Generation of a Reversible Semantics for Erlang in Maude

Generation of a Reversible Semantics for Erlang in Maude*

Giovanni Fabbretti[1][0000−0003−3002−0697], Ivan Lanese[2][0000−0003−2527−9995], and Jean-Bernard Stefani[1][0000−0003−1373−7602]

[1] Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
[2] Focus Team, Univ. of Bologna, INRIA, 40137 Bologna, Italy

Abstract. In recent years, reversibility in concurrent settings has attracted interest thanks to its diverse applications in areas such as error recovery, debugging, and biological modeling. Also, it has been studied in many formalisms, including Petri nets, process algebras, and programming languages like Erlang. However, most attempts made so far suffer from the same limitation: they define the reversible semantics in an ad-hoc fashion. To address this limit, Lanese et al. have recently proposed a novel general method to derive a concurrent reversible semantics from a non-reversible one. However, in most interesting instances the method relies on infinite sets of reductions, making doubtful its practical applicability. We bridge the gap between theory and practice by implementing the above method in Maude. The key insight is that infinite sets of reductions can be captured by a small number of schemas in many relevant cases. This happens indeed for our application: the functional and concurrent fragment of Erlang. We extend the framework with a general rollback operator, allowing one to undo an action far in the past, including all and only its consequences. We can thus use our tool, e.g., as an oracle against which to test the reversible debugger CauDEr for Erlang, or as an executable specification for new reversible debuggers.

1 Introduction

Reversible computing studies computational models which have both (standard) forward and backward notions of execution. Reversibility has attracted interest thanks to its diverse applications in areas such as debugging [7,5,19,9], robotics [21], biological modeling [4], and error-recovery [26]. In sequential systems reversibility is well understood: intuitively it corresponds to undo actions in reverse order of execution. In concurrent settings, more care is needed. In 2004, Danos and Krivine proposed the notion of *causal-consistent reversibility* [3], tailored for concurrent systems. In a concurrent execution, to undo an action causal consistency only requires to undo its causal consequences first. Actions

* The work has been partially supported by French ANR project DCore ANR-18-CE25-0007. We thank the anonymous referees for their helpful comments and suggestions. The second author also thanks INdAM-GNCS Project CUP_E55F22000270001 "Proprietà qualitative e quantitative di sistemi reversibili"

which have been temporally interleaved with such consequences, but are causally independent, can be left untouched. Thus, causal consistency undoes events only if strictly necessary, which is useful to explore concurrent programs that can be prone to state explosion. Causal-consistent reversibility has been studied in several formalisms such as process calculi [3,28,2,17], Petri nets [27,23], and the Erlang programming language [5,19,9,12]. It also led to interesting practical applications, the most prominent example being as a debugging technique as proposed in [7] and then implemented in the CauDEr debugger for Erlang [5,19,9].

Most of the reversible semantics above have been devised ad-hoc for a specific formalism. The process is usually composed of three phases: i) definition of causal dependencies between events; ii) extension of the non-reversible semantics so that enough information is kept while going forward; iii) creation of a backward semantics that allows one to undo actions in a causal-consistent manner and restore past states. Performing this process manually is time-consuming, error-prone and lacks generality.

Recently Lanese and Medić proposed a general method to automate the production of reversible semantics [14]. The method generalizes the ad-hoc approaches above and works as follows. First, causal dependencies are defined in terms of resources *consumed* and *produced*. Without focusing on the details, let us consider the following Erlang example.

$$\langle p_1, \theta, p_2 \mathbin{!} hello, me\rangle \rightarrow \langle p_1, \theta, hello, me\rangle \mid \langle p_1, p_2, hello\rangle \tag{1}$$

On the left, a process p_1 is ready to send a message *hello* (! denotes message send). When the reduction is executed the process is consumed to produce the message $\langle p_1, p_2, hello\rangle$ and the evolution of the process itself after the send. We say that the reduction consumes the process and produces the continuation and the message. Then, the non-reversible semantics taken in input is extended so that each entity is tagged with a *unique key*, and *memories* are produced each time a forward step is performed. Memories are the extra pieces of information required to restore past states of the system and together with keys they also keep track of the causal dependencies. Finally, a causal-consistent backward semantics, symmetric to the forward one, is generated.

Contributions The general method in [14] was only described theoretically. It takes in input a semantics described as a, possibly infinite, set of ground rules, making it not immediately clear that an implementation could exist. In this paper we provide such an implementation in Maude, by using schemas to capture (possibly infinite) sets of ground rules. As a case study, we use our tool to derive a causal-consistent reversible semantics for the functional and concurrent fragment of the Erlang programming language, which matches the one previously produced by hand [9].

Finally, we extend Lanese et al. approach by defining a causal-consistent rollback operator, allowing one to undo a past action including all and only its consequences, on top of the reversible semantics. Rollback is a key primitive for a concurrent causal-consistent debugger as described in [7]. In the literature,

examples of causal-consistent rollback operators abound [5,19,9]. Nonetheless these operators were always designed in an ad-hoc fashion, suffering from the same limits as the ad-hoc reversible semantics. In contrast, our definition is able to cope with all the reversible semantics we produce, thanks to their uniformity. This is beneficial and desirable, as one can change or update the underlying semantics without the need to redefine the rollback operator.

To sum up, the main contributions of this work are:

- a novel formalization of Erlang in Maude (Sec. 3);
- a tool to derive a causal-consistent reversible semantics from a non-reversible one (Sec. 4) together with a proof of correctness of the approach (Sec. 5);
- a general definition of a causal-consistent rollback operator, built on top of the reversible semantics (Sec. 6).

All the code discussed in this paper is publicly available at [29].

2 Background

2.1 The Erlang language

Erlang is a functional and concurrent programming language, it is widely used and appreciated because it is easy to learn, provides useful abstractions for concurrent and distributed programming, and because of its support for highly-available systems. Erlang implements the actor model [10], a concurrency model based on message passing. In the actor model, each process is an actor that can interact with other actors only through the exchange of messages, no memory is shared. Actors are identified by a unique pid (process identifier) and have a queue of messages which have arrived but have not yet been processed. An actor evaluates an expression, and has an environment to store variable bindings. Due to space constraints, here we only briefly describe the main concurrent primitives of Erlang, send, receive, spawn, and self, more details on the language can be found in the technical report [6].

The send primitive is written as e_1 ! e_2, where e_1 must evaluate to the pid of the receiver process and e_2 must evaluate to the payload, say v, of the message. The expression itself evaluates to v and, as a side-effect, the message is sent.

The receive $pat_1 \rightarrow exprs_1; \ldots; pat_n \rightarrow exprs_n$ end construct explores the queue of messages looking for one mathcing one of the patterns, say pat_i. If found, the corresponding branch $exprs_i$ is executed.

The spawn primitive creates a new process; it takes as argument the function f that the new process will execute, together with the parameters for f - if any. The spawn returns the (fresh) pid of the newly created process and, as a side effect, the new process is created.

Finally, function self returns the pid of the process who invoked it.

```
fmod BOOL is
  sort Bool .                                        var A : Bool .
  op true : -> Bool [ctor] .                         eq true and A = A .
  op false : -> Bool [ctor] .                        eq false and A = false .
  op _and_ : Bool Bool -> Bool [assoc ..] .          eq A and A = A .
endfm
```

Fig. 1: Maude module for Booleans[3]

2.2 Maude

Maude [22] is a programming language that efficiently implements rewriting logic [24]. Formally, a rewriting theory is a tuple (Σ, E, R), where Σ represents a collection of typed operators, E a set of equations $t = t'$, and R a set of semantic rules $t \to t'$. In both cases, t, t' are terms built from the operators in Σ.

The equational side of rewriting logic is well-suited to define the deterministic part of the model, where we define equivalence classes over terms. Equations can also be conditional, and conditions can be either the membership of the term to some kind or other equations.

Rewriting rules define the concurrent (non-deterministic) part of the programming language semantics. The set of rules R specifies how to rewrite a (parameterized) term t to another term t'. Rewriting rules can be conditional too, and conditions can be equations, as well as other rewriting conditions.

In other words the equational theory specifies which terms define the same states of a system, only using different syntactical elements, while the rewriting rules define how the system can evolve and transit from one state to another.

Let us now consider the module in Fig. 1, a sample Maude module that implements Booleans together with the `and` operation.

First, the sort `Bool` is declared. Then, the values `true` and `false` are declared as two constant operators of sort `Bool`. Successively, the `and` operation is defined as a function that takes in input some `Bool`s and produces a `Bool` as a result. Finally, the semantics of `and` is given by the equational theory defined on the right of the module. Equations are used from left to right to normalize terms. For instance, the first equation, `eq true and A = A`, is used to evaluate the `and` operator when the first argument has been normalized to `true`. For simplicity, this example does not include rewriting rules, memberships nor conditional equations.

As an additional example, we show a rewriting rule generating the Erlang reduction (1) from the Introduction:

```
< 1 | exp: 2 ! 'hello', env: {}, me: _ > =>
    < 1 | exp : 'hello', env: {}, me: _ > ||
    < sender: 1, receiver: 2, payload: 'hello' >
```

[3] Due to space reason we represented the module on two columns, usually Maude modules are single-columned.

Labels `exp` (for the expression under evaluation), `env` (for the environment) and `me` (for the module environment, containing function definitions), and similarly for messages, give names to fields. Also, the first argument in each process is the pid (pids are integers in our implementation), the special notation highlights that it can be used as identifier for the tuple. Character `_` means that the actual value is not shown.

We will define the generation of the reversible semantics as a program that, given the modules of the non-reversible semantics, produces new modules, which define the reversible semantics.

2.3 Derivation of the Reversible Semantics

The rest of this section summarizes the methodology to automatically derive a causal-consistent reversible semantics starting from a non-reversible one [14] that we will use as starting point. The approach requires that the latter is modeled as a reduction semantics that satisfies some syntactic conditions.

Format of the Input Reduction Semantics We now describe the shape that the reduction semantics taken in input must have.

The syntax must be divided in two levels: a lower level of entities on which there are no restrictions, and an upper level of systems of the following form:

$$S ::= P \mid op_n(S_1, \ldots, S_n) \mid \mathbf{0}$$

where $\mathbf{0}$ is the empty system, P any entity of the lower level and $op_n(S_1, \ldots, S_n)$ any n-ary operator to compose entities. An entity of the lower level could be, for example, a process of the system or a message traveling the network. Among the operators we always assume a binary parallel operator $\mid$.

The rules defining the operational semantics must fit the format in Fig. 2, where $\rightarrowtail$ denotes the relation defining the reduction semantics taken in input. The format contains rules to: i) allow entities to interact with each other (S-Act); ii) exploit a structural congruence (Eqv); iii) allow single entities to execute inside a context (S-Opn); iv) execute two systems in parallel (Par). While (Eqv) and (Par) are rules that must belong to the semantics, (S-Act) and (S-Opn) are schemas, and the semantics may contain any number of instances of them. In the schema (S-Act), the term $T[Q_1, \ldots, Q_m]$ denotes a generic operator of the reduction semantics taken in input. Actually, rule (Par) is an instance of schema (S-Opn), highlighting that such an instance is required. Also, reduction (1) from the Introduction is an instance of schema (S-Act). Moreover, notice that a notion of structural congruence on systems is assumed. We refer to [14] for more details on the definition of structural congruence. This is of limited relevance here, since the only structural congruence needed for Erlang is that parallel composition forms a commutative monoid, which translates to the same property in the reversible semantics.

$$(\text{S-Act})\ \frac{}{P_1 \mid \ldots \mid P_n \rightarrowtail T[Q_1, \ldots, Q_m]} \qquad (\text{Eqv})\ \frac{S \equiv S' \quad S \rightarrowtail S_1 \quad S_1 \equiv S_1'}{S' \rightarrowtail S_1'}$$

$$(\text{S-Opn})\ \frac{S_i \rightarrowtail S_i'}{op_n(S_0, \ldots, S_i, \ldots, S_n) \rightarrowtail op_n(S_0, \ldots, S_i', \ldots, S_n)} \qquad (\text{Par})\ \frac{S \rightarrowtail S'}{S \mid S_1 \rightarrowtail S' \mid S_1}$$

Fig. 2: Required structure of the semantics in input; S- rules are schemas

Methodology To obtain a forward reversible semantics, we need to track enough history and causality information to allow one to define a backward semantics exploiting it. First, the syntax of systems is updated as follows:

$$\begin{aligned} R &::= k : P \mid op_n(R_1, \ldots, R_n) \mid \mathbf{0} \mid [R\,; C] \\ C &::= T[k_1 : \bullet_1, \ldots, k_m : \bullet_m] \end{aligned}$$

Two modifications have been done. First, each entity of the system is tagged with a key k. Keys are used to distinguish identical entities with a different history. Second, the syntax is updated with another production: memories. Memories have the shape $\mu = [R; C]$, where R is the configuration of the system that gave rise to a forward step and C is a context describing the structure of the system resulting from the forward step. C acts as a link between R and the actual final configuration. In other words, memories link different states of the entities and keep track of past states of the system so that they can be restored.

Then, the forward reversible semantics is defined by decorating the rules of the non-reversible reduction semantics as depicted in Fig. 3, where $\twoheadrightarrow$ is the relation defining the forward reversible semantics. Now each time a forward step is performed each resulting entity is tagged with a fresh key, and a memory, connecting the old configuration with the new one, is produced. E.g., the forward rule corresponding to reduction (1) from the Introduction is:

$$\begin{aligned} k : \langle p_1, \theta, p_2 \,!\, hello, me\rangle \twoheadrightarrow k_1 : \langle p_1, \theta, hello, me\rangle \mid k_2 : \langle p_1, p_2, hello\rangle \mid \\ [k : \langle p_1, \theta, p_2 \,!\, hello, me\rangle\,; k_1 : \bullet_1 \mid k_2 : \bullet_2] \end{aligned}$$

Notice that the approach allows one to manage different rules since the transformation is defined in terms of the schema they must fit.

The backward rules, depicted in Fig. 4, where $\rightsquigarrow$ is the relation defining the backward reversible semantics, are symmetric to the forward ones: if a memory $\mu = [R\,; C]$ and the entities tagged with the keys in C are both available then a backward step can be performed and the old configuration R can be restored. E.g., the backward rule undoing the reduction (1) from the Introduction is:

$$\begin{aligned} k_1 : \langle p_1, \theta, hello, me\rangle \mid k_2 : \langle p_1, p_2, hello\rangle \mid \\ [k : \langle p_1, \theta, p_2 \,!\, hello, me\rangle\,; k_1 : \bullet_1 \mid k_2 : \bullet_2] \rightsquigarrow k : \langle p_1, \theta, p_2 \,!\, hello, me\rangle \end{aligned}$$

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

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

Fig. 3: Forward rules of the uncontrolled reversible semantics

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

$$\text{(B-S-Opn)}\ \frac{R_i' \rightsquigarrow R_i}{op_n(R_0, \ldots, R_i', \ldots, R_n) \rightsquigarrow op_n(R_0, \ldots, R_i, \ldots, R_n)}$$

Fig. 4: Backward rules of the uncontrolled reversible semantics

The reversible semantics produced by this approach captures causal dependencies in terms of resources produced and consumed, since, thanks to the memory, a causal link is created each time some entities are rewritten. We refer to [14] for the formal proof of the causal-consistency and of other relevant properties of the reversible semantics. We also remark that the semantics produced is uncontrolled [16], i.e., if multiple (forward and/or backward) steps are enabled at the same time there is no policy on which one to choose. Both in Fig. 3 and in Fig. 4 we omitted the rule for structural congruence as it is similar to the one in Fig. 2.

3 Formalizing Erlang in Maude

In this section we present the formalization of the semantics of the functional and concurrent fragment of Erlang in Maude. We mostly follow the semantics defined in [9]. Technically, we used as starting point the formalization of Core Erlang [1] in Maude presented in [25], which was aimed at model checking. While our formalization is quite different from theirs (e.g., we formalize a fragment of Erlang instead of one of Core Erlang), we were still able to re-use some of their modules, like the parsing module, and some of their ideas which greatly simplified the formalization task.

As in [9], our semantics of Erlang has two layers: one for expressions and one for systems. This division is quite convenient for the formalization in Maude, as we can formalize the expression level as an equational theory and then use rewriting rules to describe the system level.

The system level comprises a rewriting rule for each concurrent construct of the language and a few rules τ for sequential operations. We could define the

sequential operations as an equational theory also at the system level, however equations are applied in a fixed order, hence only one possible interleaving of sequential steps would have been considered. For rewriting rules instead all possible orders can be considered, thus enabling all possible interleavings. Notably, also a different semantics where sequential steps are defined as equations could be made reversible using the approach we describe in the next section.

Before presenting the rewriting logic, let us discuss the entities that compose an Erlang system. Processes are defined as tuples of the form:

$$\langle p, \theta, e, me \rangle$$

where p is the process pid, θ is the environment binding variables to values[4], e is the expression currently under evaluation and me is the module environment, which contains the definitions of the functions declared in the module, that p can invoke or spawn. Messages instead are defined as tuples of the form:

$$\langle p, p', v \rangle$$

where p is the pid of the sender, p' is the pid of the receiver and v is the payload. In this work, processes and messages are entities in the lower level of the semantics, denoted as P in Sec. 2.3.

A system is composed of messages and processes, using the parallel operator.

Now, let us analyze in detail the shape of the corresponding rewriting logic by first analyzing the equational theory for expressions.

3.1 Equational Theory

The theory is defined as a set of conditional (i.e., with an if clause) and unconditional equations, represented as follow

eq : [*equation-name*] $\langle l, \theta, e \rangle = \langle l', \theta', e' \rangle$	ceq : [*conditional-equation-name*] $\langle l, \theta, e \rangle = \langle l'', \theta'', e'' \rangle$ if $\langle l', \theta', e' \rangle := op(l, \theta, e) \ \wedge \langle l'', \theta'', e'' \rangle := \langle l', \theta', e' \rangle$

In the equations, to evaluate an expression e we also need two additional items: an environment θ and a label l. The environment binds each variable to its value, if any. The label communicates both i) the kind of side effect performed by the expression, if any; and ii) information on the details of the side effect back and forth between the expression level and the system level. An example of this mechanism is presented below.

Example 1 (Equation for self). The unconditional equation below describes the behavior of `self` at the expression level.

```
eq [self] : < self(pid(N)), ENV, atom("self")() > = < tau, ENV, int(N) > .
```

[4] In truth, θ is a stack of environments, such design choice is discussed in Sec. 3.2.

It reads roughly as follows: if the system level asks to check whether a `self` can be performed, communicating that the pid of the current process is `N` (via `self(pid(N))`) and the expression is actually a self (`atom("self")()`) then the expression reduces to the pid (`int(N)`) and the label becomes `tau`, denoting successful evaluation of a sequential step. ◇

Conditional equations can: either define a single step that requires some side condition (e.g., binding a variable to its value), or perform some intermediate operation (e.g., selecting an inner expression to evaluate) and then use recursively other equations (with the clause $\langle l'', \theta'', \ e'' \rangle := \langle l', \theta', e' \rangle$) to reach a canonical form. Examples of conditional equations can be found in the technical report [6].

3.2 Expression Management

One of the difficulties of formalizing Erlang lies in the manipulation of expressions. In fact, a naive management could produce unwanted results or illegal expressions.

Consider the invocation below of function

$$pow\_and\_sub(N, M) \ \rightarrow \ Z = N * N, Z - M$$

which computes the difference between the power of N and M.

$$X = pow\_and\_sub(N, M) \quad \rightarrow \quad X = Z = N * N, Z - M. \tag{2}$$

By naively replacing the function with its body, we get a syntactically correct Erlang expression, but it would not have the desired effect, as the variable X would assume the value $N * N$ instead of $Z - M$, as desired.

Similarly, constructs that produce a sequence of expressions, like case, may also produce illegal terms. Consider, e.g., the following Erlang expression:

$$\mathsf{case}\ pow\_and\_sub(N, M)\ \mathsf{of} \ldots \quad \rightarrow \quad \mathsf{case}\ Z = N * N, Z - M\ \mathsf{of} \ldots \tag{3}$$

In this case the obtained expression is illegal, as case expects a single expression and not a sequence, and would be refused by an Erlang compiler.

The solution that we adopt to solve both problems consists in wrapping the produced sequence of expressions with the construct `begin_end` (the Erlang equivalent for parentheses), which turns a sequence of expressions into a single expression.

For instance, in (2) the produced expression would be

$$X = \mathsf{begin}\ Z = N * N, Z - M\ \mathsf{end}.$$

and in this case X is correctly bound to the result of $Z - M$. This solution indeed produces the desired effect also in a real Erlang environment.

For this reason, θ, within a process tuple, is a stack of environments. Each time that a sequence of expressions is wrapped a new environment with the appropriate bindings (e.g., the function parameters) is pushed on θ. Then, each time the sequence of expression is fully evaluated, i.e., the expression looks like $\mathsf{begin}\ v\ \mathsf{end}$, then v replaces the expression and an environment is popped from θ.

```
crl [sys-send] :
    < P | exp: EXSEQ, env-stack: ENV, ASET > =>
    < P | exp: EXSEQ', env-stack: ENV', ASET > ||
    < sender: P, receiver: DEST, payload: GVALUE >
    if < DEST ! GVALUE, ENV', EXSEQ' > := < req-gen, ENV, EXSEQ > .
```

Fig. 5: System rule send

3.3 Rewriting Rules

Let us now focus on rewriting rules, which have the following general shape

$$\begin{aligned}&\mathsf{crl}: \ [\textit{conditional-rule-name}]\\ &\quad \langle p, \theta, e, me\rangle \mid E \Rightarrow \langle p, \theta', e', me\rangle \mid op(l', \langle p, \theta, e, me\rangle, E)\\ &\quad \mathsf{if}\ \langle l', \theta', e'\rangle := \langle l, \theta, e\rangle\end{aligned}$$

Here, E captures other entities of the system, if any, that may have an impact on the reduction, in particular a message that may be received. Rewriting rules are always conditional, as we always rely on the expression semantics to understand which action the selected process is ready to perform. Finally, we use *op* to apply side effects to E, determined by the label l' produced by the expression level. Example 2 below discusses the rewriting rule for send, additional examples can be found in the technical report [6].

Example 2. The rule in Fig. 5 is used to send a message. The `if` clause of the rule uses the equational theory to check if the current expression, `EXSEQ`, can perform a send of `GVALUE` to `DEST`. This exemplifies how the labels `req-gen` (a generic request about which step can be taken, more complex requests are used, e.g., for `self`, see Example 1) and `DEST ! GVALUE` serve to pass information between the system and the expression level. Using this information, side effects (in this case the send of a message) are performed at the system level. If the send can be performed, then the process evolves to evaluate the new expression `EXSEQ'` in the new environment `ENV'`, and the new message is added to the system. Here, `ASET` includes other elements of the process which are not relevant (currently, only the module environment). W.r.t. the general schema described above, here E on the left-hand side is empty, and on the right-hand side *op* will add the message to E.

Note that the rewriting rule in Sec. 2.2 is an instance of the one above. ◇

4 Generating the Reversible Semantics

We choose Maude to define the generation of the reversible semantics for two main reasons. First, Maude is well-suited to define program transformations thanks to its META-LEVEL module, which contains facilities to meta-represent a module and to manipulate it. Second, since we defined Erlang's semantics in Maude, we do not need to define a parser for it as it can be easily loaded and meta-represented by taking advantage of Maude's facilities.

```
mod SYSTEM is
...
  sort Sys .                    op #empty-system : -> Sys [ctor] .
  subsort Entity < Sys .        op _||_ : Sys Sys -> Sys [ctor ... ] .
endm
```

Fig. 6: Extract of the system module for Erlang.

4.1 Format of the Non-Reversible Semantics

As in [14], the input semantics must follow a given format so that the approach can be applied. Let us describe such format. First, the formalization must include a module named `SYSTEM` which defines the system level. As an example, Fig. 6 depicts the system module for the Erlang language. We omit elements that are not interesting in this context (namely the import of other modules).

The module defines the operators of the system level, as discussed in Sec. 2.3. For Erlang, we just have parallel composition `||` and the empty system.

All the operators in the module `SYSTEM` must take in input and produce elements of sort `Sys`. The subsort relation `Entity < Sys` must be declared as well, to specify that entities of the lower level are systems. To this end, sorts of the lower level (in Erlang, messages and processes) must be subsorts of `Entity`.

Rules of the rewriting theory that define the single steps of the reduction semantics (like in Fig. 5) must be defined under the module `TRANSITIONS`.

4.2 Transformation to the Syntax

We describe here how to transform a non-reversible syntax as described above into a reversible one, as recalled in Sec. 2.3. Roughly, we add keys and memories.

`Key` is the sort generated by the operator `key_` and `EntityWithKey` is the sort generated by the operator `_*_`, that composes an entity and a key.

To define memories, first we declare a new sort `Context` (which corresponds to C in the reversible syntax presented in Sec. 2.3), together with an operator `@:_` to create a `Context` from a key. Then, memories are added by defining the sort `Memory` and by defining an operator that builds a memory by combining the interacting entities with keys with the final configuration of sort `Context`. E.g., the memory created by the reversible version of the reduction in Sec. 2.2 is:

```
[ < 1 | exp: 2 ! 'hello', ASET> * key(0) ; @: key(0 0) || @: key:(1 0) ]
```

Here, with the variable `ASET`, we hide the process environment and the module environment since they are not interesting. `EntityWithKey`, `Context` and `Memory` are all declared as subsorts of `Sys` so that system operators can be applied to them.

```
crl [label sys-send]:
    < P | ASET, exp: EXSEQ, env-stack: ENV > * key(L)
    => < sender: P, receiver: DEST, payload: GVALUE > * key(0 L) ||
       < P | exp: EXSEQ', env-stack: ENV', ASET > * key(1 L) ||
       [< P | ASET, exp: EXSEQ, env-stack: ENV > *
        key(L) ; @: key(0 L) || @: key(1 L)]
     if < DEST ! GVALUE, ENV', EXSEQ' > := < req-gen, ENV, EXSEQ > .

rl [label sys-send]:
   < sender: P, receiver: DEST,payload: GVALUE > * key(0 L) ||
   < P | exp: EXSEQ', env-stack: ENV', ASET > * key(1 L) ||
   [< P | ASET, exp: EXSEQ, env-stack: ENV > *
    key L ; @: key(0 L) || @: key(1 L)]
   => < P | ASET, exp: EXSEQ, env-stack: ENV > * key L
```

Fig. 7: Reversible rules: send.

4.3 Generating the Reversible Semantics

The transformation to be performed over the rewriting rules is the one described in Sec. 2.3, rephrased in Maude notation. Rules must be extended to deal with entities with key, and each time a forward step is taken the resulting entities must be tagged with fresh keys and the appropriate memory must be created.

The transformation is mostly straightforward, the only tricky part concerns the generation of fresh keys. Indeed, we need a 'distributed' way to compute them, as passing around a key generator would produce spurious causal dependencies. We solved the problem as follows. Keys are lists of integers. Each time we need to produce a fresh key, to tag a new entity on the right-hand side of a rule, we take the key `L` of the first entity on the left-hand side of the rule, and we tag each of the new entities with `L` prefixed with an integer corresponding to the position of the entity on the right-hand side. Furthermore, we create the required memory.

Fig. 7 shows the reversible rules -forward and backward- for the send rule depicted in Fig. 5. In the forward rule, on the left-hand side, the process is initially tagged with a key `key(L)`, then the new entities on the right-hand side are tagged with fresh keys `key(0 L)` and `key(1 L)`, built from `key(L)`. Moreover, the rule also produces a memory binding the old and the new states.

The generation of the backward semantics is easy: a backward rule is obtained from the forward one by swapping the left- with the right-hand side and dropping the conditional branch. Indeed, the latter is not required any more because if the process has performed the forward step, as proved by the existence of a memory for it, then it can always perform the backward one. One has only to check that all the consequences of the action have been already undone. This is ensured by the presence of the entities whose keys are in the context inside the memory.

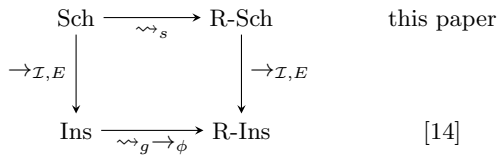

Fig. 8: Schema of the proof of correctness.

5 Correctness

This section is dedicated to prove the correctness of the generated reversible semantics. This requires to close the gap between the format of the rules expected by the general method from [14] and the actual format of the rules provided in input. In fact, the schema of the general method allows for an arbitrary number of rules, potentially infinitely many, describing the system evolution. Obviously, to efficiently describe a system, we cannot exploit infinitely many rules. Thus, in the formalization of the semantics we resorted to schemas, and we used the expression level semantics so to select only a subset of the possible instances.

For example, let us consider the following processes:

$$\langle p, \theta, 2\,!\,'hello', \_\rangle \qquad \langle p', \theta', \mathsf{case}\ 2\,!\,'hello'\ \mathsf{of}\ \ldots, \_\rangle \qquad \langle p'', \theta'', X = 2\,!\,'hello', \_\rangle$$

The three processes above are all ready to perform the same send action, even though they have a different shape, nonetheless thanks to the expression level semantics we are able to formalize their behavior in one single rewriting rule.

However, we need to prove that the instances of the corresponding reversible rules coincide with the set of reversible instances defined by the approach in [14]. That is, we need to show that the diagram in Fig. 8 commutes.

This result is needed also to ensure that our reversible semantics, defined over schemas, by construction enjoys the same desirable properties, e.g., loop lemma, as the reversible semantics defined over ground rules following [14].

Let us begin by discussing the functions on the sides of the square. First, function $\rightsquigarrow_s$ takes in input a set of non-reversible rule schemas of the form $t \rightarrow t'$ if C and generates the corresponding set of reversible (forward and backward) rule schemas. Then, $\rightarrow_{\mathcal{I},E}$ takes in input a set of (reversible or non-reversible) rule schemas and generates all possible instances using substitutions in $\mathcal{I}$, providing all the possible values for variables, and an equational theory E, allowing one to check whether the side condition C is satisfied. The side condition is then dropped. Notably, substitutions $i \in \mathcal{I}$ instantiate also key variables to lists of integers. Function $\rightarrow_{\mathcal{I},E}$ is undefined if there is some $i \in \mathcal{I}$ which is not defined on some variables of the schemas. Also, we expect the substitution to produce well-typed rules (however, we do not discuss typing here). Function $\rightsquigarrow_g$ models the general approach defined in [14]. Intuitively, $\rightsquigarrow_g$ works like $\rightsquigarrow_s$, but it takes only instances of rule schemas. Also, it adds concrete keys instead of key variables. Function $\rightarrow_\phi$ is a function mapping keys in [14], which are taken from an arbitrary set, to keys in our approach, which are lists of integers.

$\mathsf{dep}(S,k)$ when $M := \mathsf{getMem}(S,k) \rightarrow$
  $K_c := \mathsf{contextKeys}(M), R = \{k\}$
  for k_i in K_c
    $R := R \cup \mathsf{dep}(S,k_i)$

$\mathsf{dep}(S,k) \rightarrow$
  $\emptyset$

Fig. 9: Dependencies operator

These functions are formally defined in the technical report [6].
The proof of our main result below can be found in [6] as well.

Theorem 1 (Correctness). *Given functions $\rightsquigarrow_g$, $\rightsquigarrow_s$ and $\rightarrow_{\mathcal{I},E}$ in Fig. 8 such that each $i \in \mathcal{I}$ is injective on key variables, there exists a total function $\rightarrow_\phi$, injective on key variables belonging to the same rule, s.t. the square in Fig. 8 commutes, i.e., $\rightsquigarrow_s \rightarrow_{\mathcal{I},E} = \rightarrow_{\mathcal{I},E} \rightsquigarrow_g \rightarrow_\phi$*

6 Rollback Semantics

In this section we introduce a novel general causal-consistent rollback semantics built on top of the reversible backward semantics. Although general rollback semantics have been discussed in the literature [13], to the best of our knowledge this is the first general causal-consistent rollback semantics which is executable.

Causal-consistent rollback is a key primitive in causal-consistent debugging [7], which undoes a reduction of the system, possibly far in the past, including *all and only* its consequences. Intuitively it performs the smallest amount of backward steps allowing one to undo the selected action [8]. The workflow is the following: the user selects a past reduction by means of one of its unique keys (each key uniquely identifies the reduction consuming it); the set of consequences is computed; all the consequences are undone in a causal-consistent order.

Let us now describe the workflow in more detail. Given a key k in input, we want to undo the action that gave rise to the unique memory whose initial configuration contains k. First, we compute the set of keys $\mathsf{dep}(S,k)$, which contains keys identifying all the consequences of k. The dep operator, depicted in Fig. 9, recursively adds to set R the consequences of the current key, say k_1. A key k_2 is a consequence of k_1 if it occurs in the context part of the memory identified by k_1, and there exists a memory where k_2 occurs in the initial configuration. The code in Fig. 9 relies on two auxiliary functions, $\mathsf{getMem}(S,k)$ and $\mathsf{contextKeys}(M)$. The former, given a system configuration S and a key k, returns the unique memory in S containg k in its initial configuration, if any, while the latter returns the set of keys used in the context part of memory M. Notably, function $\mathsf{getMem}(S,k)$ is used as a guard: if no such memory is found, we apply the base clause on the right of the figure, returning $\emptyset$.

In the second step we need to perform backward steps to undo the computed dependencies. To this end we need to specify which backward ground rule needs to be applied. Fortunately, Maude provides a way to rewrite systems that fits

our needs: the `MetaXApply` function. `MetaXApply` given a theory $\mathcal{R}$, a term t, a rule label l and a substitution σ applies the substitution to rule l (found inside $\mathcal{R}$) and then tries to apply it anywhere possible inside t.[5] Operatively, to undo a transition it suffices to feed to `MetaXApply` the backward rules theory ($\mathcal{R}$), the current system (t), the appropriate backward rule (l), and the selected key that has to be instantiated in the rule ($\sigma = [k/K]$ where k is the concrete key and K the corresponding variable; for simplicity in the implementation we always use the leftmost key in the rule).

If `MetaXApply` can perform a rewrite for some key k then its causal consequences have already been undone. Thus, it is enough to apply `MetaXApply` to all the keys in $\mathtt{dep}(S, k)$, removing a key when the corresponding reduction is performed. When the set is emptied we have reached the desired configuration.

7 Conclusion, Related and Future Work

We defined a new executable semantics of Erlang using Maude. We also implemented a program able to transform a non-reversible semantics into a reversible one, providing an implementation of the general method described in [14]. Making the approach executable posed some challenges. E.g., [14] just declares that keys are generated fresh, while we had to provide a concrete and distributed algorithm to generate keys ensuring their freshness. Finally, we presented a causal-consistent rollback semantics build on top of the backward semantics.

This allows one to use the produced semantics as an oracle against which to test an implementation, while being confident that it correctly captures the formal specification given that it is closer to it. Indeed, we applied our framework to test the reversible debugger CauDEr [18] (forward, backward as well as rollback) on the case study described in [20], thus gaining confidence on the fact that it correctly follows the semantics in [9]. Our experiment showed no discrepancies.

Our semantics of Erlang builds on two starting points, the executable semantics of Core Erlang in Maude described in [25] as well as the reversible semantics for Erlang described in [9]. While our general approach is close to [25], moving from Core Erlang to Erlang required to update most of the code. We could have applied our approach to generate a reversible semantics for Core Erlang from the irreversible one in [25], however the resulting reversible semantics would be sequential since the semantics in [25] relies on some global data structures to simplify the implementation of the model checking analysis, which would create fake causal dependencies. Notably, translating the semantics for Erlang in [9] into Maude directly is not trivial due to its high level of abstraction. E.g., the semantics in [9] resorts to the existence of suitable contexts to identify the redex inside an expression, while we need to explicitly give an inductive definition to find the redex. We could have started from [11] (formalized using the K framework for Maude) instead of [25], however the code in [25] was better documented.

Rollback semantics have been proved of interest for debugging techniques, we find examples in [7,5,19,9]. The rollback semantics presented here differs from

[5] Technicalities have been omitted, we refer to [22] for further details.

the ones in [15,5,19,9] as it is agnostic of the underlying formalism, and from the ones in [7,13,8] as it is more concrete (to the point of being executable). We could combine generality and executability thanks to use of the Maude framework.

Let us now discuss possible future developments. First, one could apply the framework to other case studies or larger fragments of Erlang. In doing so one has to ensure that the causal-dependencies captured by the producer-consumer model used in [14] are appropriate - for example the model is not well-suited to capture causal dependencies due to shared memory. For the semantics of a larger fragment of Erlang, one could take inspiration from the one in [30].

As far as rollback is concerned, one would like to identify states by properties (e.g., when a given message has been sent), as in [5,19,9], instead of using keys.

References

1. R. Carlsson. An introduction to Core Erlang. In *Erlang Workshop*, 2001.
2. I. Cristescu, J. Krivine, and D. Varacca. A compositional semantics for the reversible π-calculus. In *LICS*, pages 388–397, 2013.
3. V. Danos and J. Krivine. Reversible communicating systems. In *CONCUR*, volume 3170 of *LNCS*, pages 292–307, 2004.
4. V. Danos and J. Krivine. Formal molecular biology done in ccs-r. *LNCS*, 180(3):31–49, July 2007.
5. G. Fabbretti, I. Lanese, and J. Stefani. Causal-consistent debugging of distributed Erlang programs. In *RC 2021*, volume 12805 of *LNCS*, pages 79–95, 2021.
6. G. Fabbretti, I. Lanese, and J.-B. Stefani. Generation of a reversible semantics for Erlang in Maude. Research Report RR-9468, Inria, Apr. 2022.
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. Causal-consistent reversible debugging: Improving cauder. In *PADL*, volume 12548 of *LNCS*, pages 145–160. Springer, 2021.
10. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI 73*, 1973.
11. J. Kőszegi. KErl: Executable semantics for Erlang. In *CEUR Workshop Proceedings 2046*, pages 144–166, 2018.
12. P. Lami, I. Lanese, J. Stefani, C. S. Coen, and G. Fabbretti. Reversibility in erlang: Imperative constructs. In *RC 2022*, volume 13354 of *LNCS*, pages 187–203. Springer, 2022.
13. I. Lanese. From reversible semantics to reversible debugging. In *RC*, volume 11106 of *LNCS*, pages 34–46. Springer, 2018.
14. I. Lanese and D. Medic. A general approach to derive uncontrolled reversible semantics. In *CONCUR*, volume 171, pages 33:1–33:24, 2020.
15. I. Lanese, C. A. Mezzina, A. Schmitt, and J. Stefani. Controlling reversibility in higher-order pi. In *CONCUR*, LNCS, pages 297–311. Springer, 2011.
16. I. Lanese, C. A. Mezzina, and J. Stefani. Controlled reversibility and compensations. In *RC 2012*, volume 7581 of *LNCS*, pages 233–240. Springer, 2012.
17. I. Lanese, C. A. Mezzina, and J. Stefani. Reversibility in the higher-order π-calculus. *Theor. Comput. Sci.*, 625:25–84, 2016.

18. I. Lanese, N. Nishida, A. Palacios, and G. Vidal. CauDEr website. URL: `https://github.com/mistupv/cauder-v2`, 2018.
19. I. Lanese, N. Nishida, A. Palacios, and G. Vidal. A theory of reversibility for Erlang. *J. Log. Algebraic Methods Program.*, 100:71–97, 2018.
20. I. Lanese, U. P. Schultz, and I. Ulidowski. Reversible computing in debugging of Erlang programs. *IT Prof.*, 24(1):74–80, 2022.
21. J. S. Laursen, U. P. Schultz, and L.-P. Ellekilde. Automatic error recovery in robot assembly operations using reverse execution. In *IROS*, pages 1785–1792, 2015.
22. All about maude, 2007.
23. H. C. Melgratti, C. A. Mezzina, and I. Ulidowski. Reversing place transition nets. *Log. Methods Comput. Sci.*, 16(4), 2020.
24. J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *CONCUR 96*, pages 331–372, Berlin, Heidelberg, 1996.
25. M. Neuhäußer and T. Noll. Abstraction and model checking of Core Erlang programs in Maude. *ENTCS*, 176(4):147–163, Jul 2007.
26. K. S. Perumalla and A. J. Park. Reverse computation for rollback-based fault tolerance in large parallel systems: Evaluating the potential gains and systems effects. *Cluster Computing*, 17(2):303–313, Jun 2014.
27. A. Philippou and K. Psara. Reversible computation in Petri nets. In *RC*, pages 84–101, 2018.
28. I. C. C. Phillips and I. Ulidowski. Reversing algebraic process calculi. *J. Log. Algebraic Methods Program.*, 73(1-2):70–96, 2007.
29. Automatic generation of reversible semantics in Maude. `https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/gfabbretti8/formalization-in-maude-of-erlang`.
30. H. Svensson, L. Fredlund, and C. B. Earle. A unified semantics for future Erlang. In *ACM SIGPLAN workshop on Erlang*, pages 23–32. ACM, 2010.