Generation of a reversible semantics for Erlang in Maude

Giovanni Fabbretti, Ivan Lanese, Jean-Bernard Stefani

▶ To cite this version:

Giovanni Fabbretti, Ivan Lanese, Jean-Bernard Stefani. Generation of a reversible semantics for Erlang in Maude. [Research Report] RR-9468, Inria - Research Centre Grenoble – Rhône-Alpes. 2022, pp.1-22. hal-03630407v3

HAL Id: hal-03630407
https://hal.inria.fr/hal-03630407v3

Submitted on 2 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Generation of a Reversible Semantics for Erlang in Maude

Giovanni Fabbretti, Ivan Lanese, Jean-Bernard Stefani

RESEARCH REPORT
N° 9468
Avril 2022
Project-Team SPADES, FOCUS

ISSN 0249-6399 ISRN INRIA/RR--9468--FR+ENG

Generation of a Reversible Semantics for Erlang in Maude

Giovanni Fabbretti*, Ivan Lanese†, Jean-Bernard Stefani*

Project-Team SPADES, FOCUS

Research Report n° 9468 — Avril 2022 — 25 pages

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 their 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 usability. We bridge the gap between theory and practice by implementing it 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 framework, e.g., as an oracle against which to test the reversible debugger CauDEr for Erlang, or as an executable specification for new reversible debuggers.

Key-words: Debugging, Maude, Programming languages, Erlang, Concurrent systems

* Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, France
† Focus Team, Univ. of Bologna/INRIA, Italy

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Montbonnot Cedex

Generation de une sémantique reversible pour Erlang en Maude

Résumé : Récemment, la réversibilité dans les systèmes concurrents a été mise à profit dans plusieurs applications tirées de domaines différents comme le débogage, la reprise sur erreurs et la modélisation des systèmes biologiques. La réversibilité a été étudiée dans plusieurs formalismes, comme les réseaux de Petri, les algèbres de processus et différents langages de programmation. Néanmoins, tous les travaux visant à développer une variante réversible de ces formalismes souffrent de la même limitation: les sémantiques ont toujours été définies de manière ad-hoc. Très récemment, Lanese et al. ont proposé une méthode générale pour définir une sémantique réversible concurrente, de manière automatique, à partir d'une sémantique opérationnelle non réversible. Cette méthode n'avait cependant pas été instrumentée. Le but de ce papier est d'en proposer une implantation, prouvée correcte, dans l'environnement de logique de réécriture Maude, et de l'appliquer à un cas d'étude: le langage de programmation Erlang.

Mots-clés : Débogage, Maude, langages de programmation, Erlang, systèmes concurrents

Contents

1 Introduction **4**

2 Background **5**
2.1 Erlang: Syntax and Semantics 5
2.2 Maude 7
2.3 Derivation of the Reversible Semantics 8
2.3.1 Format of the Input Reduction Semantics 8
2.3.2 Methodology 9

3 Formalizing Erlang in Maude **10**
3.1 Equational Theory 11
3.2 Rewriting Rules 12
3.3 Management of Environments 13

4 Generating the Reversible Semantics **14**
4.1 Format of the Non-Reversible Semantics 14
4.2 Transformation to the Syntax 15
4.3 Generating the Reversible Semantics 15

5 Correctness **16**

6 Rollback Semantics **21**

7 Conclusion **23**

1 Introduction

Reversible computing studies computational models which have both a (standard) forward and a backward notions of execution. Reversibility has attracted interest thanks to its diverse applications in areas such as debugging [6, 5, 12, 7], robotics [13], biological modeling [4], and error-recovery [18]. In sequential systems reversibility is well understood: intuitively it corresponds to undo actions in reverse order of execution. In concurrent settings, more care is required. In 2004 Danos and Krivine proposed the notion of *causal-consistent reversibility* [3], tailored for concurrent systems. With causal consistency undoing an action in a parallel execution needs only to undo the causal consequences of that action. Actions which have been temporally interleaved with these consequences but are causally independent can be left alone. Thus causal consistency ensures only the strictly necessary events in a parallel execution need to be undone which is useful to explore concurrent programs which can be prone to state explosion. Causal-consistent reversibility has then been studied in several formalisms such as process calculi [3, 20, 2, 11], Petri nets [19, 15], and the Erlang programming language [5, 12, 7]. It also leads to interesting practical applications, the most prominent example being as a debugging technique as proposed in [6] and then implemented in the CauDEr debugger for Erlang [5, 12, 7].

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 Medic proposed a general method to automate the production of reversible semantics [9]. 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.

$$\langle p_1, \theta, p_2 \; ! \; hello, me \rangle \rightarrow \langle p_1, \theta, hello, me \rangle \mid \langle p_1, p_2, hello \rangle \quad (1)$$

On the left, a process p_1 is ready to send a message *hello*. 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 [9] was only described theoretically and the semantics taken in input is assumed to be 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, bridging the gap between theory and practice by using schemas to represents the ground rules. A schema is a parameterized rule whose variables can be instantiated to obtain a ground rewriting rule (allowing the representation of an infinity of such rules). We then use the tool to derive a causal-consistent reversible semantics for the Erlang programming language, which can compare to the ones previously produced by hand.

Finally, we further extend Lanese et al. ideas by devising a causal-consistent rollback operator defined on top of the reversible semantics, which is a key primitive for a concurrent causal-consistent debugger as described in [6]. In the literature we can find examples of causal-consistent

rollback operators, like [5, 12, 7], nonetheless these examples were always designed in an ad-hoc fashion suffering the same limits as the reversible semantics. In contrast, our rollback operator is able to cope with all the reversible semantics produced by the general approach, thanks to their uniformity. This is beneficial and desirable as one can change or update the underlying semantics without the need to change the rollback one.

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

- a novel formalization of Erlang using Maude;
- a tool that derives a reversible semantics starting from a non reversible one.
- a novel automatic causal-consistent rollback semantics built on top of the reversible semantics

Paper organization We provide the reader with the required background in Section 2. In Section 3 we present the formalization of the Erlang semantics in Maude and in Section 4 we discuss how the reversible semantics is generated. In Section 5 we show the correctness of the reversible semantics. In Section 6 we present a rollback semantics built on top of the reversible semantics automatically generated. Finally, in Section 7 we give some conclusion and we hint at possible future directions.

All the code discussed in this paper is publicly available at [21].

2 Background

2.1 Erlang: Syntax and Semantics

Erlang is a functional and concurrent programming language. First introduced in 1986 by Ericsson, it has gained quite a lot of popularity since then. Today it is widely used and mostly 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 [8], 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. Indeed, central in Erlang are the `send`, `receive` and `spawn` operations, to, respectively, send a message, receive a message, and create a new actor. 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.

The rest of this section provides the reader with a basic understanding of the Erlang programming language. We begin by illustrating its syntax, depicted in Fig. 1[1].

An Erlang program can be seen as a collection of modules, where a module is a sequence of function definitions. Each function is uniquely identified by its name and by the number of formal parameters. Each function may be specified by cases via multiple definitions. The correct definition for each invocation is selected by pattern matching on parameters. The body of each definition is represented by a sequence of expressions. In the following we denote an expression with e and sequences of expressions as $e_1, \ldots, e_n$ - sequences of other syntactical elements are represented in the same manner.

Ground values in Erlang are: atoms (which are identifiers that either begin with a lowercase or are enclosed by quotes), integers and strings, as well as compositions of values using tuple and list constructs. We range over ground values using v. Tuples are denoted as $\{v_1, \ldots, v_n\}$ and lists are denoted as $[v_1|v_2]$, where v_1 is the head and v_2 the tail.

[1]We support the functional and concurrent fragment of the Erlang language.

$$
\begin{array}{rcl}
program & ::= & mod_1 \dots mod_n \\
mod & ::= & fun_def_1 \dots fun_def_n \\
fun_def & ::= & Atom([patterns]) \rightarrow exprs. \\
pat & ::= & b_value \mid Var \mid \ '\{'[patterns]'\}' \mid \ '['[patterns|patterns]']' \\
patterns & ::= & pat \ \{',' patterns\} \\
exprs & ::= & expr \ \{',' exprs\} \\
expr & ::= & b_value \mid Var \mid \ '\{'[exprs]'\}' \mid \ '['[exprs|expr]']' \\
& \mid & \mathsf{case}\ expr\ \mathsf{of}\ clseq\ \mathsf{end} \mid \mathsf{receive}\ clseq\ \mathsf{end} \mid expr\ !\ expr \\
& \mid & pat = expr \mid [Mod{:}]expr([exprs]) \\
b_value & ::= & Atom \mid Char \mid Float \mid Integer \mid String \\
clseq & ::= & pat \rightarrow exprs \ \{';' pat \rightarrow exprs\}
\end{array}
$$

Figure 1: Language syntax

Variables can store ground values. Variables, e.g., X, Age, start with a capital letter and are not enclosed in quotes. Patterns, denoted by pat, are like the ground values, but also admit the presence of variables. Patterns are used for pattern matching in the following contexts: (i) in the matching operation $e_1 = e_2$, (ii) in case statements case e of $pat_1 \rightarrow exprs_1; \dots; pat_n \rightarrow exprs_n$ end to choose the branch to evaluate according to the shape of the incoming data, (iii) in receive statements receive $pat_1 \rightarrow exprs_1; \dots; pat_n \rightarrow exprs_n$ end, to analyze the shape of the received message, and (iv) in function definitions, to give values to the formal parameters.

We start by explaining the match operation, $e_1 = e_2$. First, the expression e_2 on the right-hand side is evaluated until it becomes a ground value, occurrences of free variables, if any, raise an exception (however we do not support exception propagation and management in this work). Then, the expression on the left-hand side, e_1, is evaluated until it becomes a pattern, or a ground value in case no free variables occur in it. Then the two elements are matched against each other. Each free variable of the left-hand side is bound to the corresponding ground value of the right-hand side, ground values in corresponding position should coincide: if a mismatch occurs then an exception is raised. If no mismatch occurs then the operation evaluates to the ground value of the right-hand side and the environment is updated with the new bindings.

In the case construct, the expression e must evaluate to a ground value, then it is matched against the patterns, from top to bottom, until one that matches is found. When a match is found the environment is enriched with the new bindings and the corresponding sequence of expressions evaluated, if no match is found an exception is raised.

The behavior of the receive is similar to the one of the case, with the only difference that messages in the queue of the process are tried as ground values till the match succeeds. When a match is found the corresponding branch is selected. Contrary to the case, if no match is found then the process suspends.

Despite being - mostly - functional Erlang admits some imperative operations that produce side-effects, like the receive above, spawning a new process, and sending a message.

The syntax of message send is $e_1!e_2$, where e_1 must evaluate to the pid of the receiver process and e_2 must evaluate to the ground value that represents the payload of the message. The expression itself evaluates to the payload and, as a side-effect, the message is sent.

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

```
fmod BOOL is
  sort Bool .
  op true : -> Bool [ctor] .
  op false : -> Bool [ctor] .

  op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
  op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
  op not_ : Bool -> Bool [prec 53] .

  vars A B C : Bool .

  eq true and A = A .
  eq false and A = false .
  eq A and A = A .
  eq not false = true .
  eq not true = false .
  eq A or B = not A and not B .
  eq not not A = A .
endfm
```

Figure 2: Maude module for Booleans

2.2 Maude

Maude [14] is a programming language that efficiently implements rewriting logic [16]. Formally, a rewriting logic is a tuple (Σ, E, R), where Σ represents a collection of typed operators, E a set of equations among the operators, and R a set of semantics rules.

Using a rewriting logic is quite convenient to formalize the semantics of a language as it provides the benefits of using both an equational theory and rewriting rules.

On the one hand, the equational side of rewriting logic is well-suited to define the deterministic part of the model, where we define equivalence classes over terms. More precisely we say that two terms v and u are equivalent if under a set of equations E we can prove $E \vdash v = u$. Equations can be conditional and conditions can be either the membership of the term to some kind or other equations.

On the other hand, the rewriting rules are well-suited to 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, like equations, can be conditional and conditions can be memberships, equations, as well as other rewriting rules.

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. 2, that is an example of a Maude module that implements Booleans together with their classic operations.

First, the sort `Bool` is declared. Then, the values `true` and `false` are declared as two constant operators of sort `Bool`. Successively, the classic operations are defined as functions that take in input some `Bool`s and produce a `Bool` as a result. For example, `_and_ : Bool Bool -> Bool` defines the `and` operator that takes in input two `Bool`s and produces a `Bool`. Finally, the semantics of these operators is given by the equational theory defined at the bottom 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' >
```

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 takes in input the modules of the non-reversible semantics and produces new modules, which define the reversible semantics.

2.3 Derivation of the Reversible Semantics

The following of this section summarizes the main ideas of [9] where Lanese et al. propose a methodology to automatically derive a causal-consistent reversible semantics starting from a non-reversible one. The approach requires that the latter is modeled as a reduction semantics which satisfies some syntactic conditions.

2.3.1 Format of the Input Reduction Semantics

We now describe the shape that the reduction semantics taken as 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. 3. The format contains rules to: i) allow entities to interact with each other (Scm-Act); ii) exploit a structural congruence (Eqv); iii) allow single entities to execute inside a context (Scm-Opn); iv) execute two systems in parallel (Par). Notably, while (Eqv) and (Par) are rules that must belong to the semantics, (Scm-Act) and (Scm-Opn) are schemas, and the semantics may contain any number of instances of the schemas. Actually, rule (Par) is an instance of schema (Scm-Opn), highlighting that such an instance is required. As another example, reduction (1) from the Introduction is an instance of schema (Scm-Act). Also, notice that a notion of structural congruence on systems is assumed. We refer to [9] 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.

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

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

Figure 3: Required structure of the semantics in input; Scm- rules are schemas

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

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

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

Figure 4: Forward rules of the uncontrolled reversible semantics

2.3.2 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 the 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 processes 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. Moreover, they 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. 4. 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 format they must fit.

The backward rules, depicted in Fig. 5, 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

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

$$\text{(B-SCM-OPN)}\ \frac{R'_i \rightsquigarrow R_i}{op_n(R_0, \ldots, R'_i, \ldots, R_n) \rightsquigarrow op_n(R_0, \ldots, R_i, \ldots, R_n)} \qquad \text{(B-EQV)}\ \frac{R \equiv_c R' \quad R \rightsquigarrow R_1 \quad R_1 \equiv_c R'_1}{R' \rightsquigarrow R'_1}$$

Figure 5: Backward rules of the uncontrolled reversible semantics

can be performed and the old configuration R can be restored. E.g., the backward rule undoing the reduction (1) from the Introduction is:

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

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 [9] 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 [10], i.e., if multiple (forward and/or backward) steps are enabled at the same time there is no policy on which one to choose.

3 Formalizing Erlang in Maude

In this section we present the formalization of the semantics of Erlang in Maude. We mostly follow the semantics defined in [7]. Technically, we used as starting point the formalization of Core Erlang [1] in Maude presented in [17], which was aimed at model checking. While our formalization is quite different from the one they presented (the most notable differences are that we formalize a fragment of Erlang instead of one of Core Erlang and the use of labels), we were still able to re-use some of their modules and some of their ideas, like the internal representation of ground values, which greatly simplified the formalization task.

As in [7], 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 feature and a rewriting rule τ (actually two, for efficiency reasons) for sequential operations. While it would have been possible to define the sequential operations as an equational theory also at the system level, we take a different approach. Indeed, using rule τ is the only way to evaluate expressions (relying on the equational theory), but it forces evaluation to stop when some base cases are reached. This is more suitable to define the behavior of a (reversible) debugger, which is our intended application. Notably, also a different semantics where expressions are fully evaluated 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[2], e is the expression

[2]Actually θ is a stack of environments, later on we will clarify why.

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 the scope of this work processes and messages are entities in the lower level of the semantics. We denoted them as P in Section 2.3.

A running 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 equations which have one of the following generic forms:

$$\begin{array}{l} \mathsf{eq} : \ [equation - name] \\ \quad \langle l, \theta,\ e \rangle = \langle l', \theta',\ e' \rangle \\[1em] \mathsf{ceq} : \ [equation - name] \\ \quad \langle l, \theta,\ e \rangle = \langle l'', \theta'',\ e'' \rangle \\ \quad \mathsf{if}\ \langle l', \theta', e' \rangle := op(l, \theta, e)\ \wedge\ \langle l'', \theta'',\ e'' \rangle := \langle l', \theta', e' \rangle \end{array}$$

As we can see from above, to evaluate an expression 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 plays two roles: i) it communicates the kind of side effect performed by the expression, if any; ii) it communicates information of the details of the side effect back and forth between the expression level and the system level. Examples of this mechanism are presented below.

Two kinds of equations are used: conditional ones, featuring an `if` clause, and unconditional ones. Unconditional equations define a simple reduction of the expression and change the label to the appropriate one.

Example 3.1 (Equation for self)**.** The unconditional equation below describes the behavior of `self` at the expression level.

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

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 `INT` (via `self(pid(INT))`) and the expression is actually a self (`atom("self")()`) then the expression reduces to the pid (`int(INT)`) 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.

Example 3.2 (Conditional equation for receive)**.** Figure 6 describes the conditional equation for `receive`. It reads roughly as follows. If the system level asks whether a message with a given payload can be received (`req-receive(PAYLOAD)`) and the current expression is a receive (`receive CLSEQ end`) then one tries to match the body `CLSEQ` of the receive against the payload

```
ceq [receive] :
  < req-receive(PAYLOAD), ENV : ENVSTACK, receive CLSEQ end> =
  < received, ENV' : (ENV : ENVSTACK), begin EXSEQ end>
  if #entityMatchSuccess(EXSEQ | ENV') :=
     #entityMatch(CLSEQ | PAYLOAD | ENV ) .
```

Figure 6: Conditional equation for receive

```
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 > .
```

Figure 7: System rule send

using the environment `ENV`. If the match succeeds (that is the result of applying the operator `#entityMatch` matches the pattern `#entityMatchSuccess(EXSEQ | ENV')`) then it returns the selected clause of the receive `EXSEQ` as well as the environment enriched with the bindings from the match `ENV'`. In this case the expression can reduce to `EXSEQ` and will be evaluated in the new environment. Label `received` denotes successful reception.

3.2 Rewriting Rules

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

$$
\begin{array}{l}
\textsf{crl}: \ [rule-name] \\
\quad \langle p,\theta,e,me\rangle \mid E => \langle p,\theta',e',me\rangle \mid op(l',\langle p,\theta,e,me\rangle,E) \\
\quad \textsf{if} \ \langle l',\theta',e'\rangle := \langle l,\theta,e\rangle
\end{array}
$$

In the schema above 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 and by the information on the process. Examples 3.3 and 3.4 below show some sample rewriting rules.

Example 3.3. Let us consider the conditional rewriting rule in Fig. 7, which is used to send a message. In the conditional part of the rule we use the equational theory to check which kind of reduction the current expression `EXSEQ` can perform. If it can perform a send of a `GVALUE` to `DEST`, then the process evolves so to evaluate the new expression `EXSEQ'` in the new environment `ENV'`, and the new message is added to the system. Note that `P` is the pid of the process (as already mentioned, Maude needs a unique identifier for this notation), and `ASET` includes other elements of the process which are not relevant here (currently, only the module environment). With respect to 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. This exemplifies how the label serves to communicate information from the expression level to the system one. Using this information,

```
crl [sys-receive] :
  < P | exp: EXSEQ, env-stack: ENV, ASET > ||
  < sender: SENDER, receiver: P, payload: GVALUE > =>
  < P | exp: EXSEQ', env-stack: ENV', ASET >
if < received, ENV', EXSEQ' > :=
   < req-receive(GVALUE), ENV, EXSEQ > .
```

Figure 8: System rule receive

side effects (in this case the send of a message) are performed at the system level. Note that the rewriting rule in Section 2.2 is an instance of the one above.

Example 3.4. Let us consider rule $\mathsf{sys-receive}$ in Fig. 8, which is applied when a process receives a message. In the rule, if there is a message targeting the process, we use the equational theory (see Example 3.2) to check whether the message can be received in the current state. If this is the case then the state is updated and the message is removed from the system. This rule shows how the label can be used to bubble up information from the system level to the expression one.

3.3 Management of Environments

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.

For example, consider the function invocation

$$X = pow_and_sub(N, M) \tag{2}$$

where

$$pow_and_sub(N, M) \rightarrow Z = N * N, Z - M.$$

Simply replacing the function call with its body would produce

$$X = Z = N * N, Z - M.$$

which is a syntactically correct Erlang expression, but which would not have the desired effect, as the variable X would assume the value $N * N$ instead of $Z - M$ as desired.

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

$$\mathsf{case}\ pow_and_sub(N, M)\ \mathsf{of} \ldots \tag{3}$$

Simply replacing the function call with its body would produce

$$\mathsf{case}\ Z = N * N, Z - M\ \mathsf{of} \ldots$$

which is illegal and would be refused by an Erlang compiler.

The solution that we adopted to solve both the problems consists in wrapping the produced sequence of expressions with the construct `begin_end`, which turns a sequence of expressions into a single expression. For instance, in (2) the produced expression would be

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

```
mod SYSTEM is
...
  sort Sys .
  subsort Entity < Sys .

  op #empty-system : -> Sys [ctor] .
  op _||_ : Sys Sys -> Sys [ctor assoc comm .. ] .
...
endm
```

Figure 9: Extract of the system module for Erlang.

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.

In statement (3) the produced expression is:

$$\mathsf{case\ begin}\ Z = N * N, Z - M\ \mathsf{end\ of} \ldots$$

which is a correct Erlang term, accepted by the compiler.

Adding `begin_end` blocks requires us to properly manage the environment corresponding to each block of code. So, rather than having a single environment inside each process, we have a stack of environments, hence what we previously denoted as θ actually has the shape $\theta_1 : \ldots : \theta_n$. To sum up, whenever an expression e that produces a sequence of expressions $e_1, \ldots, e_n$ is evaluated, we wrap the sequence as begin $e_1, \ldots, e_n$ end and we push on the stack of environments the appropriate environment to evaluate $e_1, \ldots, e_n$.

If e is a function call then the appropriate environment has to bind the formal parameters of the function to the actual ones, while if e is a case or a receive statement then the appropriate environment is the previous one enriched with the bindings obtained from pattern matching.

Finally, once the subcomputation terminates producing an expression of the shape begin v end, we simply replace it with the value v and we pop one environment from the stack.

4 Generating the Reversible Semantics

We decided to define the generation of the reversible semantics in Maude, 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.

4.1 Format of the Non-Reversible Semantics

As in [9], 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. 9 depicts the system module for the Erlang language. We omit elements that are not interesting in this context (import of other modules and auxiliary functions needed to customize the translation between the module and its meta-representation).

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

Both inputs and outputs of all the operators inside the module `SYSTEM` must be of sort `Sys`. The subsort relation `Entity < Sys` must be declared as well, to specify that entities of the lower level can be used as systems. To this end, the sorts of the lower level (in Erlang, messages and processes) must be subsorts of `Entity`.

The rewriting rules of the rewriting theory that defines the single steps of the reduction semantics must be defined under the module `TRANSITIONS`. We have discussed sample rules in Figures 7 and 8.

4.2 Transformation to the Syntax

We describe here how to transform a non-reversible syntax as described above into a reversible syntax, as described in [9] and recalled in Section 2.3. Roughly, we need to add keys and memories.

Keys are a sort, and we also define the sort `EntityWithKey` composing an entity and a key using operator `*`. The subsort relation `EntityWithKey < Sys` is added to the module, so that now all system-level operators can deal with entities with key.

To define memories, first we declare a new sort `Placeholder`, together with an operator `@` to create a `Placeholder` 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 key with the final configuration, where the entities have been replaced by their placeholders. E.g., the memory created by the reversible version of the reduction in Section 2.2 is:

```
[ < 1 | exp: 2 ! 'hello', env: {}, me: _ > * key 0 ;
                          @: key 0 0 || @: key: 1 0 ]
```

The final transformation concerning the system module updates the equations to translate entities between their representation and their meta-representation so to work on entities with key.

4.3 Generating the Reversible Semantics

The transformation to be performed over the rewriting rules is the one described in Section 2.3.2, rephrased in Maude notation. Thus, 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 must have a 'distributed' way to compute them, as passing around a key generator would produce spurious causal dependencies. To solve the problem we resorted to the following idea. 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` concatenated with a number corresponding to the position of the entity on the right-hand side. Furthermore, we create the required memory.

Fig. 11 recalls rule `send` and shows the corresponding reversible rule. In the latter, 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 production of the backward semantics is easy: to produce a backward rule it is enough to swap the left- and the right-hand side of the corresponding forward reversible rule and to drop 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

```
  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 > .

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 > .
```

Figure 10: Original and forward reversible rule send

```
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
```

Figure 11: Reversible backward rule send

been already undone. This is ensured by the presence of the entities bound by the placeholders inside the memory.

The backward rule for send is shown in Figure 11.

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 [9] 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 [9]. We also need to show that

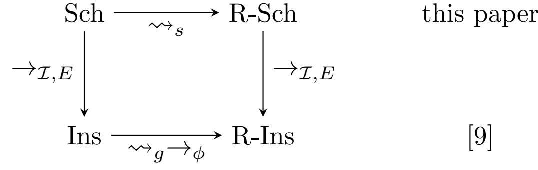

Figure 12: Schema of the proof of correctness.

by generating the reversible semantics of the schemas, as we do here, and then instantiating, we obtain the same set of instances obtained by first instantiating the original schema and then generating their we need to show that the diagram in Fig. 12 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 [9].

Let us begin by discussing the functions on the sides of the square. First, function $\rightsquigarrow_s$ takes as input a set of non-reversible rule schemas of the form $t \to t'$ if C and generates the corresponding set of reversible (forward and backward) rule schemas. Formally function $\rightsquigarrow_s$ is defined by the rules in Fig. 13. The rules make use of a number of auxiliary functions, let us analyze them. First, function $k_l(t, L)$ given a term t proceeds recursively to tag each entity of the term with the key variable L (L is the same in all the rules, we put it anyway as a parameter to highlight that the same variable is used also on the right-hand side as described below). At each recursive call L is concatenated with an $'$. This ensures key variables on the left-hand side are all distinct. Function $k_r(t, L)$ given a term t proceeds recursively to tag each entity of the term with the variable L prefixed by the entity position within the term.
Finally, function $ctx(t_r)$ given a reversible term, i.e., composed only of entities tagged with keys, proceeds recursively to substitute each entity with a $\bullet$ so to create a context for t_r.

Then, $\to_{\mathcal{I},E}$ takes as 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 $\to_{\mathcal{I},E}$ 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). Formally function $\rightsquigarrow_{\mathcal{I},E}$ computes the set of instances generated by the following inference rule:

$$\frac{t \to t' \ \mathsf{if}\ C \quad i \in \mathcal{I} \quad E \vdash i(C)}{i(t) \to i(t')}$$

With $E \vdash i(C)$ we denote that side condition C, instantiated using substitution i, holds according to the equational theory E.

The following example shows how starting from a schema by using relation $\rightsquigarrow_{\mathcal{I},E}$ we can obtain a ground instance of the schema.

Example 5.1 (Schema instantiation)**.** Let us consider the rewriting rule `tau` below, which lifts a sequential step from the equation level to the system level, and the equation about matching.

```
crl [sys-tau] :
    < P | exp: EXSEQ, env-stack: ENV, ASET >  =>
    < P | exp: EXSEQ', env-stack: ENV', ASET >
```

```
    if < tau, ENV', EXSEQ' > :=
       < req-gen, ENV, EXSEQ > .

eq [match] :
 < REQLABEL, ENVSTACK, GVALUE = GVALUE > =
 < tau, ENVSTACK, GVALUE > .
```

Then let us consider the following $i \in \mathcal{I}$:

$$\{(\texttt{EXSEQ} \mapsto \texttt{true = true}), (\texttt{EXSEQ'} \mapsto \texttt{true}), (\texttt{P} \mapsto 2), (\texttt{ENV} \mapsto \texttt{\{\}}), (\texttt{ENV'} \mapsto \texttt{\{\}})\}$$

For the sake of simplicity we ignore the components of a process inside `ASET`, as they are not relevant for this example (`ASET` currently only contains the definitions of the functions that the process is allowed to invoke).

After substituting the variables with ground values we obtain a rule with the following shape.

```
crl [sys-tau] :
    < 2 | exp: true = true, env-stack: {}, _ >  =>
    < 2 | exp: true, env-stack: {}, _ >
    if < tau, {}, true > :=
       < req-gen, {}, true = true > .
```

Then, we can use the equational theory (in this example the `match` equation depicted above) to check the validity the conditional branch. Finally, since the equation in the conditional branch is valid in the equational theory, we can drop the conditional branch and obtain the following ground rule.

```
    < 2 | exp: true = true, env-stack: {}, _ >  =>
    < 2 | exp: true, env-stack: {}, _ >
```

Function $\rightsquigarrow_g$ models the general approach defined in [9]. Intuitively, $\rightsquigarrow_g$ works like $\rightsquigarrow_s$, but it takes only instances of rule schemas. Also, it adds concrete keys instead of key variables. Formally function $\rightsquigarrow_g$ is defined by the rules in Fig. 14. The rules are quite similar to the ones in Fig. 13, only they deal with ground instances of schemas and also the function used to tag terms is different. Indeed, while for schemas we need to follow a precise algorithm to select key variables matching the approach in this paper, the general approach from [9] picks keys from an arbitrary set - what is modeled here by k_g. The selection policy is left implicit in [9]. Here, for simplicity, we assume distinct keys are used in different rules. This is not restrictive since in order to be applied keys need anyway to be α-converted, the keys on the left-hand side of rules to match keys on the running system, and the keys on the right-hand side to ensure freshness.

Function $\rightarrow_\phi$ is a function mapping keys in [9], which are taken from an arbitrary set, to keys in our approach, which are lists of integers.

We are now ready to prove our main result.

Theorem 1 (Correctness)**.** Given functions $\rightsquigarrow_g$, $\rightsquigarrow_s$and $\rightarrow_{\mathcal{I},E}$ in Fig. 12 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. 12 commutes, i.e., $\rightsquigarrow_s \rightarrow_{\mathcal{I},E} = \rightarrow_{\mathcal{I},E} \rightsquigarrow_g \rightarrow_\phi$

Proof. We refine the commuting square in Fig 15, adding sample elements of the sets at each corner. The thesis follows by showing that the equality on the bottom-right corner holds, namely

$$\frac{t \to t' \text{ if } C \quad t_r = k_l(t, L) \quad t'_r = k_r(t', L) \quad c = ctx(t'_r)}{t_r \to t'_r \mid [t_r; c] \text{ if } C}$$

$$\frac{t \to t' \text{ if } C \quad t_r = k_l(t, L) \quad t'_r = k_r(t', L) \quad c = ctx(t'_r)}{t'_r \mid [t_r; c] \to t_r}$$

Figure 13: Schemas transformation

$$\frac{gt \to gt' \qquad gt_r = k_g(gt, \mathcal{K}_l) \qquad gt'_r = k_g(gt', \mathcal{K}_r) \qquad c = ctx(gt'_r) \qquad \mathcal{K}_l \cap \mathcal{K}_r = \emptyset}{gt_r \to gt'_r \mid [gt_r; c]}$$

$$\frac{gt \to gt' \qquad gt_r = k_g(gt, \mathcal{K}_l) \qquad gt'_r = k_g(gt', \mathcal{K}_r) \qquad c = ctx(gt'_r) \qquad \mathcal{K}_l \cap \mathcal{K}_r = \emptyset}{gt'_r \mid [gt_r; c] \to gt_r}$$

Figure 14: Ground rules transformation

that there exists a total function $\to_\phi$, injective on key variables belonging to the same rule, so that for each term t

$$i(k_x(t, L)) = \phi(k_g(i(t), \mathcal{K}_x))$$

where $x \in \{l, r\}$. We proceed by case analysis on x.

Case $x = l$.
We must prove that $i(k_l(t, L)) = \phi(k_g(i(t), \mathcal{K}_l))$.
We begin by observing that, on the left-hand side, k_l tags the entities of the term t with meta-keys, which will be initialized by the seed given in input, i.e., L. Successively, i will instantiate all the variables to ground values, obtaining so a ground rule. On the right-hand side, i instantiates the variables of t to ground values, the same as the left hand-side, and then k_g proceeds to tag entities with fresh keys picked from $\mathcal{K}_l$. Notice that at this point the terms will be the same but for possibly the keys. However, all keys of a single instance are distinct both in the instances from our approach and in the ones from [9] (also thanks to injectivity of i on key variables). Also, we assumed that keys of different instances in [9] are distinct, hence we can define $\to_\phi$ so to map each key from [9] to the key in the corresponding position in our approach. By construction this is total and injective on key variables belonging to the same rule, as desired.

Case $x = r$.
The argument is similar to the one presented above. □

Example 5.2 (Commutativity of the diagram)**.** Here, we show an example, based on the rule in Example 5.1, of how by following the two paths of Fig. 15 we get the same result.

First, starting from the top-left, we can make the schema reversible by following the horizontal arrow ($\rightsquigarrow_s$), getting the following reversible schemas.

```
< P | exp: EXSEQ, env-stack: ENV, ASET > * key(L) =>
 < P | exp: EXSEQ', env-stack: ENV', ASET > key(0 L) ||
 [< P | exp: EXSEQ, env-stack: ENV, ASET > * key(L) ; @: key(0 L)]
  if < tau, ENV', EXSEQ' > :=
     < req-gen, ENV, EXSEQ > .

 < P | exp: EXSEQ', env-stack: ENV', ASET > key(0 L) ||
 [< P | exp: EXSEQ, env-stack: ENV, ASET > * key(L) ; @: key(0 L)] =>
  < P | exp: EXSEQ, env-stack: ENV, ASET > * key(L)
```

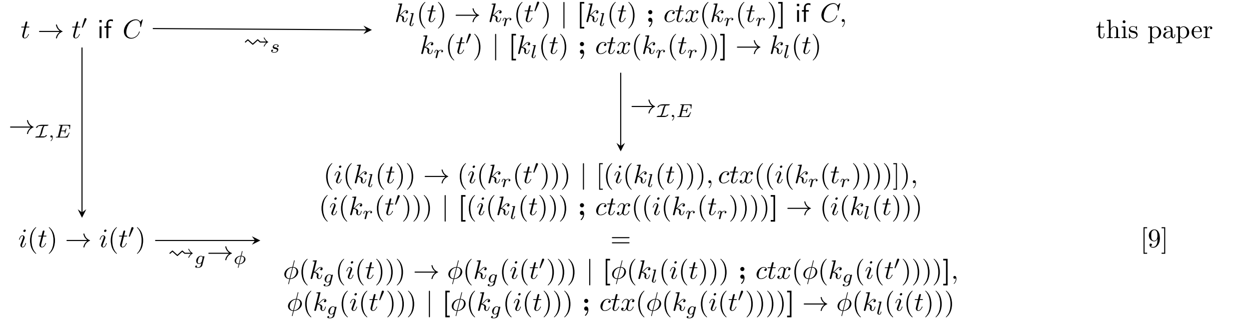

Figure 15: Schema of the proof of correctness. Seeds for k_l, k_r and k_g have been omitted for simplicity.

We can take as i:

$$\{(\texttt{EXSEQ} \mapsto \texttt{true = true}), (\texttt{EXSEQ'} \mapsto \texttt{true}), (\texttt{P} \mapsto 2), (\texttt{ENV} \mapsto \texttt{\{\}}), (\texttt{ENV'} \mapsto \texttt{\{\}}), (\texttt{L} \mapsto 0)\}$$

W.r.t. Example 5.1 we now provide a binding for `L` as well. This would make no difference in Example 5.1 since `L` did not occur in the term. By instantiating the two schemas with i we obtain the two following ground rules.

```
< 2 | exp: true = true, env-stack: {}, _ > * key(0) =>
  < 2 | exp: true, env-stack: {}, _ > * key(0 0) ||
  [< 2 | exp: true = true, env-stack: {}, _ > * key(0) ; @: key(0 0) ]

< 2 | exp: true, env-stack: {}, _ > * key(0 0) ||
[< 2 | exp: true = true, env-stack: {}, _ > * key(0) ; @: key(0 0) ]
  => < 2 | exp: true = true, env-stack: {}, _ > * key(0)
```

Now, let us resume the result in Example 5.1, where we showed how by following first the vertical arrow ($\rightarrow_{\mathcal{I},E}$) we get a ground rule. By following the horizontal arrow ($\rightsquigarrow_g$) we can transform

```
< 2 | exp: true = true, env-stack: {}, _ >  =>
< 2 | exp: true, env-stack: {}, _ >
```

into

```
< 2 | exp: true = true, env-stack: {}, _ > * key(a) =>
  < 2 | exp: true, env-stack: {}, _ > * key(b) ||
  [< 2 | exp: true = true, env-stack: {}, _ > * key(a) ; @: key(b) ]

< 2 | exp: true, env-stack: {}, _ > * key(b) ||
[< 2 | exp: true = true, env-stack: {}, _ > * key(a) ; @: key(b) ]
  => < 2 | exp: true = true, env-stack: {}, _ > * key(a)
```

i.e., in its corresponding forward and backward version. Notice that since the general approach does not impose any constraint on keys (except freshness) here we decided to represent them as lists of characters.

Finally, let us choose as ϕ:

$$\{(\texttt{key(a)} \mapsto \texttt{key(0)}), (\texttt{key(b)} \mapsto \texttt{key(0 0)}), \}$$

Thanks to $\rightarrow_\phi$ we can replace keys to recover the ones chosen by our approach, so to get

```
< 2 | exp: true = true, env-stack: {}, _ > * key(0) =>
  < 2 | exp: true, env-stack: {}, _ > * key(0 0) ||
  [< 2 | exp: true = true, env-stack: {}, _ > * key(0) ; @: key(0 0) ]

< 2 | exp: true, env-stack: {}, _ > * key(0 0) ||
[< 2 | exp: true = true, env-stack: {}, _ > * key(0) ; @: key(0 0) ]
  => < 2 | exp: true = true, env-stack: {}, _ > * key(0)
```

as desired.

6 Rollback Semantics

In this section we discuss an automatic causal-consistent rollback semantics built on top of the reversible backward semantics.

A causal-consistent rollback semantics is a semantics that reverts automatically the system back to a past state by undoing *all and only* the actions that have a causal link with such state. The user is selects a state that the system has visited by specifying one of the -potentially many-unique keys of such state and then the rollback semantics undoes automatically all the actions until such state is restored.

For the rollback semantics to work we must have a method to perform *controlled* backward steps. Here, controlled means that we can specify which process has to perform the backward step. Fortunately, Maude provides a way to rewrite systems that fits our needs: the `MetaXApply` function. `MetaXApply` takes in input seven elements:

- a theory $\mathcal{R}$
- a term t
- a rule name l
- a substitution ϕ
- a lower bound n
- an upper bound b
- a natural number m

Then, once invoked, `MetaXApply` works as follow:

- normalizes the term t with the equations in $\mathcal{R}$
- matches the normalized term t with the first $m + 1$ rules that matches l on which the substitution ϕ has been applied and discards the first m matches
- fully reduces the normalized term t with the $m + 1$ rule that matched
- returns the rewritten term

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

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

Figure 16: Dependencies operator

The lower bound n and the upper bound b define boundaries for the nesting level of the term that will be rewritten, to have no boundaries, like we desire for our case, it is enough to set n to be 0 and b to be the constant *unbounded*.

Now that we have a way to perform single step back we need a way to uniquely identify them. One possibility is to rely on the keys involved in the corresponding forward step. Each forward rule describes how the system transitions from the term on the left-hand side to the one on the right-hand one and the keys used to tag both terms are always unique. This means that each forward step is uniquely identified by the keys that it involves, hence those keys also uniquely identify the corresponding backward step. To perform a controlled backward step is enough to know one of the keys used to tag the initial configuration that gave rise to the forward step as there will exist only one backward ground instance of a schema that involves such key.

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 (σ), set b and n to have no boundaries (i.e., the rewriting can happen anywhere) and set m to 0.

Finally, given a state in input, we need to compute the set of actions that have a causal link with it. To do so we rely on the *dependencies* operator, depicted in Fig. 16, which works as follow.

Given a key k and a system configuration S we have two possibilities:

- the system contains a memory with an entity in the initial configuration tagged with k. This means that some consequences of the state are still acting on the system, to undo them we call recursively the procedure on the set of keys of the context and eventually we add to the set of computed request also k;
- the system does not contain a memory having in the initial configuration an entity tagged with k; this means no consequences of such action are acting on S.

In the definition of dep we find two auxiliary functions, let us discuss them. First, getMem, given a system configuration and a key, returns the memory that contains an entity tagged with such key, if it is found, while $\mathsf{contextKeys}$ given a memory returns the set of keys used in the context.

Once the set of backward steps (i.e., a set of keys) has been computed it suffices to non-deterministically choose a key so that `MetaXApply` can successfully perform a rewrite and continue until the set is emptied. When all the backward steps have been done the system will be in the desired state.

In the past other causal-consistent rollback semantics have been proposed, we find examples in [5, 12, 7]. The three rollback semantics all worked on different subsets of the Erlang language,

but they all followed the same principles. A system in rollback mode is identified as a system with a stack of requests - i.e., backward steps - to be satisfied. If the request on top can be undone then it is undone, otherwise some consequences of the action are still in place so we first need to undo them. The new requests are always computed ad-hoc, according to the shape of the system and of the entities involved. In our case the flavor is more general, indeed we are agnostic of the underlying system. Moreover, we also have a finer semantics, indeed thanks to the fact that we identify states through keys we can rollback to any state of the system whereas in the cited rollback semantics it is only possible to reach certain labeled states.

The automatic rollback semantics presented in this work can be thought as a base on which to build more human-friendly semantics, like the ones in [5, 12, 7], in which states are identified through labels. The advantage of having an automatic rollback semantics is that each time that the reduction semantics given in input is changed we do not need to update the rollback semantics but we only need to generate a new backward reversible one.

7 Conclusion

We presented a new formalization of the Erlang language using Maude. Having a mechanized version of the Erlang semantics makes it much easier to debug it and to become confident that the semantics correctly captures the behavior of the language. Indeed, to test the semantics, in our work, one can load an Erlang module and actually run an arbitrary program (as long as the program uses supported primitives) and the states traversed can be compared against an actual execution to make sure that the two agree.

Concretely, defining an executable semantics of a language poses also some challenges that do not rise for arbitrary semantics. For instance, the semantics in [7] 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 also implemented a program able to transform a non-reversible semantics into a reversible one, providing an implementation of the general method described in [9]. Again, ensuring that the method can be actually executed poses some challenges. E.g., [9] 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. Rollback semantics have been proved of interest for debugging techniques, we find examples in [5, 12, 7]. The rollback semantics presented here differs from the ones presented in the past as it is agnostic of the underlying formalism and it is finer, as it allows the user to reach every state that the system traversed.

Let us now discuss possible future improvements for the presented work. First, one could investigate ways to optimize the implementation of the semantics, so to be able to simulate more computationally expensive Erlang computations. For instance, right now the top-level environment also includes the bindings from the lower levels, and this increases the needed memory. Second, one could support further primitives to widen the set of executable programs. In doing so it is important to make sure that the causal-dependencies captured by the producer-consumer model used in [9] are appropriate - for example the model is not well-suited to capture dependencies due to shared memory.

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] E. Giachino, I. Lanese, and C. A. Mezzina. Causal-consistent reversible debugging. In *FASE*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2014.

[7] J. J. González-Abril and G. Vidal. Causal-consistent reversible debugging: Improving cauder. In *PADL*, volume 12548 of *LNCS*, pages 145–160. Springer, 2021.

[8] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI 73*, 1973.

[9] I. Lanese and D. Medic. A general approach to derive uncontrolled reversible semantics. In *CONCUR*, volume 171, pages 33:1–33:24, 2020.

[10] I. Lanese, C. A. Mezzina, and J. Stefani. Controlled reversibility and compensations. In *RC 2012*, volume 7581 of *LNCS*, pages 233–240. Springer, 2012.

[11] I. Lanese, C. A. Mezzina, and J. Stefani. Reversibility in the higher-order π-calculus. *Theor. Comput. Sci.*, 625:25–84, 2016.

[12] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. A theory of reversibility for Erlang. *J. Log. Algebraic Methods Program.*, 100:71–97, 2018.

[13] 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.

[14] All about maude, 2007.

[15] H. C. Melgratti, C. A. Mezzina, and I. Ulidowski. Reversing place transition nets. *Log. Methods Comput. Sci.*, 16(4), 2020.

[16] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *CONCUR 96*, pages 331–372, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[17] M. Neuhäußer and T. Noll. Abstraction and model checking of Core Erlang programs in Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):147–163, Jul 2007.

[18] 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.

[19] A. Philippou and K. Psara. Reversible computation in Petri nets. In *RC*, pages 84–101, 2018.

[20] I. C. C. Phillips and I. Ulidowski. Reversing algebraic process calculi. *J. Log. Algebraic Methods Program.*, 73(1-2):70–96, 2007.

[21] Automatic generation of reversible semantics in Maude. `https://github.com/gfabbretti8/formalization-in-maude-of-erlang`.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Montbonnot Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399