A General Approach to Derive Uncontrolled Reversible Semantics

Ivan Lanese
Focus Team, University of Bologna/Inria, Bologna, Italy
ivan.lanese@gmail.com

Doriana Medić
Focus Team, University of Bologna/Inria, Sophia Antipolis, France
doriana.medic@gmail.com

Abstract

Reversible computing is a paradigm where programs can execute backward as well as in the usual forward direction. Reversible computing is attracting interest due to its applications in areas as different as biochemical modelling, simulation, robotics and debugging, among others. In concurrent systems the main notion of reversible computing is called *causal-consistent reversibility*, and it allows one to undo an action if and only if its consequences, if any, have already been undone.

This paper presents a general and automatic technique to define a causal-consistent reversible extension for given forward models. We support models defined using a reduction semantics in a specific format and consider a causality relation based on resources consumed and produced. The considered format is general enough to fit many formalisms studied in the literature on causal-consistent reversibility, notably Higher-Order π-calculus and Core Erlang, an intermediate language in the Erlang compilation. Reversible extensions of these models in the literature are ad hoc, while we build them using the same general technique. This also allows us to show in a uniform way that a number of relevant properties, causal-consistency in particular, hold in the reversible extensions we build. Our technique also allows us to go beyond the reversible models in the literature: we cover a larger fragment of Core Erlang, including remote error handling based on links, which has never been considered in the reversibility literature.

2012 ACM Subject Classification Theory of computation → Concurrency; Computing methodologies → Concurrent computing methodologies

Keywords and phrases Reversible computing, causality, process calculi, Erlang

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2020.33

Related Version A full version of the paper is available at `https://hal.archives-ouvertes.fr/hal-02902204`.

Funding This work has been partially supported by French ANR project DCore ANR-18-CE25-0007.
Ivan Lanese: also partially supported by INdAM as member of GNCS (Gruppo Nazionale per il Calcolo Scientifico).

Acknowledgements The authors thank the reviewers for their helpful comments and suggestions.

1 Introduction

Reversible computing considers systems that can compute backward, recovering past states, as well as forward. The studies on reversible computing gained in popularity in the 60's, thanks to the observation that only irreversible actions need to produce heat [21]. Beyond obtaining computing machinery with low heat dissipation, reversible computing found its application in a wide range of fields, from biochemical modelling [6, 12, 19, 41] to simulation [8], robotics [34], programming [35, 46, 29] and program debugging [5, 16, 36, 28]. The main objective of the theoretical computer science community in this research area has been to provide a foundational understanding of reversibility. Nowadays, in the literature, there is a number of formalisms describing different approaches to reversibility with the purpose to better understand its properties and characteristics, e.g. reversible computation in process algebras [10, 42, 26], Petri Nets [40, 37], event structures [47], logic circuits [14], etc.

In a sequential system, backward computation is obtained by undoing forward actions in reverse order of execution, starting from the last one. Undoing a forward action can be seen as a backward action. In a concurrent setting, where many processes are running at the same time, identifying the last action is not an easy task, and may sometimes be impossible. Therefore, alternative approaches have been considered. Here we consider the *causal-consistent approach* [10, 42, 27], which focuses on the causality relations between actions to decide which actions can be undone. Consequently, while designing a reversible model following the causal-consistent approach, one needs to take care of storing information on the past of the system, to be able to recover past states, but also causality information, to know which forward steps can be undone at a given moment. In order to show that a reversible model follows the causal-consistent approach, a number of properties need to be proved [10]. The most relevant are the Loop Lemma, showing that each action can be undone, the Square Lemma, showing that the chosen notion of causality is compatible with the semantics, and Causal Consistency, showing that the correct information is stored. More recently [32], Causal Safety and Causal Liveness have also been proposed, stating that an action can be undone if and only if its consequences, if any, are undone beforehand.

The aim of this paper is to explore how to mechanically obtain a causal-consistent reversible extension of a given forward-only model. This is in sharp contrast with most of the reversible models in the concurrency literature, which have been defined manually. An advantage of building the reversible model in this way is that the properties mentioned before are satisfied by construction. The only other work we are aware of providing an automatic technique is [42], which considers process calculi defined in a specific SOS format [43]. Differently from [42], we focus on forward systems defined using a reduction semantics (Section 2.1). While this is more limited since it does not consider open systems, our approach can deal with systems that do not fit the model in [42]. This is the case for both our case studies, namely higher-order π [44] and Core Erlang [7].

Given a forward-only system, we aim at building its *uncontrolled* [27] causal-consistent reversible extension. Here with uncontrolled we mean that at any moment both forward actions and backward actions are possible, and there is no policy on which action to prefer. Uncontrolled semantics is the basis for a reversible model, on top of which control policies selecting the actions to be done or undone can be added [11, 24, 2, 25].

Our approach works in two main steps. First, we attach a unique identifier, called *key*, to every entity (process, messages, etc.) of the forward system, and then we enrich the model with *memories*, where past information is stored (Section 2.2). After defining our method, we show that the reversible models built using it satisfy the properties of causal-consistent reversible models discussed above (Section 3). We prove them using a novel approach [32], which consists in showing that the system satisfies a few basic axioms.

To show the generality of our method, we apply it to two case studies: higher-order π-calculus [44] (used as a running example) and Core Erlang [7] (Section 4.2). After obtaining the corresponding reversible models, we show that, while syntactically different, they have the same behaviour as the ones in the literature [26, 30]. We also show how our approach can be used to go further than what it is in the literature. As an example, we extend reversible Core Erlang to also support Core Erlang constructs for remote error handling based on links (Section 4.3). Such an extension has never been considered in the reversibility literature.

Due to space limitations, further details are in the Appendix while proofs are in [23].

2 Our Approach

In this section we formally introduce our approach. We first define the constraints that the forward-only model we take in input needs to satisfy, and then we describe how to derive the syntax and semantics of the corresponding causal-consistent reversible model.

To give a better intuition about our approach, we will use as a running example its instantiation on the asynchronous Higher-Order π-calculus [44].

2.1 Forward model

We assume a forward model equipped with a reduction semantics. The syntax of the forward model is structured in two levels. The lower level is composed by entities, e.g., processes, messages and resources, ranged over by P, Q. There are no restrictions on the syntax of the lower level. The upper level needs to follow the structure below:

$$N ::= P \ \mid \ op_n(N_1, \ldots, N_n) \ \mid \mathbf{0}$$

Essentially, a system is obtained by composing entities using composition operators op_n, where n is the operator arity. Among the composition operators we assume a binary parallel composition operator, thus $N_1 \mid N_2$ represents the parallel composition of two systems. Additionally, $\mathbf{0}$ represents the empty system. Notably, $\mathbf{0}$ is not an entity.

Below we recall the syntax of HOπ-calculus and show how it fits in our framework.

▶ **Example 1.** The classical syntax of HOπ-calculus [44] is as follows:

$$P, Q ::= a\langle P\rangle \mid a(X) \triangleright P \mid \ (P \mid Q) \ \mid \nu a\,(P) \mid X \mid \mathbf{0}$$

Process variables are represented with X and channel names with a, b, c. Process $a\langle P\rangle$ sends message P over channel a while $a(X) \triangleright P$ denotes a process which receives a message on channel a and replaces it for X inside P. There is no continuation after output since the calculus is asynchronous. We denote parallel composition with $P \mid Q$ and its neutral element with $\mathbf{0}$. Restriction of name a inside P is written $\nu a\,(P)$. The binders are $\nu a\,(P)$ and $a(X) \triangleright P$, where the scope of name a and variable X is process P. We denote the set of free names of process P with $\mathtt{fn}(P)$.

In order to fit our framework we need to separate entities from systems. In this case, an entity is any HOπ process whose topmost operator is neither a parallel composition nor a restriction nor $\mathbf{0}$. The syntax of systems is thus as follows

$$N := P \mid \ (N_1 \mid N_2) \ \mid \nu a\,(N) \mid \mathbf{0}$$

where parallel composition and $\mathbf{0}$ are the operators required by our framework and restriction is an infinite family of unary operators with one instance for each name a. ⌟

Thanks to the syntax above, a generic system can be represented as a term $T[P_1, \ldots, P_n]$, where $T[\bullet_1, \ldots, \bullet_n]$ is a context with n numbered holes built from composition operators, possibly including parallel composition, and $\mathbf{0}$. The term $T[P_1, \ldots, P_n]$ is obtained by replacing $\bullet_i$ with P_i for each $i \in \{1, \ldots, n\}$.

We complement our syntax with a structural congruence, specified by axioms of the form

$$T[P_1, \ldots, P_n] \equiv T'[P'_1, \ldots, P'_n]$$

and closed under contexts, reflexivity, symmetry and transitivity. As can be seen from the rule format, structural congruence cannot change the number of entities in a term. Also, it is understood that P_i and P'_i refer to the same entity, which can however evolve while

$$(\textsc{Scm-Act})\ \frac{}{P_1 \mid \ldots \mid P_n \rightarrowtail T[Q_1, \ldots, Q_m]} \qquad (\textsc{Eqv})\ \frac{N \equiv N' \quad N \rightarrowtail N_1 \quad N_1 \equiv N_1'}{N' \rightarrowtail N_1'}$$

$$(\textsc{Scm-Opn})\ \frac{N_i \rightarrowtail N_i'}{op_n(N_0, \ldots, N_i, \ldots, N_n) \rightarrowtail op_n(N_0, \ldots, N_i', \ldots, N_n)} \qquad (\textsc{Par})\ \frac{N \rightarrowtail N'}{N \mid N_1 \rightarrowtail N' \mid N_1}$$

Figure 1 Forward rules structure; Scm- rules are schemas.

preserving its identity. This assumption will become clearer later on, when we introduce keys (to track the identity) and the causality relation. We assume structural rules ensuring that parallel composition is associative, commutative, and has **0** as neutral element.

We illustrate below that the structural congruence of HOπ satisfies the requirements.

▶ **Example 2.** Sample HOπ structural rules are as follows, the full structural congruence is in Appendix A.

$$(\textsc{Alpha})\ \nu a\, P \equiv \nu b\, P\{b/a\} \quad \text{if } b \notin \mathtt{fn}(P) \qquad (\textsc{ParC})\ P \mid Q \equiv Q \mid P$$
$$(\textsc{ResF})\ (\nu a\, P) \mid Q \equiv \nu a\, (P \mid Q) \quad \text{if } a \notin \mathtt{fn}(Q)$$

Rule (Alpha) is α-conversion. Rule (Alpha) is seen in our framework as an infinite family of rules (and the same for rule (ResF) for scope extrusion), for each a, P and b satisfying the side condition. Hence, no side condition is needed in the instance. Note that P on the left-hand side and $P\{b/a\}$ on the right-hand side are understood to be the same entity. Rule (ParC) establishes commutativity of parallel composition as required. It exploits contexts of the form $\bullet_1 \mid \bullet_2$ and $\bullet_2 \mid \bullet_1$. ⌟

The reduction semantics of the forward model needs to have the format described in Figure 1, which includes two rules ((Par) and (Eqv)), which need to belong to the semantics, and two schemas ((Scm-Act) and (Scm-Opn)). The semantics can contain any number of instances of the schemas (possibly an infinite number), obtained by replacing all placeholders with terms of the corresponding category (e.g., P_1 with an entity, T with a context, and so on). One may notice that rule (Par) is an instance of schema (Scm-Opn): this means that such an instance is required. Anyway, being an instance, we do not need to deal with it explicitly in the following.

Rule schema (Scm-Act) allows one to specify interactions between entities. It is understood that such an interaction consumes the entities $P_1, \ldots, P_n$ and produces the entities $Q_1, \ldots, Q_m$. This intuition will be captured by keys and the causality relation. The created entities are composed in a term $T[Q_1, \ldots, Q_m]$, where T is a context built from composition operators. Rules in this schema, together with rule (Par), allowing a system to execute inside a parallel composition, define the behaviour of parallel composition. The behaviour of other operators is described by rule schema (Scm-Opn). Notably, this schema allows a single entity to execute at each step. Rule (Eqv) allows one to exploit structural congruence.

We see below how the rule for communication of HOπ fits the format given in Figure 1. The full semantics of HOπ and the explanation of how it fits the format is in Appendix A.

▶ **Example 3.** The communication rule (Act) of HOπ, where process Q is received and bound to variable X, is defined as:

$$\text{(Act)}\ \frac{}{a\langle Q\rangle \mid a(X) \triangleright P \rightarrowtail P\{Q/X\}}$$

Rule (Act) can be seen as an infinite family of rules fitting schema (Scm-Act). Notice that the number of entities in the resulting process may vary, e.g., in:

$$a\langle \nu b\,(b\langle P\rangle \mid b(Y) \triangleright Y \mid c\langle Q\rangle)\rangle \mid a(X) \triangleright X \rightarrowtail \nu b\,(b\langle P\rangle \mid b(Y) \triangleright Y \mid c\langle Q\rangle)$$

the resulting process has three entities $b\langle P\rangle$, $b(Y) \triangleright Y$ and $c\langle Q\rangle$, composed using a context $T = \nu b\,(\bullet_1 \mid \bullet_2 \mid \bullet_3)$. ⌟

▶ **Example 4.** The CCS reduction $\overline{a}.P + Q|!a.R \rightarrowtail_{CCS} P|!a.R|R$, where the output $\overline{a}$ synchronises with the replicated input $!a$ and Q is discarded, can be seen as an instance of schema (Scm-Act) as well. Indeed, the two parallel entities $\overline{a}.P + Q$ and $!a.R$ interact to produce the three entities P, $!a.R$ and R on the right-hand side (assuming P and R to be single entities). ⌟

2.2 Definition of the Reversible System

In order to define the causal-consistent reversible extension of a given system, one first needs to extend the forward semantics so to keep track of past states. This information will be used by the backward semantics. In particular, we use unique *keys* to distinguish identical entities which have different history, and *memories* to recall parts of the system which have been changed by a computational step. More in detail, each entity of a system is labelled with its unique key. Also, each step of the system produces a memory allowing one to undo it. We refer to systems extended with keys and memories as *configurations*.

▶ **Definition 5.** *The syntax of configurations R is defined by the following grammar:*

$$R ::= k : P \mid op_n(R_1, \ldots, R_n) \mid \boldsymbol{0} \mid [R\,; C] \qquad C ::= T[k_1 : \bullet_1, \ldots, k_m : \bullet_m]$$

where operators op_n are the same as in the forward system and T is a context composed of operators op_n and $\boldsymbol{0}$. Also, $\bullet_i$ are numbered holes, to be filled by the processes with keys k_i.

Intuitively, a memory $\mu = [R\,; C]$ is composed of the configuration R which gave rise to the forward step and the context C of the configuration resulting from it.

▶ **Example 6.** The syntax of the reversible HOπ-calculus is defined as:

$$R ::= k : P \mid\ (R_1 \mid R_2)\ \mid \nu a\,(R) \mid \mathbf{0} \mid [R\,; C]$$

where entities P are as in the underlying calculus and a unique key k is attached to each of them. Note that now parallel composition and restriction operators are applied to configurations. Finally, memories are also part of the syntax. ⌟

We now define the structural congruence and the forward and backward operational semantics for the reversible system. As in the original model, we can represent a reversible system as $T[k_1 : P_1, \ldots, k_n : P_n]$, where T is a context built from operators op_n and $\mathbf{0}$. The main difference w.r.t. the original calculus is that now each entity is labelled with its key. We have one structural rule for each structural rule of the original semantics, with the same context T, but now entities are labelled with keys, and keys on both sides are the same.

$$T[k_1 : P_1, \ldots, k_n : P_n] \equiv T'[k_1 : P'_1, \ldots, k_n : P'_n]$$

We define below the function $\texttt{key}(\cdot)$ that computes the set of keys in a configuration R:

$$\text{(F-SCM-ACT)}\ \frac{P_1 \mid \ldots \mid P_n \rightarrowtail T[Q_1,\ldots,Q_m] \qquad j_1,\ldots,j_m \text{ are fresh keys}}{k_1 : P_1 \mid \ldots \mid k_n : P_n \twoheadrightarrow T[j_1 : Q_1,\ldots,j_m : Q_m] \mid [k_1 : P_1 \mid \ldots \mid k_n : P_n \,; T[j_1 : \bullet_1,\ldots,j_m : \bullet_m]]}$$

$$\text{(F-SCM-OPN)}\ \frac{R_i \twoheadrightarrow R_i' \quad (\mathtt{key}(R_i') \setminus \mathtt{key}(R_i)) \cap (\mathtt{key}(R_0,\ldots,R_{i-1},R_{i+1},\ldots,R_n) = \emptyset}{op_n(R_0,\ldots,R_i,\ldots,R_n) \twoheadrightarrow op_n(R_0,\ldots,R_i',\ldots,R_n)}$$

$$\text{(F-EQV)}\ \frac{R \equiv R' \quad R \twoheadrightarrow R_1 \quad R_1 \equiv R_1'}{R' \twoheadrightarrow R_1'}$$

Figure 2 Forward rules of the uncontrolled reversible semantics.

$$\text{(B-SCM-ACT)}\ \frac{\mu = [k_1 : P_1 \mid \ldots \mid k_n : P_n \,; T[j_1 : \bullet_1,\ldots,j_m : \bullet_m]]}{T[j_1 : Q_1,\ldots,j_m : Q_m] \mid \mu \rightsquigarrow k_1 : P_1 \mid \ldots \mid k_n : P_n}$$

$$\text{(B-SCM-OPN)}\ \frac{R_i' \rightsquigarrow R_i}{op_n(R_0,\ldots,R_i',\ldots,R_n) \rightsquigarrow op_n(R_0,\ldots,R_i,\ldots,R_n)} \qquad \text{(B-EQV)}\ \frac{R \equiv R' \quad R \rightsquigarrow R_1 \quad R_1 \equiv R_1'}{R' \rightsquigarrow R_1'}$$

Figure 3 Backward rules of the uncontrolled reversible semantics.

▶ **Definition 7.** *The* set of keys *of a configuration R, written as* $\mathtt{key}(R)$*, is defined as:*

$$\mathtt{key}(k : P) = \{k\} \qquad \mathtt{key}(op_n(R_1,\ldots,R_n)) = \mathtt{key}(R_1) \cup \ldots \cup \mathtt{key}(R_n)$$
$$\mathtt{key}(\mathit{0}) = \emptyset \qquad \mathtt{key}([R\,;C]) = \mathtt{key}(R) \cup \mathtt{key}(C)$$

The forward rules of the uncontrolled reversible semantics are in Figure 2. For schemas (F-Scm-Act) and (F-Scm-Opn) we have one instance for each instance of the corresponding schema in the original semantics. For schema (F-Scm-Act) the main difference w.r.t. the original schema is that entities are labelled with keys and a memory stores information on the performed step. More precisely, entities $Q_1, \ldots Q_m$ on the right-hand side have fresh keys $j_1, \ldots, j_m$. Also, the left configuration $R = k_1 : P_1 \mid \ldots \mid k_n : P_n$ is saved in a memory $\mu = [R; C]$ together with the context $C = T[j_1 : \bullet_1, \ldots, j_m : \bullet_m]$ of the resulting configuration. In this way, the structure of the obtained system and the newly generated keys are recorded. They will be needed to perform the corresponding backward step. As far as the schema (F-Scm-Opn) is concerned, the only novelty is the side condition ensuring that keys introduced during the step are fresh for the whole system. The structural congruence rule (F-Eqv) is the same as in the original semantics (but structural congruence preserves keys).

The backward rules, depicted in Figure 3, are symmetric w.r.t. the forward ones. With rule schema (B-Scm-Act) the forward action that produced term $T[j_1 : Q_1, \ldots, j_m : Q_m]$ is undone. The past state of the system $k_1 : P_1 \mid \ldots \mid k_n : P_n$ is restored from the memory μ. The context $C = T[j_1 : \bullet_1, \ldots, j_m : \bullet_m]$ inside μ additionally ensures that all entities produced by the forward action, together with the term composing them, are available in the configuration and are consumed by the backward step.

▶ **Definition 8** (Uncontrolled reversible semantics)**.** *The reduction relation $\twoheadrightarrow$ (resp. $\rightsquigarrow$), defined as the smallest relation closed under the forward (resp. backward) rules, defines the forward (resp. backward) reversible semantics. The semantics, denoted by $\rightarrow$, is the union of the forward semantics $\twoheadrightarrow$ and the backward semantics $\rightsquigarrow$ (i.e. $\rightarrow = \twoheadrightarrow \cup \rightsquigarrow$).*

▶ **Example 9.** Below, we give the communication rule of the forward and backward reversible semantics for the HOπ-calculus. The other rules can be found in Appendix A.

$$\text{(F-ACT)}\ \frac{a\langle P\rangle \mid a(X) \triangleright P' \rightarrowtail P'\{P/X\} \qquad j_1,\dots j_m \text{ are fresh keys} \quad \text{and} \quad P'\{P/X\} = T[Q_1,\dots,Q_m]}{\begin{array}{l} k_1 : a\langle P\rangle \mid k_2 : a(X) \triangleright P' \twoheadrightarrow T[j_1 : Q_1,\dots,j_m : Q_m] \mid \\ \qquad\qquad\qquad\qquad [k_1 : a\langle P\rangle \mid k_2 : a(X) \triangleright P' \,; T[j_1 : \bullet_1,\dots,j_m : \bullet_m]] \end{array}}$$

$$\text{(B-ACT)}\ \frac{\mu = [k_1 : a\langle P\rangle \mid k_2 : a(X) \triangleright P' \,; T[j_1 : \bullet_1,\dots,j_m : \bullet_m]]}{T[j_1 : Q_1,\dots,j_m : Q_m] \mid \mu \rightsquigarrow k_1 : a\langle P\rangle \mid k_2 : a(X) \triangleright P'}$$

Using rule schema (F-Act) a configuration can execute a forward step in which the memory $\mu = [k_1 : a\langle P\rangle \mid k_2 : a(X) \triangleright P' \,; T[j_1 : \bullet_1,\dots,j_m : \bullet_m]]$, recording the prior state of the configuration and the context with the new fresh keys, is generated. After the communication we obtain the system $P'\{P/X\}$ which we can rewrite as a term $T[Q_1,\dots,Q_m]$, where $Q_1,\dots,Q_m$ are entities. Using rule (B-Act) the configuration can undo the forward step which produced the memory μ. The prior state of the system is restored from it. ⌟

3 Properties

In this section, we show that the reversible semantics defined using the approach in the previous section satisfies a number of properties expected from a causal-consistent reversible semantics. In particular, the reversible semantics is a conservative extension of the forward semantics, and it is causally consistent [10].

Since our syntax allows for a number of ill-formed terms, as commonly done, in the following we restrict the attention to reachable configurations, defined below.

▶ **Definition 10** (Initial and reachable configuration). *A configuration R is* initial *if it does not contain memories and all keys are distinct. A configuration R is* reachable *if it can be derived from an initial configuration by applying the rules in Figures 2 and 3.*

Correspondence between reversible and original semantics

In this section we prove that the forward reversible semantics is a conservative extension of the original semantics. To this end, we first define the erasing function φ that given a configuration R, by deleting histories and keys, generates a forward-only system N.

▶ **Definition 11** (Erasing function). *The function $\varphi : \mathcal{R} \to \mathcal{N}$, where $\mathcal{R}$ and $\mathcal{N}$ denote respectively the sets of configurations and of original systems, is inductively defined as follows:*

$$\varphi(k : P) = P \quad \varphi([R\,; C]) = \mathit{0} \quad \varphi(\mathit{0}) = \mathit{0} \quad \varphi(op_n(R_1,\dots,R_n)) = op_n(\varphi(R_1),\dots,\varphi(R_n))$$

Now, we can show that the forward semantics of a configuration R and the semantics of its projection on the forward system $\varphi(R)$ are strong bisimilar (Definition 28 in Appendix A).

▶ **Theorem 12.** *For each configuration R, its forward semantics and the semantics of $\varphi(R)$ are strong bisimilar.*

3.1 Concurrency and Causal Consistency

In order to prove that the defined reversible semantics is indeed causal-consistent we need to define a causality relation on our systems. We define it directly on reversible systems, for two reasons. First, keys and memories help in this respect. Second, in a reversible system the concurrency relation induces a causality relation (see [30, Def. 11 and Lemma 6]).

We extend the reduction semantics to a notion of transitions, which carry in the label information on the used resources, in form of the memory involved in the transition. Formally, we define transitions t of a system R as $t : R \xrightarrow{\mu} R'$, where μ is the memory created by the transition, if it is forward, or consumed by it, if it is backward. There, R is the *source* while R' is the *target* of the transitions t. Two transitions are *coinitial* (resp. *cofinal*) if they have the same source (resp. target), and *composable* if the target of the former is the source of the latter. A *derivation* d from the source R to the target R', written as $d : R \to^* R'$, is a sequence of composable transitions. A zero steps derivation is written ϵ.

The concurrency relation between transitions states that two coinitial transitions are concurrent if they do not share entities. Formally:

▶ **Definition 13** (Concurrent transitions). *Two coinitial transitions $t' : R \xrightarrow{\mu'} R'$ and $t'' : R \xrightarrow{\mu''} R''$ are* concurrent*, written $t' \smile_c t''$, if $\mathtt{key}(\mu') \cap \mathtt{key}(\mu'') = \emptyset$. Coinitial transitions which are not concurrent are in* conflict*.*

Notably, our notion of concurrency is extracted from the operational semantics (via its extension with keys and memories), hence it can be obtained also for those models where no notion of concurrency exists in the literature, like most mainstream programming languages.

Having fixed a notion of concurrency, we can proceed to show the Causal Consistency of the reversible semantics. To prove it, we use the recent axiomatic approach given in [32], which allows one to show a number of properties relevant for reversible calculi, such as the Parabolic Lemma (PL) and Causal Consistency (CC), by just proving a few basic axioms. The advantage is that proving the axioms is simpler than proving the results directly. Moreover, [32] introduces two new properties: Causal Safety (CS) stating that an action cannot be reversed until all actions caused by it have been reversed; and Causal Liveness (CL) saying that actions do not necessarily need to be reversed in the exact inverse order of the forward execution, but can be reversed in any order consistent with CS.

In the following we give the axioms and auxiliary definitions required by the framework of [32] necessary to show Causal Consistency, Safety and Liveness.

First, we re-formulate our framework as a Labelled Transition System with Independence (LTSI, see also [45]) $(\mathcal{R}, \mathcal{L}, \to, \iota)$, where $\mathcal{R}$ is a set of systems, $\mathcal{L}$ is the set of action labels, $\to \subset \mathcal{R} \times \mathcal{L} \times \mathcal{R}$ is a transition relation and ι is the independence relation, namely an irreflexive symmetric binary relation on transitions. In our case, $\mathcal{R}$ is the set of configurations and $\mathcal{L}$ the set of labels of our transitions. The latter include both forward and backward transitions. Also, the notion of independence is defined on coinitial transitions and it coincides with the notion of concurrency, namely $\iota = \smile_c$. A key property required by the framework in [32] is that each action is reversible, as shown by the following result.

▶ **Lemma 14** (Loop Lemma). *For every reachable configuration R and forward transition $t : R \overset{\mu}{\twoheadrightarrow} R'$, there exists a backward transition $t^\bullet : R' \overset{\mu}{\rightsquigarrow} R$ and vice versa.*

From now on we denote with $t^\bullet$ the reverse of t. The basic properties required to show causal consistency are as follows.

▶ **Definition 15** (Basic axioms).

Square Property (SP): *if $t_1 : R \xrightarrow{\mu_1} R'$ and $t_2 : R \xrightarrow{\mu_2} R''$ are two coinitial independent transitions, there exist two cofinal transitions $t_2/t_1 : R' \xrightarrow{\mu_2} R'''$ and $t_1/t_2 : R'' \xrightarrow{\mu_1} R'''$.*

Backward transitions are independent (BTI): *any two coinitial backward transitions $t_1 : R \overset{\mu_1}{\rightsquigarrow} R_1$ and $t_2 : R \overset{\mu_2}{\rightsquigarrow} R_2$ where $t_1 \neq t_2$ are independent.*

Well-foundedness (WF): *there is no infinite backward computation.*

SP states that independent transitions can be executed in any order. We follow the standard notation and write t_2/t_1 for the residual of t_2 after t_1. Coinitial backward transitions are always independent by BTI. WF ensures that each system has a finite past.

To state Causal Consistency, we first define Causal Equivalence [10], an equivalence relation between derivations which stipulates that independent transitions can be swapped while pairs of reverse transitions can be removed from the derivation. The definition is well-posed if the LTSI satisfies the Square Property.

▶ **Definition 16** (Causal equivalence). *Causal equivalence, $\sim$, is the least equivalence relation between derivations closed under composition satisfying*

$$t_1; t_2/t_1 \sim t_2; t_1/t_2 \qquad t; t^\bullet \sim \epsilon$$

We now define two properties needed for Causal Safety and Causal Liveness, namely Coinitial propagation of independence (CPI) and Coinitial independence respects events (CIRE).

▶ **Definition 17** (Coinitial propagation of independence (CPI)). *If whenever $t_1 : R \xrightarrow{\mu_1} R'$, $t_2 : R \xrightarrow{\mu_2} R''$, $t'_2 : R' \xrightarrow{\mu_2} R'''$ and $t'_1 : R'' \xrightarrow{\mu_1} R'''$ with $t_1 \smile_c t_2$, then we have $t'_2 \smile_c t_1^\bullet$.*

We introduce the notion of *event*, needed to state (CIRE), and define independence (concurrency in our case) on them.

▶ **Definition 18** (Events). *Let $(\mathcal{R}, \mathcal{L}, \to, \smile_c)$ be a LTSI satisfying SP,BTI,WF and CPI. Let $\approx$ be the smallest equivalence relation satisfying: if $t_1 : R \xrightarrow{\mu_1} R'$, $t_2 : R \xrightarrow{\mu_2} R''$, $t'_2 : R' \xrightarrow{\mu_2} R'''$ and $t'_1 : R'' \xrightarrow{\mu_1} R'''$ and $t_1 \smile_c t_2$, then $t_1 \approx t'_1$.*
The equivalence classes of forward transitions $R \overset{\mu}{\twoheadrightarrow} R'$, written $[R, \mu, R']$, are the events. The equivalence classes of reverse transitions $R \overset{\mu}{\rightsquigarrow} R'$, $[R, \mu^\bullet, R']$, are the reverse events. A labelling function l from $\to / \approx$ to $\mathcal{L}$ is defined by settings $l([R, \mu, R']) = l([R, \mu^\bullet, R']) = \mu$. Events e_1, e_2 are (coinitially) independent, written $e_1 \,\mathtt{ci}\, e_2$, iff there are coinitial transitions t_1 and t_2 such that $[t_1] = e_1$, $[t_2] = e_2$ and $t_1 \smile_c t_2$.

▶ **Definition 19** (Coinitial independence respects events (CIRE)). *If $[t_1] \,\mathtt{ci}\, [t_2]$ and t_1 and t_2 are coinitial, then $t_1 \smile_c t_2$.*

▶ **Proposition 20.** *Axioms SP, BTI, WF, CPI and CIRE hold for each instance of our framework.*

Given that our reversible semantics satisfies all the axioms, thanks to [32], all instances of our framework satisfy the Parabolic Lemma, Causal Consistency, Causal Safety and Causal Liveness, defined below.

▶ **Definition 21** (Parabolic Lemma (PL)). *Given a derivation $d : R \to^* R'$, there exists a configuration R'' such that $d' : R \rightsquigarrow^* R'' \twoheadrightarrow^* R'$ and $d \sim d'$. Also, d' is not longer than d.*

▶ **Definition 22** (Causal Consistency (CC)). *Given two coinitial derivations d_1 and d_2, $d_1 \sim d_2$ if and only if d_1 and d_2 are cofinal.*

Below, we state Causal Safety and Causal Liveness. We present them in a slightly rephrased and more intuitive form w.r.t. [32], whose presentation is however more formal.

▶ **Definition 23** (Causal Safety (CS) and Causal Liveness (CL)).
Let $\mathsf{L} = (\mathcal{R}, \mathcal{L}, \to, \smile_c)$ be a LTSI satisfying SP,BTI,WF and CPI. Take a derivation $R \xrightarrow{\mu} R' \xrightarrow{\rho}{}^* R''$. Transition $R \xrightarrow{\mu} R'$ can be undone in R'', that is there is a transition $R_1 \xrightarrow{\mu} R''$ with $(R, \mu, R') \approx (R_1, \mu, R'')$, if (CL) and only if (CS) $R \xrightarrow{\mu} R'$ is concurrent to all transitions $R' \xrightarrow{\rho}{}^* R''$ which are not undone in $R' \xrightarrow{\rho}{}^* R''$.

4 Case Studies

In this section we apply our approach to two relevant case studies from the literature, the Higher-Order π-calculus [44] and Core Erlang [7]. Causal-consistent reversible semantics for both of them are available in the literature [26, 29]. We show that the ones derived using our approach, albeit syntactically different, are equivalent to the ones in the literature. In the case of Core Erlang we go beyond the literature, which covers only the functional and concurrent fragment of Core Erlang, showing how to deal also with constructs for error handling based on links.

4.1 Reversible Semantics for Higher-Order π-calculus

In the previous sections, we already shown how to apply our approach to the Higher-Order π-calculus. We show here that the semantics derived using our approach is equivalent to the one of $\rho\pi$, the reversible HOπ in the literature [26]. Additionally, it is easy to see that the notion of concurrency induced by our approach (Definition 13) on HOπ matches the definition of concurrent transitions of [26, Definition 9].

Our reversible HOπ and the one in the literature are indeed quite close, but for a few differences. Our approach stores a context for the resulting term, in $\rho\pi$ only a key is kept. Actually, the context is always composed by parallel operators and restriction operators. The former are always collected during $\rho\pi$ backward steps, the latter are instead removed by $\rho\pi$ structural congruence when no more needed. Also, in $\rho\pi$ restrictions for keys are explicit, in our approach they are implicit. In addition, the single key kept in $\rho\pi$ is split into multiple complex tags, in direct correspondence with our keys, by $\rho\pi$ structural congruence, hence in $\rho\pi$ one key is enough.

For instance, starting from the system $R = k_1 : a\langle P_1 \mid P_2\rangle \mid k_2 : a(X) \triangleright X$, in our reversible HO$\pi$ semantics, by applying rule (F-Act), we have:

$$k_1 : a\langle P_1 \mid P_2\rangle \mid k_2 : a(X) \triangleright X \twoheadrightarrow j_1 : P_1 \mid j_2 : P_2 \mid [R\,; j_1 : \bullet_1 \mid j_2 : \bullet_2]$$

In $\rho\pi$, by applying rule (R.Fw) followed by structural congruence [26], we have:

$$R \twoheadrightarrow \nu k\,.k(P_1 \mid P_2) \mid [R\,; k] \equiv \nu k, \tilde{j}\,.(\langle j_1, \tilde{j}\rangle \cdot k : P_1 \mid \langle j_2, \tilde{j}\rangle \cdot k : P_2) \mid [R\,; k]$$

Structural congruence splits key k referring to the whole continuation into complex tags $\langle j_i, \tilde{j}\rangle \cdot k$, where $\tilde{j} = \{j_1, j_2\}$ and $i \in \{1, 2\}$. By using structural congruence, complex tags for single entities can be always generated, as in our example above.

Despite the differences, our reversible HOπ semantics and $\rho\pi$ semantics [26] are equivalent. To show this, we exploit the encoding function $(\!|\cdot|\!) : \mathcal{R} \to \mathcal{M}$ which translates a reversible HOπ configuration into a $\rho\pi$ configuration. Function $(\!|\cdot|\!)$ needs to extract the set of keys of all entities obtained by the split from the memory of our HOπ system and to construct the complex tags of $\rho\pi$ configuration. The encoding function together with other technical details can be found in Appendix A. Using the encoding function above we can show a bijective correspondence between transitions in our approach and $\rho\pi$ transitions.

▶ **Theorem 24.** *Let R be a reachable configuration of reversible HOπ with $(\!|R|\!) = M$. There is a transition $R \to R'$ in reversible HOπ iff there is a $\rho\pi$ transition $M \to M'$ with $(\!|R'|\!) \equiv M'$.*

4.2 Reversible Semantics for Erlang

In this section we apply our approach to Core Erlang [7], an intermediate step in the compilation of the concurrent and functional language Erlang. We also show the equivalence between the obtained reversible semantics and the one in [30]. As a forward model, we use the logging semantics of Core Erlang [30, Figure 14] (used also in [31]) with some minor changes: we use floating messages, as in [33], instead of a global mailbox Γ and we omit the labels of the relation $\hookrightarrow$. Indeed, labels are used in [30] to log the steps of the computation so to be able to replay it from logs [31, 30]. In our work, we are not interested in replaying from logs, therefore we do not need this information.

The semantics of Core Erlang is defined in a modular way as in [30], with relation $\rightarrow$ modelling the evaluation of expressions and relation $\hookrightarrow$ representing reductions of systems. Due to space constraints, we only present the application of our approach to selected rules of the evaluation of systems $\rightarrow$, referring to the Appendix A for the others. Since evaluation of expressions is not central for us, we refer to [30] for their description.

A Core Erlang system E is defined as a pool of processes and floating messages:

$$E := \langle p, \theta, e\rangle \mid (p, p', v) \mid \ (E_1 \mid E_2)$$

where

- $\langle p, \theta, e\rangle$ represents a process evaluating expression e in environment θ and uniquely identified by a pid (process identifier) p;
- (p, p', v) stands for a floating message carrying value v sent by the process with pid p to the one with pid p'. A floating message is a message in the system after it is sent and before it is received.

We show below rules (Send) and (Rec) of Core Erlang semantics, the full semantics is in Figure 7 of Appendix A.

$$(\textsc{Send})\ \frac{\theta, e \xrightarrow{\mathtt{send}(p',v)} \theta', e'}{\langle p, \theta, e\rangle \hookrightarrow \langle p, \theta', e'\rangle \mid (p, p', v)} \qquad (\textsc{Rec})\ \frac{\theta, e \xrightarrow{\mathtt{rec}(\kappa,\overline{cl_n})} \theta', e' \ \text{ and } \mathtt{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{(p', p, v) \mid \langle p, \theta, e\rangle \hookrightarrow \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle}$$

Roughly speaking, rule (Send) states that if the evaluation of the expression e in the premise requires as a side effect to send value v to process p', the process evolves accordingly and a corresponding message is added to the system. Dually, rule (Rec) receives a message if the expression e requires a message matching some clauses $\overline{cl_n}$ and the message at hand indeed matches one of the clauses (second premise).

Now, we can apply our approach to the Core Erlang semantics and derive a reversible semantics for it. A reversible Core Erlang configuration, denoted with R, is defined as usual by adding keys and memories to an Erlang systems, as formalised by the following grammar:

$$R ::= k : \langle p, \theta, e\rangle \mid k : (p, p', v) \ \mid \ (R_1 \mid R_2) \mid [R; C]$$

In the following, we give the forward rules (F-Send) and (F-Rec) of the reversible semantics for Erlang. The complete set of forward rules is given in Figure 8 of Appendix A.

$$(\textsc{F-Send})\ \frac{\theta, e \xrightarrow{\mathtt{send}(p',v)} \theta', e' \quad k_1, k_2 \text{ are fresh keys}}{k : \langle p, \theta, e\rangle \twoheadrightarrow k_1 : \langle p, \theta', e'\rangle \mid k_2 : (p, p', v) \mid [k : \langle p, \theta, e\rangle\, ; k_1 : \bullet_1 \mid k_2 : \bullet_2]}$$

$$(\textsc{F-Rec})\ \frac{\theta, e \xrightarrow{\mathtt{rec}(\kappa,\overline{cl_n})} \theta', e' \quad \text{and} \quad \mathtt{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i) \quad k_1 \text{ is a fresh key}}{k_2 : (p', p, v) \mid k : \langle p, \theta, e\rangle \twoheadrightarrow k_1 : \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid [k_2 : (p', p, v) \mid k : \langle p, \theta, e\rangle\, ; k_1 : \bullet_1]}$$

Actually, both rules are to be interpreted as schemas, so that premises related to the semantics of expressions and to match are used to select the allowed instances and do not occur in actual instances. E.g., an allowed instance for (F-SEND) is:

$$\frac{k_1, k_2 \text{ are fresh keys}}{k : \langle p, \theta, p'!5\rangle \twoheadrightarrow k_1 : \langle p, \theta, 5\rangle \mid k_2 : (p, p', 5) \mid [k : \langle p, \theta, p'!5\rangle \,; k_1 : \bullet_1 \mid k_2 : \bullet_2]}$$

where ! is Erlang operator for sending.

Below, we give the backward rules (B-SEND) and (B-REC) of the reversible semantics for Erlang. The complete set of backward rules is given in Figure 9 of Appendix A. When the action is undone, the prior state of the process is restored from the memory μ.

$$\text{(B-SEND)}\ k_1 : \langle p, \theta', e'\rangle \mid k_2 : (p, p', v) \mid [k : \langle p, \theta, e\rangle \,; k_1 : \bullet_1 \mid k_2 : \bullet_2] \rightsquigarrow k : \langle p, \theta, e\rangle$$

$$\text{(B-REC)}\ k_1 : \langle p, \theta', e'\rangle \mid [k_2 : (p', p, v) \mid k : \langle p, \theta, e\rangle \,; k_1 : \bullet_1] \rightsquigarrow k_2 : (p', p, v) \mid k : \langle p, \theta, e\rangle$$

The reversible semantics obtained by applying our approach to Core Erlang is not exactly the same as the one of [30]. There are two important differences. First, we are not using execution logs, that we removed from the semantics we gave in input to our approach, since we do not need them.

Another difference is in how the past information of the system is stored. In [30], a history element h is kept as part of the process $\langle p, h, \theta, e\rangle$. It contains information to recover all past states of the process. In our reversible semantics, each step generates a memory with the information needed to reverse it, and the memories are connected using keys. Also, memories are not inside processes but floating in the configuration.

In the following, we prove that, despite the differences above, the two semantics capture the same behaviours. To this end, we first show that the two semantics are based on the same notion of causality (by showing that conflicting transitions are the same) and then that they are strong back and forth barbed equivalent [26]. Here we just discuss the idea, we refer to Appendix A for the technical details.

The notion of conflict for reversible Core Erlang in [30, Definition 12] (which is an instance of the happened-before relation [20] as discussed in [30]) is defined in general terms, referring to which actions (e.g., send, ...) are performed and by which processes. Hence, it is also applicable to our reversible Core Erlang. We show below that it coincides with the definition we gave, based on keys and memories.

▶ **Theorem 25** (Causal correspondence). *Two coinitial transitions t_1 and t_2 of our reversible Core Erlang semantics are in conflict according to [30, Definition 12] iff they are in conflict according to Definition 13.*

We show below that the reversible semantics of Erlang in [30] and ours are strong back and forth barbed equivalent [26]. We let E to stand for a Core Erlang system, L for a reversible Erlang system as in [30] and R for one of our reversible Erlang configurations.

Following [33], we write $E \downarrow p$ if the system E contains a floating message targeting a process with pid p (i.e., if $(p', p, v) \mid E' \equiv E$ for some p', v and E'). We use the same notation for systems L and configurations R, writing $L \downarrow p$ and $R \downarrow p$.

We now adapt the definition of back and forth barbed bisimulation [26] to reversible Erlang. In words, two reversible semantics are back and forth barbed bisimilar if they have the same barbs and they can match each other execution steps. Formally:

▶ **Definition 26.** *Relation $\mathcal{R}$ is a strong back and forth barbed simulation if $(L, R) \in \mathcal{R}$ implies:*

- $L \downarrow p$ *implies* $R \downarrow p$
- $L \rightharpoonup L'$ *implies* $R \twoheadrightarrow R'$ *with* $(L', R') \in \mathcal{R}$
- $L \leftharpoondown L'$ *implies* $R \rightsquigarrow R'$ *with* $(L', R') \in \mathcal{R}$

Relation $\mathcal{R}$ is a strong back and forth barbed bisimulation if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong back and forth barbed simulations. Strong back and forth barbed bisimilarity is the largest strong back and forth barbed bisimulation.

Now we can state the equivalence result between the two semantics.

▶ **Theorem 27.** *The reversible semantics of Erlang in [30] and our reversible semantics of Erlang are strong back and forth barbed bisimilar.*

4.3 Reversible Link Semantics for Erlang

Here, we apply our approach to the remote error handling mechanism of Core Erlang, based on links. No reversible semantics for it exists in the literature as far as we know. Defining it correctly does not present specific technical challenges, but it requires care, hence its definition is an interesting result on its own.

We start by giving some general idea about links and their role in Erlang (see [15] for more details). A link can be seen as a bidirectional path between two processes along which error signals travel. This can be used, e.g., to signal normal or abnormal termination. A process terminates normally when its code is completely executed, or it can terminate abnormally with a "reason", meaning that some faulty behaviour occurred during the execution. In both the cases, the process signals its termination to linked processes. This gives to the receiver the role of a controller in charge of handling the termination. There are two possibilities, depending on the nature of the receiver process: it can terminate too, or, if it is a system process, it can trap the termination signal and "resolve" the faulty behaviour. For instance, it could ignore it and continue with its execution, or start a copy of the terminated process, etc. Thanks to this feature, Erlang is particularly suited to build fault-tolerant systems [1].

In Erlang, links between two processes can be created by calling either function `link`(), linking any two processes (provided they are not terminated yet) or function `spawn_link`(), which spawns a new process and links it with the parent process atomically. In this work, we concentrate on function `spawn_link`(). Function `link`() can be dealt with similarly.

We start from the Core Erlang semantics discussed in the previous section and extend it to support the functions `spawn_link`() and `process_flag`(). The latter allows one to set the state of a process to system process, i.e. a process which will trap the error signal, or non-system process. More precisely, we add to Core Erlang syntax (see [30, Section 2.1, Figure 1]) expressions `spawn_link`$(expr, [expr_1, \ldots, expr_1])$ and `process_flag`$(expr_1, expr_2)$. In our case, function `process_flag`() is always called as `process_flag`$(trap\_exit, flag)$, where the process becomes a system process if `flag` = `true`, a non-system process otherwise.

We show now a sample Erlang program to clarify the error handling mechanism described above. It calculates the sum of a given list of elements and returns *invalid* if the list contains some non-numeric element.

The execution starts by calling function `total`(), which first sets the process flag to `true`. In this way, the process will be able to trap termination signals from any process linked with it. The execution proceeds by calling function `spawn_link`(), which atomically spawns and links a new process, executing function `sumProcess`(), in charge of calculating the sum via auxiliary function `sum`(). Because of the link, when the linked process terminates, its parent process will receive an exit notification message.

Finally, function `receiveValue`() is invoked. It checks whether the computation finished without misbehaviours: if this is the case message $\{'\texttt{EXIT}', Pid, \texttt{normal}\}$ is received and the function will read the result of the computation. If an error occurred during the computation message $\{'EXIT', Pid, \{\textbf{badarith}, Stack\}\}$ is received and the function returns atom *invalid*.

$\texttt{total}(List) \rightarrow$
  $\textbf{process\_flag}(trap\_exit, true),$
  $\texttt{SumPid} = \textbf{spawn\_link}(?MODULE, \texttt{sumProcess}, [\textbf{self()}, List]),$
  $\texttt{receiveValue}(\texttt{SumPid}).$
$\texttt{sumProcess}(Pid, List) \rightarrow Pid \ ! \ \texttt{sum}(List).$
$\texttt{sum}([\,]) \rightarrow 0;$
$\texttt{sum}([H|T]) \rightarrow H + \texttt{sum}(T).$
$\texttt{receiveValue}(Pid) \rightarrow$
  receive
    $\{'EXIT', Pid, normal\} \rightarrow$
      $\textbf{receive}\ Value \rightarrow Value\ \textbf{end};$
    $\{'EXIT', Pid, \{\textbf{badarith}, Stack\}\} \rightarrow invalid$
  end.

To integrate the functions `spawn_link`() and `process_flag`(), we add to processes two pieces of information, the set of links l and the flag f. The link set l is updated when a link is created, adding the pid of the other process, or destroyed, removing it. The flag f is a Boolean, tracking whether a process is a system process or not. Also, we say that the process $\langle p, \theta, e, l, f\rangle$ is *terminated* if $e = v$ for some value v (normal termination) or $e = r$ for some reason r (faulty termination).

Formally, a Core Erlang system supporting links is defined as a pool of floating messages, live and terminated processes, with the following grammar:

$$E := \langle p, \theta, e, l, f\rangle \mid (p, p', v) \mid \ (E_1 \mid E_2)$$

By applying our approach we obtain the syntax for reversible Core Erlang supporting links below. As usual, keys and memories are added to the system.

$$R := k : \langle p, \theta, e, l, f\rangle \mid k : (p, p', v) \mid \ R_1 \mid R_2 \ \mid [R; C]$$

The new rules of the reversible semantics of Erlang supporting links are in Figure 4, but for rule (F-Nrm) which is very similar to rule (F-Err) and is given in Appendix A. We do not show the original semantics, which can however be easily deduced by removing keys and memories from the one in Figure 4. The reversible semantics includes also all the rules in Figure 8, with the only addition of the set of links l and flag f in each process, which are not affected by those rules, but for the fact that when a process is spawned its link set is initialised to empty and its flag to `false`.

Rule (F-SpLink) above is similar to rule (F-Spawn) in Figure 8, with the addition that the link between the two processes is created, by inserting the pid of the other process in the link set. Rules (F-Err) and (F-Nrm) are similar: they both model the signalling of the termination of a process p to all the processes it is linked with. In both of them links are broken, by removing pids from link sets. The effect of termination depends on whether it is a normal termination, as in rule (F-Nrm), or an error termination, as in rule (F-Err). Also,

$$(\text{F-SpLink})\ \frac{\theta, e \xrightarrow{\texttt{spawn\_link}(\kappa, f/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid} \quad k_1, k_2 \text{ are fresh keys}}{\begin{array}{c} k : \langle p, \theta, e, l, f\rangle \twoheadrightarrow k_1 : \langle p, \theta', e'\{\kappa \mapsto p'\}, l \cup \{p'\}, f\rangle \mid k_2 : \langle p', id, \texttt{apply}\ f/n\ (\overline{v_n}), \{p\}, \mathit{false}\rangle \mid \\ [k : \langle p, \theta, e, l, f\rangle \,;\, k_1 : \bullet_1 \mid k_2 : \bullet_2] \end{array}}$$

$$(\text{F-Flag})\ \frac{\theta, e \xrightarrow{\texttt{process\_flag}(\kappa, \mathit{trap\_exit}, f')} \theta', e' \quad k_1 \text{ is a fresh key}}{k : \langle p, \theta, e, l, f\rangle \twoheadrightarrow k_1 : \langle p, \theta', e'\{\kappa \mapsto f\}, l, f'\rangle \mid [k : \langle p, \theta, e, l, f\rangle \,;\, k_1 : \bullet_1]}$$

$$(\text{F-Err})\ \frac{l = \{p_1, \ldots, p_m\} \quad 1 \le i \le n \Rightarrow f_i = \texttt{true} \wedge n+1 \le i \le m \Rightarrow f_i = \texttt{false} \quad h, h_i, j_i \text{ are fresh keys}}{\begin{array}{c} k : \langle p, \theta, r, l, f\rangle \mid \prod_{1\le i\le m} k_i : \langle p_i, \theta_i, e_i, l_i, f_i\rangle \twoheadrightarrow h : \langle p, \theta, r, \emptyset, f\rangle \mid \\ \prod_{1\le i\le n} h_i : \langle p_i, \theta_i, e_i, l_i \setminus \{p\}, f_i\rangle \mid \prod_{1\le i\le n} j_i : (p, p_i, \{\texttt{'EXIT'}, p, r\}) \mid \prod_{n+1\le i\le m} h_i : \langle p_i, \theta_i, r, l_i \setminus \{p\}, f_i\rangle \mid \\ [k : \langle p, \theta, r, l, f\rangle \mid \prod_{1\le i\le m} k_i : \langle p_i, \theta_i, e_i, l_i, f_i\rangle \,;\, h : \bullet_h \mid \prod_{1\le i\le m} h_i : \bullet_{h_i} \mid \prod_{1\le i\le n} j_i : \bullet_{j_i}] \end{array}}$$

Figure 4 Forward rules of the reversible link semantics for Erlang.

a termination signal affects system processes and non-system processes differently, and this is why in the two rules we split the processes in two groups according to the value of the flag. It can be set to the desired value using rule (F-Flag).

In rule (F-Err), the process terminates for some reason r. In this case messages $\{\texttt{'EXIT'}, p, r\}$ where p is the pid of the terminated process are sent to all system processes while non-system processes are forced to terminate. We can see the latter, e.g., in non-system process $\langle p_m, \theta_m, r, l_m \setminus \{p\}, f_m\rangle$ where expressions e_m is replaced by reason r, denoting abnormal termination. A memory is generated as usual, recording the past state of the system and the configuration of the resulting processes.

In rule (F-Nrm), the process terminates normally. The only differences w.r.t. rule (F-Err) is that non-system processes are unaffected and the messages are of the form $\{\texttt{'EXIT'}, p, \texttt{normal}\}$.

Backward rules are as usual.

We need to couple the rules in Figure 4, describing the semantics of Erlang configurations, with rules describing the evaluation of expressions, as the ones in [30, Figure 11]. Two main changes are needed. First, evaluation of operators may produce either a value or an error:

$$(\text{Call3})\ \frac{\texttt{eval}(op, v_1, \ldots, v_n) = x \text{ with } x = v \vee x = r}{\theta, \texttt{call}\ op(v_1, \ldots, v_n) \xrightarrow{\tau} \theta, x}$$

Then, one also needs to add rules to evaluate the functions `spawn_link`() and `process_flag`(). They are quite standard and can be found in Appendix A.

We are working to integrate the error mechanisms above into Erlang reversible debugger CauDEr [28]. CauDEr follows the reversible semantics of Core Erlang in [30], however our results can be rephrased in that setting, as hinted at by Theorems 25 and 27.

5 Conclusion, Related and Future Work

We presented a fully automatic method to extend a given forward model to a reversible one. Notably, our approach only needs a syntax and a reduction semantics of the forward model fitting our constraints. A causal semantics is produced as a by-product of our approach (see Definition 13). We exploited our method to obtain reversible extensions of Higher-Order π and Core Erlang. We showed that the obtained reversible semantics are equivalent to the ones in the literature [26, 30]. As an illustration that our approach can go beyond the literature, we tackled Core Erlang constructs for remote error handling based on links.

Sequential systems would correspond in our framework to single entities which evolve using instances of schemas having a single entity both on the left- and on the right-hand side. While our approach would create a reversible semantics for them, undoing actions in reverse order of execution, many of our results would become trivial.

In the concurrency literature, one can find many approaches defining a single reversible formalism or studying its properties, all using techniques tailored to the chosen model (e.g., [10, 9, 26, 39, 17, 38, 3, 18, 29, 37]). Indeed, our work can be seen as a generalisation of [26, 29], which we also used as case studies. A few works present general approaches able to cope with a number of formalisms. Our work fits in this class, hence we compare it below with the other approaches of this kind we are aware of. Also, since we deal with concurrent models, we focus on approaches targeting them as well.

Beyond ours, the only work that we are aware of providing a general and fully automatic method to derive a reversible semantics is [42], which considers calculi defined in a specific SOS format. Their approach allows to deal with open systems since their semantics is SOS, while our approach based on a reduction semantics considers only close systems. On the other hand, the higher degree of abstraction provided by reduction semantics simplifies the approach and makes it applicable to a wider range of formalisms. Indeed, the approach in [42] cannot cope with our two case studies, Higher-Order π-calculus and Core Erlang, since they do not fit their SOS format.

Also [4], which presents a modular framework to define reversible extensions of models such as CCS and concurrent X-machines, can deal with open systems. Its main limitation is that it is not fully automatic. Indeed, it requires to manually refine the labels of a given LTS to ensure properties such as determinism and codeterminism. This is far from trivial.

Two abstract approaches to reversibility are [13] and [32]. The former focuses on the interplay between reversible and irreversible actions, hence its results become trivial if, like in our case, there are no irreversible actions. We exploited the latter to prove properties of reversible models built using our approach. It concentrates on deriving properties from a set of axioms but gives no indication about how to render an irreversible system reversible.

Uncontrolled reversible semantics as obtained by our approach are the foundation of a reversible model on which one can build on, by adding control mechanisms [25] such as irreversible actions [11], rollback operators [17] or energy potentials [2]. An interesting line for future work is to integrate such approaches in our framework. For rollback, we could follow the ideas in [22], which leave however open the issue of how to manage rollback targets.

Another direction for future work is to adapt our approach so to handle further forward models. For instance, we currently cannot cope with the semantics of muKlaim defined in [17], since its concurrency model includes read dependencies. In particular, our approach is based on consumed and produced resources, while in [17] resources can also be read without being consumed. More in general, our approach can cope well with message-based concurrency modelled by some form of happened-before relation [20] (e.g., beyond our case studies, CCS, π-calculus and place-transition Petri nets) but not with read-write concurrency (e.g., beyond muKlaim, imperative languages). In order to extend our method to cope with read-write concurrency, we need to identify resources which are read but not consumed.

A last direction for future work concerns reducing the memory overhead of our approach. While it is difficult to find optimisations sound for every instance, many optimisations can work on specific classes of instances. E.g., in models where the context T in the instances of schema (Scm-Act) is always composed by parallel operators only, as in Core Erlang, there is no need to store T, but it is enough to store the set of fresh keys.

References

1 Joe Armstrong. Erlang – software for a concurrent world. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, page 1. Springer, 2007. doi:10.1007/978-3-540-73589-2_1.

2 Giorgio Bacci, Vincent Danos, and Ohad Kammar. On the statistical thermodynamics of reversible communicating processes. In *Algebra and Coalgebra in Computer Science – 4th International Conference, CALCO, Winchester, UK, August 30 – September 2, 2011. Proceedings*, volume 6859 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011. doi:10.1007/978-3-642-22944-2_1.

3 Kamila Barylska, Evgeny Erofeev, Maciej Koutny, Lukasz Mikulski, and Marcin Piatkowski. Reversing transitions in bounded Petri nets. *Fundam. Inform.*, 157(4):341–357, 2018. doi:10.3233/FI-2018-1631.

4 Alexis Bernadet and Ivan Lanese. A modular formalization of reversibility for concurrent models and languages. In *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9*, pages 98–112, 2016. doi:10.4204/EPTCS.223.7.

5 Bob Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, British Columbia, Canada, June 18-21*, PLDI '00, pages 299–310, New York, NY, USA, 2000. ACM. doi:10.1145/349299.349339.

6 Luca Cardelli and Cosimo Laneve. Reversible structures. In *Computational Methods in Systems Biology, 9th International Conference, CMSB 2011, Paris, France, September 21–23. Proceedings*, pages 131–140, 2011. doi:10.1145/2037509.2037529.

7 Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. *Core Erlang 1.0.3. Language specification*, 2004. URL: https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf.

8 Christopher D. Carothers, Kalyan S. Perumalla, and Richard Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM TOMACS*, 9(3):224–253, 1999. doi:10.1145/347823.347828.

9 Ioana Cristescu, Jean Krivine, and Daniele Varacca. A compositional semantics for the reversible pi-calculus. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28*, pages 388–397. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.45.

10 Vincent Danos and Jean Krivine. Reversible communicating systems. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004. doi:10.1007/978-3-540-28644-8_19.

11 Vincent Danos and Jean Krivine. Transactions in RCCS. In *CONCUR 2005 - Concurrency Theory, 16th International Conference, San Francisco, CA, USA, August 23-26, Proceedings*, volume 3653 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005. doi:10.1007/11539452_31.

12 Vincent Danos and Jean Krivine. Formal molecular biology done in CCS-R. *Electr. Notes Theor. Comput. Sci.*, 180(3):31–49, 2007. doi:10.1016/j.entcs.2004.01.040.

13 Vincent Danos, Jean Krivine, and Paweł Sobociński. General reversibility. In *Proceedings of the 13th International Workshop on Expressiveness in Concurrency, EXPRESS 2006, Bonn, Germany, August 26*, pages 75–86, 2006. doi:10.1016/j.entcs.2006.07.036.

14 Rolf Drechsler and Robert Wille. From truth tables to programming languages: Progress in the design of reversible circuits. In *IEEE International Symposium on Multiple-Valued Logic, ISMVL*, pages 78–85, 2011. doi:10.1109/ISMVL.2011.40.

15 Erlang website. https://www.erlang.org/.

16 Elena Giachino, Ivan Lanese, and Claudio Antares Mezzina. Causal-consistent reversible debugging. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Proceedings*, pages 370–384, 2014. doi:10.1007/978-3-642-54804-8_26.

17 Elena Giachino, Ivan Lanese, Claudio Antares Mezzina, and Francesco Tiezzi. Causal-consistent rollback in a tuple-based language. *J. Log. Algebraic Methods Program.*, 88:99–120, 2017. doi:10.1016/j.jlamp.2016.09.003.

18 Eva Graversen, Iain Phillips, and Nobuko Yoshida. Event structure semantics of (controlled) reversible CCS. In *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14. Proceedings*, volume 11106 of *Lecture Notes in Computer Science*, pages 102–122. Springer, 2018. doi:10.1007/978-3-319-99498-7_7.

19 Stefan Kuhn and Irek Ulidowski. Local reversibility in a calculus of covalent bonding. *Sci. Comput. Program.*, 151:18–47, 2018. doi:10.1016/j.scico.2017.09.008.

20 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

21 Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5:183–191, 1961. doi:10.1147/rd.53.0183.

22 Ivan Lanese. From reversible semantics to reversible debugging. In *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, Proceedings*, volume 11106 of *Lecture Notes in Computer Science*, pages 34–46. Springer, 2018. doi:10.1007/978-3-319-99498-7_2.

23 Ivan Lanese and Doriana Medić. A general approach to derive uncontrolled reversible semantics (TR). Available at https://hal.archives-ouvertes.fr/hal-02902204.

24 Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Controlling reversibility in higher-order pi. In *CONCUR 2011 - Concurrency Theory - 22nd International Conference, Aachen, Germany, September 6-9. Proceedings*, volume 6901 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011. doi:10.1007/978-3-642-23217-6_20.

25 Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Controlled reversibility and compensations. In *Reversible Computation, 4th International Workshop, RC 2012, Copenhagen, Denmark, July 2-3. Revised Papers*, volume 7581 of *Lecture Notes in Computer Science*, pages 233–240. Springer, 2012. doi:10.1007/978-3-642-36315-3_19.

26 Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversibility in the higher-order π-calculus. *Theor. Comput. Sci.*, 625:25–84, 2016. doi:10.1016/j.tcs.2016.02.019.

27 Ivan Lanese, Claudio Antares Mezzina, and Francesco Tiezzi. Causal-consistent reversibility. *Bulletin of the EATCS*, 114, 2014. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/305.

28 Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. Cauder: A causal-consistent reversible debugger for Erlang. In *Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9-11, Proceedings*, volume 10818 of *Lecture Notes in Computer Science*, pages 247–263. Springer, 2018. doi:10.1007/978-3-319-90686-7_16.

29 Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. A theory of reversibility for Erlang. *J. Log. Algebraic Methods Program.*, 100:71–97, 2018. doi:10.1016/j.jlamp.2018.06.004.

30 Ivan Lanese, Adrián Palacios, and Germán Vidal. Causal-consistent replay debugging for message passing programs. In *Technical report, DSIC, Universitat Politecnica de Valencia*, 2019. URL: http://personales.upv.es/gvidal/german/forte19tr/paper.pdf.

31 Ivan Lanese, Adrián Palacios, and Germán Vidal. Causal-consistent replay debugging for message passing programs. In *Formal Techniques for Distributed Objects, Components, and Systems - 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, Proceedings*, pages 167–184, 2019. doi:10.1007/978-3-030-21759-4_10.

32 Ivan Lanese, Iain C. C. Phillips, and Irek Ulidowski. An axiomatic approach to reversible computation. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, Proceedings*, pages 442–461, 2020. doi:10.1007/978-3-030-45231-5_23.

33 Ivan Lanese, Davide Sangiorgi, and Gianluigi Zavattaro. Playing with bisimulation in Erlang. In *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*, pages 71–91, 2019. doi:10.1007/978-3-030-21485-2_6.

34 Johan Sund Laursen, Ulrik Pagh Schultz, and Lars-Peter Ellekilde. Automatic error recovery in robot assembly operations using reverse execution. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2015, Hamburg, Germany, September 28 - October 2*, pages 1785–1792. IEEE, 2015. doi:10.1109/IROS.2015.7353609.

35 Christopher Lutz. Janus: a time-reversible language. *Letter to R. Landauer.*, 1986. URL: http://tetsuo.jp/ref/janus.html.

36 James McNellis, Jordi Mola, and Ken Sykes. Time travel debugging: Root causing bugs in commercial scale software. CppCon talk, https://www.youtube.com/watch?v=l1YJTg_A914, 2017.

37 Hernán C. Melgratti, Claudio Antares Mezzina, and Irek Ulidowski. Reversing P/T nets. In Hanne Riis Nielson and Emilio Tuosto, editors, *Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, Proceedings*, volume 11533 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2019. doi:10.1007/978-3-030-22397-7_2.

38 Claudio Antares Mezzina. On reversibility and broadcast. In *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14. Proceedings*, volume 11106 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2018. doi:10.1007/978-3-319-99498-7_5.

39 Claudio Antares Mezzina and Jorge A. Pérez. Causally consistent reversible choreographies: a monitors-as-memories approach. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*, pages 127–138. ACM, 2017. doi:10.1145/3131851.3131864.

40 Anna Philippou and Kyriaki Psara. Reversible computation in Petri nets. In *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, Proceedings*, pages 84–101, 2018. doi:10.1007/978-3-319-99498-7_6.

41 Iain Phillips, Irek Ulidowski, and Shoji Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In *Reversible Computation, 4th International Workshop, RC 2012, Copenhagen, Denmark, July 2-3. Revised Papers*, pages 218–232, 2012. doi:10.1007/978-3-642-36315-3_18.

42 Iain C. C. Phillips and Irek Ulidowski. Reversing algebraic process calculi. *J. Log. Algebraic Methods Program.*, 73(1-2):70–96, 2007. doi:10.1016/j.jlap.2006.11.002.

43 Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

44 Davide Sangiorgi. Bisimulation for higher-order process calculi. *Inf. Comput.*, 131(2):141–178, 1996. doi:10.1006/inco.1996.0096.

45 Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. Models of concurrency: Towards a classification. *Theoretical Computer Science*, 170(1-2):297–348, 1996. doi:10.1016/S0304-3975(96)80710-9.

46 Ulrik Pagh Schultz. Reversible object-oriented programming with region-based memory management - work-in-progress report. In *RC*, volume 11106 of *Lecture Notes in Computer Science*, pages 322–328. Springer, 2018. doi:10.1007/978-3-319-99498-7_22.

47 Irek Ulidowski, Iain Phillips, and Shoji Yuen. Reversing event structures. *New Generation Comput.*, 36(3):281–306, 2018. doi:10.1007/s00354-018-0040-8.

A Further Technical Material

Higher-Order π-calculus and reversible HOπ-calculus

This section recalls the semantics of the HOπ-calculus [44], discusses how it fits our framework and details the reversible semantics derived for it using our approach.

The standard rules of the structural congruence for the HOπ-calculus are:

$$(\textsc{ParC})\ P \mid Q \equiv Q \mid P \qquad (\textsc{ParA})\ P \mid (Q \mid S) \equiv (P \mid Q) \mid S \qquad (\textsc{Nil})\ P \mid \mathbf{0} \equiv P$$

$$(\textsc{Alpha})\ \nu a\, P \equiv \nu b\, P\{b/a\} \quad \text{if } b \notin \mathtt{fn}(P) \qquad (\textsc{ResF})\ (\nu a\, P) \mid Q \equiv \nu a\, (P \mid Q) \quad \text{if } a \notin \mathtt{fn}(Q)$$

Rules (ParC) and (ParA) ensure that parallel composition is commutative and associative, while rule (Nil) defines $\mathbf{0}$ as neutral element as required by our framework. Rule (Alpha) is α-conversion while rule (ResF) deals with scope extrusion.

The semantics of HOπ is given by the reduction relation $\rightarrowtail$ below:

$$(\textsc{Act})\ \frac{}{a\langle Q\rangle \mid a(X) \triangleright P \rightarrowtail P\{Q/X\}} \qquad (\textsc{Par})\ \frac{P \rightarrowtail P'}{P \mid Q \rightarrowtail P' \mid Q}$$

$$(\textsc{Res})\ \frac{P \rightarrowtail P'}{\nu a\, P \rightarrowtail \nu a\, P'} \qquad (\textsc{Eqv})\ \frac{P \equiv P' \quad P \rightarrowtail Q \quad Q \equiv Q'}{P' \rightarrowtail Q'}$$

Rule (Act) is the communication rule where process Q is received and bound to variable X. Process P can execute inside a parallel or a restriction operator thanks to rules (Par) and (Res), respectively. Rules (Par) and (Eqv) are as required by our framework. Rules (Act) and (Res) can be seen as infinite families of rules fitting schemas (Scm-Act) and (Scm-Opn), respectively.

In Figure 5, we give forward and backward rules of the reversible semantics for the HOπ-calculus derived using our approach.

$$(\textsc{F-Act})\ \frac{a\langle P\rangle \mid a(X) \triangleright P' \rightarrowtail P'\{P/X\} \qquad j_1, \ldots j_m \text{ are fresh keys} \quad \text{and} \quad P'\{P/X\} = T[Q_1, \ldots, Q_m]}{\begin{array}{r} k_1 : a\langle P\rangle \mid k_2 : a(X) \triangleright P' \twoheadrightarrow T[j_1 : Q_1, \ldots, j_m : Q_m] \mid \\ [k_1 : a\langle P\rangle \mid k_2 : a(X) \triangleright P' \,; T[j_1 : \bullet_1, \ldots, j_m : \bullet_m]] \end{array}}$$

$$(\textsc{F-Par})\ \frac{R \twoheadrightarrow R' \quad (\mathtt{key}(R') \setminus \mathtt{key}(R)) \cap \mathtt{key}(R_1) = \emptyset}{R \mid R_1 \twoheadrightarrow R' \mid R_1} \qquad (\textsc{F-Res})\ \frac{R \twoheadrightarrow R'}{\nu a\,(R) \twoheadrightarrow \nu a\,(R')}$$

$$(\textsc{F-Eqv})\ \frac{R \equiv R' \quad R \twoheadrightarrow R_1 \quad R_1 \equiv R_1'}{R' \twoheadrightarrow R_1'}$$

$$(\textsc{B-Act})\ \frac{\mu = [k_1 : a\langle P\rangle \mid k_2 : a(X) \triangleright P' \,; T[j_1 : \bullet_1, \ldots, j_m : \bullet_m]]}{T[j_1 : Q_1, \ldots, j_m : Q_m] \mid \mu \rightsquigarrow k_1 : a\langle P\rangle \mid k_2 : a(X) \triangleright P'} \qquad (\textsc{B-Par})\ \frac{R' \rightsquigarrow R}{R' \mid R_1 \rightsquigarrow R \mid R_1}$$

$$(\textsc{B-Res})\ \frac{R' \rightsquigarrow R}{\nu a\,(R') \rightsquigarrow \nu a\,(R)} \qquad (\textsc{B-Eqv})\ \frac{R \equiv R' \quad R \rightsquigarrow R_1 \quad R_1 \equiv R_1'}{R' \rightsquigarrow R_1'}$$

Figure 5 Forward and backward rules of the reversible semantics for the Higher-Order π-calculus.

Properties

This section contains further technical material related to Section 3. We start by defining strong bisimilarity between the forward semantics of a reversible configuration R and the semantics of its projection on the original model $\varphi(R)$.

▶ **Definition 28.** *A relation $\mathcal{R}$ between reversible configurations R and forward-only systems N is a* strong bisimulation *whenever for each $(R, N) \in \mathcal{R}$:*

- *if $R \twoheadrightarrow R'$, then $N \rightarrowtail N'$ with $(R', N') \in \mathcal{R}$;*
- *if $N \rightarrowtail N'$, then $R \twoheadrightarrow R'$ with $(R', N') \in \mathcal{R}$.*

Strong bisimilarity is the largest strong bisimulation.

Notably, only forward actions of the reversible systems need to be matched.

Concurrency and Causal Consistency. To fit the framework of [32], we re-formulate our framework as a labelled transition system enriched with an independence relation (LTSI) [45, Definition 3.7].

▶ **Definition 29.** *A* labelled transition system *(LTS) is a structure $(\mathcal{R}, \mathcal{L}, \rightarrow)$, where $\mathcal{R}$ is a set of systems, $\mathcal{L}$ is the set of action labels and $\rightarrow \subset \mathcal{R} \times \mathcal{L} \times \mathcal{R}$ is a transition relation.*

▶ **Definition 30.** *A* labelled transition system with independence *(LTSI) is a structure $(\mathcal{R}, \mathcal{L}, \rightarrow, \iota)$, where $(\mathcal{R}, \mathcal{L}, \rightarrow)$ is an LTS and ι is the independence relation (an irreflexive symmetric binary relation on transitions).*

In our case, $\mathcal{R}$ is the set of configurations and $\mathcal{L}$ the set of labels of our transitions. The latter include both forward and backward transitions. Also, the notion of independence is defined on coinitial transitions and it coincides with the notion of concurrency, namely $\iota = \smile_c$.

Given that our reversible semantics satisfies all the required axioms (Proposition 20), thanks to [32], all instances of our framework satisfy the Parabolic Lemma, Causal Consistency, Causal Safety and Causal Liveness.

▶ **Proposition 31.** *For every instance of our framework, the LTSI $(\mathcal{R}, \mathcal{L}, \rightarrow, \smile_c)$ satisfies the Parabolic Lemma, Causal Consistency, Causal Safety and Causal Liveness.*

Correspondence between our reversible HOπ and $\rho\pi$

This section contains technical material necessary to define the correspondence between our reversible HOπ and $\rho\pi$ (Section 4.1).

In the following, we recall the definition of thread normal form from [26, Lemma 1] using which, by exploiting structural congruence, unique keys are generated for each primitive thread process in a configuration (primitive thread processes are entities in our terminology).

▶ **Definition 32** (Thread normal form). *For any closed configuration M in $\rho\pi$, we have*

$$M \equiv \nu\tilde{u} \prod_{i \in I} (\kappa_i : \rho_i) \prod_{j \in J} [\mu_j \,; k_j] \qquad \textit{with} \quad \rho_i = a_i\langle P_i\rangle \quad \textit{or} \quad \rho_i = a_i(X_i) \triangleright P_i$$

We now present an encoding from our reversible HOπ to $\rho\pi$. The encoding works in two steps: a first step explores memories in our configurations to find related sets of keys, the second step uses the gathered information to actually perform the translation. The first step is done by function $\mathtt{col}(R)$, which computes (possibly annotated) sets of keys, one for each memory $[R' \,; C]$ in R. If C contains more than one key, the extracted set of keys is annotated with a fresh key k, what is denoted with $\mathtt{key}(C)_k$. The formal definition of function $\mathtt{col}(R)$ is in Figure 6. Note that the result of function $\mathtt{col}(R)$ is a set of possibly annotated sets of keys. We denote with $\tilde{h}_k$ and $\tilde{h}$ sets $\mathtt{key}(C)_k$ and $\mathtt{key}(C)$, respectively. We also assume to have a fresh key generator, giving us fresh keys k as needed.

$$
\begin{aligned}
\mathtt{col}(\nu a\,(R)) &= \mathtt{col}(R)\\
\mathtt{col}(R_1 \mid R_2) &= \mathtt{col}(R_1) \cup \mathtt{col}(R_2)\\
\mathtt{col}([R\,;C]) &= \{\mathtt{key}(C)_k\} \quad \text{where } k \text{ is fresh} \quad \text{if } |\mathtt{key}(C)| > 1\\
\mathtt{col}([R\,;C]) &= \{\mathtt{key}(C)\} \qquad\qquad \text{if } |\mathtt{key}(C)| = 1\\
\mathtt{col}(k : P) &= \emptyset\\
\mathtt{col}(\mathbf{0}) &= \emptyset
\end{aligned}
$$

Figure 6 Function $\mathtt{col}()$.

The second step performs the translation of a given configuration R and uses as a parameter a set of sets of keys S as above, which is initialised as $S = \mathtt{col}(R)$. Let us denote with $\mathcal{M}$ the set of all $\rho\pi$ [26] configurations in normal form and with $\mathcal{R}$ the set of all configurations obtained by applying our approach to the HOπ-calculus. The encoding function $(\!|\cdot|\!) : \mathcal{R} \to \mathcal{M}$, is defined as:

$$
\begin{aligned}
(\!|R|\!) &= \nu K\,(\!|R|\!)_{\mathtt{col}(R)} \qquad \text{where } K = (\bigcup_{\tilde{h}_k \in \mathtt{col}(R)} \tilde{h} \cup \{k\}) \cup \bigcup_{\{h\} \in \mathtt{col}(R)} \{h\}\\
(\!|\nu a\,R|\!)_S &= \nu a\,(\!|R|\!)_S\\
(\!|R_1 \mid R_2|\!)_S &= (\!|R_1|\!)_S \mid (\!|R_2|\!)_S\\
(\!|[R\,;C]|\!)_S &= [(\!|R|\!)_S\,;(\!|C|\!)_S]\\
(\!|h : P|\!)_S &= \langle h, \tilde{h}\rangle \cdot k : P \qquad \text{if} \quad h \in \tilde{h} \text{ for some } \tilde{h}_k \in S\\
(\!|h : P|\!)_S &= h : P \qquad \text{if} \quad h \notin \tilde{h} \text{ for all } \tilde{h}_k \in S\\
(\!|C|\!)_S &= k \qquad \text{if } \mathtt{key}(C)_k \in S\\
(\!|C|\!)_S &= h \qquad \text{if } \mathtt{key}(C) = \{h\}
\end{aligned}
$$

Let us comment on it. The first rule computes the parameter $\mathtt{col}(R)$ containing information on keys, to be used in the rest of the translation, and creates restrictions for all the keys occurring in it. The other rules just propagate the set S, till one of the last 4 rules applies. The first two deal with keys labelling processes: it the key belongs to a non-singleton set, then it is replaced by a complex tag, otherwise it is left unchanged. The two last rules remove the context C in the memory, which is not needed in $\rho\pi$, replacing it with a key. If C contains only one key, this is the key used. If it contains more than one key instead the fresh key k generated for the set of keys is used. The keys in the set will become complex tags, carrying k so to make the connection between the memory and all the processes created by the corresponding transition.

Let us show a simple example to clarify how the translation works.

▶ **Example 33.** Let us consider the system produced by the sample transition in Section 4.1:

$$R' = j_1 : P_1 \mid j_2 : P_2 \mid [R\,; j_1 : \bullet_1 \mid j_2 : \bullet_2]$$

$$(\textsc{Seq})\ \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\langle p, \theta, e\rangle \hookrightarrow \langle p, \theta', e'\rangle} \qquad (\textsc{Rec})\ \frac{\theta, e \xrightarrow{\mathtt{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \text{and} \quad \mathtt{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{(p', p, v) \mid \langle p, \theta, e\rangle \hookrightarrow \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle}$$

$$(\textsc{Send})\ \frac{\theta, e \xrightarrow{\mathtt{send}(p', v)} \theta', e'}{\langle p, \theta, e\rangle \hookrightarrow \langle p, \theta', e'\rangle \mid (p, p', v)} \qquad (\textsc{Self})\ \frac{\theta, e \xrightarrow{\mathtt{self}(\kappa)} \theta', e'}{\langle p, \theta, e\rangle \hookrightarrow \langle p, \theta', e'\{\kappa \mapsto p\}\rangle}$$

$$(\textsc{Spawn})\ \frac{\theta, e \xrightarrow{\mathtt{spawn}(\kappa, f/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\langle p, \theta, e\rangle \hookrightarrow \langle p, \theta', e'\{\kappa \mapsto p'\}\rangle \mid \langle p', id, \mathtt{apply}\ f/n\ (\overline{v_n})\rangle} \qquad (\textsc{Par})\ \frac{E \hookrightarrow E' \quad \mathtt{pid}(E') \cap \mathtt{pid}(E_1) = \emptyset}{E \mid E_1 \hookrightarrow E' \mid E_1}$$

Figure 7 System rules of standard Core Erlang.

We have $\mathtt{col}(R') = \{\{j_1, j_2\}_k\}$ where k is a fresh key. We now have:

$$\begin{aligned}
(\!|R'|\!) =& \nu j_1, j_2, k\, (\!|R'|\!)_{\mathtt{col}(R')} = \\
=& \nu j_1, j_2, k\, (\!|j_1 : P_1|\!)_{\mathtt{col}(R')} \mid (\!|j_2 : P_2|\!)_{\mathtt{col}(R')} \mid (\!|[R\,; j_1 : \bullet_1 \mid j_2 : \bullet_2]|\!)_{\mathtt{col}(R')} \\
=& \nu j_1, j_2, k\, \langle j_1, \{j_1, j_2\}\rangle \cdot k : P_1 \mid \langle j_2, \{j_1, j_2\}\rangle \cdot k : P_2 \mid \\
& \qquad [(\!|R|\!)_{\mathtt{col}(R')}\,; (\!|j_1 : \bullet_1 \mid j_2 : \bullet_2|\!)_{\mathtt{col}(R')}] \\
=& \nu j_1, j_2, k\, \langle j_1, \{j_1, j_2\}\rangle \cdot k : P_1 \mid \langle j_2, \{j_1, j_2\}\rangle \cdot k : P_2 \mid [R\,; k]
\end{aligned}$$

where R is unchanged since it only contains keys not occurring in $\mathtt{col}(R')$.

Classic and reversible semantics for Core Erlang

This section recalls a reduction semantics for Core Erlang [30] and presents forward and backward rules of the reversible semantics for Core Erlang obtained using approach (Section 4.2).

In Figure 7 we give the reduction semantics for Core Erlang. Rule (Spawn) adds a new process with a fresh pid p', initialised with an empty environment id, into the system. The function $\mathtt{pid}(\cdot)$ used in rule (Par) extracts the set of pids of processes in a given system. It is used to ensure that the pid of the newly spawned process is fresh. We refer to [30] for a detailed description of the rules.

The forward rules of the reversible semantics are given in Figure 8. Rule (F-Par) allows configurations to execute as part of a larger configuration with the additional condition that keys generated by the execution are not part of the parallel configuration.

Backward rules of the reversible semantics are given in Figure 9. Notably, to capture exactly the instances produced by our approach some side conditions would be needed. E.g., in rule B-Par one would need condition $\mathtt{pid}(R') \cap \mathtt{pid}(R'') = \emptyset$. However, such conditions are always satisfied in reachable configurations.

Reversible link semantics for Erlang

In this section we give the additional rules of the reversible link semantics for Erlang (Section 4.3), namely system rule (F-Nrm) as well as rules describing the evaluation of functions `spawn_link()` and `process_flag()`. In rule (Flag), f is a Boolean value.

$$(\text{F-Seq})\ \frac{\theta, e \xrightarrow{\tau} \theta', e' \quad k_1 \text{ is a fresh key}}{k : \langle p, \theta, e\rangle \twoheadrightarrow k_1 : \langle p, \theta', e'\rangle \mid [k : \langle p, \theta, e\rangle\ ; k_1 : \bullet_1]}$$

$$(\text{F-Send})\ \frac{\theta, e \xrightarrow{\mathtt{send}(p',v)} \theta', e' \quad k_1, k_2 \text{ are fresh keys}}{k : \langle p, \theta, e\rangle \twoheadrightarrow k_1 : \langle p, \theta', e'\rangle \mid k_2 : (p, p', v) \mid [k : \langle p, \theta, e\rangle\ ; k_1 : \bullet_1 \mid k_2 : \bullet_2]}$$

$$(\text{F-Rec})\ \frac{\theta, e \xrightarrow{\mathtt{rec}(\kappa,\overline{cl_n})} \theta', e' \quad \text{and} \quad \mathtt{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i) \quad k_1 \text{ is a fresh key}}{k_2 : (p', p, v) \mid k : \langle p, \theta, e\rangle \twoheadrightarrow k_1 : \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid [k_2 : (p', p, v) \mid k : \langle p, \theta, e\rangle\ ; k_1 : \bullet_1]}$$

$$(\text{F-Spawn})\ \frac{\theta, e \xrightarrow{\mathtt{spawn}(\kappa, f/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid} \quad \text{and} \quad k_1, k_2 \text{ are fresh keys}}{k : \langle p, \theta, e\rangle \twoheadrightarrow k_1 : \langle p, \theta', e'\{\kappa \mapsto p'\}\rangle \mid k_2 : \langle p', id, \mathtt{apply}\ f/n\ (\overline{v_n})\rangle \mid [k : \langle p, \theta, e\rangle\ ; k_1 : \bullet_1 \mid k_2 : \bullet_2]}$$

$$(\text{F-Self})\ \frac{\theta, e \xrightarrow{\mathtt{self}(\kappa)} \theta', e' \quad k_1 \text{ is a fresh key}}{k : \langle p, \theta, e\rangle \twoheadrightarrow k_1 : \langle p, \theta', e'\{\kappa \mapsto p\}\rangle \mid [k : \langle p, \theta, e\rangle\ ; k_1 : \bullet_1]}$$

$$(\text{F-Par})\ \frac{R \twoheadrightarrow R' \quad \mathtt{pid}(R') \cap \mathtt{pid}(R'') = \emptyset \quad \text{and} \quad (\mathtt{key}(R') \setminus \mathtt{key}(R)) \cap \mathtt{key}(R'') = \emptyset}{R \mid R'' \twoheadrightarrow R' \mid R''}$$

Figure 8 Forward rules of the reversible semantics for Erlang.

$$(\text{B-Seq})\ k_1 : \langle p, \theta', e'\rangle \mid [k : \langle p, \theta, e\rangle\ ; k_1 : \bullet_1] \rightsquigarrow k : \langle p, \theta, e\rangle$$

$$(\text{B-Send})\ k_1 : \langle p, \theta', e'\rangle \mid k_2 : (p, p', v) \mid [k : \langle p, \theta, e\rangle\ ; k_1 : \bullet_1 \mid k_2 : \bullet_2] \rightsquigarrow k : \langle p, \theta, e\rangle$$

$$(\text{B-Rec})\ k_1 : \langle p, \theta', e'\rangle \mid [k_2 : (p', p, v) \mid k : \langle p, \theta, e\rangle\ ; k_1 : \bullet_1] \rightsquigarrow k_2 : (p', p, v) \mid k : \langle p, \theta, e\rangle$$

$$(\text{B-Spawn})\ k_1 : \langle p, \theta', e'\rangle \mid k_2 : \langle p', id, e''\rangle \mid [k : \langle p, \theta, e\rangle\ ; k_1 : \bullet_1 \mid k_2 : \bullet_2] \rightsquigarrow k : \langle p, \theta, e\rangle$$

$$(\text{B-Self})\ k_1 : \langle p, \theta', e'\rangle \mid [k : \langle p, \theta, e\rangle\ ; k_1 : \bullet_1] \rightsquigarrow k : \langle p, \theta, e\rangle \qquad (\text{B-Par})\ \frac{R' \rightsquigarrow R}{R' \mid R'' \rightsquigarrow R \mid R''}$$

Figure 9 Backward rules of the reversible semantics for Erlang.

$$(\text{F-Nrm})\ \frac{l = \{p_1, \ldots, p_m\} \quad 1 \le i \le n \Rightarrow f_i = \mathtt{true}\ \wedge\ n+1 \le i \le m \Rightarrow f_i = \mathtt{false} \quad h, h_i, j_i \text{ are fresh keys}}{\begin{array}{c} k : \langle p, \theta, v, l, f\rangle \mid \prod_{1 \le i \le m} k_i : \langle p_i, \theta_i, e_i, l_i, f_i\rangle \twoheadrightarrow h : \langle p, \theta, v, \emptyset, f\rangle \mid \\ \prod_{1 \le i \le n} h_i : \langle p_i, \theta_i, e_i, l_i \setminus \{p\}, f_i\rangle \mid \prod_{1 \le i \le n} j_i : (p, p_i, \{\text{'}\mathtt{EXIT}\text{'}, p, \mathtt{normal}\}) \mid \prod_{n+1 \le i \le m} h_i : \langle p_i, \theta_i, e_i, l_i \setminus \{p\}, f_i\rangle \mid \\ [k : \langle p, \theta, v, l, f\rangle \mid \prod_{1 \le i \le m} k_i : \langle p_i, \theta_i, e_i, l_i, f_i\rangle\ ; h : \bullet_h \mid \prod_{1 \le i \le m} h_i : \bullet_{h_i} \mid \prod_{1 \le i \le n} j_i : \bullet_{j_i}] \end{array}}$$

$$(\text{Spawn\_Link1})\ \frac{\theta, e_i \xrightarrow{l} \theta', e_i' \quad i \in \{1, \ldots, n\}}{\theta, \mathtt{spawn\_link}(a/n, [\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}]) \xrightarrow{l} \theta', \mathtt{spawn\_link}(a/n, [\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}])}$$

$$(\text{Spawn\_Link2})\ \theta, \mathtt{spawn\_link}(a/n, [\overline{v_n}]) \xrightarrow{\mathtt{spawn\_link}(\kappa, a/n, [\overline{v_n}])} \theta, \kappa$$

$$(\text{Flag})\ \theta, \mathtt{process\_flag}(trap\_exit, f) \xrightarrow{\mathtt{process\_flag}(\kappa, trap\_exit, f)} \theta, \kappa$$