Reversible Computing in Debugging of Erlang Programs

Ivan Lanese, Ulrik Schultz, Irek Ulidowski

Department: Software Technology
Editor: Markus Schordan

Reversible Computing in Debugging of Erlang Programs

I. Lanese
University of Bologna/INRIA

U. P. Schultz
University of Southern Denmark

I. Ulidowski
University of Leicester

***Abstract*—Reversible computation is a computing paradigm where execution can progress backwards as well as in the usual, forward direction. It has found applications in many areas of computer science, such as circuit design, programming languages, simulation, modelling of biochemical reactions, debugging and robotics. In this article, we give an overview of reversible computation focusing on its use in reversible debugging of concurrent programs written in the Erlang programming language.**

INTRODUCTION

There are many situations when it would be useful to go back in time to make different choices or to alter events. In addition to time travel one would also wish to carry memory of the present into the past. Otherwise, after travelling back into the past, one could follow the same path of events. Our experience of time and the current knowledge of how the Universe works suggest that time travel is (currently) not possible. There is evidence, however, that basic physical processes are *reversible* at a microscopic scale, for example, covalent chemical reactions are bidirectional. Once we consider any larger systems we cannot observe many naturally occurring meaningful reversible processes.

The situation is more favourable in man-made systems. We are able to simulate time travel by making artificial systems (broadly speaking) reversible, namely by enriching such systems with the ability to undo computation or actions. Sometimes reversibility is an integral part of systems as, for example, in data compression and decompression algorithms or in the Fourier transform, which is its own inverse [1]. More often, we need to re-design or re-engineer systems to make them reversible. Unlike in the natural world, we would like to think that we can exert some control over artificial systems. Envisaged applications in areas such as low-power computing, simulation, robotics and debugging have recently motivated research on how to add reversibility to existing systems and software.

Reversibility has interested scientists for some

 Published by the IEEE Computer Society

time now. Landauer has discovered in the 1960s that erasing information in computers requires energy and that loss of information, such as erasing a value stored in a variable, during computation is manifested by release of heat [2]. The scientists thought then, and are still thinking, that if we could build logic circuits and, ultimately, hardware that reduces or even avoids completely the need to remove information, then computers would be more energy efficient. Subsequently, Fredkin and Toffoli developed reversible universal logic gates as an alternative to the traditional CMOS technology gates [3]. These reversible gates have recently become elementary gates in quantum computing. There has been a significant amount of research on reversible computing since the discovery of reversible logic gates, including on alternative reversible logics [4], reversible sequential and quantum circuits and hardware [5], reversible arithmetic logic unit (ALU) [6], and on software to support them. This means that it is possible to design, manufacture and program reversible computing devices.

Apart from this original motivation for *physical reversibility*, there are many other reasons for, and benefits of, *logical* reversibility [7]. The latter form of reversibility concerns enhancing systems and software with the ability to undo (or simulate undoing of) computation. Logical reversibility can be implemented in hardware via reversible gates and with the help of other reversible devices such as the mentioned ALU. It can be programmed using reversible programming languages such as Janus [1], which is both forward and backward deterministic. However, since the majority of software and hardware currently in use is irreversible, logical reversibility is most of the time only simulated. There are, e.g., techniques for simulating reversibility of traditional imperative programming languages such as C [8]. Researchers have also discovered the basics of how to reverse computation of concurrent programs and systems [9], [10], [11], [12].

The purpose of this article is to introduce the topic of reversible computation by presenting a case study where logical reversibility has made a difference. This case study, and more generally, reversible computation research in Europe was partially supported by COST Action IC1405 on Reversible Computation - Extending Horizons of Computing [13]. We shall touch gently on the theories we have developed and explain how they assisted us in solving practical problems of the case study.

Our case study concerns debugging of concurrent programs written in the Erlang programming language.

REVERSIBLE DEBUGGING

Programs frequently misbehave, and once a misbehaviour, e.g., a wrong output, in executing a program is observed, the aim is to discover its causes so to correct the program. The use of reversibility in debugging is quite natural: debugging amounts to find the bug in a program, that is the wrong line of code, which causes a visible misbehaviour. The bug precedes the misbehaviour, hence a sensible way to find the bug is to execute the program backwards from where the misbehaviour is seen. This is the approach of reversible debugging in sequential systems, which can be summarised as follows: keep undoing the computation steps of the program from the point of misbehaviour until the lines of code that caused the bug in the first place are reached. This contrasts with the usual approach employed by forward debuggers: one needs to put a breakpoint before the bug, and then execute step-by-step forwards from the breakpoint until the bug is found. This approach has the limitation that the position of the bug is not known, hence it is not easy to find where to put the breakpoint. An early breakpoint requires a lot of step-by-step execution which is time consuming, while a late breakpoint does not allow one to find the bug and requires to restart the execution with an earlier breakpoint. This analysis of traditional debugging has motivated the study and the development of reversible debuggers both at the industrial level (e.g., Microsoft Time Travel Debugger [14] or UDB of Undo [15]) and in the open source community (GDB supports reversible debugging since version 7.0). Such debuggers focus on sequential programs, hence they can be built on top of the theory of sequential reversibility, which roughly prescribes that to reverse a sequential program one needs to undo its actions in reverse order of completion. Sequential reversibility is nowadays well understood.

Reversing Concurrent Programs

The analysis of reversibility and reversible debugging in concurrent systems is quite recent [16]. In such systems the execution of different actions can overlap, hence it is not (always) possible to find an order of completion to reverse. Even when possible, it may not make sense: if two processes P_1 and P_2 compute without interacting then their actions are independent, and a misbehaviour in P_1 cannot depend on a bug in P_2. As a result, when looking for the bug causing the misbehaviour in P_1, there is no need to undo actions of P_2, even if they took place in the time interval under analysis. These ideas are captured by the notion of *causal-consistent reversibility* [9] and its mantra:

> Any action can be undone, provided that its consequences, if any, are undone first.

Equivalently, independent actions can be undone in any order, and causally dependent actions need to be undone in reverse order: first the consequences, then the causes. An interesting property of such a reversibility theory is that any state reachable by some mixture of forward and backward steps is also reachable from the initial state using only forward steps. This makes sense for debugging, since one would like reversibility to help finding bugs (and wrong states) reachable in a forward computation, not to create new states.

We will exemplify these ideas using the causal-consistent reversible debugger CauDEr[1] [17], [18]. CauDEr is a step-by-step interpreter for the concurrent and functional language Erlang with additional support for reversibility, which follows quite closely the approach outlined above.

Bank Account Case Study

We will now show how to use CauDEr to debug a subtle misbehaviour in a small case study modelling a bank account. The Erlang code for the bank account system is given in Fig. 1.

The code features several Erlang actors: an *accountManager* keeping the balance of an account, a *meManager* providing mutual exclusion, two *deposit*s, each adding 100 to the same account, and a *checkBalance* displaying the account balance. Since the balance starts from 0 one expects a final balance of 200, and this is what happens in most of the cases. Sometimes, however, the final balance is 100, and we want to find the cause of this misbehaviour.

The bug is quite subtle, and even in such a small piece of code it may not be trivial to spot it without the help of some debugging tool. It will become very difficult to find a similar bug if it occurs inside a large application.

With a standard debugger, as discussed above, one should decide where to put a breakpoint to start a step-by-step forward execution looking for the bug. This is a non trivial choice.

Fig. 2 shows a screenshot of CauDEr while debugging the scenario above. CauDEr provides facilities for forward and backward execution, and for debugging. In the screenshot, one of the processes executing the code snippet is shown. Beyond standard information about the state of the processes, one can see further kinds of information regarding

- the past of the process in the History frame,
- the future of the process in the Log frame (here we are replaying inside the debugger an execution following logs obtained in the real Erlang environment),
- the main actions executed in the system in the Trace tab, and
- the effect of the last rollback in the Roll Log tab (the content of the Roll Log tab is not visible in the screenshot).

The first direct application of the theory of reversible debugging is to allow the user to step or run the program both backwards and forwards. In this setting, one has to specify which process needs to execute forwards or backwards, and this can be decided either manually by the user, or automatically using a suitable scheduler.

Thus, we can just run the program to completion (or to line 33 in *checkBalance*), where the wrong value is printed, and then execute backwards the corresponding actor. With a few steps backwards, or just by looking at the code, we easily see that the wrong value comes from the receive action at line 32.

Here a more refined form of backward exploration, called rollback or just *roll* [19], comes

[1] https://github.com/mistupv/cauder-v2

```
1  -module(bank).
2  -export([main/0,meManager/0,accountManager/1,deposit/2,checkBalance/2]).
3
4  main() ->
5      MePid = spawn(?MODULE,meManager,[]),
6      APid = spawn(?MODULE,accountManager,[0]),
7      spawn(?MODULE,deposit,[MePid,APid]),
8      spawn(?MODULE,deposit,[MePid,APid]),
9      spawn(?MODULE,checkBalance,[MePid,APid]).
10
11 meManager() ->
12     receive {requestMe,Pid} -> Pid!grantMe end,
13     receive {releaseMe} -> meManager() end.
14
15 accountManager(Val) ->
16     receive {setBalance,NewVal} -> accountManager(NewVal);
17             {getBalance,Pid} -> Pid!{balance,Val} end,
18     accountManager(Val).
19
20 deposit(MePid,APid) ->
21     MePid!{requestMe,self()},
22     receive grantMe ->
23             APid!{getBalance,self()},
24             receive {balance,X} ->
25                     APid!{setBalance,X+100},
26                     MePid!{releaseMe} end end.
27
28 checkBalance(MePid,XPid) ->
29     MePid!{requestMe,self()},
30     receive grantMe ->
31             XPid!{getBalance,self()},
32             receive {balance,X} ->
33                     io:format("Account balance:~b~n",[X]),
34                     MePid!{releaseMe} end end.
```

Figure 1. Bank account system

handy. The roll operator allows one to undo a selected action, possibly far in the past, including all and only its consequences. Indeed, all the consequences have to be undone according to the causal-consistent mantra to avoid reaching states which could not be reached in a forward computation. Undoing only such actions allows us to focus on the processes actually involved in the computation under analysis since other processes cannot be the cause of the bug. From another point of view, the roll operator executes the shortest backward causal-consistent computation undoing the target action.

In our case study, we see from the History frame (Fig. 2 shows exactly this state) that message 22 (messages are labelled with unique identifiers) was received at line 32. We can ask for a roll of the send of message 22, which performs the minimal backward computation leading to a state where message 22 could have been sent. The Roll Log tab (Fig. 3), showing which actions have been undone, also tells us that message 22 has been sent by process 2, that is the *accountManager*. In general, we can use rollback to follow causality links backward, e.g., if the message takes a wrong value from some variable X, then we could use Roll variable to go where the variable X has been assigned (this may involve undoing consequences on other actors as well). In our case study, the wrong value is in the state of the *accountManager*. We could use the roll operator to check where the wrong value has been assigned, but an easier way to understand the issue is to look at the History frame (Fig. 4). The frame shows that the *accountManager* has sent a message $\{$balance,0$\}$ twice, and has received twice a message $\{$setBalance,100$\}$. We can use a Roll receive to find out who received the second message $\{$balance,0$\}$, which has identifier 17. The message has been received by a *deposit* process, and we may notice by looking at its code that there is no synchronisation ensuring that the balance is set before the mutual exclusion is released. Indeed, the messages at lines 25

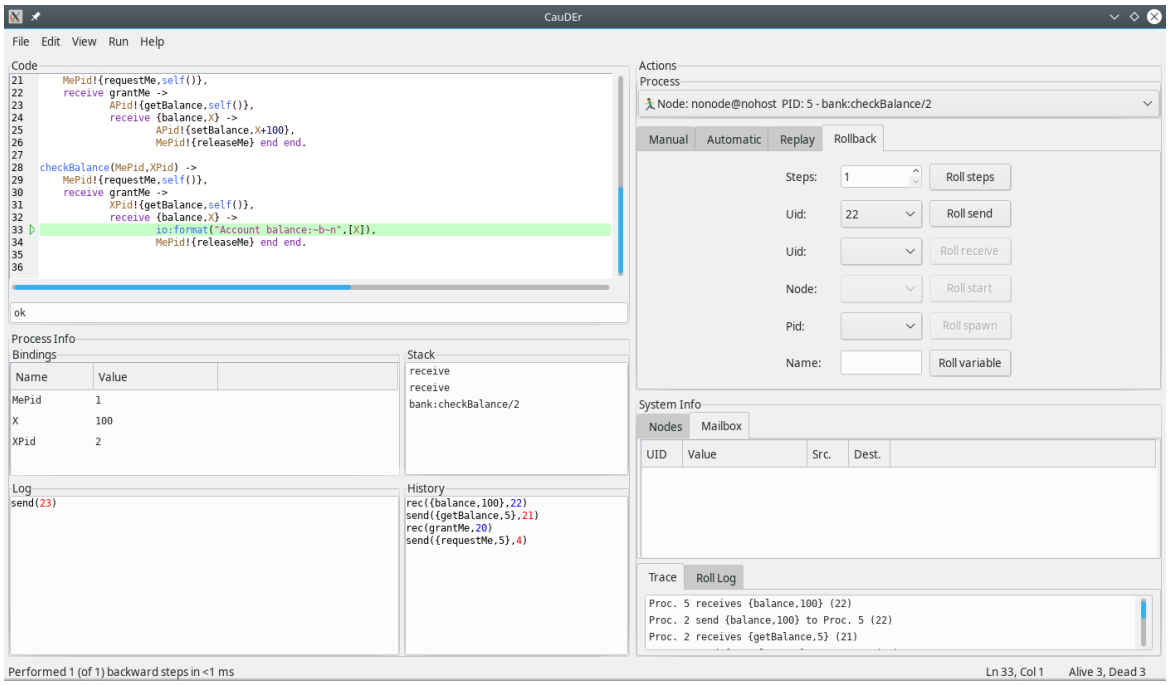

Figure 2. CauDEr screenshot

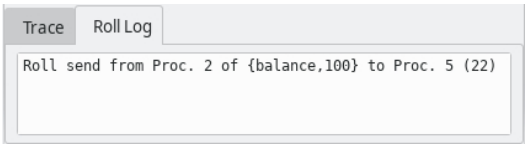

Figure 3. Roll log tab after roll receive of message 22

Figure 4. History after roll receive of message 22

and 26 are concurrent and can arrive in any order, which is the bug we were looking for. This can also have been checked by observing that we can replay the release of mutual exclusion (by sending the correspondent message and having *meManager* receive it) without the need to replay the receive of the *setBalance*. This shows another advantage of causal-consistent reversibility (w.r.t. mainstream reversible debuggers which linearise the execution): it keeps and uses causality information which can highlight missing synchronisations, like in this case, or undesired dependencies.

We have now found the bug, and we can fix it by adding a further synchronisation ensuring that mutual exclusion is released only after the *accountManager* has been updated.

The approach above emerges from theoretical studies on causal-consistent reversibility, and ideally it can be applied to any concurrent programming language.

The current CauDEr implementation only supports the functional, concurrent and distributed fragment of Erlang. This leaves out practically important features like error handling, hot code swap, timeouts and others. While these features can be supported, and some of them will be indeed added in the near future, they require both a detailed study to understand their causal and reversible semantics (such as the one provided for the functional and concurrent fragment in [20]), and an implementation effort. In some cases the causal semantics can be quite complex, and approximations need to be taken. A safe approximation is to add some fake causal dependences, thus making some processes that were originally independent become related. This pushes the user towards analysing more processes than strictly needed, but it is safe in that the process which contains the bug will always be among the ones

that will be explored by going backwards from the misbehaviour using roll. We remark however that fake dependencies may hide missing real dependencies.

Concerning performance, in order to enable causal-consistent reversibility CauDEr needs to keep a large amount of both causal and historical information about the computation, and managing it causes considerable time overheads. Hence, CauDEr currently slows down considerably if the size and complexity of the considered program grows. However, smart optimisations could be used to reduce the tracked information and the time overhead. By analogy with the sequential setting, we think large improvements are possible. Indeed, on the same sequential benchmark (a Quake 2 frame forward execution) and system GDB 7.6 took 34 minutes while the heavily optimised UndoDB 4.1.3840 took just 6.29 milliseconds (to be compared to 1.19 milliseconds in a normal execution)[2].

CONCLUSION

Reversing of computation is conceptually and technically a challenging task even if we only consider logical reversibility. We have illustrated significant potential benefits of reversibility in debugging of concurrent programs written in Erlang. We have presented briefly some of the recently developed theoretical foundations for the case study, describing mainly how reversibility helps. Exploring this application area helped us to exemplify the richness of different forms of reversibility.

Causal-consistent reversibility has enabled us to undo executions of concurrent Erlang programs to assist in debugging. In this setting, the developers are fully in control of where undoing of execution can lead us to. As a result, we have the following strong property: any reachable (by an arbitrary combination of reverse and forward steps) state is forwards reachable. This property does not hold in segments of (reversible) computation of some physical systems and processes such as, for example, robots and biochemical reactions. A case study of programming industrial robots for assembly operations, where classical AI planning is enriched with an underlying reversible execution model to increase robustness and versatility, is presented in [21]. Such robots are controlled through programming, but we cannot be certain of their actions since they interact with an unpredictable physical world. Contrary to causal-consistent reversibility, we have seen that some inverses of indirectly reversible operations may lead to new "get-out-of-trouble" states, albeit temporarily, which are *not* forwards reachable. Such states are needed due to the irreversibility of the physical world with which the robot interacts.

There are also other forms of reversibility suitable for different applications. Probably the best known is *backtracking*[3], where steps of computation are undone in the inverse order of execution. It has been used to undo concurrent C-like programs for debugging [22]. Finally, there are computations where the causes of some actions can be undone before those actions are themselves undone, seemingly breaking the cause-effect relationship. This is a common phenomenon in biochemical reactions such as catalytic reactions and signalling pathways, and is called *out-of-causal order* reversibility [23].

ACKNOWLEDGMENT

The authors acknowledge partial support from COST Action IC1405 on Reversible Computation - Extending Horizons of Computing. The first author has also been partially supported by French ANR project DCore ANR-18-CE25-0007.

REFERENCES

1. T. Yokoyama, H. B. Axelsen, and R. Glück, "Principles of a reversible programming language," in *Proceedings of CF 2008*. ACM, 2008, pp. 43–54.
2. R. Landauer, "Irreversibility and heat generated in the computing process," *IBM Journal of Research and Development*, vol. 5, 1961.
3. E. Fredkin and T. Toffoli, "Conservative logic," *International Journal of Theoretical Physics*, vol. 21, pp. 219–253, 1982.
4. K. Morita, *Theory of Reversible Computing*, ser. Monographs in Theoretical Computer Science. Springer, 2017.

[2]https://www.jwhitham.org/2015/05/review-undodb-reversible-debugger.html

[3]We refer here to backtracking in the context of reversible computing, the same term can also be used in other contexts with related but different meanings.

5. A. De Vos, *Reversible Computing - Fundamentals, Quantum Computing, and Applications*. Wiley, 2010.
6. M. K. Thomsen, R. Glück, and H. B. Axelsen, "Reversible arithmetic logic unit for quantum arithmetic," *Journal of Physics A: Mathematical and Theoretical*, vol. 43, no. 38, p. 382002, 2010.
7. C. Bennett, "Logical reversibility of computation," *IBM Journal of Research and Development*, vol. 17, 1973.
8. K. Perumalla, *Introduction to Reversible Computing*. CRC Press, 2014.
9. V. Danos and J. Krivine, "Reversible communicating systems," in *Proceedings of CONCUR 2004*, ser. LNCS, vol. 3170. Springer, 2004, pp. 292–307.
10. I. Phillips and I. Ulidowski, "Reversing algebraic process calculi," *Journal of Logic and Algebraic Programming*, vol. 73, pp. 70–96, 2007.
11. I. Lanese, C. Mezzina, and J.-B. Stefani, "Reversing higher-order pi," in *Proceedings of CONCUR 2010*, ser. LNCS, vol. 6269. Springer, 2010, pp. 478–493.
12. I. Lanese, I. Phillips, and I. Ulidowski, "An axiomatics approach to reversible computation," in *Proceedings of FOSSACS 2020*, ser. LNCS, vol. 12077. Springer, 2020, pp. 442–461.
13. I. Ulidowski, I. Lanese, U. P. Schultz, and C. Ferreira, Eds., *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, ser. LNCS. Springer, 2020, vol. 12070.
14. J. McNellis, J. Mola, and K. Sykes, "Time travel debugging: Root causing bugs in commercial scale software," CppCon talk, https://www.youtube.com/watch?v=l1YJTg_A914, 2017.
15. Undo, *UDB - reverse debugger for C/C++*, https://undo.io, 2020.
16. E. Giachino, I. Lanese, and C. A. Mezzina, "Causal-consistent reversible debugging," in *Proceedings of FASE 2014*, ser. LNCS, vol. 8411. Springer, 2014, pp. 370–384.
17. I. Lanese, N. Nishida, A. Palacios, and G. Vidal, "CauDEr: A causal-consistent reversible debugger for Erlang," in *Proceedings of FLOPS 2018*, ser. LNCS, vol. 10818. Springer, 2018, pp. 247–263.
18. J. J. González-Abril and G. Vidal, "Causal-consistent reversible debugging: Improving CauDEr," in *PADL*, 2021, to appear.
19. I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani, "Controlling reversibility in higher-order pi," in *Proceedings of CONCUR 2011*, ser. LNCS, vol. 6901. Springer, 2011, pp. 297–311.
20. I. Lanese, N. Nishida, A. Palacios, and G. Vidal, "A theory of reversibility for Erlang," *Journal of Logical and Algebraic Methods in Programming*, vol. 100, pp. 71–97, 2018.
21. I. Lanese, U. P. Schultz, and I. Ulidowski, "Reversible execution for robustness in embodied AI and industrial robots," *IT Professional*, vol. X, pp. 01–08, 2021.
22. J. Hoey and I. Ulidowski, "Reversing imperative parallel programs and debugging," in *Proceedings of Reversible Computation 2019*, ser. LNCS, vol. 11497. Springer, 2019, pp. 108–127.
23. I. Phillips, I. Ulidowski, and S. Yuen, "A reversible process calculus and the modelling of the ERK signalling pathway," in *Proceedings of Reversible Computation 2012*, ser. LNCS, vol. 7581. Springer, 2013, pp. 218–232.

Ivan Lanese is an Associate Professor at the University of Bologna, Italy. He acted as the vice chair of the COST Action IC1405 on Reversible Computation. He is interested in formal methods and concurrency theory, with special focus on reversibility and reversible debugging. Further information is available at https://www.unibo.it/sitoweb/ivan.lanese/en. He can be contacted at ivan.lanese@gmail.com.

Ulrik P. Schultz is a Professor in Aerial Robotics at the University of Southern Denmark. He recently participated in the COST Action IC1405 on Reversible Computing where he chaired the Working Group on Applications. He is interested in programming languages for robotics, his full biography is available at http://www.sdu.dk/staff/ups and he can be contacted at ups@sdu.dk

Irek Ulidowski is an Associate Professor at the University of Leicester in England. He chaired the COST Action IC1405 on Reversible Computation. His interests include reversible computation and its application, and concurrent systems. Further information is available at https://www.cs.le.ac.uk/people/iu3. He can be contacted at irekulidowski@gmail.com.