \text{ and } id(S_1') \cap id(S_2) = \emptyset}{S_1 \mid S_2 \rightarrowtail_e S_1' \mid S_2}$$

Fig. 1: Standard semantics

A floating message thus represents a message that has been already sent but not yet delivered (i.e., the message is in the network). Furthermore, in this work, process mailboxes are abstracted away for simplicity, thus a floating message can also represent a message that is actually stored in a process mailbox.[2]

Systems range over by S, S', S_1, etc. Here, the parallel composition operator is denoted by "|" and considered commutative and associative. Therefore, two systems are considered equal if they are the same up to associativity and commutativity.

As in [23,25], the semantics of the language is defined in a modular way, so that the labeled transition relations $\rightarrow$ and $\rightarrowtail$ model the evaluation of expressions (or statements) and the evaluation of systems, respectively.

In the following, we skip the definition of the local semantics ($\rightarrow$) since it is not necessary for our developments; we refer the interested reader to [23,14]. The rules of the operational semantics that define the reduction of systems is shown in Figure 1. The transition steps are labeled with the pid of the selected process and the considered action: seq, send, rec, or $\mathsf{spawn}(p')$, where p' is the pid of the spawned process. Let us briefly explain these rules:

- Sequential, local steps are dealt with rule *Seq*. Here, we just propagate the reduction from the local level to the system level.
- Rule *Send* applies when the local evaluation requires sending a message as a side effect. The local step $s \xrightarrow{\mathsf{send}(p',v)} s'$ is labeled with the information that must flow from the local level to the system level: the pid of the target process, p', and the message value, v. The system rule then adds a new message of the form (p, p', v) to the system, where p is the pid of the sender, p' the pid of the target process, and v the message value.

[2] In Erlang, the order of messages sent directly from process p to process p' is preserved when they are all delivered; see [7, Section 10.8]. We ignore this constraint for simplicity, but could be ensured by introducing triples of the form (p, p', vs) where vs is a queue of messages instead of a single message.

- In order to receive a message, the situation is somehow different. Here, we need some information to flow both from the local level to the system level (the clauses cs of the receive statement) and vice versa (the selected clause, cs_i, if any). For this purpose, in rule *Receive*, the label of the local step includes a special variable κ —a sort of *future*— that denotes the position of the receive expression within state s. The rule then checks if there is a floating message v addressed to process p that matches one of the constraints in cs. This is done by the auxiliary function $\mathsf{matchrec}$, which returns the selected clause cs_i of the receive statement in case of a match (the details are not relevant here). Then, the reduction proceeds by binding κ in s' with the selected clause cs_i, which we denote by $s'[\kappa \leftarrow cs_i]$.
- Rule *Spawn* also requires a bidirectional flow of information. Here, the label of the local step includes the future κ and the state of the new process s_0. The rule then produces a fresh pid, p', adds the new process $\langle p', s_0\rangle$ to the system, and updates the state s' by binding κ to p' (since spawn reduces to the pid of the new process), which we denote by $s'[\kappa \leftarrow p']$.
- Finally, rule *Par* is used to lift an evaluation step to a larger system [26]. The auxiliary function *id* takes a system S and returns the set of pids in S, in order to ensure that new pids are indeed fresh in the complete system.

In the following, $\rightarrowtail^*$ denotes the transitive and reflexive closure of $\rightarrowtail$. Given systems S_0, S_n, we let $S_0 \rightarrowtail^* S_n$ denote a *derivation* under the standard semantics. When we want to consider the individual steps of a derivation, we often write $S_0 \rightarrowtail_{p_1,a_a} S_1 \rightarrowtail_{p_2,a_2} \ldots \rightarrowtail_{p_n,a_n} S_n$. A reduction step usually consists of a number of applications of rule *Par* until a process, or a combination of a process and a message, is selected, so that one of the remaining rules can be applied (*Seq*, *Send*, *Receive* or *Spawn*). In the following, we often omit the steps with rule *Par* and only show the reductions on the selected process, i.e., $a_i \in \{\mathsf{seq}, \mathsf{send}, \mathsf{rec}, \mathsf{spawn}(p')\}$.

An *initial* system has the form $\langle p, s_0\rangle$, i.e., it contains a single process. A system S' is *reachable* if there exists a derivation $S \rightarrowtail^* S'$ such that S is an initial system. A derivation $S \rightarrowtail^* S'$ is *well-defined* under the standard semantics if S is a reachable system.

The semantics in Figure 1 applies to a significant subset of the programming language Erlang [5], as described, e.g., in [14,25]. However, for clarity, in the examples we will consider a much simpler notation which only shows some relevant information regarding the concurrent actions performed by each process. This is just a textual representation which makes explicit process interaction but is not the actual program. In particular, we describe the concurrent actions of a process by means of the following items:

- $p \leftarrow \mathsf{spawn}()$, for process spawning, where p is the (fresh) pid returned by the call to spawn and assigned to the new process;
- $\mathsf{send}(p, v)$, for sending a message, where p is the pid of the target process and v the message value (which could be a tuple including the process own pid in order to get a reply, a common practice in Erlang);
- $\mathsf{rec}(v)$, for receiving message v.

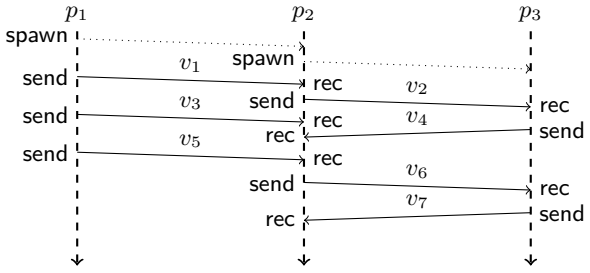

Fig. 2: Graphical representation of a reduction (time flows from top to bottom)

We will ignore sequential actions in the examples since they are not relevant for the purpose of this paper.

Example 1. Consider, for instance, a system with three processes with pids p_1 (the initial one), p_2, and p_3, which perform the following actions:[3]

proc p_1	**proc** p_2	**proc** p_3
$p_2 \leftarrow \mathsf{spawn}()$	$p_3 \leftarrow \mathsf{spawn}()$	$\mathsf{rec}(v_2)$
$\mathsf{send}(p_2, v_1)$	$\mathsf{rec}(v_1)$	$\mathsf{send}(p_2, v_4)$
$\mathsf{send}(p_2, v_3)$	$\mathsf{send}(p_3, v_2)$	$\mathsf{rec}(v_6)$
$\mathsf{send}(p_2, v_5)$	$\mathsf{rec}(v_3)$	$\mathsf{send}(p_2, v_7)$
	$\mathsf{rec}(v_4)$	
	$\mathsf{rec}(v_5)$	
	$\mathsf{send}(p_3, v_6)$	
	$\mathsf{rec}(v_7)$	

A (partial) derivation under the standard semantics (representing a particular interleaving of the processes' actions) may proceed as follows, where we underline the selected process and message (if any) at each step:

$$
\begin{aligned}
&\underline{\langle p_1, s[p_2 \leftarrow \mathsf{spawn}()]\rangle} \\
&\rightarrowtail_{p_1,\mathsf{spawn}(p_2)} \langle p_1, s[\mathsf{send}(p_2, v_1)]\rangle \mid \underline{\langle p_2, s[p_3 \leftarrow \mathsf{spawn}()]\rangle} \\
&\rightarrowtail_{p_2,\mathsf{spawn}(p_3)} \underline{\langle p_1, s[\mathsf{send}(p_2, v_1)]\rangle} \mid \langle p_2, s[\mathsf{rev}(v_1)]\rangle \mid \langle p_3, s[\mathsf{rec}(v_2)]\rangle \\
&\rightarrowtail_{p_1,\mathsf{send}} \langle p_1, s[\mathsf{send}(p_2, v_3)]\rangle \mid \underline{\langle p_2, s[\mathsf{rec}(v_1)]\rangle} \mid \langle p_3, s[\mathsf{rec}(v_2)]\rangle \mid \underline{(p_1, p_2, v_1)} \\
&\rightarrowtail_{p_2,\mathsf{rec}} \langle p_1, s[\mathsf{send}(p_2, v_3)]\rangle \mid \underline{\langle p_2, s[\mathsf{send}(p_3, v_2)]\rangle} \mid \langle p_3, s[\mathsf{rec}(v_2)]\rangle \\
&\rightarrowtail_{p_2,\mathsf{send}} \langle p_1, s[\mathsf{send}(p_2, v_3)]\rangle \mid \langle p_2, s[\mathsf{rec}(v_3)]\rangle \mid \underline{\langle p_3, s[\mathsf{rec}(v_2)]\rangle} \mid \underline{(p_2, p_3, v_2)} \\
&\rightarrowtail_{p_3,\mathsf{rec}} \underline{\langle p_1, s[\mathsf{send}(p_2, v_3)]\rangle} \mid \langle p_2, s[\mathsf{rec}(v_3)]\rangle \mid \langle p_3, s[\mathsf{send}(p_2, v_4)]\rangle \\
&\rightarrowtail_{p_1,\mathsf{send}} \langle p_1, s[\mathsf{send}(p_2, v_5)]\rangle \mid \underline{\langle p_2, s[\mathsf{rec}(v_3)]\rangle} \mid \langle p_3, s[\mathsf{send}(p_2, v_4)]\rangle \mid \underline{(p_1, p_2, v_3)} \\
&\rightarrowtail \ldots
\end{aligned}
$$

[3] Note that, as mentioned before, we consider that processes can be dynamically spawned. Therefore, fresh pids must be sent to other processes in order to be known. This is abstracted away in our simple notation.

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

$$(Commit)\ \frac{}{\theta, \mathsf{commit}(\tau) \xrightarrow{\mathsf{commit}(\tau)} \theta, ok} \qquad (Rollback)\ \frac{}{\theta, \mathsf{rollback}(\tau) \xrightarrow{\mathsf{rollback}(\tau)} \theta, ok}$$

Fig. 3: Rollback recovery operators

where a state of the form $s[op]$ denotes an arbitrary state where the next operation to be reduced is op. We omit some intermediate steps which are not relevant here. A graphical representation of the complete reduction can be found in Fig. 2.

We note that programs can exhibit an iterative behavior through recursive function calls.[4] This is hidden in our examples since we do not show explicitly sequential operations.

3 Checkpoint-Based Rollback Recovery

In this section, we present our approach to rollback recovery in a message-passing concurrent language. Essentially, our approach is based on defining an instrumented semantics with two modes: a "normal" mode, which proceeds similarly to the standard semantics, and a "reversible" mode, where actions can be undone and, thus, can be used for rollbacks.

3.1 Basic Operators

We consider three explicit operators to control rollback recovery: check, commit, and rollback. Intuitively speaking, they proceed as follows:

- check introduces a *checkpoint* for the current process. The reduction of check returns a fresh identifier, τ, associated to the checkpoint; note that nested checkpoints are possible.
- commit(τ) can then be used to discard the state saved in checkpoint τ. In our context, check implies turning the reversible mode on and commit turning it off (when no more active checkpoints exist).
- Finally, rollback(τ) starts a backward computation, undoing all the actions of the process (and their causal dependencies from other processes) up to the call to check (including) that introduced τ.

The local reduction rules for the new operators are very simple and can be found in Figure 3. Here, we consider that a local state consists of an environment (a variable substitution) and an expression (to be evaluated), but it could

[4] For instance, a typical server process is defined by a function that waits for a client request, process it, and then makes a recursive call, possibly with a modified *state*; this is a common pattern in Erlang.

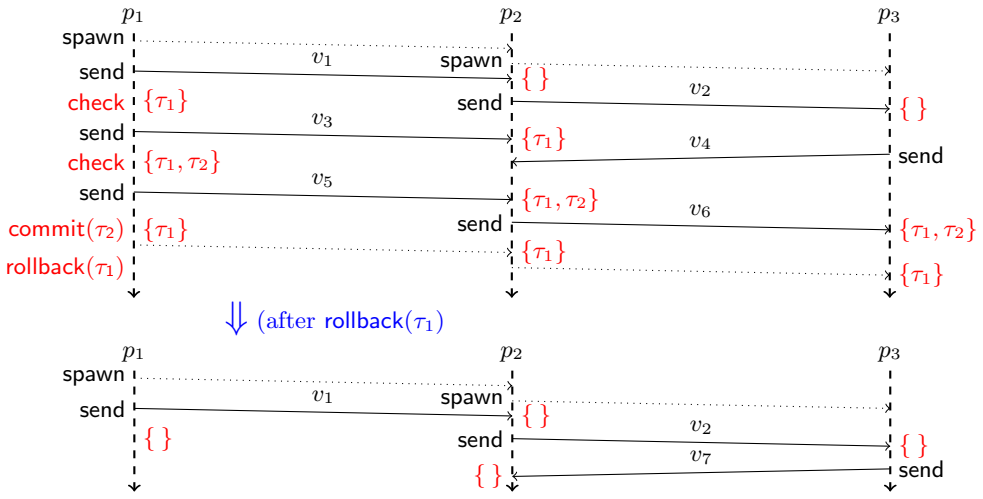

Fig. 4: Graphical representation of a reduction with check, commit, and rollback

be straightforwardly extended to other state configurations (e.g., configurations that also include a stack, as in [15]).

Rule *Check* reduces the call to a future, κ, which also occurs in the label of the transition step. As we will see in the next section, the corresponding rule in the system semantics will perform the associated side-effect (creating a checkpoint) and will also bind κ with the (fresh) identifier for this checkpoint. Rules *Commit* and *Rollback* just pass the corresponding information to the system semantics in order to do the associated side effects. Both rules reduce the call to the constant "*ok*" (an atom used in Erlang when a function call does not return any value).

Example 2. Consider again the program in Example 1. Here, we let $\tau \leftarrow \mathsf{check}$ denote that τ is the (fresh) identifier returned by the call to check. We now add a couple of checkpoints, a commit, and a rollback to process p_1:

proc p_1	**proc** p_2	**proc** p_3
$p_2 \leftarrow \mathsf{spawn}()$	$p_3 \leftarrow \mathsf{spawn}()$	$\mathsf{rec}(v_2)$
$\mathsf{send}(p_2, v_1)$	$\mathsf{rec}(v_1)$	$\mathsf{send}(p_2, v_4)$
$\tau_1 \leftarrow \mathsf{check}$	$\mathsf{send}(p_3, v_2)$	$\mathsf{rec}(v_6)$
$\mathsf{send}(p_2, v_3)$	$\mathsf{rec}(v_3)$	$\mathsf{send}(p_2, v_7)$
$\tau_2 \leftarrow \mathsf{check}$	$\mathsf{rec}(v_4)$	
$\mathsf{send}(p_2, v_5)$	$\mathsf{rec}(v_5)$	
$\mathsf{commit}(\tau_2)$	$\mathsf{send}(p_3, v_6)$	
$\mathsf{rollback}(\tau_1)$	$\mathsf{rec}(v_7)$	

A graphical representation of the new execution can be found in Fig. 4. Intuitively speaking, it proceeds as follows:

- Process p_1 calls function check, which creates a checkpoint with identifier τ_1. This checkpoint is propagated to p_2 when sending message v_3, so p_2 turns the reversible mode on too.
- Then, p_1 creates another checkpoint, τ_2, so we have two active checkpoints. These checkpoints are propagated to p_2 by message v_5 and also to p_3 by message v_6. At this point, all three processes have the reversible mode on.
- Now, p_1 calls commit(τ_2), so checkpoint τ_2 is not active anymore in p_1. This is also propagated to both p_1 and p_2. Nevertheless, the reversible mode is still on in all three processes since τ_1 is still alive.
- Then, p_1 calls rollback(τ_1) so p_1 undoes all its actions up to (and including) the first call to check. For this rollback to be causal consistent, p_2 rolls back to the point immediately before receiving message v_3, and p_3 to the point immediately before receiving message v_6.
- Finally, all three processes are back to normal, irreversible mode (no checkpoint is active), and p_3 sends message v_7 to p_2. The actions in the second diagram are all irreversible.

3.2 A Reversible Semantics for Rollback Recovery

Let us now consider the design of a *reversible* semantics for rollback recovery. Essentially, the operators can be modeled as follows:

- The reduction of check creates a checkpoint, which turns on the reversible mode of a process as a side-effect (assuming it was not already on). As in [23,25], reversibility is achieved by defining an appropriate *Landauer embedding* [19], i.e., by adding a history of the computation to each process configuration.[5] A checkpoint is propagated to other processes when a causally dependent action is performed (i.e., spawn and send); following the terminology of [3], these checkpoints are called *forced* checkpoints.
- A call of the form commit(τ) removes τ from the list of *active* checkpoints of a process, turning the reversible mode off when the list of active checkpoints is empty. Forced checkpoints in other processes with the same identifier τ (if any) are also removed from the corresponding sets of active checkpoints.
- Finally, the reduction of rollback(τ) involves undoing all the steps of a given process up to the checkpoint τ *in a causal consistent way*, i.e., possibly also undoing causally dependent actions from other processes.

Unfortunately, this apparently simple model presents a problem. Consider, for instance, a process p that performs the following actions:

$$p := \{\overbrace{\mathsf{check}(\tau_1), \mathsf{send}(p', v), \underbrace{\mathsf{check}(\tau_2), \ldots, \mathsf{commit}(\tau_1)}}, \ldots, \mathsf{rollback}(\tau_2), \ldots\} \qquad (*)$$

Here, we can observe that the pairs check-commit and check-rollback are not well balanced. As a consequence, we commit checkpoint τ_1 despite the fact that a

[5] For clarity of exposition, the complete state is saved at each state. Nevertheless, an optimized history could also be defined; see, e.g., [28,32].

rollback like rollback(τ_2) may bring the computation back to the point immediately before check(τ_2), where τ_1 should be alive, thus producing an inconsistent state. We could recover τ_1 when undoing the call commit(τ_1). However, this is not only a *local* problem, since commit(τ_1) may have also removed the checkpoint from other processes (p', in the example).

We could solve the above problem by considering that a call to commit introduces new causal dependencies. Intuitively speaking, one could treat the propagation of commit to other process as the sending of a message. Therefore, a causal consistent rollback would often require undoing all subsequent actions in these processes before undoing the call to commit. Unfortunately, this solution would require all processes to keep the reversible mode on all the time, which is precisely what we want to avoid.

A very simple workaround comes immediately to mind: require the programmer to write well-balanced pairs check-commit and check-rollback. In this case, a rollback that undo a call to commit will also undo the corresponding call to check, so the inconsistent situation above is no longer possible. However, this solution is not acceptable when forced checkpoints come into play. For instance, in example (*) above, if we replace check(τ_2) by a receive operation from some process where checkpoint τ_2 is active, the same inconsistent state can be reproduced. In this case, though, we cannot require the programmer to avoid such situations since they are unpredictable.

For all the above, in this work we do not impose any constraint on the use of the new operators but propose the following solution: When a call of the form commit(τ) occurs, we check in the process' history whether τ is the *last* active checkpoint of the process (either proper or forced). If this is the case, the commit is executed. Otherwise, it is *delayed* until the condition is met.

Our configurations will now have three additional fields: the set of active checkpoints, the set of delayed commits, and a history (for reversibility):

Definition 1 (rollback configuration). *A* forward *configuration is defined as a tuple* $(\mathcal{C}, \mathcal{D}, h, \langle p, s\rangle)$ *where* $\mathcal{C}$ *is a set of (active) checkpoint identifiers,* $\mathcal{D}$ *is a set of delayed commits,* h *is a history, and* $\langle p, s\rangle$ *is a pid and a (local) state, similarly to the standard semantics. A* backward *configuration has the form* $(\!(\mathcal{C}, \mathcal{D}, h, \langle p, s\rangle)\!)^\tau$ *where* τ *is the target of a rollback request.*

A (rollback) configuration *is either a forward or a backward configuration.*

As for the *messages*, they now have the form $(\mathcal{C}, p, p', \{\ell, v\})$. Here, we can distinguish two differences w.r.t. the standard semantics: first, we add a set of active checkpoints, $\mathcal{C}$, which should be propagated to the receiver as *forced* checkpoints; secondly, the message value is wrapped with a tag to uniquely identify it (as in [23], in order to distinguish messages with the same value).

As in the standard semantics, a *system* is then a parallel composition of (rollback) configurations and floating messages.

In the following, we let $[\,]$ denote an empty list and $x : xs$ a list with head x and tail xs. A process *history* h is then represented by a list of the following elements: seq, send, rec, spawn, check, and commit. Each one is denoted by a

term containing enough information to undo the corresponding reduction step,[6] except for the case of commit, whose side-effects are irreversible, as argued above. To be more precise,

- all terms store the current state (s).
- for sending a message, the corresponding term also stores the pid of the target process (p') and the message tag (ℓ);
- for receiving a message, the term also stores two sets of active checkpoints (the active ones and those received from a message as forced checkpoints), the pid of the sender (p'), the message tag (ℓ), and the message value (v);
- for spawning a process, the corresponding term also includes the fresh pid of the new process (p');
- and, finally, for check and commit, it also stores the checkpoint identifier (τ).

A delayed commit is represented as a triple $\langle \tau, h, P \rangle$, where τ is a checkpoint identifier, h is a history, and P is a set of pids (the pids of the processes where a forced checkpoint τ has been propagated).

Forward Rules. The *forward* reduction rules of the rollback semantics are shown in Figure 5. The main difference with the standard semantics is that, now, some process configurations include a *history* with enough information to undo any reduction step. Here, we follow the same strategy as in [23,25] in order to define a scheme for reversible debugging.[7]

Essentially, the first four rules of the semantics can behave either as the standard semantics or as a *reversible* semantics, depending on whether $\mathcal{C}$ is empty or not. For conciseness, we avoid duplicating all rules by introducing the auxiliary function add to update the history only when there are active checkpoints: $\mathsf{add}_{\mathcal{C}}(a, h) = h$ if $\mathcal{C} = \emptyset$, and $\mathsf{add}_{\mathcal{C}}(a, h) = a : h$ otherwise.

As mentioned above, the reversible mode is propagated through message sending and receiving. This is why messages now include the set of active checkpoints $\mathcal{C}$. As can be seen in rule *Receive*, the process receiving the message updates its active checkpoints with those in the message. This is necessary for rollbacks to be causally consistent. Note that, in the associated term in the history, $\mathsf{rec}(\mathcal{C}' \setminus \mathcal{C}, \mathcal{C}', s, p', \ell, v)$, $\mathcal{C}' \setminus \mathcal{C}$ denotes the *forced* checkpoints introduced by the received message.

Similarly, the reversible mode is also propagated by process spawning: rule *Spawn* adds the current set of active checkpoints $\mathcal{C}$ (which might be empty) to the new process.

[6] The reader can compare the rules in Fig. 5 and their inverse counterpart in Fig. 6.

[7] In contrast to [23,25], however, we do not need to undo every possible step, but only those steps that are performed when there is at least one active checkpoint. This is why we added $\mathcal{C}$: to store the set of *active* checkpoints (i.e., checkpoints without a corresponding commit/rollback yet). Observe that we might have several active checkpoints not only because nested checkpoints are possible, but because of *forced* checkpoints propagated by process spawning and message sending.

$$(\underline{Seq}) \quad \frac{s \xrightarrow{\mathsf{seq}} s' \text{ and } \mathsf{add}_{\mathcal{C}}(\mathsf{seq}(s), h) = h'}{(\mathcal{C}, \mathcal{D}, h, \langle p, s\rangle) \hookrightarrow_{p,\underline{\mathsf{seq}}} (\mathcal{C}, \mathcal{D}, h', \langle p, s'\rangle)}$$

$$(\underline{Send}) \quad \frac{s \xrightarrow{\mathsf{send}(p',v)} s',\ \ell \text{ is a fresh symbol, and } \mathsf{add}_{\mathcal{C}}(\mathsf{send}(s, p', \ell), h) = h'}{(\mathcal{C}, \mathcal{D}, h, \langle p, s\rangle) \hookrightarrow_{p,\underline{\mathsf{send}}(\ell)} (\mathcal{C}, p, p', \{\ell, v\}) \mid (\mathcal{C}, \mathcal{D}, h', \langle p, s'\rangle)}$$

$$(\underline{Receive}) \quad \frac{s \xrightarrow{\mathsf{rec}(\kappa,cs)} s',\ \mathsf{matchrec}(cs, v) = cs_i, \text{ and } \mathsf{add}_{\mathcal{C}}(\mathsf{rec}(\mathcal{C}' \setminus \mathcal{C}, \mathcal{C}', s, p', \ell, v), h) = h'}{(\mathcal{C}', p', p, \{\ell, v\}) \mid (\mathcal{C}, \mathcal{D}, h, \langle p, s\rangle) \hookrightarrow_{p,\underline{\mathsf{rec}}(\ell)} (\mathcal{C} \cup \mathcal{C}', \mathcal{D}, h', \langle p, s'[\kappa \leftarrow cs_i]\rangle)}$$

$$(\underline{Spawn}) \quad \frac{s \xrightarrow{\mathsf{spawn}(\kappa,s_0)} s',\ p' \text{ is a fresh pid, and } \mathsf{add}_{\mathcal{C}}(\mathsf{spawn}(s, p'), h) = h'}{(\mathcal{C}, \mathcal{D}, h, \langle p, s\rangle) \hookrightarrow_{p,\underline{\mathsf{spawn}}(p')} (\mathcal{C}, \mathcal{D}, h', \langle p, s'[\kappa \leftarrow p']\rangle) \mid (\mathcal{C}, [\,], \langle p', s_0\rangle)}$$

$$(\underline{Check}) \quad \frac{s \xrightarrow{\mathsf{check}(\kappa)} s' \text{ and } \tau \text{ is a fresh identifier}}{(\mathcal{C}, \mathcal{D}, h, \langle p, s\rangle) \hookrightarrow_{p,\underline{\mathsf{check}}(\tau)} (\mathcal{C} \cup \{\tau\}, \mathcal{D}, \mathsf{check}(\tau, s) : h, \langle p, s'[\kappa \leftarrow \tau]\rangle)}$$

$$(\underline{Commit}) \quad \frac{s \xrightarrow{\mathsf{commit}(\tau)} s',\ last_\tau(h) = true,\ dp_\tau(h) = P, \text{ and } propagate(\tau, P)}{(\mathcal{C}, \mathcal{D}, h, \langle p, s\rangle) \hookrightarrow_{p,\underline{\mathsf{commit}}(\tau)} (\mathcal{C} \setminus \{\tau\}, \mathcal{D}, \mathsf{commit}(\tau, s) : h, \langle p, s'\rangle)}$$

$$(\underline{Delay}) \quad \frac{s \xrightarrow{\mathsf{commit}(\tau)} s',\ last_\tau(h) = false, \text{ and } dp_\tau(h) = P}{(\mathcal{C}, \mathcal{D}, h, \langle p, s\rangle) \hookrightarrow_{p,\underline{\mathsf{delay}}(\tau)} (\mathcal{C} \setminus \{\tau\}, \mathcal{D} \cup \{\langle \tau, h, P\rangle\}, \mathsf{commit}(\tau, s) : h, \langle p, s'\rangle)}$$

$$\frac{last_\tau(h') = true \text{ and } propagate(\tau, P)}{(\mathcal{C}, \mathcal{D} \cup \{\langle \tau, h', P\rangle\}, h, \langle p, s\rangle) \hookrightarrow (\mathcal{C}, \mathcal{D}, h, \langle p, s\rangle)}$$

$$(\underline{Rollback}) \quad \frac{s \xrightarrow{\mathsf{rollback}(\tau)} s'}{(\mathcal{C}, \mathcal{D}, h, \langle p, s\rangle) \hookrightarrow_{p,\underline{\mathsf{rollback}}(\tau)} ((\mathcal{C}, \mathcal{D}, h, \langle p, s'\rangle))^\tau}$$

$$(\underline{Par}) \quad \frac{S_1 \hookrightarrow_l S_1' \quad \text{and} \quad id(S_1') \cap id(S_2) = \emptyset}{S_1 \mid S_2 \hookrightarrow_l S_1' \mid S_2}$$

Fig. 5: Rollback recovery semantics: forward rules

As for the new rules, *Check* produces a fresh identifier, τ, and binds the future, κ, with this identifier. Moreover, it adds τ to the current set of active checkpoints. In particular, if $\mathcal{C}$ were empty, this step would turn the reversible mode on.

Commit includes two transition rules, depending on whether the commit can be done or it should be delayed. For this purpose, we use the auxiliary Boolean function *last*. Here, a call of the form $last_\tau(h)$ checks whether τ is the last checkpoint of the process according to history h, i.e., whether the last check or rec term in h has either the form $\mathsf{check}(\tau, s')$ or $\mathsf{rec}(\mathcal{C}', \ldots)$ with $\tau \in \mathcal{C}'$. Note that we do not need to consider forced checkpoints introduced by process spawning since they cannot occur after a call to check (they are always introduced when creating the process). Then, if the call to function *last* returns *true*, we remove the checkpoint identifier from $\mathcal{C}$ and from all processes where this checkpoint was propagated (as a forced checkpoint). Here, the auxiliary funcion dp_τ takes a history and returns all pids which have causal dependencies with the current process according to h, i.e.,

- $p \in dp_\tau(h)$ if $\mathsf{spawn}(s, p)$ occurs in h;
- $p \in dp_\tau(h)$ if $\mathsf{send}(s, p, \ell)$ occurs in h.

Now, we want to propagate the effect of commit to all processes in $dp_\tau(h)$ in order to remove τ from their set of active checkpoints. One could formalize this process by means of a few more transition rules and a new kind of configuration. However, for simplicity, we represent it by means of an auxiliary function, *propagate*. To be more precise, $propagate(\tau, P)$ always returns *true* while performing the following side-effects:

1. for each $p' \in P$, we look for the process with pid p', say $(\mathcal{C}', \mathcal{D}', h', \langle p', s' \rangle)$;
2. if $\tau \notin \mathcal{C}'$ (it is not an active checkpoint of process p'), we are done;
3. otherwise ($\tau \in \mathcal{C}'$), we remove τ from $\mathcal{C}'$ and repeat the process, i.e., we compute $P' = dp_\tau(h')$ and call $propagate(\tau, P')$.

Termination is ensured since the number of processes is finite and a process where τ is not active will eventually be reached. In practice, commit propagation can be implemented by sending (asynchronous) messages to the involved processes. Note that the semantics would be sound even if commit operations were not propagated, so doing it is essentially a matter of efficiency (a sort of *garbage collection* to avoid recording actions that are not really necessary).

On the other hand, if the call to function *last* returns *false*, the checkpoint is moved from $\mathcal{C}$ to $\mathcal{D}$ as a delayed commit (second rule of *Commit*). Eventually, rule *Delay* becomes applicable, which proceeds similarly to the first rule of *Commit* but considering the delayed commit. We do not formalize a particular strategy for firing rule *Delay*, but one could assume a simple strategy where this rule is only considered when some checkpoint is removed from the set of active checkpoints of a process.

Rule *Rollback* simply changes the forward configuration to a backward configuration, also adding the superscript τ to *drive* the rollback. Therefore, the forward rules are no longer applicable to this process (and the backward rules in Figure 6 can be applied instead).

Finally, rule *Par* is identical to that in the standard semantics. The only difference is that, now, function $id(S)$ returns the set of pids, message tags, and checkpoints in S.

Backward Rules. Let us now present the backward rules of the rollback semantics: First, rule *Seq* applies when the history is headed by a term of the form $\mathsf{seq}(s)$. It simply removes this element from the history and recovers state s.

Rule *Send* distinguishes two cases. If the message with tag ℓ is a floating message (so it has not been *received*), then we remove the message from the system and recover the saved state. Otherwise (i.e., the message has been consumed by the target process p'), the rollback mode is propagated to process p', which will go backwards up to the receiving of the message; once the floating message is back into the system, the first rule applies.

Rule *Receive* also distinguishes two cases. In both of them, the message is put back into the system as a floating message and the recorded state is recovered.

$$(\overline{Seq}) \quad (\!(\mathcal{C}, \mathcal{D}, \mathsf{seq}(s) : h, \langle p, s' \rangle)\!)^{\tau} \hookrightarrow_{p,\overline{\mathsf{seq}}} (\!(\mathcal{C}, \mathcal{D}, h, \langle p, s \rangle)\!)^{\tau}$$

$$(\overline{Send}) \quad (\mathcal{C}, p, p', \{\ell, v\}) \mid (\!(\mathcal{C}, \mathcal{D}, \mathsf{send}(s, p', \ell) : h, \langle p, s' \rangle)\!)^{\tau} \hookrightarrow_{p,\overline{\mathsf{send}(\ell)}} (\!(\mathcal{C}, \mathcal{D}, h, \langle p, s \rangle)\!)^{\tau}$$

$$(\!(\mathcal{C}, \mathcal{D}, \mathsf{send}(s, p', \ell) : h, \langle p, s' \rangle)\!)^{\tau} \mid (\mathcal{C}', \mathcal{D}', h', \langle p', s'' \rangle) \hookrightarrow (\!(\mathcal{C}, \mathcal{D}, \mathsf{send}(s, p', \ell) : h, \langle p, s' \rangle)\!)^{\tau} \mid (\!(\mathcal{C}', \mathcal{D}', h', \langle p', s'' \rangle)\!)^{\tau}$$

$$(\overline{Receive}) \quad (\!(\mathcal{C} \cup \mathcal{C}', \mathsf{rec}(\mathcal{C}'', \mathcal{C}', s, p', \ell, v) : h, \langle p, s' \rangle)\!)^{\tau} \hookrightarrow_{p,\overline{\mathsf{rec}(\ell)}} (\mathcal{C}', p', p, \{\ell, v\}) \mid (\mathcal{C}, \mathcal{D}, h, \langle p, s \rangle) \quad \text{if } \tau \in \mathcal{C}''$$

$$(\!(\mathcal{C} \cup \mathcal{C}', \mathsf{rec}(\mathcal{C}'', \mathcal{C}', s, p', \ell, v) : h, \langle p, s' \rangle)\!)^{\tau} \hookrightarrow_{p,\overline{\mathsf{rec}(\ell)}} (\mathcal{C}', p', p, \{\ell, v\}) \mid (\!(\mathcal{C}, \mathcal{D}, h, \langle p, s \rangle)\!)^{\tau} \quad \text{if } \tau \notin \mathcal{C}''$$

$$(\overline{Spawn}) \quad (\!(\mathcal{C}, \mathcal{D}, \mathsf{spawn}(s, p') : h, \langle p, s' \rangle)\!)^{\tau} \mid (\!(\mathcal{C}, \emptyset, [\,], \langle p', s_0 \rangle)\!)^{\tau} \hookrightarrow_{p,\overline{\mathsf{spawn}(p')}} (\!(\mathcal{C}, \mathcal{D}, h, \langle p, s \rangle)\!)^{\tau}$$

$$(\!(\mathcal{C}, \mathcal{D}, \mathsf{spawn}(s, p') : h, \langle p, s' \rangle)\!)^{\tau} \mid (\mathcal{C}', \mathcal{D}', h', \langle p', s'' \rangle) \hookrightarrow (\!(\mathcal{C}, \mathcal{D}, \mathsf{spawn}(s, p') : h, \langle p, s' \rangle)\!)^{\tau} \mid (\!(\mathcal{C}', \mathcal{D}', h', \langle p', s'' \rangle)\!)^{\tau}$$

$$(\overline{Check}) \quad (\!(\mathcal{C}, \mathcal{D}, \mathsf{check}(\tau, s) : h, \langle p, s' \rangle)\!)^{\tau} \hookrightarrow_{p,\overline{\mathsf{check}(\tau)}} (\mathcal{C} \setminus \{\tau\}, \mathcal{D}, h, \langle p, s \rangle)$$

$$(\!(\mathcal{C}, \mathcal{D}, \mathsf{check}(\tau', s) : h, \langle p, s' \rangle)\!)^{\tau} \hookrightarrow_{p,\overline{\mathsf{check}(\tau')}} (\!(\mathcal{C} \setminus \{\tau'\}, \mathcal{D}, h, \langle p, s \rangle)\!)^{\tau} \quad \text{if } \tau \neq \tau'$$

$$(\overline{Commit}) \quad (\!(\mathcal{C}, \mathcal{D}, \mathsf{commit}(\tau', s) : h, \langle p, s' \rangle)\!)^{\tau} \hookrightarrow_{p,\overline{\mathsf{commit}(\tau')}} (\!(\mathcal{C} \cup \{\tau'\}, \mathcal{D}, h, \langle p, s \rangle)\!)^{\tau} \text{ if } \langle \tau', \_, \_ \rangle \notin \mathcal{D}$$

$$(\!(\mathcal{C}, \mathcal{D} \cup \{\langle \tau', h', P \rangle\}, \mathsf{commit}(\tau', s) : h, \langle p, s' \rangle)\!)^{\tau} \hookrightarrow_{p,\overline{\mathsf{commit}(\tau')}} (\!(\mathcal{C} \cup \{\tau'\}, \mathcal{D}, h, \langle p, s \rangle)\!)^{\tau}$$

(*) We assume the side condition $\tau \in \mathcal{C}$ holds in all rules.
(**) The second rule of $\overline{Send}$ only applies when the message tagged with ℓ has been received by p' according to history h'.

Fig. 6: Rollback recovery semantics: backward rules

They differ in that the first rule considers the case where τ is a forced checkpoint introduced by the received message. In this case, we undo the step and the process resumes its forward computation. Otherwise (i.e., τ was introduced somewhere else), we undo the step but keep the rollback mode for the process.

Rule $\overline{Spawn}$ proceeds in a similar way as rule $\overline{Send}$: if the spawned process is already in its initial state with an empty history, it is simply removed from the system. Otherwise, the reversible mode is propagated to the spawned process p' (note that if p' is already in reversible mode, rule $\overline{Spawn}$ does not apply).

Rule $\overline{Check}$ applies when we reach a checkpoint in the process' history. If the checkpoint has the same identifier of the initial rollback operator, τ, the job is done and the process resumes its forward computation after undoing one last step. Otherwise (i.e., the checkpoint in the history has a different identifier, τ'), we undo the step, also removing τ' from the set of active checkpoints, but keep the rollback mode.

Finally, rule $\overline{Commit}$ considers two cases: either the commit has been executed (and, thus, the rollback will eventually undo the associated check too), or the commit was delayed (see Example 3 below).

Example 3. Consider again the program in Example 2, where we now switch the arguments of commit and rollback in process p_1 in order to illustrate the use of delayed commits (p_2 and p_3 remain the same as before):[8]

proc p_1	**proc** p_2	**proc** p_3
$p_2 \leftarrow \mathsf{spawn}()$	$p_3 \leftarrow \mathsf{spawn}()$	$\mathsf{rec}(\ell_2)$
$\mathsf{send}(p_2, \ell_1)$	$\mathsf{rec}(\ell_1)$	$\mathsf{send}(p_2, \ell_4)$
$\tau_1 \leftarrow \mathsf{check}$	$\mathsf{send}(p_3, \ell_2)$	$\mathsf{rec}(\ell_6)$
$\mathsf{send}(p_2, \ell_3)$	$\mathsf{rec}(\ell_3)$	$\mathsf{send}(p_2, \ell_7)$
$\tau_2 \leftarrow \mathsf{check}$	$\mathsf{rec}(\ell_4)$	
$\mathsf{send}(p_2, \ell_5)$	$\mathsf{rec}(\ell_5)$	
$\mathsf{commit}(\tau_1)$	$\mathsf{send}(p_3, \ell_6)$	
$\mathsf{rollback}(\tau_2)$	$\mathsf{rec}(\ell_7)$	

In this case, the sequence of configurations of p_1 would be as follows:[9]

$$
\begin{aligned}
&(\emptyset, \emptyset, [\,], \langle p_1, s[p_2 \leftarrow \mathsf{spawn}]\rangle)\\
&\hookrightarrow (\emptyset, \emptyset, [\mathsf{spawn}(p_2)], \langle p_1, s[\mathsf{send}(p_2, \ell_1)]\rangle)\\
&\hookrightarrow (\emptyset, \emptyset, [\mathsf{send}(p_2, \ell_1), \mathsf{spawn}(p_2)], \langle p_1, s[\tau_1 \leftarrow \mathsf{check}]\rangle)\\
&\hookrightarrow (\{\tau_1\}, \emptyset, [\mathsf{check}(\tau_1), \mathsf{send}(p_2, \ell_1), \mathsf{spawn}(p_2)], \langle p_1, s[\mathsf{send}(p_2, \ell_3)]\rangle)\\
&\hookrightarrow (\{\tau_1\}, \emptyset, [\mathsf{send}(p_2, \ell_3), \mathsf{check}(\tau_1), \mathsf{send}(p_2, \ell_1), \ldots], \langle p_1, s[\tau_2 \leftarrow \mathsf{check}]\rangle)\\
&\hookrightarrow (\{\tau_1, \tau_2\}, \emptyset, [\mathsf{check}(\tau_2), \mathsf{send}(p_2, \ell_3), \mathsf{check}(\tau_1), \ldots], \langle p_1, s[\mathsf{send}(p_2, \ell_5)]\rangle)\\
&\hookrightarrow (\{\tau_1, \tau_2\}, \emptyset, [\mathsf{send}(p_2, \ell_5), \mathsf{check}(\tau_2), \mathsf{send}(p_2, \ell_3), \ldots], \langle p_1, s[\mathsf{commit}(\tau_1)]\rangle)\\
&\hookrightarrow (\{\tau_2\}, \{\langle \tau_1, h, \{p_2\}\rangle\}, [\mathsf{commit}(\tau_1), \mathsf{send}(p_2, \ell_5), \ldots], \langle p_1, s[\mathsf{rollback}(\tau_2)]\rangle)\\
&\hookrightarrow ((\{\tau_2\}, \{\langle \tau_1, h, \{p_2\}\rangle\}, [\mathsf{commit}(\tau_1), \mathsf{send}(p_2, \ell_5), \ldots], \langle p_1, s[\ldots]\rangle))^{\tau_2}\\
&\hookrightarrow ((\{\tau_1, \tau_2\}, \emptyset, [\mathsf{send}(p_2, \ell_5), \mathsf{check}(\tau_2), \mathsf{send}(p_2, \ell_3), \ldots], \langle p_1, s[\ldots]\rangle))^{\tau_2}\\
&\hookrightarrow ((\{\tau_1, \tau_2\}, \emptyset, [\mathsf{check}(\tau_2), \mathsf{send}(p_2, \ell_3), \mathsf{check}(\tau_1), \ldots], \langle p_1, s[\ldots]\rangle))^{\tau_2}\\
&\hookrightarrow (\{\tau_1\}, \emptyset, [\mathsf{send}(p_2, \ell_3), \mathsf{check}(\tau_1), \mathsf{send}(p_2, \ell_1), \mathsf{spawn}(p_2)], \langle p_1, s[\ldots]\rangle)\\
&\hookrightarrow \ldots
\end{aligned}
$$

Here, the call $\mathsf{commit}(\tau_1)$ cannot be executed since the last checkpoint of process p_1 is τ_2. Therefore, it is added as a delayed checkpoint. Then, we have a call $\mathsf{rollback}(\tau_2)$ which undo the last steps of p_1 (as well as some steps in p_2 and p_3 in order to keep causal consistency, which we do not show for simplicity).

Soundness. In the following, we assume a *fair* selection strategy for processes, so that each process is eventually reduced. Furthermore, we only consider *well-defined* derivations where, e.g., the calls $\mathsf{commit}(\tau)$ and $\mathsf{rollback}(\tau)$ can only be made by the same process that created the checkpoint τ; furthermore, a process can only have one action for every checkpoint τ, either $\mathsf{commit}(\tau)$ or $\mathsf{rollback}(\tau)$, but not both.

Soundness is then proved by projecting the configurations of the rollback semantics to configurations of either the standard semantics (function *sta*) or a

[8] Furthermore, terms send and rec now include message tags instead of values.

[9] For clarity, we omit some of the arguments of history items and some other information which is not relevant for the example.

pure reversible semantics (function *rev*). Then, we prove that every step under the rollback semantics has a counterpart either under the standard or under the reversible semantics, after applying the corresponding projections. Formally,[10]

Theorem 1. *Let d be a well-defined derivation under the rollback semantics. Then, for each step $S \hookrightarrow S'$ in d we have either $sta(S) \rightarrowtail^{=} sta(S')$ or $rev(S) \rightleftharpoons^{=} rev(S')$.*

We have also proved that every computation between a checkpoint and the corresponding rollback is indeed reversible for well-defined derivations; see the companion technical report, [37], for more details. We leave the study of other interesting results of our rollback semantics (e.g., minimality and some partial completeness) for future work.

4 Related Work

There is abundant literature on checkpoint-based rollback recovery to improve fault tolerance (see, e.g., the survey by Elnozahy *et al* [3] and references there in). In contrast to most of them, our distinctive features are the extension of the underlying language with *explicit* operators for rollback recovery, the automatic generation of forced checkpoints (somehow similarly to communication-induced checkpointing [35]), and the use of a reversible semantics. Also, we share some similarities with the checkpointing technique for fault-tolerant distributed computing of [8,17], although the aim is in principle different: their goal is the definition of a new programming model where globally consistent checkpoints can be created (rather than extending an existing message-passing programming language with explicit operators for rollback recovery). Indeed, the use of some form of reversibility is mentioned in [8] as future work.

The idea of using some form of reversible computation for rollback recovery is not new. E.g., Perumalla and Park [33] already suggested it as an alternative to other traditional techniques based on checkpointing. In contrast to our work, the authors focus on empirically analyzing the trade-off between fault tolerance based on checkpointing and on reversible computation (i.e., memory vs run time), using a particular example (a particle collision application). Moreover, since the application is already reversible, no Landauer embedding is required.

The introduction of a rollback construct in a causal-consistent concurrent formalism can be traced back to [11,12,21,27,22]. In these works, however, the authors focus on a different formalism and, moreover, no explicit checkpointing operator is considered. These ideas are then transferred to an Erlang-like language in [31,23], where an explicit checkpoint operator is introduced. However, in contrast to our work, all actions are recorded into a history (i.e., it has no

[10] We denote by $\rightarrow^{=}$ the reflexive closure of a binary relation $\rightarrow$, i.e., $(\rightarrow^{=}) = (\rightarrow \cup =)$. We consider the reflexive closure in the claim of Theorem 1 since some steps under the rollback semantics have no counterpart under the standard or reversible semantics. In these cases, the projected configurations remain the same.

way of turning the reversible mode off).[11] In other words, a checkpoint is just a mark in the execution, but it is not propagated to other processes (as our forced checkpoints) and cannot be removed (as a call to commit does in our approach). More recent formulations of the reversible semantics for an Erlang-like language include [24,25,26,20], but the checkpoint operator has not been considered again (the focus is on reversible debugging instead).

The standard semantics in Figure 1 is trivially equivalent to that considered in [25] except for some minor details: First, [25] considers an operational semantics where the complete system is represented in each transition rule. Here, we follow the simpler and more elegant formulation of [26]. For instance, following the style of [25], rule *Send* would have the following form:

$$\frac{s \xrightarrow{\mathsf{send}(p',v)} s'}{\Gamma; \langle p, s\rangle \mid \Pi \rightarrowtail_{p,\mathsf{send}(\ell)} \Gamma \cup \{(p,p',v)\}; \langle p, s'\rangle \mid \Pi}$$

In this case, messages are stored in a global mailbox, Γ, and an expression like "$\langle p, s\rangle \,|\, \Pi$" represents all the processes in the system, i.e., $\langle p, s\rangle$ is a distinguished process (where reduction applies) and Π is the parallel composition of the remaining processes. In contrast, we have *floating* messages and select a process to be reduced by applying (repeteadly) rule *Par*. The possible reductions, though, are the same in both cases. There are other, minor differences, like considering a rule to deal with the predefined function *self* (which returns the pid of a process), and representing a state by a pair θ, e (environment,expression)

Another difference with the reversible semantics in [23,25] is that we consider a single transition relation for systems ($\hookrightarrow$). This relation aims at modeling an actual execution in which a process proceeds normally forwards but a call to rollback forces it to go backwards temporarily (a situation that can be propagated to other processes in order to be causally consistent). In contrast, [23,25] considers first an *uncontrolled* semantics ($\rightharpoonup$ and $\leftharpoondown$) which models *all* possible forward and backward computations. Then, a *controlled* semantics is defined on top of it to drive the steps of the uncontrolled semantics in order to satisfy both replay and rollback requests.

On a different line of work, Vassor and Stefani [36] formally studied the relation between rollback recovery and causal-consistent reversible computation. In particular, they consider the relation between a distributed checkpoint/rollback scheme based on (causal) logging (Manetho [4]) and a causally-consistent reversible version of π-calculus with a rollback operator [21]. Their main conclusion is that the latter can simulate the rollback recovery strategy of Manetho. Our aim is somehow similar, since we also simulate a checkpoint-based rollback recovery strategy using a reversible semantics, but there are also some significant differences: the considered language is different (a variant of π-calculus vs an Erlang-like language), they only consider a fixed number of processes (while

[11] Our ability to turn the reversible mode on/off is useful not only to model rollback recovery, but can also constitute the basis of a *selective* reversible debugging scheme where only some computations—those of interest—are traced.

we accept dynamic process spawning) and, moreover, no explicit operators are considered (i.e., our approach is more oriented to introduce a new programming feature rather than proving a theoretical result).

Very recently, Mezzina, Tiezzi and Yoshida [29] have introduced a rollback recovery strategy for session-based programming. The authors propose a number of rollback recovery primitives for session-based programming. Besides considering a different setting (a variant of π-calculus), their approach is also limited to a fixed number of parties (no dynamic processes can be added at run time), and nested checkpoints are not allowed. Furthermore, the checkpoints of [29] are not automatically propagated to other causally consistent processes (as our forced checkpoints); rather, they introduce a *compliance check* at the type level to prevent undesired situations.

Our work also shares some similarities with [34], which presents a hybrid model combining message-passing concurrency and software transactional memory. However, the underlying language is different and, moreover, their transactions cannot include process spawning (which must be delayed).

Finally, Fabbretti, Lanese and Stefani [6] have introduced a calculus to formally model distributed systems subject to crash failures, where recovery mechanisms can be encoded by a small set of primitives. This work can be seen as a reworking and extension of the previous work by Francalanza and Hennessy [9]. Here, a variant of π-calculus is considered. Furthermore, the authors focus on crash recovery without relying on a form of checkpointing, in contrast to our approach.

5 Conclusions and Future Work

In this work, we have defined a rollback-recovery strategy for a message-passing concurrent programming language without the need for a central coordination. For this purpose, we have extended the underlying language with three explicit operators: check, commit, and rollback. Our approach is based on a *selective* reversible semantics where every process may go both forwards and backwards (during a rollback). Checkpoints are automatically propagated to other processes so that backward computations are causally consistent. Finally, we have proved the soundness of our rollback semantics w.r.t. a *pure* reversible semantics.

As for future work, we will consider the definition of a *shorcut* version of the rollback semantics where only the state in a checkpoint is recorded (rather than all the states between a checkpoint and the corresponding commit/rollback) so that a rollback recovers the saved state in one go (rather than in a stepwise manner). This extension will be essential to make our approach feasible in practice. A possible implementation for Erlang applications including check, commit, and rollback could be based on a program instrumentation. It will likely require introducing a wrapper for each process in order to record the process' history, turning the reversible mode on/off, propagating forced checkpoints and commits, etc. For this purpose, we will explore the run-time monitors of [10], which play a similar role in their scheme for reversible choreographies.

Acknowledgements. The author would like to thank Ivan Lanese and Adrián Palacios for their useful remarks and discussions on a preliminary version of this work. I would also like to thank the anonymous reviewers of FACS 2023 for their suggestions to improve this paper.

References

1. Aman, B., et al.: Foundations of reversible computation. In: Ulidowski, I., Lanese, I., Schultz, U.P., Ferreira, C. (eds.) Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405, Lecture Notes in Computer Science, vol. 12070, pp. 1–40. Springer (2020). https://doi.org/10.1007/978-3-030-47361-7_1
2. Danos, V., Krivine, J.: Reversible communicating systems. In: CONCUR. LNCS, vol. 3170, pp. 292–307. Springer (2004)
3. Elnozahy, E.N., Alvisi, L., Wang, Y., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3), 375–408 (2002)
4. Elnozahy, E.N., Zwaenepoel, W.: Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. IEEE Trans. Computers **41**(5), 526–531 (1992). https://doi.org/10.1109/12.142678
5. Erlang website. URL: `https://www.erlang.org/` (2021)
6. Fabbretti, G., Lanese, I., Stefani, J.B.: A behavioral theory for crash failures and erlang-style recoveries in distributed systems. Tech. Rep. RR-9511, INRIA (2023), `https://hal.science/hal-04123758`
7. Frequently Asked Questions about Erlang. Available at `http://erlang.org/faq/academic.html` (2018)
8. Field, J., Varela, C.A.: Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005). pp. 195–208. ACM (2005)
9. Francalanza, A., Hennessy, M.: A theory of system behaviour in the presence of node and link failure. Inf. Comput. **206**(6), 711–759 (2008). https://doi.org/10.1016/j.ic.2007.12.002
10. Francalanza, A., Mezzina, C.A., Tuosto, E.: Reversible choreographies via monitoring in erlang. In: Bonomi, S., Rivière, E. (eds.) Proceedings of the 18th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2018), held as part of DisCoTec 2018. Lecture Notes in Computer Science, vol. 10853, pp. 75–92. Springer (2018). https://doi.org/10.1007/978-3-319-93767-0_6
11. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: Gnesi, S., Rensink, A. (eds.) Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE 2014). Lecture Notes in Computer Science, vol. 8411, pp. 370–384. Springer (2014)
12. Giachino, E., Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent reversibility in a tuple-based language. In: Daneshtalab, M., Aldinucci, M., Leppänen, V., Lilius, J., Brorsson, M. (eds.) Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015. pp. 467–475. IEEE Computer Society (2015)
13. Glück, R., Lanese, I., Mezzina, C.A., Miszczak, J.A., Phillips, I., Ulidowski, I., Vidal, G.: Towards a taxonomy for reversible computation approaches. In: Kutrib,

M., Meyer, U. (eds.) Reversible Computation. pp. 24–39. Springer Nature Switzerland, Cham (2023)
14. González-Abril, J.J., Vidal, G.: Causal-Consistent Reversible Debugging: Improving CauDEr. Tech. rep., DSIC, Universitat Politècnica de València (2020), https://gvidal.webs.upv.es/confs/padl21/tr.pdf
15. González-Abril, J.J., Vidal, G.: Causal-Consistent Reversible Debugging: Improving CauDEr. In: Morales, J.F., Orchard, D.A. (eds.) Proceedings of the 23rd International Symposium on Practical Aspects of Declarative Languages (PADL 2021). Lecture Notes in Computer Science, vol. 12548, pp. 145–160. Springer (2021). https://doi.org/10.1007/978-3-030-67438-0_9
16. Hewitt, C., Bishop, P.B., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Nilsson, N.J. (ed.) Proceedings of the 3rd International Joint Conference on Artificial Intelligence. pp. 235–245. William Kaufmann (1973), http://ijcai.org/Proceedings/73/Papers/027B.pdf
17. Kuang, P., Field, J., Varela, C.A.: Fault tolerant distributed computing using asynchronous local checkpointing. In: Boix, E.G., Haller, P., Ricci, A., Varela, C. (eds.) Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control (AGERE! 2014). pp. 81–93. ACM (2014)
18. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563
19. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development **5**, 183–191 (1961)
20. Lanese, I., Medic, D.: A general approach to derive uncontrolled reversible semantics. In: Konnov, I., Kovács, L. (eds.) 31st International Conference on Concurrency Theory, CONCUR 2020. LIPIcs, vol. 171, pp. 33:1–33:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.CONCUR.2020.33
21. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.: Controlling reversibility in higher-order pi. In: Katoen, J., König, B. (eds.) Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR 2011). Lecture Notes in Computer Science, vol. 6901, pp. 297–311. Springer (2011)
22. Lanese, I., Mezzina, C.A., Stefani, J.: Reversibility in the higher-order π-calculus. Theor. Comput. Sci. **625**, 25–84 (2016)
23. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. Journal of Logical and Algebraic Methods in Programming **100**, 71–97 (2018). https://doi.org/10.1016/j.jlamp.2018.06.004
24. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay debugging for message passing programs. In: Pérez, J.A., Yoshida, N. (eds.) Proceedings of the 39th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2019). Lecture Notes in Computer Science, vol. 11535, pp. 167–184. Springer (2019). https://doi.org/10.1007/978-3-030-21759-4_10
25. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay reversible semantics for message passing concurrent programs. Fundam. Informaticae **178**(3), 229–266 (2021). https://doi.org/10.3233/FI-2021-2005
26. Lanese, I., Sangiorgi, D., Zavattaro, G.: Playing with bisimulation in Erlang. In: Boreale, M., Corradini, F., Loreti, M., Pugliese, R. (eds.) Models, Languages, and Tools for Concurrent and Distributed Programming – Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 11665, pp. 71–91. Springer (2019). https://doi.org/10.1007/978-3-030-21485-2_6

27. Lienhardt, M., Lanese, I., Mezzina, C.A., Stefani, J.B.: A reversible abstract machine and its space overhead. In: Giese, H., Rosu, G. (eds.) Proceedings of the Joint 14th IFIP WG International Conference on Formal Techniques for Distributed Systems (FMOODS 2012) and the 32nd IFIP WG 6.1 International Conference (FORTE 2012). Lecture Notes in Computer Science, vol. 7273, pp. 1–17. Springer (2012)
28. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: Hinze, R., Ramsey, N. (eds.) Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007. pp. 47–58. ACM (2007)
29. Mezzina, C.A., Tiezzi, F., Yoshida, N.: Rollback recovery in session-based programming. In: Jongmans, S., Lopes, A. (eds.) Proceedings of the 25th IFIP WG 6.1 International Conference on Coordination Models and Languages, COORDINATION 2023. Lecture Notes in Computer Science, vol. 13908, pp. 195–213. Springer (2023). https://doi.org/10.1007/978-3-031-35361-1_11
30. Milner, R.: A Calculus of Communicating Systems. Springer LNCS 92 (1980)
31. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: Hermenegildo, M., López-García, P. (eds.) Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016). Lecture Notes in Computer Science, vol. 10184, pp. 259–274. Springer (2017). https://doi.org/10.1007/978-3-319-63139-4_15
32. Nishida, N., Palacios, A., Vidal, G.: Reversible computation in term rewriting. J. Log. Algebraic Methods Program. **94**, 128–149 (2018). https://doi.org/10.1016/j.jlamp.2017.10.003
33. Perumalla, K.S., Park, A.J.: Reverse computation for rollback-based fault tolerance in large parallel systems - evaluating the potential gains and systems effects. Clust. Comput. **17**(2), 303–313 (2014). https://doi.org/10.1007/s10586-013-0277-4
34. Swalens, J., Koster, J.D., Meuter, W.D.: Transactional actors: communication in transactions. In: Jannesari, A., de Oliveira Castro, P., Sato, Y., Mattson, T. (eds.) Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems, SEPSSPLASH 2017. pp. 31–41. ACM (2017). https://doi.org/10.1145/3141865.3141866
35. Tsai, J., Wang, Y.: Communication-induced checkpointing protocols and rollback-dependency trackability: A survey. In: Wah, B.W. (ed.) Wiley Encyclopedia of Computer Science and Engineering. John Wiley & Sons, Inc. (2008). https://doi.org/10.1002/9780470050118.ecse059
36. Vassor, M., Stefani, J.B.: Checkpoint/Rollback vs Causally-Consistent Reversibility. In: Kari, J., Ulidowski, I. (eds.) Reversible Computation. pp. 286–303. Springer International Publishing, Cham (2018). https://doi.org/978-3-319-99498-7_20
37. Vidal, G.: From reversible computation to checkpoint-based rollback recovery for message-passing concurrent programs. CoRR **abs/2309.04873** (2023), `https://arxiv.org/abs/2309.04873`