Automatic Generation of a Reversible Semantics for Erlang in Maude

Giovanni Fabbretti $^1$, Ivan Lanese $^2$, Jean-Bernard Stefani$^1$

$^1$SPADES Team, INRIA and $^2$Univ. of Bologna

DCORE, 20-01-2022
Reversibility is the ability to execute a program not only in the canonical forward direction but in a backward manner as well.

Thanks to its growing interest reversibility has been investigated in various settings, like Petri-nets, ccs, pi-calculus, Erlang, $\mu$-klaim, etc.
The majority of the reversible semantics have always been devised ad hoc. A process that is error-prone, requires time and doesn’t scale well.

Lanese et al. recently proposed a general method to produce a reversible semantics given a non-reversible one. The pros are symmetric to the cons listed above.
The general method proposed by Lanese et al. lacked an implementation which we propose here, by using the Maude programming language.

To test it we also devised a novel formalization of Erlang in Maude, hence our two main contributions are:

- A new mechanized formalization of Erlang in Maude
- A concrete implementation in Maude that generates reversible semantics
Table of Contents

1 Introduction
2 Background
3 Contribution
4 Conclusion
Before diving into the details of our contribution let us discuss the various ingredients required:

- Erlang
- Maude
- The general method
Erlang
Erlang, developed in 1986 by Ericsson, is a concurrent, distributed, functional programming language, based on message passing.

It is probably the most popular programming language that implements the actor model.

Here we are mostly interested in the main concurrent primitives:

- spawn
- send
- receive
Maude
Maude is a programming language that efficiently implements a rewriting logic.

A rewriting logic is a tuple \((\Sigma, E, R)\) where:

- \(\Sigma\) is a collection of typed operators
- \(E\) is a set of equations
- \(R\) is a set of rewriting rules
Maude: an example

fmod BOOL is
  sort Bool .

  op true : -> Bool .
  op false : -> Bool .
  op _and_ : Bool Bool -> Bool .

  var A : Bool

  eq true and A = A .
  eq false and A = false .
  eq A and A = A .

endfm
General Method
The general method expects in input a reduction semantics together with its syntax and as a result produces a reversible version of it, where causal dependencies are captured in terms of resources produced and consumed.

\[ P \]

\[ t \rightarrow t' \]

Above \( t \) is consumed to produce \( t' \).
Syntax

The reduction semantics must have a two level syntax. On the lower level there are no constraints, the upper level must be of the following shape.

\[ S ::= P \mid \mathit{op}_n(S_1, \ldots, S_n) \mid 0 \]
The rules of the reduction semantics must fit the following schemas.

\[
\begin{align*}
\text{(Scm-Act)} & \quad \frac{P_1 | \ldots | P_n \rightarrow T[Q_1, \ldots, Q_m]}{P_1 | \ldots | P_n \rightarrow T[Q_1, \ldots, Q_m]} \\
\text{(Scm-Opn)} & \quad \frac{S_i \rightarrow S_i'}{op_n(S_0, \ldots, S_i, \ldots, S_n) \rightarrow op_n(S_0, \ldots, S_i', \ldots, S_n)} \\
\text{(Eqv)} & \quad \frac{S \equiv_c S'}{S \rightarrow S_1 \quad S_1 \equiv_c S_1'} \\
\text{(Par)} & \quad \frac{S \rightarrow S'}{S \mid S_1 \rightarrow S_1'}
\end{align*}
\]
Keys and Memories

To make the semantics reversible we resort to the use of keys and memories.

Keys are attached to each entity of the lower level and are used to uniquely distinguish them.

Memories are produced each time a step forward is taken, they are used to bind two states of the system and to store configurations so that they can be restored later on.
The reversible syntax has the following shape.

\[
R ::= k : P | op_n(R_1, \ldots, R_n) | 0 | [R ; C]
\]
\[
C ::= T[k_1 : \bullet_1, \ldots, k_m : \bullet_m]
\]
The forward reversible rules of the reduction semantics have the following shape.

\[(F-\text{SCM-Act})\quad \frac{k_1 : P_1 | \ldots | k_n : P_n \rightarrow T[j_1 : Q_1, \ldots, j_m : Q_m] | [k_1 : P_1 | \ldots | k_n : P_n ; T[j_1 : \bullet_1, \ldots, j_m : \bullet_m]]}{j_1, \ldots, j_m \text{ are fresh keys}}\]

\[(F-\text{SCM-Opn})\quad \frac{R_i \rightarrow R_i'}{\left(\text{keys}(R_i') \setminus \text{keys}(R_i)\right) \cap (\text{keys}(R_0, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n) = \emptyset \quad \text{op}_n(R_0, \ldots, R_{i-1}, R_i, \ldots, R_n) \rightarrow \text{op}_n(R_0, \ldots, R_i', \ldots, R_n)}\]

\[(F-\text{EQV})\quad \frac{R \equiv_c R' \quad R \rightarrow R_1 \quad R_1 \equiv_c R_1'}{R' \rightarrow R_1'}\]
Backward Rules

The backward reversible rules of the reduction semantics have the following shape.

\[
(B-SCM-Act) \quad \frac{\mu = [k_1 : P_1 | \ldots | k_n : P_n ; T[j_1 : \bullet_1, \ldots, j_m : \bullet_m]]}{T[j_1 : Q_1, \ldots, j_m : Q_m] | \mu \leadsto k_1 : P_1 | \ldots | k_n : P_n}
\]

\[
(B-SCM-OPN) \quad \frac{R_i' \leadsto R_i}{op_n(R_0, \ldots, R_i', \ldots, R_n) \leadsto op_n(R_0, \ldots, R_i, \ldots, R_n)}
\]

\[
(B-EQV) \quad \frac{R \equiv_c R' \quad R \leadsto R_1 \quad R_1 \equiv_c R_1'}{R' \leadsto R_1'}
\]
A Concrete Example

(non-reversible rule)
\[ \langle p_1, \theta, p_2 ! hello \rangle \rightarrow \langle p_1, \theta, hello \rangle \mid \langle p_1, p_2, hello \rangle \]

(forward reversible rule)
\[ k : \langle p_1, \theta, p_2 ! hello \rangle \rightarrow \]
\[ k_1 : \langle p_1, \theta, hello \rangle \mid k_2 : \langle p_1, p_2, hello \rangle \mid [k : \langle p_1, \theta, p_2 ! hello \rangle ; k_1 : \bullet_1 \mid k_2 : \bullet_2] \]

(backward rule)
\[ k_1 : \langle p_1, \theta, hello \rangle \mid k_2 : \langle p_1, p_2, hello \rangle \mid [k : \langle p_1, \theta, p_2 ! hello \rangle ; k_1 : \bullet_1 \mid k_2 : \bullet_2] \]
\[ \sim \rightarrow k : \langle p_1, \theta, p_2 ! hello \rangle \]
Table of Contents

1 Introduction
2 Background
3 Contribution
4 Conclusion
Erlang semantics
A Two Layer Semantics

We implemented the Erlang semantics as a two layer semantics:

- A set of equations for the expression semantics, defined over the tuples <LABEL, ENV, EXPR>
- A set of rewriting rules for the system semantics defined over system configurations, i.e., processes running in parallel with messages
Example of Equations

\[
eq \text{[match]} : \\
\quad \langle \text{REQLABEL}, \text{ENVSTACK}, \text{GVALUE} = \text{GVALUE} \rangle = \\
\quad \langle \tau, \text{ENVSTACK}, \text{GVALUE} \rangle .
\]

\[
\text{ceq [receive]} : \\
\quad \langle \text{req-receive(PAYLOAD), ENV : ENVSTACK, receive CLSEQ end} \rangle = \\
\quad \langle \text{received, ENV'} : (\text{ENV : ENVSTACK}), \text{begin EXSEQ end} \rangle \\
\quad \text{if } \#\text{entityMatchSuccess(EXSEQ | ENV')} := \\
\quad \#\text{entityMatch(CLSEQ | PAYLOAD | ENV)} .
\]
While managing expressions we need to be careful as a naive handling could cause unwanted effects.

\[ pow\_and\_sub(N, M) \rightarrow Z = N \times N, Z - M. \]

\[ X = pow\_and\_sub(N, M) \Rightarrow_{\text{wrong}} X = Z = N \times N, Z - M. \]

\[ X = pow\_and\_sub(N, M) \Rightarrow X = \text{begin } Z = N \times N, Z - M \text{ end.} \]
Rewriting Rules

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 [sys-self] :
< P | exp: EXSEQ, env-stack: ENV, ASET > =>
< P | exp: EXSEQ’, env-stack: ENV’, ASET >
if < tau, ENV’, EXSEQ’ > :=
< self(P), ENV, EXSEQ > .
Automatic Generation of the Reversible Semantics
Input Format: Entities

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

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

mod SYSTEM is
...
sorts Memory Context Sys .

subsort EntityWithKey Memory Context < Sys .

op @:_ : Key -> Context [ctor] .
op [_;_] : Sys Context -> Memory [ctor frozen .. ] .

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

...
endm
Rewriting Rules: Send and Forward Reversible Send

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 [fwd 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 > .

Giovanni Fabbretti, Ivan Lanese, Jean-Bernard Stefani
Automatic Generation of a Reversible Semantics for Erlang in Maude
Rewriting Rules: Backward Send

\[
\text{crl [bwd sys-send]}:
\begin{align*}
&< \text{sender: P, receiver: DEST, payload: GVALUE} > * \text{key(0 L)} || \\
&< P \mid \text{exp: EXSEQ', env-stack: ENV', ASET} > * \text{key(1 L)} || \\
&[< P \mid \text{ASET, exp: EXSEQ, env-stack: ENV} > * \text{key(L)} ; \\
@: \text{key(0 L)} || @: \text{key(1 L)}] \Rightarrow \\
&< P \mid \text{ASET, exp: EXSEQ, env-stack: ENV} > * \text{key(L)}
\end{align*}
\]
Table of Contents

1. Introduction
2. Background
3. Contribution
4. Conclusion
Ongoing Work

Figure: Schema of the proof of correctness.
Future Work

Still many directions to be explored, one could for instance:

- Optimize the implementation of the Erlang semantics
- Widen the set of supported primitives
- Introduce support for read dependencies in the general method
Thank you for the attention!