Causal-Consistent Replay Debugging for Message Passing Programs

Ivan Lanese, Adrián Palacios & Germán Vidal

Università Bologna & Universitat Politècnica de València

(paper presented at FORTE 2019)

Roadmap

Goal: debugging technique for concurrent programs

1. A simple (eager) functional language with message-passing concurrency (subset of Erlang)
2. Logging semantics: records the order in which messages are delivered to each process
3. Reversible semantics: allows us to explore back and forth the recorded execution in a causal-consistent way (i.e., an action cannot be undone until all the actions that depend on it have already been undone)
4. Controlled (replay/rollback) semantics: where the user can specify the actions to replay/undo → CauDEr

Roadmap

Goal: debugging technique for concurrent programs

1. A simple (eager) functional language with message-passing concurrency (subset of Erlang)
2. Logging semantics: records the order in which messages are delivered to each process
3. Reversible semantics: allows us to explore back and forth the recorded execution in a causal-consistent way (i.e., an action cannot be undone until all the actions that depend on it have already been undone)
4. Controlled (replay/rollback) semantics: where the user can specify the actions to replay/undo → CauDEr

Roadmap

Goal: debugging technique for concurrent programs

1. A simple (eager) functional language with message-passing concurrency (subset of Erlang)
2. Logging semantics: records the order in which messages are delivered to each process
3. Reversible semantics: allows us to explore back and forth the recorded execution in a causal-consistent way (i.e., an action cannot be undone until all the actions that depend on it have already been undone)
4. Controlled (replay/rollback) semantics: where the user can specify the actions to replay/undo → CauDEr

Roadmap

Goal: debugging technique for concurrent programs

1. A simple (eager) functional language with message-passing concurrency (subset of Erlang)
2. Logging semantics: records the order in which messages are delivered to each process
3. Reversible semantics: allows us to explore back and forth the recorded execution in a causal-consistent way (i.e., an action cannot be undone until all the actions that depend on it have already been undone)
4. Controlled (replay/rollback) semantics: where the user can specify the actions to replay/undo → CauDEr

Roadmap

Goal: debugging technique for concurrent programs

1. A simple (eager) functional language with message-passing concurrency (subset of Erlang)
2. Logging semantics: records the order in which messages are delivered to each process
3. Reversible semantics: allows us to explore back and forth the recorded execution in a causal-consistent way (i.e., an action cannot be undone until all the actions that depend on it have already been undone)
4. Controlled (replay/rollback) semantics: where the user can specify the actions to replay/undo → CauDEr

The language

We consider a simple functional and concurrent programming language similar to Erlang

- No shared memory, only message passing (asynchronous communication)
- Each process has a local queue (mailbox)
- A system is a collection of processes

Sequential Erlang in 5 examples

append/2

```
append([H|T], L) -> [H|append(T,L)];
append([],    L) -> L.
```

Variables start with an uppercase letter

Function names and atoms (i.e., constants) start with a lowercase letter

Alternative definition:

append/2

```
append(A,B) -> case A of
                 [H|T]  -> [H|append(T,L)];
                 []     -> L
               end.
```

Sequential Erlang in 5 examples

append/2

```
append([H|T], L) -> [H|append(T,L)];
append([],    L) -> L.
```

Variables start with an uppercase letter

Function names and atoms (i.e., constants) start with a lowercase letter

Alternative definition:

append/2

```
append(A,B) -> case A of
                 [H|T]    -> [H|append(T,L)];
                 []       -> L
               end.
```

Sequential Erlang in 5 examples

toint/1

```
toint({s,N})  ->  int(N)+1;
toint(zero)   ->  0.
```

E.g., `toint({s,{s,{s,zero}}})` evaluates to `3`

No user-defined algebraic data types (so we cannot write `s(s(s(zero)))`)

Main data types: numbers, atoms, lists, and tuples

Sequential Erlang in 5 examples

factorial/1

```
factorial(N) when N > 0  ->  N * factorial(N - 1);
factorial(1)             ->  0.
```

Besides pattern matching, we can have guards

Only built-in functions are allowed in guards

Sequential Erlang in 5 examples

minmax/1

```
minmax(L) -> Min = lists:min(L),
             Max = lists:max(L),
             {Min,Max}.
```

Sequence $e_1, \ldots, e_n$ evaluates all expressions, returns the evaluation of e_n

Equation `pat = exp` evaluates `exp` and perform pattern matching with `pat`

Sequential Erlang in 5 examples

inclist/1

```
inclist(L) -> lists:map(fun(X) -> X + 1 end, L).
```

Higher-order functions

Anonymous functions

No partial applications

Concurrency features

- spawn/1 and spawn/3: creates a new process as a side-effect and returns the pid of the new process
- self/0: returns the pid of the current process
- pid ! val: sends val to process pid as a side-effect and returns val
- receive ... end: waits for a message that matches some pattern (otherwise, blocks execution) and returns the expression in the selected branch

Concurrent Erlang in 1 example

```
main()     -> S = spawn(fun() -> server([]) end),
              client(S).

client(S)  -> S ! {self(), {add, paper}},
              S ! {self(), {add, pencil}},
              S ! {self(), take},
              receive
                X -> X
              end.

server(L)  -> receive
                {_, {add, Item}} -> server([Item|L]);
                {C, take} -> C ! hd(L), server(tl(L))
              end.
```

From Erlang to Core Erlang

Core Erlang is an intermediate representation used during the compilation of Erlang programs

It is a convenient representation for defining analyses and other tools

Not as readable as Erlang...

From Erlang to Core Erlang

erlang

```
a(42) -> ok;
a(N)  -> M = N + 1, a(M).
```

core erlang

```
'a'/1 = fun(_@c0) ->
    case _@c0 of
      <42> when 'true' -> 'ok'
      <_@c2> when 'true' -> let <_@c3> = call 'erlang':'+'(N, 1)
                              in apply 'a'/1 (_@c3)
    end
```

Essentially: one clause per function, case for pattern matching, let for sequences, apply for function applications, call for built-in calls, etc

From Erlang to Core Erlang

erlang

```
a(42) -> ok;
a(N)  -> M = N + 1, a(M).
```

core erlang

```
'a'/1 = fun(_@c0) ->
    case _@c0 of
      <42> when 'true' -> 'ok'
      <_@c2> when 'true' -> let <_@c3> = call 'erlang':'+'(N, 1)
                            in apply 'a'/1 (_@c3)
    end
```

Essentially: one clause per function, case for pattern matching, let for sequences, apply for function applications, call for built-in calls, etc

Core Erlang syntax

We consider a subset of Core Erlang with this syntax:

$$
\begin{array}{rcl}
\mathit{Module} & ::= & \mathrm{module}\ \mathit{Atom} = \mathit{fun}_1, \ldots, \mathit{fun}_n \\
\mathit{fun} & ::= & \mathit{fname} = \mathrm{fun}\ (X_1, \ldots, X_n) \rightarrow \mathit{expr} \\
\mathit{fname} & ::= & \mathit{Atom} / \mathit{Integer} \\
\mathit{lit} & ::= & \mathit{Atom} \mid \mathit{Integer} \mid \mathit{Float} \mid [\,] \\
\mathit{expr} & ::= & \mathit{Var} \mid \mathit{lit} \mid \mathit{fname} \mid [\mathit{expr}_1 | \mathit{expr}_2] \mid \{\mathit{expr}_1, \ldots, \mathit{expr}_n\} \\
& \mid & \mathrm{call}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \mid \mathrm{apply}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \\
& \mid & \mathrm{case}\ \mathit{expr}\ \mathrm{of}\ \mathit{clause}_1; \ldots; \mathit{clause}_m\ \mathrm{end} \\
& \mid & \mathrm{let}\ \mathit{Var} = \mathit{expr}_1\ \mathrm{in}\ \mathit{expr}_2 \mid \mathrm{receive}\ \mathit{clause}_1; \ldots; \mathit{clause}_n\ \mathrm{end} \\
& \mid & \mathrm{spawn}(\mathit{expr}, [\mathit{expr}_1, \ldots, \mathit{expr}_n]) \mid \mathit{expr}_1\ !\ \mathit{expr}_2 \mid \mathrm{self}() \\
\mathit{clause} & ::= & \mathit{pat}\ \mathrm{when}\ \mathit{expr}_1 \rightarrow \mathit{expr}_2 \\
\mathit{pat} & ::= & \mathit{Var} \mid \mathit{lit} \mid [\mathit{pat}_1 | \mathit{pat}_2] \mid \{\mathit{pat}_1, \ldots, \mathit{pat}_n\}
\end{array}
$$

Core Erlang syntax

We consider a subset of Core Erlang with this syntax:

$$
\begin{array}{rcl}
\mathit{Module} & ::= & \mathsf{module}\ \mathit{Atom} = \mathit{fun}_1, \ldots, \mathit{fun}_n \\
\mathit{fun} & ::= & \mathit{fname} = \mathsf{fun}\ (X_1, \ldots, X_n) \rightarrow \mathit{expr} \\
\mathit{fname} & ::= & \mathit{Atom}/\mathit{Integer} \\
\mathit{lit} & ::= & \mathit{Atom} \mid \mathit{Integer} \mid \mathit{Float} \mid [\,] \\
\mathit{expr} & ::= & \mathit{Var} \mid \mathit{lit} \mid \mathit{fname} \mid [\mathit{expr}_1 | \mathit{expr}_2] \mid \{\mathit{expr}_1, \ldots, \mathit{expr}_n\} \\
& \mid & \mathsf{call}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \mid \mathsf{apply}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \\
& \mid & \mathsf{case}\ \mathit{expr}\ \mathsf{of}\ \mathit{clause}_1; \ldots; \mathit{clause}_m\ \mathsf{end} \\
& \mid & \mathsf{let}\ \mathit{Var} = \mathit{expr}_1\ \mathsf{in}\ \mathit{expr}_2 \mid \mathsf{receive}\ \mathit{clause}_1; \ldots; \mathit{clause}_n\ \mathsf{end} \\
& \mid & \mathsf{spawn}(\mathit{expr}, [\mathit{expr}_1, \ldots, \mathit{expr}_n]) \mid \mathit{expr}_1\ !\ \mathit{expr}_2 \mid \mathsf{self}() \\
\mathit{clause} & ::= & \mathit{pat}\ \mathsf{when}\ \mathit{expr}_1 \rightarrow \mathit{expr}_2 \\
\mathit{pat} & ::= & \mathit{Var} \mid \mathit{lit} \mid [\mathit{pat}_1 | \mathit{pat}_2] \mid \{\mathit{pat}_1, \ldots, \mathit{pat}_n\}
\end{array}
$$

Some preliminary definitions

Definition (process)

//no local queue!

A process is a triple $\langle p, \theta, e \rangle$ where

- p is the pid of the process
- θ is an environment
- e is the expression to be reduced

Definition (system)

A system is denoted by $\Gamma; \Pi$, where

- Γ models the network & local queues (global mailbox)
- Π is a pool of processes

Γ is a multiset of triples $(sender\_pid, target\_pid, message)$

We often use $\Gamma; \langle p, \theta, e \rangle \ \& \ \Pi$ to denote an arbitrary system

Some preliminary definitions

Definition (process)

//no local queue!

A process is a triple $\langle p, \theta, e \rangle$ where

- p is the pid of the process
- θ is an environment
- e is the expression to be reduced

Definition (system)

A system is denoted by $\Gamma; \Pi$, where

- Γ models the network & local queues (global mailbox)
- Π is a pool of processes

Γ is a multiset of triples $(sender\_pid, target\_pid, message)$

We often use $\Gamma; \langle p, \theta, e \rangle$ & Π to denote an arbitrary system

Core Erlang Semantics

Two-level (reduction) semantics:

- Semantics of expressions (sequential & concurrent)
- Semantics of systems

For concurrent actions, we face the following problems:

1. we don't know the result of the actions (fresh variables)
2. we must perform side effects (labels)

Labels

- At expression level, transitions for concurrent actions are labelled with enough information
- At system level, labels are used to perform the associated actions

Core Erlang Semantics

Two-level (reduction) semantics:

- Semantics of expressions (sequential & concurrent)
- Semantics of systems

For concurrent actions, we face the following problems:

1. we don't know the result of the actions (fresh variables)
2. we must perform side effects (labels)

Labels

- At expression level, transitions for concurrent actions are labelled with enough information
- At system level, labels are used to perform the associated actions

Expression semantics: sequential expressions

$$(Var)\ \frac{}{\theta, X \xrightarrow{\tau} \theta, \theta(X)} \qquad (Tuple)\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e'_i}{\theta, \{\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}\} \xrightarrow{\ell} \theta', \{\overline{v_{1,i-1}}, e'_i, \overline{e_{i+1,n}}\}}$$

$$(List1)\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e'_1}{\theta, [e_1|e_2] \xrightarrow{\ell} \theta', [e'_1|e_2]} \qquad (List2)\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e'_2}{\theta, [v_1|e_2] \xrightarrow{\ell} \theta', [v_1|e'_2]}$$

$$(Let1)\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e'_1}{\theta, \text{let } X = e_1 \text{ in } e_2 \xrightarrow{\ell} \theta', \text{let } X = e'_1 \text{ in } e_2} \qquad (Let2)\ \frac{}{\theta, \text{let } X = v \text{ in } e \xrightarrow{\tau} \theta[X \mapsto v], e}$$

$$(Case1)\ \frac{\theta, e \xrightarrow{\ell} \theta', e'}{\begin{array}{l}\theta, \text{case } e \text{ of } cl_1; \ldots; cl_n \text{ end} \\ \xrightarrow{\ell} \theta', \text{case } e' \text{ of } cl_1; \ldots; cl_n \text{ end}\end{array}} \qquad (Case2)\ \frac{\text{match}(v, cl_1, \ldots, cl_n) = \langle \theta_i, e_i \rangle}{\theta, \text{case } v \text{ of } cl_1; \ldots; cl_n \text{ end} \xrightarrow{\tau} \theta\theta_i, e_i}$$

$$(Apply1)\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e'_i \quad i \in \{1, \ldots, n\}}{\theta, \text{apply } a/n\,(\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \xrightarrow{\ell} \theta', \text{apply } a/n\,(\overline{v_{1,i-1}}, e'_i, \overline{e_{i+1,n}})}$$

$$(Apply2)\ \frac{\mu(a/n) = \text{fun } (X_1, \ldots, X_n) \to e}{\theta, \text{apply } a/n\,(v_1, \ldots, v_n) \xrightarrow{\tau} \{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}, e}$$

Sending a message

(expression semantics)

$$(\mathit{Send1})\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1 \,!\, e_2 \xrightarrow{\ell} \theta', e_1' \,!\, e_2} \qquad \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, v_1 \,!\, e_2 \xrightarrow{\ell} \theta', v_1 \,!\, e_2'}$$

$$(\mathit{Send2})\ \frac{}{\theta, v_1 \,!\, v_2 \xrightarrow{\mathsf{send}(v_1, v_2)} \theta, v_2}$$

(system semantics)

$$(\mathit{Send})\ \frac{\theta, e \xrightarrow{\mathsf{send}(p', v)} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma \cup \{(p, p', v)\}; \langle p, \theta', e'\rangle \& \Pi}$$

Sending a message

(expression semantics)

$$(Send1)\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1 \,!\, e_2 \xrightarrow{\ell} \theta', e_1' \,!\, e_2} \qquad \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, v_1 \,!\, e_2 \xrightarrow{\ell} \theta', v_1 \,!\, e_2'}$$

$$(Send2)\ \frac{}{\theta, v_1 \,!\, v_2 \xrightarrow{\mathsf{send}(v_1, v_2)} \theta, v_2}$$

(system semantics)

$$(Send)\ \frac{\theta, e \xrightarrow{\mathsf{send}(p', v)} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma \cup \{(p, p', v)\}; \langle p, \theta', e'\rangle \& \Pi}$$

Spawning a process

(expression semantics)

$$(\mathit{Spawn})\ \frac{}{\theta, \mathsf{spawn}(a/n, [v_1, \ldots, v_n]) \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta, \kappa}$$

(system semantics)

$$(\mathit{Spawn})\ \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma; \langle p, \theta', e'\{\kappa \mapsto p'\}\rangle \& \langle p', \theta', \mathsf{apply}\ a/n\ (\overline{v_n})\rangle \& \Pi}$$

Spawning a process

(expression semantics)

$$(\mathit{Spawn})\ \frac{}{\theta, \mathsf{spawn}(a/n, [v_1, \ldots, v_n]) \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta, \kappa}$$

(system semantics)

$$(\mathit{Spawn})\ \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, \theta, e \rangle \& \Pi \hookrightarrow \Gamma; \ \langle p, \theta', e'\{\kappa \mapsto p'\} \rangle \& \langle p', \theta', \mathsf{apply}\ a/n\ (\overline{v_n}) \rangle \& \Pi}$$

Receiving a message

(expression semantics)

$$(\mathit{Receive})\ \frac{}{\theta, \mathsf{receive}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta, \kappa}$$

(system semantics)

$$(\mathit{Receive})\ \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\Gamma \cup \{(p', p, v)\}; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma; \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \& \Pi}$$

Receiving a message

(expression semantics)

$$(\mathit{Receive})\ \dfrac{}{\theta, \mathsf{receive}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta, \kappa}$$

(system semantics)

$$(\mathit{Receive})\ \dfrac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\Gamma \cup \{(p', p, v)\}; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma; \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \& \Pi}$$

Logging semantics

In concurrent languages, replaying a particular computation might be difficult (even impossible) given the nondeterminism of the language

We tag messages with unique identifiers

$$v \mapsto \{v, \ell\}, \quad \text{where } \ell \text{ is fresh}$$

A log $\mathcal{L}(d)$ of a derivation d is a sequence of items spawn(p), send(ℓ) or rec(ℓ) for each process in d

(logs are local to each process)

Logging semantics

In concurrent languages, replaying a particular computation might be difficult (even impossible) given the nondeterminism of the language

We tag messages with unique identifiers

$$v \;\mapsto\; \{v, \ell\}, \quad \text{where } \ell \text{ is fresh}$$

A log $\mathcal{L}(d)$ of a derivation d is a sequence of items spawn(p), send(ℓ) or rec(ℓ) for each process in d

(logs are local to each process)

(*Seq*)

$$\frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{seq}} \Gamma; \langle p, \theta', e'\rangle \mid \Pi}$$

(*Send*)

$$\frac{\theta, e \xrightarrow{\mathsf{send}(p', v)} \theta', e' \text{ and } \ell \text{ is a fresh symbol}}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{send}(\ell)} \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \theta', e'\rangle \mid \Pi}$$

(*Receive*)

$$\frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \;\text{ and }\; \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{rec}(\ell)} \Gamma; \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid \Pi}$$

(*Spawn*)

$$\frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \;\text{ and }\; p' \text{ is a fresh pid}}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{spawn}(p')} \Gamma; \langle p, \theta', e'\{\kappa \mapsto p'\}\rangle \mid \langle p', id, \mathsf{apply}\ a/n\ (\overline{v_n})\rangle \mid \Pi}$$

(*Self*)

$$\frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{self}} \Gamma; \langle p, \theta', e'\{\kappa \mapsto p\}\rangle \mid \Pi}$$

(implemented by a program instrumentation)

Causally equivalent derivations

$t_1 = (s_1 \hookrightarrow_{p_1,r_1} s_1')$ happened before $t_2 = (s_2 \hookrightarrow_{p_2,r_2} s_2')$, in symbols $t_1 \rightsquigarrow t_2$, if one of the following conditions holds:

- $p_1 = p_2$ and t_1 comes before t_2;
- $r_1 = \mathsf{spawn}(p)$ and $p_2 = p$;
- $r_1 = \mathsf{send}(\ell)$ and $r_2 = \mathsf{rec}(\ell)$.

t_1 and t_2 are independent if $t_1 \not\rightsquigarrow t_2$ and $t_2 \not\rightsquigarrow t_1$

d_1 and d_2 are causally equivalent ($d_1 \approx d_2$) if d_1 can be obtained from d_2 by switching consecutive independent transitions

Given (coinitial) derivations d_1 and d_2, $\mathcal{L}(d_1) = \mathcal{L}(d_2)$ iff $d_1 \approx d_2$

Causally equivalent derivations

$t_1 = (s_1 \hookrightarrow_{p_1,r_1} s'_1)$ happened before $t_2 = (s_2 \hookrightarrow_{p_2,r_2} s'_2)$, in symbols $t_1 \rightsquigarrow t_2$, if one of the following conditions holds:

- $p_1 = p_2$ and t_1 comes before t_2;
- $r_1 = \mathrm{spawn}(p)$ and $p_2 = p$;
- $r_1 = \mathrm{send}(\ell)$ and $r_2 = \mathrm{rec}(\ell)$.

t_1 and t_2 are independent if $t_1 \not\rightsquigarrow t_2$ and $t_2 \not\rightsquigarrow t_1$

d_1 and d_2 are causally equivalent ($d_1 \approx d_2$) if d_1 can be obtained from d_2 by switching consecutive independent transitions

Given (coinitial) derivations d_1 and d_2, $\mathcal{L}(d_1) = \mathcal{L}(d_2)$ iff $d_1 \approx d_2$

Reversible Semantics

Processes have the form $\langle p, \omega, h, \theta, e \rangle$
with ω a *log* and h a *history*

A history h is a sequence of terms headed by constructors seq, send, rec, spawn, and self, and whose arguments are the information required to (deterministically) undo the step

Uncontrolled forward semantics

(*Send*)

$$\frac{\theta, e \xrightarrow{\mathsf{send}(p',v)} \theta', e'}{\begin{array}{l}\Gamma; \langle p, \mathsf{send}(\ell):\omega, h, \theta, e\rangle \mid \Pi \\ \rightharpoonup_{p,\mathsf{send}(\ell),\{\mathsf{s},\ell\Uparrow\}} \Gamma \cup \{(p, p', \{v,\ell\})\}; \\ \qquad \langle p, \omega, \mathsf{send}(\theta, e, p', \{v,\ell\}):h, \theta', e'\rangle \mid \Pi\end{array}}$$

(*Receive*)

$$\frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta', e' \text{ and } \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\begin{array}{l}\Gamma \cup \{(p', p, \{v,\ell\})\}\langle p, \mathsf{rec}(\ell):\omega, h, \theta, e\rangle \mid \Pi \\ \rightharpoonup_{p,\mathsf{rec}(\ell),\{\mathsf{s},\ell\Downarrow\}} \Gamma; \langle p, \omega, \mathsf{rec}(\theta, e, p', \{v,\ell\}):h, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid \Pi\end{array}}$$

(*Spawn*)

$$\frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa,a/n,[\overline{v_n}])} \theta', e' \text{ and } \omega' = \mathsf{trace}(d, p')}{\begin{array}{l}\Gamma; \langle p, \mathsf{spawn}(p'):\omega, h, \theta, e\rangle \mid \Pi \\ \rightharpoonup_{p,\mathsf{spawn}(p'),\{\mathsf{s},\mathsf{sp}_{p'}\}} \Gamma; \langle p, \omega, \mathsf{spawn}(\theta, e, p'):h, \theta', e'\{\kappa \mapsto p'\}\rangle \\ \qquad \mid \langle p', \omega', [\,], id, \mathsf{apply}\ a/n\ (\overline{v_n})\rangle \mid \Pi\end{array}}$$

Uncontrolled forward semantics

(*Receive*)

$$\frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \;\text{ and }\; \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\begin{array}{l}\Gamma \cup \{(p', p, \{v, \ell\})\}\langle p, \mathsf{rec}(\ell)\!:\!\omega, h, \theta, e\rangle \mid \Pi \\ \rightharpoonup_{p,\mathsf{rec}(\ell),\{\mathsf{s},\ell^{\Downarrow}\}} \Gamma; \langle p, \omega, \mathsf{rec}(\theta, e, p', \{v, \ell\})\!:\!h, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid \Pi\end{array}}$$

Uncontrolled backward semantics

$$(\overline{Send})\quad \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \omega, \mathsf{send}(\theta, e, p', \{v, \ell\}) : h, \theta', e' \rangle \mid \Pi \leftharpoondown_{p, \mathsf{send}(\ell), \{\mathsf{s}, \ell \Uparrow\}} \Gamma; \langle p, \mathsf{send}(\ell) : \omega, h, \theta, e \rangle \mid \Pi$$

$$(\overline{Receive})\quad \Gamma; \langle p, \omega, \mathsf{rec}(\theta, e, p', \{v, \ell\}) : h, \theta', e' \rangle \mid \Pi \leftharpoondown_{p, \mathsf{rec}(\ell), \{\mathsf{s}, \ell \Downarrow\} \cup \mathcal{V}} \Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \mathsf{rec}(\ell) : \omega, h, \theta, e \rangle \mid \Pi$$

$$\text{where } \mathcal{V} = \mathcal{D}om(\theta') \backslash \mathcal{D}om(\theta)$$

$$(\overline{Spawn})\quad \Gamma; \langle p, \omega, \mathsf{spawn}(\theta, e, p') : h, \theta', e' \rangle \mid \langle p', \omega', [\,], id, e'' \rangle \mid \Pi \leftharpoondown_{p, \mathsf{spawn}(p'), \{\mathsf{s}, \mathsf{sp}_{p'}\}} \Gamma; \langle p, \mathsf{spawn}(p') : \omega, h, \theta, e \rangle \mid \Pi$$

Uncontrolled backward semantics

$$(\overline{Receive}) \quad \begin{array}{l} \Gamma; \langle p, \omega, \mathsf{rec}(\theta, e, p', \{v, \ell\}) : h, \theta', e' \rangle \mid \Pi \\ \quad \leftharpoondown_{p, \mathsf{rec}(\ell), \{\mathsf{s}, \ell^{\Downarrow}\} \cup \mathcal{V}} \Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \mathsf{rec}(\ell) : \omega, h, \theta, e \rangle \mid \Pi \\ \hfill \text{where } \mathcal{V} = \mathcal{D}om(\theta') \backslash \mathcal{D}om(\theta) \end{array}$$

Some results...

Coinitial derivations are cofinal
iff they are causally equivalent

Misbehaviors are preserved
by all causally equivalent derivations

Controlled replay/rollback semantics

We allow the user to start a replay/rollback until a particular action is performed, e.g.,

- $\{p, \mathsf{s}\}$: one step backward/forward of process p
- $\{p, \ell^{\Uparrow}\}$: a backward/forward derivation of process p up to the sending of the message tagged with ℓ
- $\{p, \ell^{\Downarrow}\}$: a backward/forward derivation of process p up to the reception of the message tagged with ℓ
- $\{p, \mathsf{sp}_{p'}\}$: a backward/forward derivation of process p up to the spawning of the process with pid p'
- $\{p, X\}$: a backward derivation of process p up to the introduction of variable X
- ...

Controlled semantics takes a stack of requests (initially one)

It is defined as a layer on top of the uncontrolled semantics:

- If a process can perform a step satisfying the request on top of the stack → do it and remove the request
- If a process can perform a step but it doesn't satisfy the request → update the system but keep the request
- If a step on the process is not possible → track dependencies and add new requests on top of the stack

Controlled semantics takes a stack of requests (initially one)

It is defined as a layer on top of the uncontrolled semantics:

- If a process can perform a step satisfying the request on top of the stack → do it and remove the request
- If a process can perform a step but it doesn't satisfy the request → update the system but keep the request
- If a step on the process is not possible → track dependencies and add new requests on top of the stack

Controlled semantics takes a stack of requests (initially one)

It is defined as a layer on top of the uncontrolled semantics:

- If a process can perform a step satisfying the request on top of the stack $\rightarrow$ do it and remove the request
- If a process can perform a step but it doesn't satisfy the request $\rightarrow$ update the system but keep the request
- If a step on the process is not possible $\rightarrow$ track dependencies and add new requests on top of the stack

Controlled semantics takes a stack of requests (initially one)

It is defined as a layer on top of the uncontrolled semantics:

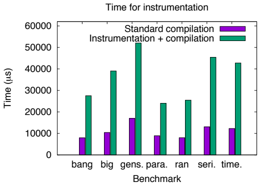
- If a process can perform a step satisfying the request on top of the stack → do it and remove the request
- If a process can perform a step but it doesn't satisfy the request → update the system but keep the request
- If a step on the process is not possible → track dependencies and add new requests on top of the stack

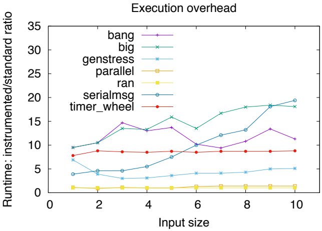
Reversible debugging

Two components: code instrumentation (logging) + causal-consistent reversible debugger (CauDEr)

https://github.com/mistupv/tracer

https://github.com/mistupv/cauder/tree/replay





A Note on the implementation

Current prototypes show good potential, but more implementation effort is still required:

- move from Core Erlang to Erlang (or add pretty printing)
- graphical representation of traces
- consider more Erlang features: links, monitors, built-in's, input/output, behaviours, etc

Conclusions & future work

Promising approach for (causal-consistent) reversible debugging of message passing concurrent programs

Most ideas are applicable to other concurrent languages

Some ideas for future work:

- deal with (partially) unknown modules, trusted components, etc
- combine it with program slicing / automatic bug location
- keep improving the implementation

Conclusions & future work

Promising approach for (causal-consistent) reversible debugging of message passing concurrent programs

Most ideas are applicable to other concurrent languages

Some ideas for future work:

- deal with (partially) unknown modules, trusted components, etc
- combine it with program slicing / automatic bug location
- keep improving the implementation

Thanks for your attention!

Questions?