Reversible Debugging of Erlang Programs in CauDEr*

Ivan Lanese
Olas Team, University of Bologna/INRIA
Bologna, Italy
ivan.lanese@unibo.it

Germán Vidal
Universitat Politècnica de València
Valencia, Spain
gvidal@dsic.upv.es

Abstract

This talk presents the notion of causal-consistent reversible debugging and its instance on Erlang provided by CauDEr. Reversible debugging allows us to explore an execution back and forth looking for a bug. Causal-consistent debugging tailors this approach to concurrent systems so that actions can be undone in any order as long as their consequences, if any, are undone first.

CCS Concepts

• **Computing methodologies** → **Concurrent programming languages**; • **Software and its engineering** → **Concurrent programming languages**; **Software testing and debugging**.

Keywords

Reversible computing, Debugging, Concurrency

ACM Reference Format:
Ivan Lanese and Germán Vidal. 2024. Reversible Debugging of Erlang Programs in CauDEr. In *Proceedings of the 2nd ACM International Workshop on Future Debugging Techniques (DEBT '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3678720.3685319

1 Introduction

Reversible debugging allows us to explore an execution back and forth looking for a bug causing a visible misbehavior. Going backwards in a sequential computation amounts to undo the actions of a standard (forward) execution in reverse order. In the case of concurrent programs, this approach may lead to explore an execution taking into account which processes were executed at a given point in time—which depends on scheduling policies and the speed of processors and cores, and provides few clues about which processes to consider when looking for a bug. Here, we consider *causal-consistent* reversible debugging [5], which allows us to leverage information on causal dependencies between processes for debugging. In particular, it allows one to explore the tree of causes of a visible misbehavior when looking for a bug, disregarding unrelated processes that just happened to be executed in the same time interval.

*This work has been partially supported by MSCA-PF project 101106046 - ReGraDe-CS, by Generalitat Valenciana under grant CIPROM/2022/6 (FassLow), and by MCIN/AEI/10.13039/501100011033 under grant PID2019-104735RB-C41.

DEBT '24, September 19, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1110-7/24/09
https://doi.org/10.1145/3678720.3685319

2 Causal-Consistent Debugging

Nowadays, most of the software is concurrent (and frequently distributed) with many processes interacting to reach some common goal. This makes debugging harder for many reasons. First of all, a bug may become visible only in some particular execution, originating from a specific interleaving among the actions of different processes. Replicating the execution that produced the misbehavior can be very difficult or even impossible in practice. Second, the bug may not be in the same process showing a misbehavior, but actually in a different one that interacted wrongly with the one showcasing the misbehavior. These difficulties make debugging concurrent software particularly hard.

Standard debuggers allow one to run a program, possibly step-by-step, or till some breakpoint, while inspecting the state to understand whether anything is wrong. The standard debugging technique consists in guessing the area of the program where the bug might be, adding a breakpoint just before, and then executing the program step-by-step from this breakpoint while looking for the bug. This approach depends critically on the correctness of the guess: if the bug is much further ahead, a tedious step-by-step execution is necessary (or just to continue to a later breakpoint). If the bug precedes the breakpoint, part of the state will already be wrong and one needs to restart execution including an earlier breakpoint.

In the case of concurrent systems, further problems arise. First, restarting the execution with an earlier breakpoint may result in a different interleaving, perhaps missing the misbehavior and making it impossible to find the bug. Second, if there are many processes, it can be very complex to determine which process may contain the bug and should thus be analyzed.

A key observation here is that the execution of the bug *precedes* and indeed *causes* the visible misbehavior, while possibly being in a different process. Reversibility exploits the first part of the observation: one can execute the program till the visible misbehavior occurs, and then execute it backward looking for the bug. This avoids the need to guess where the bug is typical of standard debugging.

Danos and Krivine showed [3] that in concurrent systems executing forward actions in reverse order may be impossible—many actions can occur at the same time, hence no total order may exist—and it is in general meaningless. Indeed, concurrent actions may occur in a given order even if they are unrelated. To solve these issues, Danos and Krivine proposed *causal-consistent* reversibility, which prescribes that any action can be undone provided that its consequences, if any, are undone beforehand.

As shown in [5], this approach is suitable for debugging, since it allows one to focus on a process of interest, e.g., the one showing the misbehavior, as well as the ones that interact with it, disregarding independent processes that just happened to be executed in the same time interval. A further step taken in [5] is to leverage the

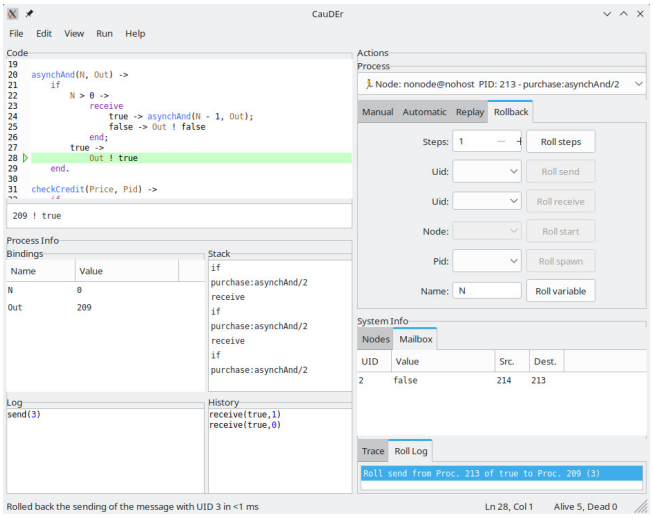

Figure 1: CauDEr screenshot

rollback operator in [8] for debugging. This operator allows one to undo an action (possibly far in the past) together with all *and only* its consequences. It can also be defined as the execution of the *shortest* sequence of backward causal-consistent steps undoing the target action. Rollbacks can be exploited in debugging as follows. When one spots a wrong behavior, say a wrong value of some variable x, (s)he can ask to undo its immediate cause: the last assignment to x. Either this operation leads to a wrong line of code—the bug one is looking for—or to a correct line of code that takes a wrong value from previous instructions, e.g., $x = y + 1$, which is correct but y has a wrong value as well. In this last case, one needs to iterate the procedure till the bug is found. This approach allows one to navigate the tree of causes of the misbehavior, which includes the bug, possibly inspecting different processes. Indeed, if for instance the wrong value assigned to x comes from a message, one can undo the sending of this message, which might be an action of a different process. Notably, unrelated processes are never considered.

3 Debugging Erlang Programs

The approach above is language agnostic, while however collapsing on standard reversible debugging for sequential languages. To make the theory concrete, it has been applied to the functional and concurrent language Erlang. This language has a clean concurrency model based on actors and has been applied in a number of relevant software projects worldwide [2]. The result of this line of work has materialized in CauDEr [1, 6, 9], a reversible debugger for Erlang. It implements the causal-consistent reversible semantics described above by means of an instrumented interpreter for the language (no native features of the Erlang/OTP environment are used).

A screenshot of CauDEr is shown in Fig. 1. The top-left presents the code under analysis. The top-right provides several debugging commands to control the execution. In particular, it currently shows the rollback tab, allowing to trigger the rollback of relevant actions like, e.g., a variable definition (since Erlang is functional, variables can only be assigned once). The bottom part provides various information on the state. Beyond the usual ones, CauDEr presents a history of each process describing past actions that can be used as targets for rollbacks. It also shows the roll log, which describes which actions have been actually undone due to the last rollback. This is particularly useful for spotting spurious or missing causal dependencies. For instance, if one rolls back an action of some process that was in a mutual exclusion region, and as a result no action of the process currently in the mutual exclusion region is undone, it means that mutual exclusion is not enforced properly.

Finally, CauDEr can take the trace of a (buggy) execution as input, obtained through the instrumentation of the source code. In this way, one can replay a particular execution—or any causally equivalent one—in the debugger, exploring it back and forth, thus solving the problem of replicability mentioned before [10].

4 Concluding Remarks

We have presented causal-consistent reversible debugging and its concrete instance provided by CauDEr. While the approach is promising, CauDEr is still a prototype supporting only the core features of the language (including higher-order functions, process spawning, message sending and receiving, and a few built-ins). Extending it to cover the full language and its libraries is time-consuming, and not trivial either. Indeed, it requires one to define a causal semantics of the full language, understand which information must be saved to enable reversibility, and ensure that actions are undone only after their consequences have been undone. This proved to be not trivial in the case of distribution [4] and of imperative primitives [7]. Another interesting line of research involves formalizing and implementing an extension of CauDEr in order to also show message races, following the approach in [11].

References

[1] CauDEr 2024. CauDEr repository. Available at https://github.com/mistupv/cauder.
[2] F. Cesarini. 2019. Which companies are using Erlang, and why? URL: https://erlang-solutions.com/blog/which-companies-are-using-erlang-and-why.
[3] V. Danos and J. Krivine. 2004. Reversible communicating systems. In *CONCUR (LNCS, Vol. 3170)*. Springer, 292–307. https://doi.org/10.1007/978-3-540-28644-8_19
[4] G. Fabbretti, I. Lanese, and J.-B. Stefani. 2021. Causal-Consistent Debugging of Distributed Erlang Programs. In *Reversible Computation (LNCS, Vol. 12805)*. Springer, 79–95. https://doi.org/10.1007/978-3-030-79837-6_5
[5] E. Giachino, I. Lanese, and C.A. Mezzina. 2014. Causal-Consistent Reversible Debugging. In *FASE (LNCS, Vol. 8411)*. Springer, 370–384. https://doi.org/10.1007/978-3-642-54804-8_26
[6] J.J. González-Abril and G. Vidal. 2021. Causal-Consistent Reversible Debugging: Improving CauDEr. In *PADL (LNCS, Vol. 12548)*. Springer, 145–160. https://doi.org/10.1007/978-3-030-67438-0_9
[7] P. Lami, I. Lanese, J.-B. Stefani, C. Sacerdoti Coen, and G. Fabbretti. 2024. Reversible debugging of concurrent Erlang programs: Supporting imperative primitives. *Journal of Logical and Algebraic Methods in Programming* 138 (2024), 100944. https://doi.org/10.1016/j.jlamp.2024.100944
[8] I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. 2011. Controlling Reversibility in Higher-Order Pi. In *CONCUR (LNCS, Vol. 6901)*. Springer, 297–311. https://doi.org/10.1007/978-3-642-23217-6_20
[9] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. 2018. CauDEr: A Causal-Consistent Reversible Debugger for Erlang. In *FLOPS (LNCS, Vol. 10818)*. Springer, 247–263. https://doi.org/10.1007/978-3-319-90686-7_16
[10] I. Lanese, A. Palacios, and G. Vidal. 2021. Causal-Consistent Replay Reversible Semantics for Message Passing Concurrent Programs. *Fundam. Informaticae* 178, 3 (2021), 229–266. https://doi.org/10.3233/FI-2021-2005
[11] G. Vidal. 2022. Computing Race Variants in Message-Passing Concurrent Programming with Selective Receives. In *FORTE (LNCS, Vol. 13273)*. Springer, 188–207. https://doi.org/10.1007/978-3-031-08679-3_12