CauDEr: Causal-Consistent Debugging of Erlang

Giovanni Fabbretti [1], Ivan Lanese [2]

[1]SPADES Team, INRIA and [2]Univ. of Bologna

Progetto GNCS, 23-09-2021

Introduction

CauDEr is a reversible causal-consistent debugger for the Erlang programming language.

Distinctive features of CauDEr:

- Reversibility of multi-process systems
- Causal-consistent rollback

Motivations

Concurrent and distributed systems are everywhere and both are well known for their intrinsic difficulties.

Hence we need effective tools while writing code.

The Erlang language

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.

Erlang owes its success to three aspects: the support of concurrency and distribution, the facilities to do error-handling and the OTP libraries.

Some of the supported features

The debugger currently supports a subset of Erlang.
In particular it supports constructs for:

- Concurrency (spawn, send and receive)
- Distribution (creation of new nodes, reading the nodes, remote spawn)

Reversibility

Causal Consistency: before undoing an action we must ensure that all of its consequences, if any, have been undone.

Demo

Demo

Table of Contents

Causal dependencies

We say that there is a dependency between two consecutive actions in two cases:

- they cannot be executed in the opposite order
- by executing them in the opposite order the result would change

Concurrent dependencies

Supported constructs:

- $\text{spawn}/2, 3$
- $\text{send}/2$
- receive

Dependencies:

- An action of a process depends on its (successful) spawn
- A receive depends on its send

Distributed dependencies

Constructs:

- spawn/4
- start/1, 2
- nodes/0

Dependencies:

1. a (successful) spawn on node *nid* depends on the start of *nid*;
2. a (successful) start of node *nid* depends on previous failed spawns on the same node, if any (if we swap the order, the spawn will succeed);
3. a failed start of node *nid* depends on its (successful) start;
4. a nodes reading a set Ω depends on the start of all nids in Ω, if any.

Semantics Structure

Definition (Process)

A process is defined as: $\langle nid, p, h, \theta, e\rangle$ where

- nid is the name of the node
- p is the process id
- h is the process history
- θ is the process environment
- e is the expression to evaluate

Semantics Structure

Definition (System)

A system is defined as $\Gamma; \Pi; \Omega$ where

- Γ is the global mailbox
- Π is a pool of processes
- Ω is the set of connected nodes

Operational Semantics

CauDEr implements three semantics:

- a forward semantics that defines Erlang's behavior and stores information in the history
- a backward semantics that ensures that we undo only actions whose consequences have been already undone
- a rollback semantics which automatically undoes all the consequences of an action

Forward And Backward Semantics

$$(StartS) \quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, nid')} \theta', e' \quad nid' \notin \Omega}{\boxed{\Gamma; \langle nid, p, h, \theta, e \rangle \mid \Pi; \Omega} \rightharpoonup_{p,\mathsf{start}(nid'),\{s, st_{nid'}\}} \Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\{\kappa \mapsto nid'\}\rangle \mid \Pi; \{nid'\} \cup \Omega}$$

$$(\overline{StartS}) \quad \begin{array}{c} \Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e' \rangle \mid \Pi; \Omega \cup \{nid'\} \\ \leftharpoondown_{p,\mathsf{start}(nid'),\{s, st_{nid'}\}} \Gamma; \langle nid, p, h, \theta, e \rangle \mid \Pi; \Omega \\ \text{if } spawns(nid', \Pi) = [\,] \land reads(nid', \Pi) = [\,] \land failed\_starts(nid', \Pi) = [\,] \end{array}$$

Forward And Backward Semantics

$$(StartS)\quad \frac{\boxed{\theta, e \xrightarrow{\mathsf{start}(\kappa, nid')} \theta', e' \quad nid' \notin \Omega}}{\begin{array}{c}\Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \rightharpoonup_{p, \mathsf{start}(nid'), \{s, st_{nid'}\}} \\ \Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\{\kappa \mapsto nid'\}\rangle \mid \Pi; \{nid'\} \cup \Omega\end{array}}$$

$$(\overline{StartS})\quad \begin{array}{c}\Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\rangle \mid \Pi; \Omega \cup \{nid'\} \\ \leftharpoondown_{p, \mathsf{start}(nid'), \{s, st_{nid'}\}} \Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \\ \text{if } spawns(nid', \Pi) = [\,] \land reads(nid', \Pi) = [\,] \land failed\_starts(nid', \Pi) = [\,]\end{array}$$

Forward And Backward Semantics

$$(StartS)\quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, nid')} \theta', e' \quad nid' \notin \Omega}{\Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \rightharpoonup_{p,\mathsf{start}(nid'),\{s, st_{nid'}\}} \boxed{\Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\{\kappa \mapsto nid'\}\rangle \mid \Pi; \{nid'\} \cup \Omega}}$$

$$(\overline{StartS})\quad \begin{array}{c} \Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\rangle \mid \Pi; \Omega \cup \{nid'\} \\ \leftharpoondown_{p,\mathsf{start}(nid'),\{s, st_{nid'}\}} \Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \\ \text{if } spawns(nid', \Pi) = [\,] \wedge reads(nid', \Pi) = [\,] \wedge failed\_starts(nid', \Pi) = [\,] \end{array}$$

Forward And Backward Semantics

$$(StartS)\quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, nid')} \theta', e' \quad nid' \notin \Omega}{\Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \rightharpoonup_{p,\mathsf{start}(nid'),\{s, st_{nid'}\}} \Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\{\kappa \mapsto nid'\}\rangle \mid \Pi; \{nid'\} \cup \Omega}$$

$$(\overline{StartS})\quad \boxed{\Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\rangle \mid \Pi; \Omega \cup \{nid'\}} \leftharpoondown_{p,\mathsf{start}(nid'),\{s, st_{nid'}\}} \Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega$$

$$\text{if } spawns(nid', \Pi) = [\,] \wedge reads(nid', \Pi) = [\,] \wedge failed\_starts(nid', \Pi) = [\,]$$

Forward And Backward Semantics

$$(\mathit{StartS}) \quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, nid')} \theta', e' \quad nid' \notin \Omega}{\begin{array}{c}\Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \rightharpoonup_{p, \mathsf{start}(nid'), \{s, st_{nid'}\}} \\ \Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\{\kappa \mapsto nid'\}\rangle \mid \Pi; \{nid'\} \cup \Omega\end{array}}$$

$$(\overline{\mathit{StartS}}) \quad \begin{array}{c}\Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\rangle \mid \Pi; \Omega \cup \{nid'\} \\ \leftharpoondown_{p, \mathsf{start}(nid'), \{s, st_{nid'}\}} \Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \\ \text{if } \boxed{\mathit{spawns}(nid', \Pi) = [\,] \wedge \mathit{reads}(nid', \Pi) = [\,] \wedge \mathit{failed\_starts}(nid', \Pi) = [\,]}\end{array}$$

Forward And Backward Semantics

$$(\mathit{StartS})\quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, \mathit{nid}')} \theta', e' \quad \mathit{nid}' \notin \Omega}{\begin{array}{c}\Gamma; \langle \mathit{nid}, p, h, \theta, e\rangle \mid \Pi; \Omega \rightharpoonup_{p, \mathsf{start}(\mathit{nid}'), \{s, st_{\mathit{nid}'}\}} \\ \Gamma; \langle \mathit{nid}, p, \mathsf{start}(\theta, e, \mathsf{succ}, \mathit{nid}') : h, \theta', e'\{\kappa \mapsto \mathit{nid}'\}\rangle \mid \Pi; \{\mathit{nid}'\} \cup \Omega\end{array}}$$

$$(\overline{\mathit{StartS}})\quad \begin{array}{c}\Gamma; \langle \mathit{nid}, p, \mathsf{start}(\theta, e, \mathsf{succ}, \mathit{nid}') : h, \theta', e'\rangle \mid \Pi; \Omega \cup \{\mathit{nid}'\} \\ \leftharpoondown_{p, \mathsf{start}(\mathit{nid}'), \{s, st_{\mathit{nid}'}\}} \boxed{\Gamma; \langle \mathit{nid}, p, h, \theta, e\rangle \mid \Pi; \Omega} \\ \text{if } \mathit{spawns}(\mathit{nid}', \Pi) = [\,] \wedge \mathit{reads}(\mathit{nid}', \Pi) = [\,] \wedge \mathit{failed\_starts}(\mathit{nid}', \Pi) = [\,]\end{array}$$

Rollback Semantics

The rollback semantics allows us to reach a past state of the computation of the system, such past state is specified as an action performed by a process.

Some of the considered requests are:

- $\{p, \lambda^{\Downarrow}\}$: the receive of the message uniquely identified by λ;
- $\{p, st_{nid}\}$: the successful start of node nid;
- $\{p, sp_{p'}\}$: the spawn of process p'.

A system in rollback mode is denoted as $[\![\mathcal{S}]\!]_{\Psi}$

Rollback Semantics

$$\frac{\mathcal{S} \leftharpoondown_{p,r,\Psi'} \mathcal{S}' \;\wedge\; \psi \in \Psi'}{[\![\mathcal{S}]\!]_{\{p,\psi\}:\Psi} \rightsquigarrow [\![\mathcal{S}']\!]_{\Psi}} \qquad \frac{\mathcal{S} \leftharpoondown_{p,r,\Psi'} \mathcal{S}' \;\wedge\; \psi \notin \Psi'}{[\![\mathcal{S}]\!]_{\{p,\psi\}:\Psi} \rightsquigarrow [\![\mathcal{S}']\!]_{\{p,\psi\}:\Psi}}$$

$$\frac{\mathcal{S} = \Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \;\wedge\; \mathcal{S} \not\leftharpoondown_{p,r,\Psi'} \;\wedge\; \{p', \psi'\} = \text{bwd\_dep}(\langle nid, p, h, \theta, e\rangle, \mathcal{S})}{[\![\mathcal{S}]\!]_{\{p,\psi\}:\Psi} \rightsquigarrow [\![\mathcal{S}']\!]_{\{p',\psi'\}:\{p,\psi\}:\Psi}}$$

Rollback semantics: dependencies operator

The dependencies operator does pattern matching on the history item and given the system computes the request to undo the consequences.

$$\text{bwd\_dep}(<\_,\_,\text{nodes}(\_,\_,\Omega') : h,\_,\_>,\_;\Pi;\{nid'\}\cup\Omega) \quad = \{parent(nid',\Pi), st_{nid'}\}$$
where $nid' \notin \Omega'$

Table of Contents

Future work

Many features are still to be covered, e.g., links, failures, I/O.
As a side-product of studying how to do reversibility when such features are considered we wish to obtain:

- an abstraction of the language to reason while leaving out technical details
- a compact and readable way to formalize the language (this time with the technical details)
- automatic ways to obtain reversible semantics starting from non-reversible ones (once the dependencies are well-understood)

The end

Thank you for the attention!