Causal-Consistent Debugging and Replay in Core Erlang

Ivan Lanese
Focus research group
Computer Science and Engineering Department
University of Bologna/INRIA
Bologna, Italy

Joint work with Adrian Palacios, German Vidal and Naoki Nishida
Roadmap

- Causal-consistent reversible debugging
- Causal-consistent replay
- Demo (by German)
- Formal specification
- Future directions
Reversible debugging of actor systems

- Debugging is the central topic of the DCore project
- In particular, debugging for actor systems
- Actor systems are concurrent
  - Misbehaviors may depend on the scheduling
  - Bugs may be in a different process than the one showing the misbehavior
- In Dcore, we would like to build on the work we did on CauDEr to tackle the project objectives
- CauDEr is a causal-consistent reversible and replay debugger targeting Erlang
Why Erlang can be good for this project

- German has already presented Erlang and its semantics
- Real language with a simple functional core
  - Let us look at an hello world example
  - Close to Scala + Akka
  - Much simpler than Java + Akka
- Its semantics has already been deeply studied
- CauDEr is a nice starting point
- Potential risk: the DCore proposal had more emphasis on Akka than on Erlang, ANR may complain
CauDEr: an overview

- A causal-consistent reversible debugger for Core Erlang
- Supports the fragment of Core Erlang presented by German
- Written in Erlang
- Includes a tracer to log a concurrent computation in the real execution environment and replay it inside the debugger
- Supported by a formal specification at the level of operational semantics
CauDEr: where to find further information

- CauDEr available at https://github.com/mistupv/cauder
- Tracer available at https://github.com/mistupv/tracer
- Described in a series of papers by (subsets of) Lanese, Nishida, Palacios & Vidal
  - LOPSTR 2016: reversible semantics of Erlang, preliminary version
  - JLAMP 2018: reversible semantics of Erlang
  - FLOPS 2018: CauDEr
  - FORTE 2019: tracer and replay
Causal-consistent reversibility

- Causal-consistent reversibility [Danos & Krivine, CONCUR 2004] is the main notion of reversibility for concurrent systems
  - Any action can be undone, provided that its consequences (if any) are undone beforehand
  - Concurrent actions can be undone in any order, but causal-dependent actions are undone in reverse order
Reversible debugging

- Extends classical debugging with the ability to explore an execution not only forward but also backward
- Supported for instance by GDB
- Operators such as “execute n steps backward”
- Avoids the classical “Oh no, I put the breakpoint too late” exclamation
  - Just execute backward from where the program stopped
Reversible debugging for concurrent systems

- In concurrent systems, one should select which process should go back (or forward)
  - Manually, or by providing a scheduler

- The selected process may not be able to go back n steps unless some other process also goes back
  - E.g., cannot undo a send unless the process that received the message undoes the receive
  - In this case the “go back n steps” command of CauDEr just stops
Reversible debugging and causality

- Causal-consistency relates backward computations with causality
- Debugging amounts to find the bug that caused a given misbehavior
- CauDEr supports the following debugging strategy: follow causality links backward from misbehavior to bug
  - Causal-consistent reversible debugging
  - Originally proposed in [Giachino, Lanese & Mezzina, FASE 2014]
  - Supported by the roll primitive
The **roll** primitive

- Causal-consistent debugging based on **roll** \( n \) \( pid \)
- Undoes the last \( n \) steps of process \( pid \)...
- ... in a causal-consistent way
  - Before undoing an action one has to undo all (and only) its consequences
  - The debugger automatically finds and undoes the consequences
- A single **roll** may cause undoing steps in many processes
- We can provide different interfaces for **roll** helping the user to select suitable \( n \) and \( pid \)
  - one for each kind of misbehavior in the language
Different interfaces for **roll**

- One interface for each possible misbehavior
- In Erlang:
  - **Wrong value in a variable**: `roll var id` goes to the state just before the variable `id` has been created
  - **Unexpected message**: `roll send msgId` goes to the state where the message `msgId` has been sent
  - **Wrong message received**: `roll rec msgId` goes to the state where `msgId` has been received
  - **Unexpected process**: `roll spawn pid` goes to the state where process `pid` has been created
Using roll-like primitives

- The programmer can follow causality links backward
- The procedure can be iterated till the bug is found
- E.g., at some point, in process p, x = 5 while we were expecting x = 10
  - Roll var x goes back to where x has been created
  - E.g., x taken from a message with msgId 23
  - If the message has the wrong value, use Roll send 23 to explore further backward
  - If a wrong message has been taken due to a wrong pattern, then the bug has been found
Properties of roll-like primitives

- Only relevant steps are undone
  - Thanks to causal consistency we undo only consequences of the target action

- No need for the programmer to know which process or expression originated the misbehavior
  - The primitives find them automatically

- Looking at which processes are involved in a roll execution may give useful information
  - The involvement of an unexpected process means that an interference has happened
The need for replay

- CauDEr allows the user to go back in the execution looking for the causes of a given misbehavior but...
- If the misbehavior occurs in an actual execution in production environment it is difficult to reproduce it inside the debugger
  - Common problem in debugging of concurrent systems
  - Due to nondeterminism
- If during debugging one goes too much backward, it would be good to be able to go forward again with the guarantee to replay the same misbehaviors
- Causal-consistent replay solves both these problems
Causal-consistent rollback

- It allows one to undo any action, provided that its consequences (if any) are undone beforehand.
- Concurrent actions can be undone in any order, but causal-dependent actions are undone in reverse order.
Causal-consistent **replay**

- It allows one to **redo** any action, provided that its **causes** (if any) are **redone** beforehand.
- Concurrent actions can be **redone** in any order, but causal-dependent actions are **redone** in **original** order.
Causal-consistent replay

- It allows one to redo any action, provided that its causes (if any) are redone beforehand.
- Concurrent actions can be redone in any order, but causal-dependent actions are redone in original order.
- It is the dual of causal-consistent rollback.
- It allows one to redo actions which are in the future w.r.t. the current state of the computation.
- The choice of the future action to redo depends on the (mis)behavior we want to replay.
- How do we know the relevant future actions?
Logging

- Future actions are taken from real executions
- We built a tracer that instruments an Erlang program and produces a log for each process
- We log only concurrency-related actions
- Unique identifiers are attached to messages to match sends with receives
- The log has the form
  \{pid, spawn, 74\}
  \{73, send, 5\}
  \{75, receive, 7\}
  ...
- Can also be seen as one log per process
Replay in CauDEr

- CauDEr can now take a log and allow the user to explore the logged execution
  - undo selected past actions (and their consequences)
  - redo selected future actions (and their causes)

- We always replay a computation causal equivalent to the original one
  - That is, equal up to swap of concurrent actions
  - The log should contain enough information to allow one to do this

- This is enough to replay the (mis)behaviors of the original computation
Demo, by German
Log semantics

- The log of a computation is obtained by adding labels to relevant rules of the system semantics of Erlang
- The sequence of the labels corresponds to the log

\[
(Send) \quad \frac{\theta, e \xrightarrow{send(p',v)} \theta', e'}{\Gamma; \langle p, \theta, e \rangle \mid \Pi \xrightarrow{p,send(\ell)} \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \theta', e' \rangle \mid \Pi}
\]

\[
(Receive) \quad \frac{\theta, e \xrightarrow{rec(\kappa,cl_n)} \theta', e'}{\Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \theta, e \rangle \mid \Pi \xrightarrow{p,rec(\ell)} \Gamma; \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\} \rangle \mid \Pi}
\]

\[
(Spawn) \quad \frac{\theta, e \xrightarrow{spawn(\kappa,a/n,\overline{v_n})} \theta', e'}{\Gamma; \langle p, \theta, e \rangle \mid \Pi \xrightarrow{p,spawn(p')} \Gamma; \langle p', \theta', e'\{\kappa \mapsto p'\} \rangle \mid \langle p', id, apply a/n (\overline{v_n}) \rangle \mid \Pi}
\]
Both replay and rollback are specified in two steps

Uncontrolled semantics: which forward/backward steps are legal at any given point
  - It allows to replay any computation causal equivalent to the original one
  - Equal up to swap of concurrent actions and of introduction/removal of pairs do/undo or undo/redo

Controlled semantics: which forward/backward steps are needed to replay/undo a selected future/past action
Replay uncontrolled semantics

- The syntax of processes also includes their log
- Fresh message/process identifiers are taken from logs
- Only steps compatible with the log are allowed
  - In receive we can only take the expected message

\[
\begin{align*}
\text{(Send)} & \quad \frac{\theta, e \xrightarrow{\text{send}(p', v)} \theta', e'}{\Gamma; \langle p, \text{send}(\ell) + \omega, \theta, e \rangle \mid \Pi \xrightarrow{p, \text{send}(\ell)} \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \omega, \theta', e' \rangle \mid \Pi}
\\
\text{(Receive)} & \quad \frac{\theta, e \xrightarrow{\text{rec}(\kappa, \overline{c_l})} \theta', e' \text{ and } \text{matchrec}(\theta, \overline{c_l}, v) = (\theta_i, e_i)}{\Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \text{rec}(\ell) + \omega, \theta, e \rangle \mid \Pi}
\\
& \quad \xrightarrow{p, \text{rec}(\ell)} \Gamma; \langle p, \omega, \theta' \theta_i, e' \{\kappa \mapsto e_i\} \rangle \mid \Pi}
\\
\text{(Spawn)} & \quad \frac{\theta, e \xrightarrow{\text{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \text{ and } \omega' = \mathcal{L}(d, p')}{\Gamma; \langle p, \text{spawn}(p') + \omega, \theta, e \rangle \mid \Pi \xrightarrow{p, \text{spawn}(p')} \Gamma; \langle p, \omega, \theta', e' \{\kappa \mapsto p'\} \rangle \mid \Pi}
\\& \quad \langle p', \omega', \text{id}, \text{apply } a/n (\overline{v_n}) \rangle \mid \Pi}
\end{align*}
\]
Rollback uncontrolled semantics

- We need history information to go back
  - Each process has its own history
- Much more detailed than the log
- E.g., at each step we store the previous expression and state, and for some actions also further information
  - Ok, this could be optimized a lot...
- All the information needed to recover the past configuration
- History is computed going forward, and consumed going backward
Computing history

\[
\text{(Seq)} \quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, \omega, h, \theta, e \rangle \mid \Pi \xrightarrow{p, \text{seq}} \Gamma; \langle p, \omega, \text{seq}(\theta, e) + h, \theta', e' \rangle \mid \Pi}
\]

\[
\text{(Send)} \quad \frac{\theta, e \xrightarrow{\text{send}(p', v)} \theta', e'}{\Gamma; \langle p, \text{send}(\ell) + \omega, h, \theta, e \rangle \mid \Pi \xrightarrow{p, \text{send}(\ell)} \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \omega, \text{send}(\theta, e, p', \{v, \ell\}) + h, \theta', e' \rangle \mid \Pi}
\]

\[
\text{(Receive)} \quad \frac{\theta, e \xrightarrow{\text{rec}(\kappa, c\ell_n)} \theta', e' \text{ and } \text{matchrec}(\theta, c\ell_n, v) = (\theta_i, e_i)}{\Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \text{rec}(\ell) + \omega, h, \theta, e \rangle \mid \Pi \xrightarrow{p, \text{rec}(\ell)} \Gamma; \langle p, \omega, \text{rec}(\theta, e, p', \{v, \ell\}) + h, \theta', e' \{\kappa \mapsto e_i\} \rangle \mid \Pi}
\]

\[
\text{(Spawn)} \quad \frac{\theta, e \xrightarrow{\text{spawn}(\kappa, a/n, \{\overline{v_n}\})} \theta', e' \text{ and } \omega' = \mathcal{L}(d, p')}{\Gamma; \langle p, \text{spawn}(p') + \omega, h, \theta, e \rangle \mid \Pi \xrightarrow{p, \text{spawn}(p')} \Gamma; \langle p, \omega, \text{spawn}(\theta, e, p') + h, \theta', e' \{\kappa \mapsto p'\} \rangle \mid \Pi \xrightarrow{p', \omega', (, id, apply\ a/n\ (\overline{v_n})} \Gamma; \langle p', \omega', (, id, apply\ a/n\ (\overline{v_n}) \rangle \mid \Pi}
\]
Exploiting history

- Send and spawn can only be undone if dependencies are undone too
  - Send requires the sent message to be available in $\Gamma$
  - Spawn requires the target process to be in the initial state

\[
\begin{align*}
\text{(Seq)} & \quad \Gamma; \langle p, \omega, \text{seq}(\theta, e) + h, \theta', e' \rangle \mid \Pi \xleftarrow{p, \text{seq}} \Gamma; \langle p, \omega, h, \theta, e \rangle \mid \Pi \\
& \quad \text{where } \mathcal{V} = \text{Dom}(\theta') \setminus \text{Dom}(\theta) \\
\text{(Send)} & \quad \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \omega, \text{send}(\theta, e, p', \{v, \ell\}) + h, \theta', e' \rangle \mid \Pi \\
& \quad \xleftarrow{p, \text{send}(\ell)} \Gamma; \langle p, \text{send}(\ell) + \omega, h, \theta, e \rangle \mid \Pi \\
\text{(Receive)} & \quad \Gamma; \langle p, \omega, \text{rec}(\theta, e, p', \{v, \ell\}) + h, \theta', e' \rangle \mid \Pi \\
& \quad \xleftarrow{p, \text{rec}(\ell)} \Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \text{rec}(\ell) + \omega, h, \theta, e \rangle \mid \Pi \\
& \quad \text{where } \mathcal{V} = \text{Dom}(\theta') \setminus \text{Dom}(\theta) \\
\text{(Spawn)} & \quad \Gamma; \langle p, \omega, \text{spawn}(\theta, e, p') + h, \theta', e' \rangle \mid \langle p', \omega', (), id, e'' \rangle \mid \Pi \\
& \quad \xleftarrow{p, \text{spawn}(p')} \Gamma; \langle p, \text{spawn}(p') + \omega, h, \theta, e \rangle \mid \Pi
\end{align*}
\]
Controlled semantics

- Rollback and replay are sequences of uncontrolled steps
- We use a recursive algorithm (modeled as a stack machine) to select the steps
- To rollback an action A in process p
  - Start undoing actions in p
  - If A is undone then stop
  - If it is not possible to undo an action due to a dependency on action A1 in p1 then rollback A1 in p1, then continue undoing A
- Replay is analogous
Properties of the uncontrolled semantics

- Uncontrolled semantics satisfies the classical properties of reversible calculi
- Loop lemma: each step can be undone
- Parabolic lemma: each computation is causal equivalent to a backward one followed by a forward one
  - Hence, no new states are introduced by reversibility
- Causal consistency theorem: two coinitial computations are cofinal iff they are causal equivalent
Properties of the controlled semantics

- Controlled rollback/replay are minimal sequences of uncontrolled steps undoing/redoing the target action
- Allows one to leverage results from the uncontrolled semantics
  - E.g., no new states are introduced by reversibility
Usefulness for debugging

- All computations in the debugger are causal equivalent to the logged one
- A local error is visible in the debugger iff it is visible in the original computation
  - Local errors are errors that involve a single process or message
Future directions

- Support Erlang instead of Core Erlang
  - Not technically difficult, but time consuming

- Support a larger subset of the language
  - Distribution, constructs for fault tolerance, ...

- Improve efficiency
  - In particular, we are currently working on reducing the time overhead due to logging
  - Particularly critical since logging needs to be done in production environment
Finally

Thanks!

Questions?