Reversibility in Erlang: Imperative Constructs

Pietro Lami, Ivan Lanese, Jean-Bernard Stefani, Claudio Sacerdoti Coen, Giovanni Fabbretti

Reversibility in Erlang: Imperative Constructs*

Pietro Lami[1][0000−0002−1841−387X], Ivan Lanese[2][0000−0003−2527−9995], Jean-Bernard Stefani[1][0000−0003−1373−7602], Claudio Sacerdoti Coen[3][0000−0002−4360−6016], and Giovanni Fabbretti[1][0000−0003−3002−0697]

[1] Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
[2] Focus Team, Univ. of Bologna, INRIA, 40126 Bologna, Italy
[3] Univ. of Bologna, 40126 Bologna, Italy

Abstract. A relevant application of reversibility is causal-consistent reversible debugging, which allows one to explore concurrent computations backward and forward to find a bug. This approach has been put into practice in CauDEr, a causal-consistent reversible debugger for the Erlang programming language. CauDEr supports the functional, concurrent and distributed fragment of Erlang. However, Erlang also includes imperative features to manage a map (shared among all the processes of a same node) associating process identifiers to names. Here we extend CauDEr and the related theory to support such imperative features. From a theoretical point of view, the added primitives create different causal structures than those derived from the concurrent Erlang fragment previously handled in CauDEr, yet we show that the main results proved for CauDEr are still valid.

Keywords: Debugging · Erlang · Reversible computing · Causality

1 Introduction

Reversible computing is a programming paradigm in which programs run both forwards (the standard computation) and backwards. Any forward computation in a reversible language can be undone with a finite number of backward steps. Reversible computing has applications in many areas, such as low-power computing [13], simulation [1], robotics [19], biological modeling [20] and others. We are particularly interested in applying reversible computing to debugging [2].

In a sequential system, undoing forward actions in reverse order of completion starting from the last one produces a backward computation. Undoing a forward action can be seen as a backward action. In a concurrent environment, one cannot easily decide which is the last action since many actions can be executed at the same time, and a total order of actions may not be available. Even if a total order exists, undoing actions in reverse order may be too restrictive since the

* The work has been partially supported by French ANR project DCore ANR-18-CE25-0007 and by INdAM – GNCS 2020 project *Sistemi Reversibili Concorrenti: dai Modelli ai Linguaggi*. We thank the anonymous referees for their helpful comments and suggestions.

order of execution of concurrent actions may depend on the relative speed of the processors executing them and has no impact on the final state. For instance, when looking for a bug causing a visible misbehavior in a concurrent system, independent actions may be disregarded since they cannot contain the bug.

The first definition of reversibility in a concurrent setting has been proposed by Danos and Krivine [4]: *causal-consistent reversibility*. In short, it states that any action can be undone provided that all its effects (if any) have been undone.

The idea of a causal-consistent reversible debugger was introduced in [7]. The main concept of [7] is to use causal-consistent reversibility to explore backward a concurrent execution starting from a visible misbehavior looking for the bug causing it. The CauDEr debugger [2], described in [16,22,6], applies these ideas to provide a reversible debugger for the functional, concurrent and distributed fragment of the Erlang programming language [5].

Here, we extend CauDEr and its underlying theory by adding the support for some primitives that are not considered in the previous versions. These primitives, namely register, unregister, whereis and registered, provide imperative behaviors inside the Erlang language whose core is functional. More precisely, they define a map linking process identifiers (pids) to names. They make it possible to add, delete and read elements from the map. From the technical point of view, supporting these primitives is not trivial since they introduce causal dependencies that are different from those originating from the functional and concurrent fragment of Erlang considered in [16,17,22]. In particular, read actions commute, but do not commute with add and delete actions. Such causal dependencies cannot be reliably represented in the general approach to derive reversible semantics for a given language presented in [14], because the approach in [14] considers a causal relation based on resources consumed and produced only, and does not support read operations. Similar dependencies are considered in [6], to model the set of nodes in an Erlang network, but this model does not include a delete operation, while we consider one. Similar dependencies are also used in [8] to study operations on shared tuple spaces in the framework of the coordination language Klaim, however they only access single tuples, while we also access multiple tuples or check for the absence of a given tuple. Also, their work is in the context of an abstract calculus and has never been implemented.

The paper is structured as follows. Section 2 briefly recalls the reversible semantics on which CauDEr is based [22]. Then, in Section 3, we extend the reversible semantics of Erlang to support imperative features. In Section 4 we describe our extension to CauDEr. Finally, in Section 5 we discuss related work and conclude the paper with hints for future work. Due to space constraints we omit some technicalities, for proofs and further details we refer the interested reader to the companion technical report [12].

2 Background

We build our technical development on the reversible semantics for Erlang in [22]. We give below a quick overview of it, while referring to [22] for further details.

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{mod}_1 \ldots \mathit{mod}_n \\
\mathit{mod} & ::= & \mathit{fun\_def}_1 \ldots \mathit{fun\_def}_n \\
\mathit{fun\_def} & ::= & \mathit{fun\_rule}\{';' \mathit{fun\_rule}\}'.' \\
\mathit{fun\_rule} & ::= & \mathit{Atom}\ \mathit{fun} \\
\mathit{fun} & ::= & ([\mathit{exprs}])\ [\mathsf{when}\ \mathit{expr}] \to \mathit{exprs} \\
\mathit{exprs} & ::= & \mathit{expr}\ \{','\ \mathit{expr}\} \\
\mathit{expr} & ::= & \mathit{atomic} \mid \mathit{Var} \mid\ '\{'[\mathit{exprs}]'\}' \mid\ '['[\mathit{exprs}|\mathit{exprs}]']' \mid \mathsf{if}\ \mathit{if\_clauses}\ \mathsf{end} \\
& \mid & \mathsf{case}\ \mathit{expr}\ \mathsf{of}\ \mathit{cr\_clauses}\ \mathsf{end} \mid \mathsf{receive}\ \mathit{cr\_clauses}\ \mathsf{end} \mid \mathit{expr}\ !\ \mathit{expr} \\
& \mid & \mathit{pattern} = \mathit{expr} \mid [\mathit{Mod}:]\mathit{expr}([\mathit{exprs}]) \mid \mathit{fun\_expr} \mid \mathit{Op exprs} \\
\mathit{atomic} & ::= & \mathit{Atom} \mid \mathit{Char} \mid \mathit{Float} \mid \mathit{Integer} \mid \mathit{String} \\
\mathit{if\_clauses} & ::= & \mathit{expr} \to \mathit{exprs}\ \{';'\ \mathit{expr} \to \mathit{exprs}\} \\
\mathit{cr\_clause} & ::= & \mathit{pattern}\ [\mathsf{when}\ \mathit{expr}] \to \mathit{exprs}\ \{';' \mathit{pattern}\ [\mathsf{when}\ \mathit{expr}] \to \mathit{exprs}\} \\
\mathit{fun\_expr} & ::= & \mathsf{fun}\ \mathit{fun}\ \{';' \mathit{fun}\}\ \mathsf{end} \\
\mathit{patterns} & ::= & \mathit{pattern}\ \{','\ \mathit{pattern}\} \\
\mathit{pattern} & ::= & \mathit{atomic} \mid \mathit{Var} \mid\ '\{'[\mathit{patterns}]'\}' \mid\ '['[\mathit{patterns}|\mathit{pattern}]']'
\end{array}
$$

Fig. 1. Language syntax

The language syntax. Erlang is a functional, concurrent and distributed programming language based on the actor paradigm [10] (concurrency based on asynchronous *message-passing*).

The syntax of the language is shown in Fig. 1. A program is a collection of module definitions, a module is a collection of function definitions, a function is a mapping between the function name and the function expression. An expression can be a variable, an atom, a list, a tuple, a call to a function, a case expression, an if expression, or a pattern matching equation. We distinguish expressions and patterns. Here, patterns are built from atomic values, variables, tuples and lists. When we have a case *expr* of *cr_clauses* end expression we first evaluate *expr* to a value, say v, then we search for a clause that matches v and such that the guard when *expr* is satisfied. If one is found then the case construct evaluates to the clause expression. The if expression is very similar to the evaluation of the *case* expression just described. Pattern matching is written as $pattern\ =\ expr$. Then, $expr_1\ !\ expr_2$ allows a process to send a message to another one. Expression $expr_1$ must evaluate either to a pid or to an atom (identifying the receiver process) and $expr_2$ evaluates to the message payload, indicated with v. The whole function evaluates to v and, as a side-effect, the message will be sent to the target process. The complementary operation of message sending is receive *cr_clauses* end. This construct takes a message targeting the process that matches one of the clauses. If no message is found then the process suspends.

Erlang includes a number of built-in functions (BIFs). In [22], they only consider self, which returns the process identifier of the current process, and spawn, that creates a new process. BIFs supporting distribution are considered in [6]. For a deeper discussion we refer to [22,6].

The language semantics. Here we describe the semantics of the language. We begin by providing the definitions of *process* and *system*.

$$(Op)\ \frac{\mathsf{eval}(op, v_1, \ldots, v_n) = v}{\theta, C[op\ (v_1, \ldots, v_n)], S \xrightarrow{\tau} \theta, C[v], S}$$

Fig. 2. A sample rule belonging to the expression level.

Definition 1 (Process). *A process is a tuple $\langle p, \theta, e, S\rangle$, where p is the process pid, θ is the process environment, e is the expression under evaluation and S is a stack of process environments.*

Stack S is used to store away the process state to start a sub-computation of the expression under evaluation and then to restore it, once the sub-computation ends. We refer to [22] for a discussion on why it is needed.

Definition 2 (System). *A system is a tuple $\Gamma; \Pi$. Γ is the global mailbox, that is a set of messages of the form $(sender\_pid, receiver\_pid, payload)$. Π is the pool of running processes, denoted by an expression of the form*

$$\langle p_1, \theta_1, e_1, S_1\rangle \mid \ldots \mid \langle p_n, \theta_n, e_n, S_n\rangle$$

where "|" is an associative and commutative parallel operator.

The semantics in [22] is defined in a modular way, similarly to the one presented in [16,6]: there is a semantics for the expression level and one for the system level. This approach simplifies the design of the reversible semantics since only the system one needs to be updated. The expression semantics is defined as a labeled transition relation, where the label describes side-effects (e.g., creation of a message) or requests of information to the system level. The semantics is a classical call-by-value semantics for a higher-order language. Fig. 2 shows a sample rule of the expression level: the *Op* rule, used to evaluate arithmetic and relational operators. This rule uses the auxiliary function **eval** to evaluate the expression and an evaluation context C to find the redex in a larger term.

The system semantics uses the label from the expression level to execute the associated side-effect or to provide the necessary information. Below we list the labels used in the expression semantics:

– τ, denoting the evaluation of a (sequential) expression without side-effects;
– $\mathsf{send}(v_1, v_2)$, where v_1 and v_2 represent, respectively, the pid of the sender and the value of the message;
– $\mathsf{rec}(\kappa, \overline{cl_n})$, where $\overline{cl_n}$ denotes the n clauses of a receive expression;
– $\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])$, where a/n represents the name and arity of the function executed by the spawned process, while $[\overline{v_n}]$ is the list of its parameters.

Symbol κ is a placeholder for the result of the evaluation, not known at the expression level, that the system rules will replace with the correct value.

For space reasons, we do not show here the system rules, which are available in [22]. We show instead below how sample rules are extended to support reversibility.

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

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

Fig. 3. A sample rule belonging to the forward semantics and its counterpart.

A reversible semantics. Two relations describe the reversible semantics: one forward ($\rightharpoonup$) and one backward ($\leftharpoondown$). The former extends the system semantics using a *Landauer embedding* [13]. The latter proceeds in the opposite direction and allows us to undo an action by ensuring causal consistency, thus before undoing an action we ensure that all its consequences have been undone.

Syntactically, every process is extended with a history, denoted with h, which stores the information needed in the backward semantics to *undo* an action. In the semantic rules we highlight the history in red. The history is composed of *history items*, to distinguish the last rule executed by a process and track the related information. The history items introduced in [22] are:

$$\{\tau(\theta, e, S), \mathsf{send}(\theta, e, S, \{v, \lambda\}), \mathsf{rec}(\theta, e, S, p, \{v, \lambda\}), \mathsf{spawn}(\theta, e, S, p), \mathsf{self}(\theta, e, S)\}$$

Fig. 3 shows a sample rule from the forward semantics and its counterpart from the backward semantics. W.r.t. the standard semantics, here messages also carry a unique identifier λ, without which messages with the same value could not be distinguished. This choice is discussed in [16].

In the premises of rule *Send*, we can see the expression-level semantics in action, transitioning from configuration (θ, e, S) to (θ', e', S'). The forward semantics uses the corresponding label to determine the associated side-effect: the message $(p, p', \{v, \lambda\})$ is added to the set of messages Γ. Also, the history of process p is enriched with the corresponding history item.

The reverse rule, $\overline{\mathit{Send}}$, can be applied only when all the consequences of the *Send*, in particular the reception of the sent message, have been undone. Such constraint is enforced by requiring the message to be in Γ. Then we can remove the message $(p, p', \{v, \lambda\})$ from Γ and restore p to the previous state.

3 Reversible Erlang with Imperative Primitives

Syntax of imperative primitives. In our extension, atoms and pids are central. An atom is a literal constant. Pid is an abbreviation for process identifier: each process is identified by a pid. In Erlang, a pid can be associated to an atom. Thus, one can refer the process, e.g., when specifying the target of a message, using the associated atom instead of the pid. On the one hand, an atom is more meaningful than a pid for a human. On the other hand, this allows one to decide which process plays a given role. E.g., if a process crashes another one can be registered under the same atom so that the replacement is transparent to other processes (provided that they use the atom to interact). All pairs $\langle atom, pid\rangle$ form a map, shared among the processes of the same node (we consider here a single node, we discuss in Section 4 how to deal with multiple nodes).

Our extension is based on the syntax in Fig. 1, but we add the following built-in functions (BIFs):

- $\mathsf{register}/2$ (where /2 denotes the arity): given an atom a and a pid p, it inserts the pair $\langle a, p\rangle$ in the map and returns the atom **true**. If either the atom a or the pid p is already registered, an exception is raised;
- $\mathsf{unregister}/1$: given an atom a, it removes the (unique) pair $\langle a, p\rangle$ from the map and returns **true** if the atom a is found, raises an exception otherwise;
- $\mathsf{whereis}/1$: given an atom, it returns the associated pid if it exists, the atom **undefined** otherwise;
- $\mathsf{registered}/0$: returns a list (possibly empty) of all the atoms in the map.

3.1 Semantics of imperative features

Standard semantics of imperative features. According to the official documentation [5], the BIFs above are implemented in Erlang using request and reply signals between the process and the manager of the map. To simplify the modelization, we opted to implement these BIFs as synchronous actions. This choice does not alter the possible behaviors since the behavior visible to Erlang users is determined by the order in which the request messages are processed at the manager. We begin by providing the updated definition of *system* (the definition of process is unchanged).

Definition 3 (System). *A system is a tuple* $\Gamma; \Pi; \mathsf{M}$. Γ *and* Π *are as in Def. 2.* M *is a set of registered pairs atom-pid of the form* $\{\langle a_1, p_1\rangle; \ldots; \langle a_n, p_n\rangle\}$, *where* a_i *are atoms and* p_i *pids. Given an atom* a, M_a *is the set* $\{\langle a, p\rangle | \langle a, p\rangle \in \mathsf{M}\}$*; given a pid* p, M_p *is the set* $\{\langle a, p\rangle | \langle a, p\rangle \in \mathsf{M}\}$.

Sets M_a and M_p contain at most one element.

As in the previous section, we have a double-layered semantics: one level for expressions ($\rightarrow$) and one for systems ($\hookrightarrow$).

To simplify the presentation w.r.t. [22], we extend rule *Op* (Fig. 4) to deal also with built-in functions. To this end, we extend the operator **eval** to produce also the label for functions with side-effects. We define **eval** on them as:

- $\mathsf{eval}(\mathsf{self}) = (\kappa, \mathsf{self}(\kappa))$;
- $\mathsf{eval}(\mathsf{spawn}, \mathsf{fun}() \rightarrow exprs\ \mathsf{end}) = (\kappa, \mathsf{spawn}(\kappa, exprs))$;
- $\mathsf{eval}(\mathsf{register}, atom, pid) = (\kappa, \mathsf{register}(\kappa, atom, pid))$;
- $\mathsf{eval}(\mathsf{unregister}, atom) = (\kappa, \mathsf{unregister}(\kappa, atom))$;
- $\mathsf{eval}(\mathsf{whereis}, atom) = (\kappa, \mathsf{whereis}(\kappa, atom))$;
- $\mathsf{eval}(\mathsf{registered}) = (\kappa, \mathsf{registered}(\kappa))$.

On sequential expressions **eval** returns (v, τ), with v the result of the evaluation.

Thanks to our extension, rule *Op* in Fig. 4 covers all function invocations, including BIFs with side effects, while in [16,22] each such BIF requires a dedicated

$$(Op)\ \frac{\mathsf{eval}(op, v_1, \ldots, v_n) = (v, label)}{\theta, C[op\ (v_1, \ldots, v_n)], S \xrightarrow{label} \theta, C[v], S}$$

Fig. 4. Standard semantics: evaluation of function applications, revised.

rule. Furthermore, new BIFs with side effects can be added without changing the expression level (function **eval** needs to be updated though).

For space reasons, the other rules for evaluating expressions are collected in the companion technical report [12].

The semantics of the system level can be found in [22], the only difference is that the system now includes the shared map M. Equivalently, the rules describing the imperative primitives can be obtained from the ones in Fig. 5, which describes the forward semantics, by dropping the red part. Rules are divided into *write rules* (above the line), which modify the map, and *read rules* (below the line), that only read it. This has an impact on their concurrent behavior, as described later on. We highlight in blue the parts related to the map.

In all the rules, the tuple representing the system includes the map M, where we store all the registered pairs atom-pid.

Rule *RegisterS* defines the success case of the $\mathsf{register}$ BIF, which adds the tuple $\langle a, p' \rangle$ to the map. The $\mathsf{register}$ fails either when the atom a or the pid p' are already used, or when the pid p' refers to a dead process (this is checked by predicate $\mathsf{isAlive}$), as described by rule *RegisterF*. Similarly, for the $\mathsf{unregister}$, the success case corresponds to rule *UnregisterS*, which removes from the map the (unique) pair atom-pid for a given atom a. The failure case, when there is no pid registered under atom a, corresponds to rule *UnregisterF*. Both failure cases replace the current expression with ϵ and the current stack with $[\,]$. This denotes an uncaught exception (in this paper we do not consider exception handling). The predicate $\mathsf{isAlive}$ takes a pid p and the pool of running processes and controls that the process with pid p is alive ($\langle p, \theta, e, S \rangle$ with $e \neq \bot$).

Rules *SendS* and *SendF* define the behavior of send actions when the receiver is identified with an atom. The former is fired when the receiver is registered in the map, resulting in the addition of the message to Γ, the latter when it is not, resulting in an uncaught exception.

Rules *Whereis1*, *Whereis2* and *Registered* define the behavior of the respective primitives; these rules read M without modifying it. Rule *Registered* uses the auxiliary function $\mathsf{registered}$. We define it as: $\mathsf{registered}(\mathsf{M}) = [a_1, \ldots, a_n]$ where $\mathsf{M} = \{\langle a_1, p_1 \rangle, \ldots, \langle a_n, p_n \rangle\}$.

Finally, we have two rules dealing with process termination. If the pid of the process is not registered on the map, rule *End* simply changes the expression to $\bot$, denoting a terminated process. Otherwise, rule *EndUn* applies, additionally removing the pid from the map.

Reversible semantics. The definition of the forward semantics poses a number of challenges, due to the need of balancing two conflicting requirements when defining the history information to be stored. On the one hand, we need to keep

$$(\mathit{RegisterS})\ \frac{\theta, e, S \xrightarrow{\mathsf{register}(\kappa,a,p')} \theta', e', S' \qquad t \text{ fresh} \qquad \mathsf{M}_a = \emptyset \qquad \mathsf{M}_{p'} = \emptyset \qquad \mathsf{isAlive}(p', \Pi)}{\Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{regS}(\theta, e, S, \{\langle a, p', t, \top\rangle\}){:}h, \theta', e'\{\kappa \to \mathsf{true}\}, S'\rangle \mid \Pi; \mathsf{M} \cup \{\langle a, p', t, \top\rangle\}}$$

$$(\mathit{UnregisterS})\ \frac{\theta, e, S \xrightarrow{\mathsf{unregister}(\kappa,a)} \theta', e', S' \qquad \mathsf{M}_a = \{\langle a, p', t, \top\rangle\}}{\Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{del}(\theta, e, S, \mathsf{M}_a, \mathsf{M}^a \cup M^{p'}){:}h, \theta', e'\{\kappa \to \mathsf{true}\}, S'\rangle \mid \Pi; \mathsf{M} \setminus \mathsf{M}_a \cup \mathrm{kill}(\mathsf{M}_a)}$$

$$(\mathit{EndUn})\ \frac{e \text{ is a value } \lor e = \epsilon \qquad \mathsf{M}_p = \{\langle a, p, t, \top\rangle\}}{\Gamma; \langle p, h, \theta, e, [\,]\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{del}(\theta, e, [\,], \mathsf{M}_p, \mathsf{M}^a \cup \mathsf{M}^p){:}h, \theta, \bot, [\,]\rangle \mid \Pi; \mathsf{M} \setminus \mathsf{M}_p \cup \mathrm{kill}(\mathsf{M}_p)}$$

$$(\mathit{RegisterF})\ \frac{\theta, e, S \xrightarrow{\mathsf{register}(\kappa,a,p')} \theta', e', S' \qquad \mathsf{M}_a \neq \emptyset \lor \mathsf{M}_{p'} \neq \emptyset \lor \neg\ \mathsf{isAlive}(p', \Pi)}{\Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{readS}(\theta, e, S, \mathsf{M}_a \cup \mathsf{M}_{p'}){:}h, \theta, \epsilon, [\,]\rangle \mid \Pi; \mathsf{M}}$$

$$(\mathit{UnregisterF})\ \frac{\theta, e, S \xrightarrow{\mathsf{unregister}(\kappa,a)} \theta', e', S' \qquad \mathsf{M}_a = \emptyset}{\Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{readF}(\theta, e, S, a, \mathsf{M}^a){:}h, \theta, \epsilon, [\,]\rangle \mid \Pi; \mathsf{M}}$$

$$(\mathit{SendS})\ \frac{\theta, e, S \xrightarrow{\mathsf{send}(a,v)} \theta', e', S' \qquad \lambda \text{ fresh} \qquad \mathsf{M}_a = \{\langle a, p', t, \top\rangle\}}{\Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma \cup \{(p, p', \{v, \lambda\})\}; \langle p, \mathsf{sendS}(\theta, e, S, \{v, \lambda\}, \mathsf{M}_a){:}h, \theta', e', S'\rangle \mid \Pi; \mathsf{M}}$$

$$(\mathit{SendF})\ \frac{\theta, e, S \xrightarrow{\mathsf{send}(a,v)} \theta', e', S' \quad \mathsf{M}_a = \emptyset}{\Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{readF}(\theta, e, S, a, \mathsf{M}^a){:}h, \theta, \epsilon, [\,]\rangle \mid \Pi; \mathsf{M}}$$

$$(\mathit{Whereis1})\ \frac{\theta, e, S \xrightarrow{\mathsf{whereis}(\kappa,a)} \theta', e', S' \qquad \mathsf{M}_a = \{\langle a, p', t, \top\rangle\}}{\Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{readS}(\theta, e, S, \mathsf{M}_a){:}h, \theta', e'\{\kappa \to p'\}, S'\rangle \mid \Pi; \mathsf{M}}$$

$$(\mathit{Whereis2})\ \frac{\theta, e, S \xrightarrow{\mathsf{whereis}(\kappa,a)} \theta', e', S' \qquad \mathsf{M}_a = \emptyset}{\Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{readF}(\theta, e, S, a, \mathsf{M}^a){:}h, \theta', e'\{\kappa \to \mathsf{undefined}\}, S'\rangle \mid \Pi; \mathsf{M}}$$

$$(\mathit{Registered})\ \frac{\theta, e, S \xrightarrow{\mathsf{registered}(\kappa)} \theta', e', S' \qquad \mathsf{registered}(\mathsf{M}) = \mathit{atoms}}{\Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{readM}(\theta, e, S, \mathsf{M}){:}h, \theta', e'\{\kappa \to \mathit{atoms}\}, S'\rangle \mid \Pi; \mathsf{M}}$$

$$(\mathit{End})\ \frac{e \text{ is a value } \lor e = \epsilon \qquad \mathsf{M}_p = \emptyset}{\Gamma; \langle p, h, \theta, e, [\,]\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{readF}(\theta, e, [\,], p, \mathsf{M}^p){:}h, \theta, \bot, [\,]\rangle \mid \Pi; \mathsf{M}}$$

Fig. 5. Forward reversible semantics (standard semantics by dropping the red part).

enough information to be able to define a corresponding backward semantics. This requires to understand when all the consequences of an action have been undone, and to restore the state prior to its execution. On the other hand, we need to avoid storing information allowing one to distinguish computations obtained by only swapping independent actions (this would invalidate Lemma 2, as discussed in Example 3).

We first extend the definition of system.

Definition 4 (System). *A system is a tuple $\Gamma; \Pi; \mathsf{M}$. Γ and Π are as in Def. 3. Now each element of M is a quadruple $\langle a, p, t, s\rangle$ where a and p are as in Def. 3, t is a unique identifier for the tuple and s can be either $\top$ or $\bot$.*

Unique identifiers t are used to distinguish identical tuples existing at different times. For example, if we have two successful pairs of register and unregister operations of the same tuple, without a unique identifier we would not know which unregister operation is connected to which register. This information is relevant since the tuple generates a causal link between a register and the corresponding unregister. This justification is similar to the one for unique identifiers λ for messages, discussed in [17].

Tuples whose last field is $\top$ match the ones in the standard semantics, we call them *alive* tuples. Those with $\bot$ are *ghost* tuples, namely alive tuples that have been removed from the map in a past forward action. We will discuss their need in Example 2.

Given an atom a, M^a is the set $\{\langle a, p, t, \bot\rangle | \langle a, p, t, \bot\rangle \in \mathsf{M}\}$; similarly, given a pid p, $\mathsf{M}^p = \{\langle a, p, t, \bot\rangle | \langle a, p, t, \bot\rangle \in \mathsf{M}\}$. Dually, from now on, sets M_a and M_p include only alive tuples. We define function kill, which takes a map and sets to $\bot$ the last field of all its tuples.

We describe below the forward and backward semantics of the imperative primitives. The semantics of other constructs is as in the original work [22], but for the introduction of the global map M.

The forward semantics is defined in Fig. 5. The following history items have been added to describe the imperative features: regS, readS, readF, sendS, readM, and del. Notably, readS is created by both rules *RegisterF* and *Whereis*1 (which both read some alive tuples), readF is created by rules *UnregisterF*, *SendF*, *Whereis*2 and *End* (which all require the absence of some alive tuple), del is created by both rules *UnregisterS* and *EndUn* (which both turn an alive a tuple $\langle \_, \_, \_, \top\rangle$ into a ghost $\langle \_, \_, \_, \bot\rangle$).

All the new history items, like the old ones, carry the old state θ, e, S, thus allowing the backward computation to restore it. Furthermore, they carry some additional information to enable us to understand their causal dependencies:

- regS carries the tuple inserted in the map;
- readS carries the read tuple(s);
- sendS carries the read tuple as well, but also the sent message;
- readF carries the atom or the pid which the rule tried to read and the ghost tuples for such atom or pid, if any;
- readM carries the whole map read by the rule;
- del carries the removed tuple and the ghost tuples on the same atom or pid.

Fig. 6 presents the backward semantics. In previous works [6,22,17] there is one backward rule for each forward rule. Here, we were able to define one backward rule for each kind of history item, thus some backward rule covers more than one forward rule. This is possible because the history item contains enough information to correctly reverse forward rules with similar effects. E.g., both rules *RegisterF* and *Whereis*1 read information from the map, and the

$$(\overline{RegisterS})\quad \Gamma; \langle p, \mathsf{regS}(\theta, e, S, \{\langle a, p', t, \top\rangle\}){:}h, \theta', e', S'\rangle \mid \Pi; \mathsf{M} \cup \{\langle a, p', t, \top\rangle\} \leftharpoondown \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \quad \text{if } \mathsf{readop}(t, \Pi) = \emptyset$$

$$(\overline{Del})\quad \Gamma; \langle p, \mathsf{del}(\theta, e, S, \{\langle a, p', t, \top\rangle\}, \mathsf{M}_1){:}h, \theta', e', S'\rangle \mid \Pi; \mathsf{M} \cup \{\langle a, p', t, \bot\rangle\} \leftharpoondown \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \cup \{\langle a, p', t, \top\rangle\}$$
$$\text{if } \mathsf{M}_a = \emptyset \wedge \mathsf{M}_{p'} = \emptyset \wedge \mathsf{readmap}(\mathsf{M} \cup \{\langle a, p', t, \bot\rangle\}, \Pi) = \emptyset \wedge \mathsf{readfail}(t, \Pi) = \emptyset \wedge \mathsf{M}_1 = \mathsf{M}^a \cup \mathsf{M}^{p'}$$

$$(\overline{ReadS})\quad \Gamma; \langle p, \mathsf{readS}(\theta, e, S, \mathsf{M}_1){:}h, \theta', e', S'\rangle \mid \Pi; \mathsf{M} \leftharpoondown \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \quad \text{if } \mathsf{M}_1 \subseteq \mathsf{M}$$

$$(\overline{SendS})\quad \Gamma \cup \{(p, p', \{v, \lambda\})\}; \langle p, \mathsf{sendS}(\theta, e, S, \{v, \lambda\}, \mathsf{M}_1){:}h, \theta', e', S'\rangle \mid \Pi; \mathsf{M} \leftharpoondown \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \quad \text{if } \mathsf{M}_1 \subseteq \mathsf{M}$$

$$(\overline{ReadF})\quad \Gamma; \langle p, \mathsf{readF}(\theta, e, S, \iota, \mathsf{M}_1){:}h, \theta', e', S'\rangle \mid \Pi; \mathsf{M} \leftharpoondown \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \quad \text{if } \mathsf{M}_\iota = \emptyset \wedge \mathsf{M}_1 = \mathsf{M}^\iota$$

$$(\overline{ReadM})\quad \Gamma; \langle p, \mathsf{readM}(\theta, e, S, \mathsf{M}_1){:}h, \theta', e', S'\rangle \mid \Pi; \mathsf{M} \leftharpoondown \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \quad \text{if } \mathsf{M}_1 = \mathsf{M}$$

Fig. 6. Backward reversible semantics.

history item tracks the read information. Hence, a single rule can exploit this information to check that the same read information is still available in the map.

Rule $\overline{RegisterS}$ undoes the corresponding forward action, removing the element that was added by it. To this end, rule $\overline{RegisterS}$ requires that the element added from the corresponding forward rule is still in the map (ensuring that possible deletions of the same tuple have been undone) and, as a side condition, that no process performed a read operation on a tuple with unique identifier t. This last condition is checked by the predicate $\mathsf{readop}(t, \Pi)$, which scans the histories of processes in Π looking for such reads.

Rule $\overline{Del}$ undoes either rule $UnregisterS$ or rule $EndUn$, turning a ghost tuple back into an alive one. Let us discuss its side conditions. The first two conditions require that in M there is no alive tuple on the same atom a or process p'. The third one ensures that no process performed a registered getting M, while the fourth that no process read a ghost tuple with identifier t. Finally, we require ghost tuples on both a and p' to be the same as when the corresponding forward action has been performed. The last condition ensures that rule $\overline{Del}$ will not commute with pairs of operations that add and then delete tuples on the same atom or pid, e.g., a pair register-unregister. This is needed to satisfy the properties described in Section 3.2, such as causal consistency.

Rule $\overline{ReadS}$ reverses rules $Whereis1$ and $RegisterF$. The only side conditions requires that the element(s) read by the forward rule must be alive. Rule $\overline{SendS}$ is analogous, but it also requires that the sent message is in Γ.

Rule $\overline{ReadF}$ undoes actions from rules $UnregisterF$, $SendF$, $Whereis2$, and End. As a side condition, we require that no alive tuple matching ι - which is either a pid or an atom - exists and that the ghost tuples related to ι are the same as when the corresponding forward action triggered.

Rule $\overline{ReadM}$ is used to undo rule $Registered$. It requires that the map M_1 stored in the history is exactly the current map M.

3.2 Properties

Here we discuss some properties of the reversible semantics introduced in the previous section. Since most of the properties are related to causality, we need to study the concurrency model of the imperative primitives. Notably, this is not specific to reversibility and the same notion can be useful in other contexts, e.g., to find races [9].

To study concurrency for the imperative primitives we define for each history item k the set of resources (atoms and pids) read or written by the corresponding transition. The idea is that two transitions (including at least a forward one) are in conflict on the map if they both access the same resource and at least one of the accesses is a write (*RegisterS*, *UnregisterS*, *EndUn*). To obtain k, we indicate with $t = (s \rightleftharpoons_{p,r,k} s')$ a (forward or backward) transition from system s to system s', where p is the pid of the process performing the action, r is the applied rule and k the item added or removed to/from the history. We call computation a sequence of consecutive transitions, and denote with ϵ the empty computation. Two transitions are co-initial if they start from the same state, co-final if they end in the same state.

Definition 5 (Resources read or written). *We define functions* read(k) *and* write(k) *as follows:*

k	read(k)	write(k)
$\mathsf{regS}(\theta, \mathsf{e}, \mathsf{S}, \{\langle \mathsf{a}, \mathsf{p}, \mathsf{t}, \top \rangle\})$	$\emptyset$	$\{a, p\}$
$\mathsf{del}(\theta, \mathsf{e}, \mathsf{S}, \{\langle \mathsf{a}, \mathsf{p}, \mathsf{t}, \top \rangle\}, \mathsf{M})$	$\emptyset$	$\{a, p\}$
$\mathsf{readS}(\theta, \mathsf{e}, \mathsf{S}, \mathsf{M})$	$\{a \vert \mathsf{M}_a \neq \emptyset\} \cup \{p \vert \mathsf{M}_p \neq \emptyset\}$	$\emptyset$
$\mathsf{sendS}(\theta, \mathsf{e}, \mathsf{S}, \{\mathsf{v}, \lambda\}, \{\langle \mathsf{a}, \mathsf{p}, \mathsf{t}, \top \rangle\})$	$\{a, p\}$	$\emptyset$
$\mathsf{readF}(\theta, \mathsf{e}, \mathsf{S}, \iota, \mathsf{M})$	$\{\iota\}$	$\emptyset$
$\mathsf{readM}(\theta, \mathsf{e}, \mathsf{S}, \mathsf{M})$	$\{a \vert a \textit{ is an atom}\}$	$\emptyset$

Intuitively, items regS and del write on the resources a and p of the tuple added or removed. Item readS reads one or two tuples, and accesses in read modality all the involved pids and atoms. Item sendS just reads the atom and pid of the accessed tuple. Item readF accesses in read modality either an atom or a pid, as tracked in the history item. Finally, item readM exactly stores the current map, and needs to be in conflict with any transition writing on the map, even if it writes a tuple with atom and pid not previously used. Hence, we have chosen as read resources the set of all possible atoms, independently on whether they are currently used or not. We could also store all possible pids, but this will not impact the semantics, since each write access touches on an atom.

Definition 6 (Concurrent transitions). *Two co-initial transitions, $t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$, are* in conflict *if one of these conditions hold:*

- *if no transition is on the map, we refer to [17, Definition 12];*
- *if exactly one transition is on the map, they are in conflict if they are taken by the same process, namely $p_1 = p_2$, and a $\overline{SendS}$ is in conflict with a receive of the same message;*

– *if both transitions are on the map, and at least one is forward, then they are in conflict iff* $\mathrm{read}(k_1) \cap \mathrm{write}(k_2) \neq \emptyset$, $\mathrm{read}(k_2) \cap \mathrm{write}(k_1) \neq \emptyset$ *or* $\mathrm{write}(k_1) \cap \mathrm{write}(k_2) \neq \emptyset$*;*

Two co-initial transitions are concurrent if they are not in conflict.

Intuitively, concurrent transitions can be executed in any order (we will formalize this in Lemma 2). Notably, co-initial backward transitions are never in conflict.

Example 1 (Conflicting register*).* Consider a system S where two processes, say p_1 and p_2, try to register two different pids under the same atom a, and a is not already present in M (recall that an atom can be associated to one pid only). In this scenario the order in which the two actions are performed matters, because the first process to perform the action succeeds, while the second is doomed to fail. The two possibilities lead us to two states of the system, one where p_1 has succeeded and p_2 failed, say S', and the other where p_2 succeeded and p_1 failed, say S''. Clearly $S' \neq S''$, hence the two operations are in conflict. Indeed, $\mathrm{write}(k_1) \cap \mathrm{write}(k_2) = \{a\} \neq \emptyset$. ◊

Example 2 (Register followed by delete). Consider a system S where a process, say p_1, can do a registered operation. Another process, say p_2, performs a (successful) register followed by a delete operation (e.g., unregister) of a same tuple. In the standard semantics, executing first p_1 and then p_2 or vice versa would lead to the same state. If we were not using ghost tuples, the histories of p_1 and p_2 would be the same as well. However, we want to distinguish these two computations, since undoing the unregister would change the result of the registered, hence they cannot commute (cfr. Lemma 2). Ghost tuples are our solution to this problem. We get a similar behavior also if we consider, instead of the registered operation, any other read operation involving the added tuple. ◊

We can now discuss some relevant properties of the reversible semantics. As standard (see, e.g., [17] and the notion of consistency in [15]) we restrict to reachable systems, namely systems obtained from a single process with empty history (and empty Γ and M) via some computation. First, each transition can be undone.

Lemma 1 (Loop Lemma). *For every pair of reachable systems,* s_1 *and* s_2*, we have* $s_1 \rightharpoonup s_2$ *iff* $s_2 \leftharpoondown s_1$*.*

Let us denote with $\underline{t}$ the transitions undoing t, which exists thanks to the Loop Lemma. Next lemma shows that concurrent transitions can be executed in any order. It can be seen as a safety check on the notion of concurrency.

Lemma 2 (Square lemma). *Given two co-initial 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 transitions* $t_2/t_1 = (s_1 \rightleftharpoons_{p_2,r_2,k_2} s_3)$*,* $t_1/t_2 = (s_2 \rightleftharpoons_{p_1,r_1,k_1} s_3)$*. Graphically:*

$$\begin{tikzcd}
s \arrow[r, shift left, harpoon, "p_1{,}r_1{,}k_1"] \arrow[r, shift right, harpoon', leftarrow] \arrow[d, shift right, harpoon', "p_2{,}r_2{,}k_2"'] \arrow[d, shift left, harpoon, leftarrow] & s_1 \arrow[r, phantom, "\Longrightarrow"] & s \arrow[r, shift left, harpoon, "p_1{,}r_1{,}k_1"] \arrow[r, shift right, harpoon', leftarrow] \arrow[d, shift right, harpoon', "p_2{,}r_2{,}k_2"'] \arrow[d, shift left, harpoon, leftarrow] & s_1 \arrow[d, shift left, harpoon, "p_2{,}r_2{,}k_2"] \arrow[d, shift right, harpoon', leftarrow] \\
s_2 & & s_2 \arrow[r, shift left, harpoon] \arrow[r, shift right, harpoon', leftarrow, "p_1{,}r_1{,}k_1"'] & s_3
\end{tikzcd}$$

Next example shows that in order to ensure that the Square Lemma holds the semantics needs to be carefully crafted, in particular one should avoid to store information allowing to distinguish the order of execution of concurrent transitions.

Example 3 (Information carried by the register *history item).* If the history item of the register would contain the whole map, it would be impossible to swap the register action with an unregister action even if on a tuple with different pid and atom, because of the Square Lemma (Lemma 2). Indeed, the Square Lemma requires to reach the same state after two concurrent transitions are executed, regardless of their order. If we save the whole map in the history item of the register, we would reach two different states:

- if we execute the register operation first, the saved map would include the tuple that the unregister operation will delete;
- if we execute the unregister operation first, the map saved by the register will not contain the deleted tuple. ◊

We now want to prove causal-consistency [4,18], which essentially states that we store the correct amount of causal and history information.

Definition 7 (Causal Equivalence). *Let $\asymp$ be the smallest equivalence on computations closed under composition and satisfying:*

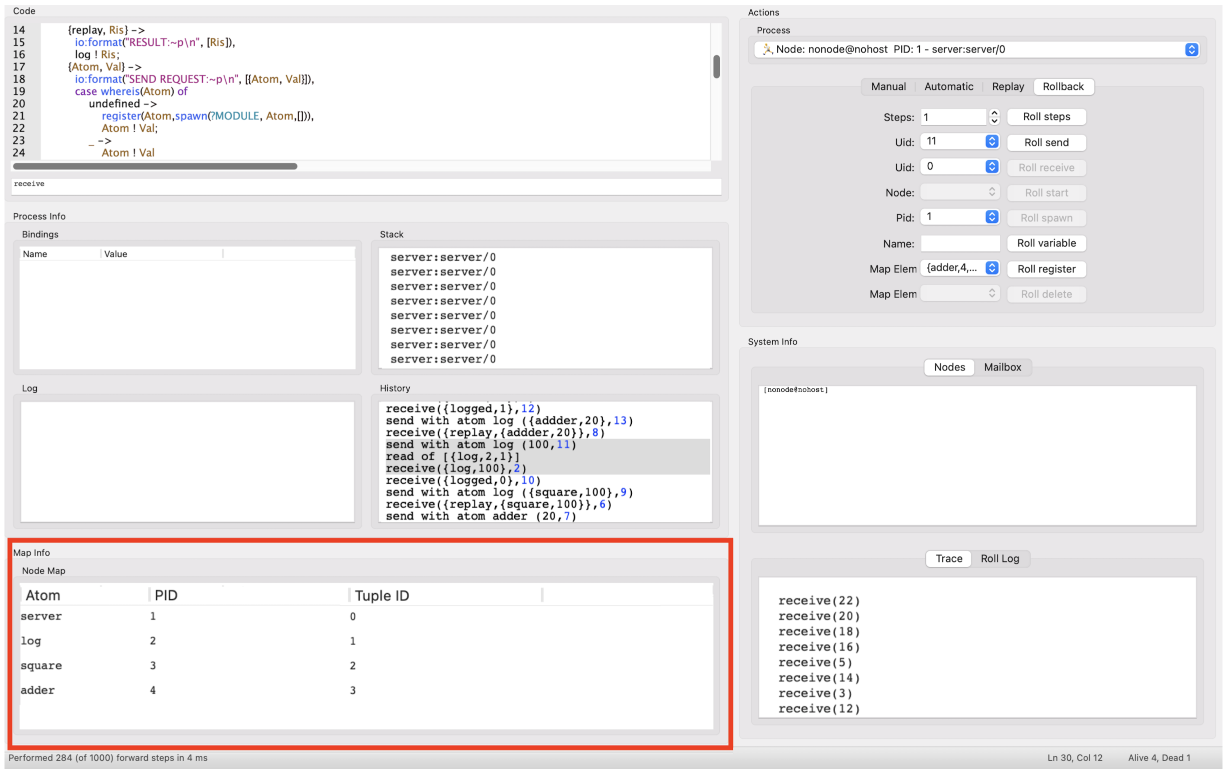
1. *if $t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$ are concurrent and $t_3 = (s_1 \rightleftharpoons_{p_2,r_2,k_2} s_3)$, $t_4 = (s_2 \rightleftharpoons_{p_1,r_1,k_1} s_3)$ then $t_1t_3 \asymp t_2t_4$;*
2. *$t\underline{t} \asymp \epsilon$ and $\underline{t}t \asymp \epsilon$*

Intuitively, computations are causal equivalent if they differ only for swapping concurrent transitions and for adding do-undo or undo-redo pairs of transitions.

Definition 8 (Causal Consistency). *Two co-initial computations are co-final iff they are causal equivalent.*

Intuitively, if co-initial computations are co-final then they have the same causal information and can reverse in the same ways: we want computations to reverse in the same ways iff they are causal equivalent.

In order to prove causal consistency, we rely on the theory developed in [18]. It considers a transition system with forward and backward transitions which satisfies the Loop Lemma and has a notion of independence. The latter is concurrency in our case. The theory allows one to reduce the proof of causal consistency and of other relevant properties to the validity of five axioms: Square Property (SP), Backward Transitions are Independent (BTI), Well-Foundedness (WF), Co-initial Propagation of Independence (CPI) and Co-initial Independence Respects Event (CIRE). SP is proved in Lemma 2, BTI corresponds to the observation that two backward transitions are always concurrent (see Def. 6), and WF requires backward computations to be finite. WF holds since each backward transition consumes an history item, which are in a finite number. CPI and



```
    {replay, Ris} ->
      io:format("RESULT:~p\n", [Ris]),
      log ! Ris;
    {Atom, Val} ->
      io:format("SEND REQUEST:~p\n", [{Atom, Val}]),
      case whereis(Atom) of
        undefined ->
          register(Atom,spawn(?MODULE, Atom,[])),
          Atom ! Val;
        _ ->
          Atom ! Val
```

```
receive
```

Process: Node: nonode@nohost PID: 1 - server:server/0

Steps: 1
Uid: 11
Uid: 0
Node:
Pid: 1
Name:
Map Elem: {adder,4,...
Map Elem:

Name	Value

```
server:server/0
server:server/0
server:server/0
server:server/0
server:server/0
server:server/0
server:server/0
server:server/0
```

```
receive({logged,1},12)
send with atom log ({addder,20},13)
receive({replay,{addder,20}},8)
send with atom log (100,11)
read of [{log,2,1}]
receive({log,100},2)
receive({logged,0},10)
send with atom log ({square,100},9)
receive({replay,{square,100}},6)
send with atom adder (20,7)
```

```
[nonode@nohost]
```

Atom	PID	Tuple ID
server	1	0
log	2	1
square	3	2
adder	4	3

```
receive(22)
receive(20)
receive(18)
receive(16)
receive(5)
receive(14)
receive(3)
receive(12)
```

Performed 284 (of 1000) forward steps in 4 ms   Ln 30, Col 12   Alive 4, Dead 1

Fig. 7. A screenshot of CauDEr.

CIRE hold thanks to [18, Prop. 5.4] because the notion of concurrency is defined in terms of transition labels only. Hence, causal consistency follows from [18, Prop. 3.6]. We obtain as well a number of other properties (a list can be found in [18, Table 1]), including various forms of causal safety and causal liveness, that intuitively say that a transition can be undone iff its consequences have been undone. We refer to [18] for precise definitions and further discussion.

4 CauDEr with Imperative Primitives

We exploited the theory presented above to extend CauDEr [2,16,22,6], a Causal-consistent reversible Debugger for Erlang. CauDEr is written in Erlang and provides a graphical interface for user interaction. Previously CauDEr supported only the sequential, concurrent and distributed fragment of Erlang, and we added support for the imperative primitives. The updated code can be found at [3].

While the theory discussed so far does not consider distribution, we extended the version of CauDEr supporting distribution [6], where systems can be composed of multiple nodes. As far as the imperative primitives are concerned, the only difference is that each node has its own map, shared only among its processes.

Fig. 7 shows a snapshot of the new version of CauDEr. The interface is organized as follows. On the left, from top to bottom, we can see (i) the program under debugging, (ii) the state, history and log (log is not discussed in this paper) of the selected process, (iii) the map of the node of the current process (the main novelty in the interface due to our extension, highlighted with a red square). On the top-right we can find execution controls (they are divided in multiple tabs, here we see the tab about rollback, described below), and on the bottom-right information on the system structure and on the execution.

CauDEr works as follows: the user selects the Erlang source file, then CauDEr loads the program and shows the source code to the user. Then, the user can choose the function that will act as an entry point, specify its arguments, and select the identifier of the node where the first process should run. The user can perform single steps on some process (both forward and backward), n steps in the chosen direction in automatic (a scheduler decides which process will execute each step), or use the rollback operator.

The rollback operator allows one to undo a selected action (e.g., the send of a given message) far in the past, including all and only its consequences. This is convenient to look for a bug causing a visible misbehavior, as described below. The semantics of the rollback operator roughly explores the graph of consequences of the target action, and undoes them in a causal-consistent order using the backward semantics. A formalization of the rollback operator can be found in the companion technical report [12].

Case study. We consider as a case study a simple server dispatching requests to various mathematical services, and logging the results of the evaluation on a logger. Services can be stateful, and are spawned only when there is a first request for them. Our example includes two stateless services, computing the square and the logarithm, respectively, and a stateful service adding all the numbers it receives. The logger keeps track of the values it receives, and answers each request with the sequential number of the element in the log. For space reasons we omit the full code of our case study, which anyway can be found either in the companion technical report [12] or in the repository [3].

In our sample scenario, we invoke the program with the list of requests [{square, 10}, {adder, 20}, {log, 100}, {adder, 30}, {adder, 100}].

The two first requests are successfully answered, while the request to compute the logarithm of 100 is not. By checking the history of the server (this is exactly the one shown in Fig. 7, relevant items are grayed, most recent items are on top) we notice that the request has been sent by the server as message 11. By using CauDEr rollback facilities to undo the send of message 11 (including all and only its consequences), one notice that the send has been performed at line 24 (also visible in the screenshot, upon rollback the line becomes highlighted), which is used for already spawned services. This is wrong since this is the first request for a logarithm. One can now require to rollback the register of atom log (used as target of the send). We can now see that the system logger has been registered under this atom in the main function:

```
register (log,spawn(?MODULE, logger,[0,[]])),
```

This is wrong. The bug is that the same atom has been used both for the system logger and for logarithm service.

Finding such a bug without the support of reversibility, and rollback in particular, would not be easy. Also, rollback allows us to go directly to points of interest (e.g., where atom log has been registered), even if we do not know which process performed the action. Hence, debugging via rollback scales better than standard techniques to larger programs, where finding the bug without reversibility would be even more difficult.

5 Conclusion, Related and Future Work

We have extended CauDEr and the underlying reversible semantics of Erlang to support imperative primitives used to associate names to pids. This required to distinguish write accesses from read accesses to the map, since the latter commute while the former do not. Also, the interplay between delete and read operations required us to keep track of removed tuples. Notably, a similar approach needs to be used to define the reversible semantics of imperative languages, such as C or Java.

While we discussed most related work, in particular work on reversibility in Erlang, in the Introduction, we mention here some related approaches. Indeed, reversibility of imperative languages with concurrency has been considered, e.g., in [11]. There however actions are undone (mostly) in reverse order of completion, hence their approach does not fit causal-consistent reversibility. Generation of reversible code is also studied in the area of parallel simulation, see, e.g., [21], but there reversed code is sequential, and concurrency is added on top of it by the simulation algorithm. Also, this thread of research lacks theoretical results.

The current approach, as well as the theory in [18] on which we rely to prove properties, defines independence as a binary relation on transitions. We plan to extend this approach in future work by defining independence as a binary relation on sequences of transitions, since we found cases where single transitions do not commute, while sequences can.

For instance, a $\mathsf{registered}()$ does not commute with either $\mathsf{register}(a, \_)$ or $\mathsf{unregister}(a)$, but it can commute with their composition since the set of registered tuples is the same before and after. Notably, covering this case would require to extend the theory in [18] as well.

References

1. C. D. Carothers, K. S. Perumalla, and R. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *TOMACS*, 9(3):224–253, 1999.
2. CauDEr repository. Available at `https://github.com/mistupv/cauder`, 2022.
3. CauDEr with imperative primitives repository. Available at `https://github.com/PietroLami/cauder`, 2022.

4. V. Danos and J. Krivine. Reversible communicating systems. In *CONCUR*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004.
5. Erlang/OTP 24.1.5. Available at `https://www.erlang.org/doc/index.html`.
6. G. Fabbretti, I. Lanese, and J. Stefani. Causal-consistent debugging of distributed Erlang programs. In *Reversible Computation*, LNCS, pages 79–95. Springer, 2021.
7. E. Giachino, I. Lanese, and C. A. Mezzina. Causal-consistent reversible debugging. In *FASE*, volume 8411 of *LNCS*, pages 370–384. Springer, 2014.
8. E. Giachino, I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-consistent rollback in a tuple-based language. *J. Log. Algebraic Methods Program.*, 88:99–120, 2017.
9. J. J. González-Abril and G. Vidal. A lightweight approach to computing message races with an application to causal-consistent reversible debugging. *CoRR*, 2021.
10. C. Hewitt, P. B. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245. William Kaufmann, 1973.
11. J. Hoey and I. Ulidowski. Reversible imperative parallel programs and debugging. In *Reversible Computation*, volume 11497 of *LNCS*, pages 108–127. Springer, 2019.
12. P. Lami, I. Lanese, J. Stefani, C. Sacerdoti Coen, and G. Fabbretti. Improving CauDEr with imperative features - Technical Report. `https://hal.archives-ouvertes.fr/hal-03655372`, 2022.
13. R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.
14. I. Lanese and D. Medic. A general approach to derive uncontrolled reversible semantics. In *CONCUR*, volume 171 of *LIPIcs*, pages 33:1–33:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
15. I. Lanese, C. A. Mezzina, and J. Stefani. Reversibility in the higher-order π-calculus. *Theor. Comput. Sci.*, 625:25–84, 2016.
16. I. Lanese, N. Nishida, A. Palacios, and G. Vidal. CauDEr: A causal-consistent reversible debugger for Erlang. In *FLOPS*, LNCS, pages 247–263. Springer, 2018.
17. I. Lanese, N. Nishida, A. Palacios, and G. Vidal. A theory of reversibility for Erlang. *J. Log. Algebraic Methods Program.*, 100:71–97, 2018.
18. I. Lanese, I. C. C. Phillips, and I. Ulidowski. An axiomatic approach to reversible computation. In *FoSSaCS*, volume 12077 of *LNCS*, pages 442–461. Springer, 2020.
19. J. S. Laursen, U. P. Schultz, and L. Ellekilde. Automatic error recovery in robot assembly operations using reverse execution. In *IROS*, pages 1785–1792. IEEE, 2015.
20. I. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In *RC*, volume 7581 of *LNCS*, pages 218–232. Springer, 2012.
21. M. Schordan, T. Oppelstrup, D. R. Jefferson, and P. D. Barnes Jr. Generation of reversible C++ code for optimistic parallel discrete event simulation. *New Gener. Comput.*, 36(3):257–280, 2018.
22. G. Vidal and J. J. González-Abril. Causal-consistent reversible debugging: Improving CauDEr. In *PADL*, volume 12548 of *LNCS*, pages 145–160. Springer, 2021.