# Causal Debugging for Concurrent Systems

Ivan Lanese, Gregor Gössler

# Causal Debugging for Concurrent Systems

Ivan Lanese[1(✉)] and Gregor Gössler[2]

[1] Olas Team, University of Bologna, INRIA, 40137 Bologna, Italy
ivan.lanese@gmail.com
[2] Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

**Abstract.** Debugging concurrent systems is notoriously hard, since bugs may manifest only for some interleavings among the processes' execution, and since debugging them may involve analyzing multiple processes. We claim that two key ingredients for such an analysis are reversible execution, to explore a faulty computation back and forward, and causal analysis, to identify the causes of a visible misbehavior. In this talk we focus in particular on the use of reversible execution, as enabled by CauDEr, a reversible debugger for concurrent Erlang programs.

## 1 Introduction

As soon as one learns how to program, (s)he is faced with the challenges of debugging. Indeed, every programmer has experienced long and tedious debugging sessions, possibly ended with the discovery of very trivial bugs. While experience allows one to reduce the amount of trivial bugs, one is faced with programs which are more and more complex, and time spent in debugging remains consistent.

Concurrent programs may contain very hard to find bugs, the so called Heisenbugs, whose distinctive feature is that they manifest by producing some visible misbehavior only for some specific interleaving among the processes of a concurrent application. Interleavings are highly sensitive to factors beyond programmer's control, such as processor(s) speed, scheduling policy, interrupts, and others, what makes extremely hard to reproduce a desired scheduling. Folklore says that a concurrent program can run flawlessly for days in the programmer's premises, and fail as soon as started in front of the customer. Reality behind this folklore is that moving from programmer's to customer's machine may change some of the factors above, hence increasing the probability of failures, and of course, when such failures really occur, programmers experiencing them share the story, reinforcing the legend above.

In the present paper we do not discuss how to make such bugs manifest in the programmer's premises, but we tackle a related problem: how to catch those

The work has been partially supported by French ANR project DCore ANR-18-CE25-0007. The first author has also been partially supported by MSCA-PF project 101106046—ReGraDe-CS and by INdAM – GNCS 2023 project RISICO, code CUP_E53C22001930001.


T. Æ. Mogensen and Ł. Mikulski (Eds.): RC 2024, LNCS 14680, pp. 3–9, 2024.
https://doi.org/10.1007/978-3-031-62076-8_1

bugs, after they manifest at least once. Indeed, this includes two problems: how to reliably replay the corresponding misbehaviors after they manifest once, and how to find the bugs from their visible effect. We will show how to leverage the theory of causal-consistent reversibility [3] to this effect, and how this can be concretely done on Erlang programs using the reversible debugger CauDEr [1,10].

## 2 Rollback and Replay

As mentioned, we assume that we have been lucky enough to have seen a bug to manifest. Unfortunately, the bug may manifest only if a specific interleaving among actions occurs. If we keep no information on the execution, even if we provide the same input to the program, we have no guarantee that the bug occurs again. For instance, a process may expect 3 messages which can be either `True` or `False`, and the bug manifests, e.g., since the program prints `True` on the screen while the correct output would be `False`, only if the two first messages contain value `True`, and the last one value `False`. Note that here we have two sources of different behaviors: the values of the messages, and the order in which they are received. For simplicity assume that the input to the program determines the value of the messages, and that there is no synchronization so that the order in which they arrive only depends on the scheduling. Remembering the input allows us to replay an execution where two messages carry value `True` and one carries value `False`, but not to ensure the `False` message to be received last. A log of a computation allows us to keep track of all the events which may cause non-determinism, and executing a program (with the same inputs) by allowing actions only if they are compatible with the log allows us to reproduce the same behaviors. In order to identify messages, we assign them unique identifiers. In our example, assume the messages with `True` have identifiers 1 and 2, and the one with `False` has identifier 3. A log of the faulty computation states, e.g., that messages have been received in order 2, 1, 3. Any computation satisfying this condition will produce a wrong behavior, hence manifesting the bug.

We are now able to replay the computation and be sure to showcase the bug, causing a wrong output printed from the program. Clearly, chances are that the bug is not in the print instruction, but in some previous computation providing a wrong value to the print instruction. Since we are in a concurrent system, the computation may involve multiple processes, interacting with each other. Any such interaction creates a causal link between actions of (possibly different) processes, and actions of a single process are all linked by the program flow. Looking backwards, any action, specifically the print providing a wrong value, is determined by a tree of causes, spanning multiple processes. The bug is for sure one of these actions. Some actions, and even some processes, may not be involved in the tree, hence looking at them is not helpful to find the bug. Causal-consistent debugging [5] suggests the following algorithm to find a bug: explore the tree of causes backwards, starting from the visible misbehavior.

At every step, look at the involved data[1]: if all the data are correct, then try another branch, if the data are all wrong, go further backwards, if the data entering the action are correct, and the ones exiting from it are wrong, then the action is wrong and it is the bug looked for. In order to support such a debugging strategy, one should be able to go backwards to explore a past action in some branch of the tree of causes, go forward again if the selected branch does not contain the bug, while remaining in a computation showcasing the bug, and then go backwards again along another branch. Clearly reversible computation comes handy here. More precisely, we need to be able to find the direct cause of some wrong part of the state, such as the last assignment to a variable with a wrong value, or the send of a given wrong message, and to go back where such an action has been executed. Note that, in particular, the send of a given wrong message is in general not in the same process where the message is received, hence this requires to possibly explore multiple processes. In order to go back we will exploit *causal-consistent rollback* [9], which allows one to undo a given action, including all and only its consequences. Dually, we can go forward again using *causal-consistent replay* [11], which replays a given action (e.g., the print showcasing the bug), including all and only its causes, while respecting the log. These two primitives allows one to explore a computation backward and forward, remaining in a setting manifesting the bug, while undoing or redoing the minimal number of actions to reach a state where the target action has been undone or redone.

*Causal Analysis.* As a complementary technique to rollback and replay, we have been exploring the use of causal analysis to enable the debugger to automatically answer queries of the form "why is the output of my program `True`?". Indeed, the approach above does not distinguish among the causes of an action, requiring the programmer to explore all of them. Using causal analysis we can identify one or more events in the program execution that actually caused the wrong outcome [6]. The analysis is *counterfactual*, that is, based on the analysis of alternative executions, in order to determine whether the latter would have changed the outcome. As a particular case, the analysis is able to blame the outcome on non-deterministic choices in the program execution. Furthermore, it enjoys desirable properties such as stability of the analysis result under semantic equivalence of the program. The integration of causal analysis and rollback and replay is still future work.

The approach described above is general and can be ideally applied to any programming language. We will describe below how to apply rollback and replay in the concrete case of the Erlang programming language.

We are currently working on instantiating on Erlang the causal analysis as well, relying on a novel fine-grained semantics [2] which carefully describes the possible orders in which messages (and signals) are handled, which is the main source of non-determinism in Erlang.

[1] For simplicity, the description here focuses on wrong values, but a similar approach can be applied to control flow.

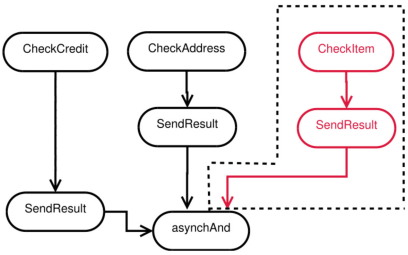


**Fig. 1.** The purchase workflow.

## 3 Causal-Consistent Debugging in CauDEr

The example above is not artificial, and it captures the essence of an actual case study, originally described in [12]. A simple Erlang implementation showcasing the misbehavior can be found in CauDEr [1] library of case studies. The case study implements a procedure for handling purchase orders, whose behavior is depicted in Fig. 1.

Before an order is placed, two conditions must be verified: the availability of the customer's credit, and the completeness of the address for delivery. The two independent checks CheckCredit and CheckAddress are performed concurrently. The results of the checks are sent to the asynchAnd procedure as soon as they are available. The asynchAnd procedure performs a short-circuit evaluation of n-ary AND: it waits for up to n values, but as soon as one is `False`, it immediately produces `False` as a result. If n `True` values are received, it sends the value `True` as a result.

Assume we perform perfective maintenance on this software: to avoid that clients wait for long periods of time, we make sure that the item is actually in stock before concluding the purchase. Thus, a new procedure checkItem implementing such a check is incorporated in the system (as shown in the dashed part of Fig. 1).

During testing, the program behaves as expected, however, when deployed on client's premises, it sometimes misbehaves. Indeed, it sometimes returns value `True` even if one of the conditions fails.

The bug is indeed dependent on the schedule, thus by running the program one most likely obtains the correct result, namely that the purchase is not authorized if at least one condition fails.

One can then run the program, instrumented by CauDEr tracer so to produce a log of the computation, and if (s)he is able to capture the misbehavior once, then (s)he can analyze it at will inside CauDEr, being sure that the misbehavior always manifests.

CauDEr interface can be seen in Fig. 2. We can see the code on the top-left, some useful commands on the top right, and information on the system and on the selected process (the one executing the asynchAnd) on the bottom.

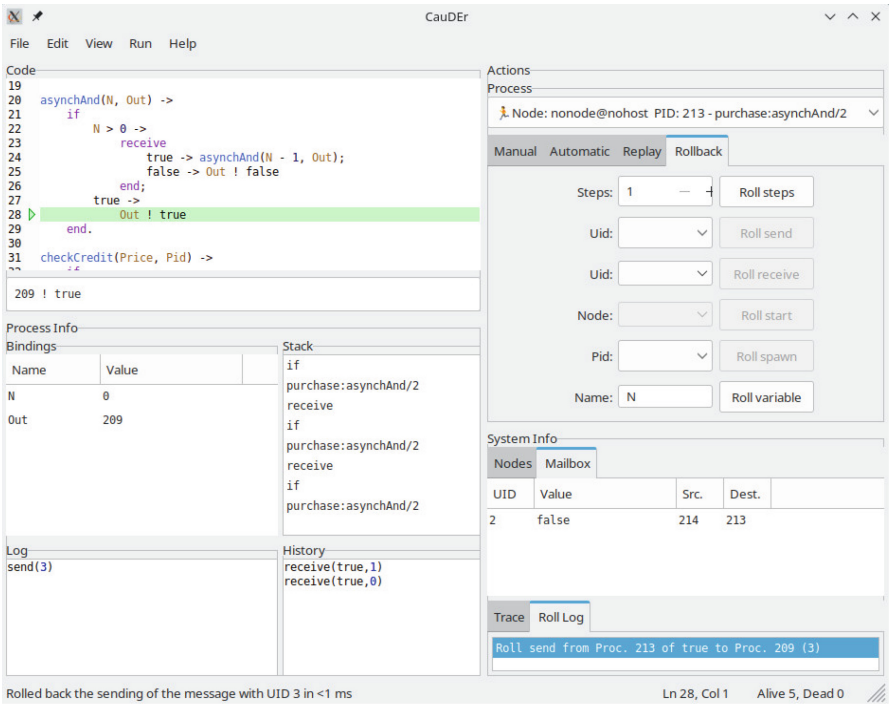


**Fig. 2.** CauDEr at work.

First one can select to replay the full log, so to execute the program till the end. Indeed the bug manifests: one check failed, yet the program reports that all the checks succeeded.

The result is taken from a receive, and it was inside message number 3. Then, we can use rollback to undo the send of message number 3. We reach exactly the state depicted in the figure. We can see, as expected, that the message comes from process asynchAnd. Looking at the history of the process (History frame in the center at the bottom) we see that the message has been sent after receiving only two values, what should be enough to understand which is the error. If we fail to notice this information in the history, we can execute backward step-by-step the code of the asynchAnd, and reasonably noticing the bug, when we reach the first invocation and look at the parameters.

## 4 Conclusion

We have shown, albeit in a minimal example, how to exploit causal-consistent debugging to find elusive bugs. We stress here a few aspects:

– a log provides a compact way to capture a family of computations which show some desired behavior, in particular the manifestation of a bug;

– backward execution in general, and rollback in particular, allow one to explore the tree of causes of a misbehavior, looking for the bug;
– rollback allows one to automatically navigate among processes, disregarding unrelated processes, hinting at the possibility that this debugging technique scales better than traditional approaches to programs with many processes.

While we believe this approach is very promising, being able to apply it to real large applications is far from trivial. Indeed, one needs to understand the causal semantics of the underlying language, to store history and causal information, and to exploit it to drive the computation.

In the case of a language like Erlang, the causal semantics for send, receive and spawn is relatively easy (it is essentially the happened-before relation [8]), but coping with other aspects has proved to require a detailed and non trivial analysis. This is indeed what happened to support primitives deadling with distribution [4], and primitives used to manage an imperative store associating names to process identifiers [7]. Notably, the latter highlighted challenges that would also emerge in languages with shared memory.

Managing history information requires a large amount of time and space overhead already in the sequential case, as shown by GDB, and smart optimizations are needed to reduce it as shown by UndoDB [13].

In order to use all this information to drive execution, CauDEr has been written essentially as a step-by-step interpreter of Erlang programs. As a result, constructs need to be implemented one by one, and one needs to follow Erlang evolution to avoid becoming outdated. This is a suitable strategy for a proof of concept, but an industrial implementation should be integrated with Erlang runtime support to avoid the above issues. However, this integration needs to be very deep, and it is not enough, e.g., to build on top of a classical debugger or a tracer.

## References

1. CauDEr repository (2024). https://github.com/mistupv/cauder
2. Kong Win Chang, A., Feret, J., Gössler, G.: A semantics of core Erlang with handling of signals. In: Kulahcioglu Ozkan, B., Fernandez-Reyes, K. (eds.) Proceedings of the 22nd ACM SIGPLAN International Workshop on Erlang, Erlang 2023, Seattle, WA, USA, 4 September 2023, pp. 31–38. ACM (2023)
3. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_19
4. Fabbretti, G., Lanese, I., Stefani, J.-B.: Causal-consistent debugging of distributed Erlang programs. In: Yamashita, S., Yokoyama, T. (eds.) RC 2021. LNCS, vol. 12805, pp. 79–95. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79837-6_5
5. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 370–384. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_26

6. Gössler, G., Stefani, J.-B.: Causality analysis and fault ascription in component-based systems. Theoret. Comput. Sci. **837**, 158–180 (2020)
7. Lami, P., Lanese, I., Stefani, J.-B., Sacerdoti Coen, C., Fabbretti, G.: Reversible debugging of concurrent Erlang programs: supporting imperative primitives. J. Log. Algebraic Methods Program. **138**, 100944 (2024)
8. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
9. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.-B.: Controlling reversibility in higher-order pi. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 297–311. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_20
10. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: a causal-consistent reversible debugger for Erlang. In: Gallagher, J.P., Sulzmann, M. (eds.) FLOPS 2018. LNCS, vol. 10818, pp. 247–263. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90686-7_16
11. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay reversible semantics for message passing concurrent programs. Fundam. Inform. **178**(3), 229–266 (2021)
12. Stanley, T., Close, T., Miller, M.S.: Causeway: a message-oriented distributed debugger. Technical report, HP Labs (2009). HP Labs tech report HPL-2009-78
13. Undo Software. 6 things you need to know about time travel debugging. https://undo.io/resources/6-things-time-travel-debugging. Accessed April 2024