Reversible computing is a computing paradigm where execution can progress backward 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, modeling 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.

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 traveling 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 favorable 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. More often, we need to redesign or reengineer 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 time now. Landauer 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. The scientists thought then, and are still thinking, that if we could build logic circuits and, ultimately, hardware that reduces or even completely avoids 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. 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, reversible sequential and quantum circuits and hardware, reversible arithmetic logic unit (ALU), 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. 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, which is both forward and backward deterministic. However, since the majority of software and hardware currently in use is irreversible, logical reversibility is only simulated most of the time. There are, for example, techniques for simulating reversibility of traditional imperative programming languages such as C. Researchers have also discovered the basics of how to reverse computation of concurrent programs and systems.

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. 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 misbehavior, 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 finding the bug in a program, that is, the wrong line of code, which causes a visible misbehavior. The bug precedes the misbehavior; hence, a sensible way to find the bug is to execute the program backward from where the misbehavior was seen. This is the approach of reversible debugging in sequential systems, which can be summarized as follows: keep undoing the computation steps of the program from the point of misbehavior 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 forward 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 or UDB of Undo) 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. 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 misbehavior in \( P_1 \) cannot depend on a bug in \( P_2 \). As a result, when looking for the bug causing the misbehavior 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 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, which 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 misbehavior in a small case study modeling a bank account. The Erlang code for the bank account system is given in Figure 1.

The code features several Erlang actors: an `accountManager` keeping the balance of an account,
a `meManager` providing mutual exclusion, two deposits, 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 misbehavior.

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 nontrivial choice.

Figure 2 shows a screenshot of CauDER while debugging the abovementioned scenario. 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 backward and forward. In this setting, one has to specify which process needs to execute forward or backward, 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 backward the corresponding actor. With a few steps backward, 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, comes 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 (Figure 2 shows exactly this state) that message 22 (messages are labeled 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 (see Figure 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 (see Figure 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 synchronization ensuring that the balance is set before the mutual exclusion is released. Indeed, the messages at lines 25 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

---

**FIGURE 2.** CauDEr screenshot.

**FIGURE 3.** Roll log tab after roll receive of message 22.
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 linearize the execution): it keeps and uses causality information which can highlight missing synchronizations, like in this case, or undesired dependencies.

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

The abovementioned approach emerges from theoretical studies on causal-consistent reversibility, and ideally it can be applied to any concurrent programing 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 the work of Lanese et al., 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 toward analyzing 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 backward from the misbehavior 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 optimizations 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 optimized UndoDB 4.1.3840 took just 6.29 milliseconds (to be compared to 1.19 milliseconds in a normal execution).

**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 recent 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 forward 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 programing 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 the work of Lanese et al. Such robots are controlled through programing, 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 forward 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, where steps of computation are undone in the inverse order of execution. It has been used to undo concurrent C-like programs for debugging. Finally,

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

5. 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.
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 signaling pathways, and is called out-of-causal order reversibility.23

ACKNOWLEDGMENTS

The authors acknowledge partial support from COST Action IC1405 on Reversible Computation - Extending Horizons of Computing. Ivan Lanese has also been partially supported by French ANR project DCore ANR-18-CE25-0007 and by INdAM – GNCS 2020 project Sistemi Reversibili Concorrenti: dai Modelli ai Linguaggi. Irek Ulidowski has also been partially supported by JSPS Invitational Fellowship for Research in Japan ID S21050.

REFERENCES


**IVAN LANESE** is currently an Associate Professor with the University of Bologna, Bologna, Italy. He acted as the Vice Chair of the COST Action IC1405 on Reversible Computation. His research interests include 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 is the corresponding author of this article. Contact him at ivan.lanese@gmail.com.

**ULRIK P. SCHULTZ** is currently a Professor in aerial robotics with the University of Southern Denmark, Odense, Denmark. He recently participated in the COST Action IC1405 on Reversible Computing where he chaired the Working Group on Applications. His research interests include programming languages for robotics, his full biography is available at http://www.sdu.dk/staff/ups. Contact him at ups@sdu.dk.

**IREK ULIDOWSKI** is currently an Associate Professor with the University of Leicester, Leicester, U.K. 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/iru3. Contact him at irekulidowski@gmail.com.