From Reversible Semantics to Reversible Debugging

Ivan Lanese

From Reversible Semantics to Reversible Debugging*

Ivan Lanese[1]

Focus Team, University of Bologna/INRIA
ivan.lanese@gmail.com

Abstract. This paper presents a line of research in reversible computing for concurrent systems. This line of research started in 2004 with the definition of the first reversible extensions for concurrent process calculi such as CCS, and is currently heading to the production of practical reversible debuggers for concurrent languages such as Erlang. Main questions that had to be answered during the research include the following. Which is the correct notion of reversibility for concurrent systems? Which history information needs to be stored? How to control the basic reversibility mechanism? How to exploit reversibility for debugging? How to apply reversible debugging to real languages?

1 Introduction

Reversible computing is a computing paradigm where programs can be executed not only in the standard, forward direction, but also backward, recovering previous states of the computation. It has its origin in physics, with the observation by Landauer that only irreversible computations need to consume energy [24] fostering applications of reversible computing in low-power computing. Since then, it has attracted interest in fields as diverse as, e.g., hardware design [16], computational biology [8], program debugging [2], quantum computing [4], discrete simulation [10] and robotics [32]. This paper concentrates on the application area of debugging. Indeed, debugging is a time consuming and costly activity, hence improvements on debugging techniques may have huge practical impact. According to a 2014 study [43], the cost of debugging software amounts to $312 billion annually. Another recent study, by the Judge Business School of the University of Cambridge, UK, estimates that the time spent in debugging is 49.9% of total programming time, and evaluates the potential impact of reversible debugging to a saving of $81 billion per year [6]. Currently, reversible debugging for sequential languages is well understood, and has found applications both in industrial products, such as Windows Time Travel debugger [34], and in free software, such as GDB, which supports reversible debugging features since version 7.0.

* This work has been partially supported by COST Action IC1405 on Reversible Computation - extending horizons of computing. We thank Germán Vidal, Adrián Palacios and Irek Ulidowski for their useful comments and help.

However, nowadays most of the software is concurrent, either since the platform is distributed, as for the Internet or the Cloud, or to overcome the advent of the power wall [40]. Unfortunately, as we will show, reversibility in a concurrent setting is inherently different from reversibility in a sequential setting, hence sequential debugging techniques either do not apply to concurrent scenarios, or require to sequentialize the execution. This last approach, used for instance by UndoDB [42], on the one hand causes a large time overhead, and on the other hand loses information about concurrency and causality. This is a relevant drawback, since this information is at the heart of many bugs appearing in concurrent systems, notably the so called Heisenbugs, which also happen to be the most tricky to find [23], since they may appear or not in a computation depending on the speed of execution of the different processes[1]. Hence, the ultimate target of this paper is to explain the ideas underlying an approach to reversible debugging of concurrent systems called *causal-consistent reversible debugging* [18], and embodied in the CauDEr [30,29] debugger for a subset of the Erlang programming language. However, we will focus not only on the destination, but also on the journey, highlighting relevant questions, mostly theoretical, that had to be answered to put us in the place today to tackle the problem of reversible debugging of concurrent systems:

- Which is the correct notion of reversibility for concurrent systems? (Section 2)
- Which history information needs to be stored? (Section 3)
- How to control the basic reversibility mechanism? (Section 4)
- How to exploit reversibility for debugging? (Section 5)
- How to apply reversible debugging to real languages? (Section 6)

We will conclude the paper with an overview of CauDEr (Section 7) and an analysis of problems that remain open (Section 8).

2 Which notion of reversibility for concurrent systems?

Reversibility for sequential systems can be formulated as "recursively undo the last action", that is to undo a computation performing first action A, then action B, and finally action C, one has to undo first C, then B, and finally A. This definition is based on the notion of last action, that is on a total order provided by timing, which unfortunately is not well defined in concurrent systems, where the time span of execution of different actions may overlap. Hence, in order to deal with concurrent systems one has to find a different definition. Such a definition has been first proposed in [14] under the name of *causal-consistent reversibility*: "recursively undo any action provided that its consequences, if any, are undone beforehand". Notably, this definition is not based on timing, but on *causality*. This definition specifies which actions can be undone. This is complemented

[1] We use the term process to denote a sequential flow of execution, actually referring to both processes and/or threads.

by the so called Loop Lemma, also first proposed in [14, Lemma 6], stating essentially that executing one action and then undoing it should lead back to the starting state. These two definitions together ensure a main property of causal-consistent reversibility, namely that each state that can be reached from an initial state (that is, a state where no action has been performed yet, thus where no action can be undone) by mixing forward and backward steps can also be reached by forward steps only. Notably, other definitions of reversibility for concurrent systems exist [37], but they do not have (nor want to have) this property. This property has been proved in a number of formalisms [14,36,28,19,35] (mostly in a more general form called the Parabolic Lemma or the Rearranging Lemma). Below, however, we show it to be a very general, language-independent result, depending only on the definition of causal-consistent reversibility and on the Loop Lemma. The one below is the only technical part in this paper.

We assume a set of actions ranged over by $A, B, \ldots, A^{-1}, B^{-1}, \ldots$, where A^{-1} is the action undoing action A. We consider traces, that is finite sequences of actions, ranged over by σ. Trace concatenation is denoted by juxtaposition, and the empty trace is denoted by ϵ. A trace is forward if it contains no undo actions. We consider a symmetric switchability relation $\sim$, telling when the order of execution of two actions is irrelevant, and formalizing the concurrency model of the system under analysis.

Following [14, Definition 9], we define the *causal equivalence relation* $\approx$ as the smallest equivalence relation on traces closed under trace concatenation and satisfying the axioms below:

$$AA^{-1}{\approx}\epsilon \qquad A^{-1}A{\approx}\epsilon \qquad AB{\approx}BA \text{ iff } A{\sim}B$$

The first two axioms formalize the Loop Lemma, while the last one formalizes a notion of concurrency: two switchable actions can be executed in any order. We consider causal-consistent initial traces, defined as follows.

Definition 1 (Initial trace). *A trace σ is* initial *if actions can be undone only if they have been first performed. Formally, given an action A and a trace σ_p, $\mathrm{count}_A(\sigma_p)$ denotes the number of occurrences of A in σ_p. A trace σ is initial iff for each prefix σ_p of σ and each action A we have*

$$\mathrm{count}_A(\sigma_p) \geq \mathrm{count}_{A^{-1}}(\sigma_p)$$

Definition 2 (Causal-consistent trace). *A trace σ is* causal consistent *if actions can be undone only if their consequences, if any, are undone beforehand. Formally, fixed a trace σ, we denote with A_n the n-th occurrence of A in σ. A trace σ is causal consistent iff for each action occurrence A_n and each action occurrence B_m between A_n and A_n^{-1} either $A_n{\sim}B_m$ (B_m is not a consequence) or B_m^{-1} is before A_n^{-1} (B_m has been undone).*

We can now prove our result.

Theorem 1. *Each initial causal-consistent trace σ is causal equivalent to a forward one.*

Proof. The proof is by induction on the number of undo action occurrences in σ. If it is 0 then the thesis follows. Note that since σ is initial then for each undo action occurrence A_n^{-1} there is a unique action occurrence A_n in σ, and A_n precedes A_n^{-1}. Among all the undo action occurrences, select one such that the distance between the undo action occurrence and the corresponding forward action occurrence is minimal. Then, since σ is causal consistent, action occurrence A_n can be switched with all the action occurrences till A_n^{-1} (the other case would violate minimality). When the two action occurrences are adjacent they can be both removed using the Loop Lemma, reducing the number of undo action occurrences. The thesis follows by inductive hypothesis. □

3 Which history information needs to be stored?

We know from the previous section what we want to achieve, namely undoing actions in a causal-consistent way, but not how to achieve it. In general, an action can be undone only if it causes no loss of information, otherwise there are many possible predecessor states, and it is not possible to know which one is the actual predecessor. For instance, a simple assignment such as $X = 0$ loses the past value of variable X, hence states with any value of X are possible predecessor states. Thus, in order to reverse $X = 0$ one needs to keep history information, in particular the previous value of X. Note that an assignment such as $X{+}{=}\ 1$ does not cause any loss of information, and it can be undone simply by executing $X{-}{=}\ 1$. This observation is at the base of reversible languages such as Janus [45], which are obtained by restricting to only constructs which do not lose information. While Janus is an imperative language, the same approach has been applied also to object-oriented languages such as ROOPL [20], and functional languages such as RFun [44]. However, we want to debug existing languages, not to build new ones, and all mainstream languages cause loss of information, hence we cannot use the Janus approach. Thus, we need some history information.

However, there is some history information that we do not want. For instance, storing how many times an action has been undone would violate the Loop Lemma, since by doing and undoing the action such a counter would not go back to the starting value. Forgetting all information about undone actions, as required by the Loop Lemma, is a strong limitation, and in many cases one wants to drop such a restriction (see [25] for a detailed analysis). However, it is meaningful for debugging, since one is only interested in the state of the program under analysis, and not in how reversibility has been exploited to reach it, hence we will satisfy the constraint above.

We have shown an example of history information that would violate the Loop Lemma. More in general, there is a result, called the *causal-consistency theorem* [14, Theorem 1], characterizing which history information needs to be kept in causal-consistent reversibility. It states that two computations starting from the same state end in the same state iff they are causal equivalent. Apparently this has nothing to do with history information. However, history information is part of the state, hence ending in the same state also means having the same

history information. The right to left implication states that one cannot count how many times an action has been done and undone like above, and also that one cannot record the order in which actions are performed (otherwise swapping switchable actions would change the final state). The left to right direction states that if one would be able to reach the same state in two ways which are not causal consistent, then it must add history information to tell them apart. This may happen only in languages with nondeterminism, possibly due to concurrency coupled with interaction[2]. We give an example by using a simple nondeterministic assignment: assumes that $X = \{0, 1\}$ nondeterminstically assigns to X either 0 or 1. The program $X = \{0, 1\}; \texttt{if } X = 0 \texttt{ then } X = 5 \texttt{ else } X = 5$ has two possible forward computations, leading to the same state and not causal equivalent. Hence, history information needs to be added to distinguish the two final states.

The result above tells us, for a given language and notion of concurrency, which history information needs to be stored. Provided the correct history information, reversing single steps is normally not a too difficult task. By doing this, one obtains the *uncontrolled causal-consistent semantics* of the language, specifying how actions can be done and undone according to causal-consistent reversibility. This approach has been applied to many process calculi and programming languages, including CCS [14] and a family of similar calculi [36], CCS with broadcast synchronization [35], π-calculus [12], higher-order π [28], Klaim [19], μOz [33] and Core Erlang [31]. We remark however that the results above does not tell how to *efficiently* track and store history information, and how to efficiently exploit it for undoing actions.

4 How to control the basic reversibility mechanism?

In the previous sections we discussed how to undo actions, and which actions can be undone, but we have not discussed how to decide whether to go forward or backward, and which actions to undo when multiple ones can be undone according to causal-consistent reversibility. One needs control mechanisms to this end, to be defined on top of an uncontrolled reversible semantics. We refer to [27] for a categorization of control mechanisms, while here we discuss the so called `roll` operator, introduced in [26]. The `roll` operator answers a natural question: how to undo an arbitrary past action A, without violating causal-consistent reversibility? Since dependent actions have to be undone beforehand, undoing an arbitrary action needs to result in undoing all the tree of its causal consequences, from leaves to the root. One can see this as a sequence of uncontrolled steps which

[2] Concurrency alone is not enough. If the natural notion of switchability, allowing one to switch all pairs of concurrent actions, is chosen then all possible traces are causal equivalent

can naturally be implemented as a visit of the tree of consequences of action A:

```
Roll(A){
  for each B ∈ {direct consequences of A}
  {
    Roll(B)
  }
  A^-1
}
```

Notably, the definition above is language independent. Implementing the `roll` operator, and more in general operators to control reversibility, in terms of sequences of uncontrolled steps is important, since they inherit the nice properties of causal-consistent reversibility. For instance, we are guaranteed that using the `roll` operator never leads to states unreachable in forward computations.

5 How to exploit reversibility for debugging?

We discussed till now about reversibility, but not about debugging. However, the `roll` operator we presented above is at the basis of *causal-consistent reversible debugging*, introduced in [18]. Broadly speaking, the debugging process works as follows: given a visible misbehavior of the program, say a wrong output on the screen, one has to find the line of code containing the bug that caused such a misbehavior. Notably, the line of code performing the output may very well be correct, simply receiving wrong values from past elaborations. Also, the line of code performing the output and the one containing the bug may be in different processes, yet there should be some causality chain connecting the two, possibly including communication among processes. What we want is a technique that given a misbehavior points out its immediate causes. We can then examine them, to understand whether they are wrong, or we need to go to the causes of the causes, and iterate the procedure.

How to exactly find the causes of the misbehavior depends on the kind of misbehavior. Some common cases are the following.

Wrong value in a variable: in this case the cause is the last assignment to the variable (or its definition, if the language is functional).

Wrong value in a message: this refers to languages based on message passing, and in this case the cause is the send of the message. History information must be precise enough to find for each message the corresponding send, distinguishing in particular between equal messages. This can be obtained by adding unique identifiers to messages. Otherwise, all send operations that may have sent the message need to be inspected.

Unexpected message received: again this refers to languages based on message passing, and in this case the cause is the receive of the message. The difference between this case and the previous one is subtle: here we have an unexpected message, in the previous case the message was the desired

one but it contained wrong values. Receiving an unexpected message may happen, for instance, if the receive performs pattern matching to select the message, and the pattern is wrong. In case of doubt a safe approach is to first undo the receive, and, if it looks correct, go further back to the send.

Unexpected process: an action has been performed by a process that was not supposed to exist; in this case the cause is the action that created the process, which is of course in a different process.

The analysis above captures the idea of causal-consistent reversible debugging: follow the causes of a visible misbehavior towards the bug, possibly jumping from one process to the other. One can provide dedicated support to such a debugging strategy by exploiting the `roll` operator and the history information used to undo actions. First, history information is explored looking for the action A which is the direct cause of the misbehavior, then the `roll` operator is used to undo action A, thus going back to the state where A was executed. Such an approach can automatically find which is the process that performed action A, and which thus needs to be inspected. Analyzing the action A itself allows the programmer to decide whether the code of action A is correct or not, while inspecting the state allows him to find whether the error is due to wrong data from previous actions, and analysis should go further back.

6 How to apply reversible debugging to real languages?

Applying the techniques above to real languages involves a number of challenges. First, one needs to understand the notion of causality for the considered language. Causality has been deeply studied for process calculi, and such a study highlighted a number of subtleties emerging when considering mobility, as proved by the different notions of causality that exist in π-calculus [5,15,13]. We are not aware of any deep analysis of the topic in the setting of real programming languages (but for the one we did for Erlang [31]), hence this issue needs to be tackled preliminarily. As a second step, one needs to find the correct form of history information matching the chosen causality notion and satisfying the constraints defined in Section 3. Finally, one can define a reversible semantics for the chosen language and concurrency notion. Only at this point the actual implementation of the debugger can start. Of course, still many problems remain to be settled, e.g. concerning efficiency, usability, etc., but we concentrate here on the problems above.

As common in computer science, modular approaches can help in solving difficult problems. It may be the case that in the chosen programming language most of the constructs are not related to concurrency. In particular, this happens in languages based on message passing, where only primitives such as send, receive and spawn deal with concurrency. This is not the case in languages based on shared memory, since there any assignment and any access to a variable is potentially an interaction between threads. In languages based on message passing one can deal with sequential constructs using well-known approaches

from sequential reversibility, and only needs to study in depth primitives for concurrency.

Approximation can also be used. It may be the case that for a fixed set of observables it is difficult to characterize whether two actions are switchable. Consider for instance two processes reading messages from the same mailbox queue. Intuitively, the two read operations are not switchable. Yet they are switchable if the read values are the same, or if they are not used in the following computation, or if they result in the same following computation. This is a very complex behavior, but it can be approximated by saying that two read operations from the same queue never commute. The approximation corresponds to choose a concurrency model which is not the most concurrent one, but one that has a few fake dependences. This simplifies considerably both the theoretical work and the implementation, yet it may slow down the debugging process, since in order to perform a rollback more actions than really needed are undone. Furthermore, this may suggest possible interferences between processes which are actually independent, thus the programmer may try to find whether the bug is due to such an interference, what is obviously not possible.

7 CauDEr, a causal-consistent debugger for Erlang

We ended in the last section the description of the challenges faced to get causal-consistent reversible debuggers for real languages, and we describe now the current state of the topic. We present here CauDEr [30,29], which is currently the only existing causal-consistent reversible debugger for a real language, namely Erlang. We point out that CauDEr actually works on Core Erlang [9], but since Erlang is mapped into Core Erlang, it can actually deal with Erlang programs. We also remark that there is a causal-consistent debugger for μOz called CaReDeb [18,17], but μOz can hardly be considered a real language. There is also a causal-consistent debugger for actors programmed using Scala and the Akka library [3], called Actoverse [39,1], but it allows one to go back only to states where calls to the Akka library are performed. We will describe CauDEr below, yet we note that CauDEr is not the end of the journey, but actually only the beginning: CauDEr is just a prototype, and further work is needed in many directions. We will come back to this point in the final section.

The fact that CauDEr tackles Erlang is not a coincidence: Erlang is a concurrent and functional language based on message passing, more precisely on the actor paradigm [21]. Because of this, the sequential and the concurrent part are clearly separated. As discussed in Section 6, this greatly simplifies the development of the debugger.

CauDEr allows the user to load an Erlang program, and execute it. The Erlang program is first translated into Core Erlang, hence the whole debugging session takes place at the Core Erlang level. While being more constrained and less compact, Core Erlang is close enough to Erlang so that the user can work at this level with limited effort.

CauDEr provides three execution modalities: in the Manual modality the user selects for each step the process and the direction of execution, and the step is executed if it is enabled according to causal-consistent reversibility. In the Automatic modality the user selects a direction of execution and a number of steps, and the process is selected by a scheduler. In the Rollback modality the user can undo selected past actions in a causal-consistent way as described in Section 5. Broadly speaking, the Automatic modality is used to execute the program till a misbehavior appears, the Rollback modality to look for the bug, and the Manual modality for exploring the area where the bug is expected to be.

At each step the user can explore the state of the program, which is composed by a global mailbox, containing messages traveling in the network, and a set of processes. Processes include a process identifier, a local mailbox containing messages that reached the process but have not yet been retrieved, the history of the part of the computation already performed, the environment for variables, and the expression under evaluation.

Furthermore, a Trace, that is an abstract view of the concurrent events (send, receive and spawn) performed during the computation, is presented. Such a view is a total order of actions, thus it violates the constraints of causal-consistent reversibility: a more suitable view would present the partial order on these events defined by the causality relation. This is indeed an item in our future work.

Another relevant piece of information is shown when a rollback is executed: the Roll Log, that is the sequence of actions that have been undone as a consequence. This is particularly useful to spot missing or spurious dependences among actions. Indeed, if by rollbacking an action of process $P1$ some action of process $P2$ is also undone, but the two processes were supposed to be independent, then an interference between the two processes happened, and this may very well be the cause of the bug.

We will conclude this section by describing a typical debugging session in CauDEr. We consider as example a simplified version of the Transmission Control Protocol, taken from the github repository of the EDD debugger [41,7]. The expected behavior of the program is roughly as follows.

The program starts one server and two clients. Clients can start the connection to the server by sending SYN messages. One of the clients tries to connect on a port which is not available, and gets an RST message, meaning that the connection has been rejected. The other client tries to connect on a port which is instead available, and gets a SYN-ACK message. It answers with an ACK message followed by a message containing actual data. The server receives the data and sends them to the main process. See [41] for the details of the code.

By running the program (using Automatic modality) we immediately discover that the server has ended. It is enough to rollback it one step to see that the last action performed is to send an RST message. It is easy to see by looking at the code that after this send a recursive call of the server is missing.

Once this bug is fixed, we get a new misbehavior. By running the program we get as result of the `main` function the value `error_ack`, notifying that a

wrong ACK message has been received, while we were expecting a `client2` atom, representing the communicated data. By looking at the history of the `main` process we see that we received the value `error_ack` from message 7. We hence decide to rollback the send of the message. From the Roll Log we know that the message has been sent from process 5, which is a process spawned by the server when a SYN message on an available port is received to manage the new connection. We decide to go back using the Manual modality. At some point it seems we are not able to proceed futher back. This is due to an approximation of the causality relation, requiring, for undoing a receive, the local queue to be the same as when the receive was performed. Thus to solve the problem we need either to move back message 6 to the global mailbox, or to perform the rollback of the receive of message 5, which does this for us before undoing the receive.

We can now compare message 5 with the patterns of the receive, and discover that it does not match the main pattern, hence it triggers the error pattern. The reason for the mismatch is that the pattern expects an `ack` atom in second position, while it is in fifth position in the message. From the specification of our protocol we know that the `ack` atom should be in the second position, hence the error is in the message. We can now rollback the send of the message, and we discover it comes from process 4, that is one of the clients. By inspecting the code of the send we immediately discover the error: the values in the second and in the fifth positions were swapped. After fixing this second error the program works as expected.

8 Conclusion and future work

We described the challenges that put us today in the position of tackling the problem of causal-consistent reversible debugging of real languages, and a preliminary approach in this direction, namely the CauDEr debugger.

However, the problem is far from being solved, and many open issues remain. First, CauDEr deals only with a subset of Erlang, including the functional part and the actor primitives. Relevant aspects such as error handling and distribution are not covered. Second, efficiency needs to be improved in order to tackle large Erlang applications. We point out that efficiency is a limitation for many reversible debuggers also in the sequential setting (e.g., GDB), since building efficient reversible debuggers requires an hard work on tedious and ad-hoc optimizations. Third, currently CauDEr concentrates on the rollback primitives, yet more classic primitives such as breakpoints or exploration of the call stack would be a nice complement to them. Fourth, state visualization could be improved, allowing the programmer to zoom on the part of the code or of the state of interest. Finally, it would be very important to be able to capture a computation in the real Erlang execution environment (by instrumenting the source code) as a log and to replay the execution inside CauDEr. This would allow one to replay Heisenbugs and analyze them.

We described above the main development directions for CauDEr. We point out, however, that how this approach can be applied to other languages, and

to shared-memory languages in particular, is also a relevant direction for future research. Currently, the only approach supporting causal-consistent reversibility in a shared-memory setting is in the context of the coordination language μKlaim [19]. An approach to reverse a while language with interleaving parallelism (but no interaction between parallel processes) has been presented in [22], but it forces backward execution in reverse order. Refined approaches to reverse C++ code [38] and x86 assembly code under Linux [11] have been studied in the context of parallel discrete event simulation, but they do not consider concurrency and focus on recovering some specific past state. Nevertheless they can be an interesting starting point to apply our techniques to real shared-memory languages.

References

1. Actoverse website. `https://github.com/45deg/Actoverse`
2. Akgul, T., Mooney III, V.J.: Assembly instruction level reverse execution for debugging. ACM Trans. Softw. Eng. Methodol. 13(2), 149–198 (2004)
3. Akka. `http://akka.io/`
4. Altenkirch, T., Grattage, J.: A functional quantum programming language. In: LICS. pp. 249–258. IEEE Computer Society (2005)
5. Boreale, M., Sangiorgi, D.: A fully abstract semantics for causality in the pi-calculus. Acta Inf. 35(5), 353–400 (1998)
6. Britton, T., Jeng, L., Carver, G., Cheak, P., Katzenellenbogen, T.: Reversible debugging software – quantify the time and cost saved using reversible debuggers. `http://www.roguewave.com` (2012)
7. Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S.: EDD: A declarative debugger for sequential Erlang programs. In: TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 581–586. Springer (2014)
8. Cardelli, L., Laneve, C.: Reversible structures. In: CMSB. pp. 131–140. ACM (2011)
9. Carlsson, R., Gustavsson, B., Johansson, E., Lindgren, T., Nyström, S.O., Pettersson, M., Virding, R.: Core Erlang 1.0.3. Language specification (2004), available at URL: `https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf`
10. Carothers, C.D., Perumalla, K.S., Fujimoto, R.: Efficient optimistic parallel simulations using reverse computation. ACM Transactions on Modeling and Computer Simulation 9(3), 224–253 (1999)
11. Cingolani, D., Pellegrini, A., Quaglia, F.: Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES. In: PADS. pp. 211–222. ACM (2015)
12. Cristescu, I.D., Krivine, J., Varacca, D.: A compositional semantics for the reversible pi-calculus. In: LICS. pp. 388–397. IEEE Press (2013)
13. Cristescu, I.D., Krivine, J., Varacca, D.: Rigid families for CCS and the π-calculus. In: ICTAC. Lecture Notes in Computer Science, vol. 9399, pp. 223–240. Springer (2015)
14. Danos, V., Krivine, J.: Reversible communicating systems. In: CONCUR. Lecture Notes in Computer Science, vol. 3170, pp. 292–307. Springer (2004)
15. Degano, P., Priami, C.: Non-interleaving semantics for mobile processes. Theor. Comput. Sci. 216(1-2), 237–270 (1999)

16. Frank, M.P.: Introduction to reversible computing: motivation, progress, and challenges. In: 2nd Conference on Computing Frontiers. pp. 385–390. ACM (2005)
17. Giachino, E., Lanese, I., Mezzina, C.A.: CaReDeb website. URL: `http://www.cs.unibo.it/caredeb`
18. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: FASE. Lecture Notes in Computer Science, vol. 8411, pp. 370–384. Springer (2014)
19. Giachino, E., Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent rollback in a tuple-based language. J. Log. Algebr. Meth. Program. 88, 99–120 (2017)
20. Haulund, T., Mogensen, T.Æ., Glück, R.: Implementing reversible object-oriented language features on reversible machines. In: RC. Lecture Notes in Computer Science, vol. 10301, pp. 66–73. Springer (2017)
21. Hewitt, C., Bishop, P.B., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: IJCAI. pp. 235–245. William Kaufmann (1973)
22. Hoey, J., Ulidowski, I., Yuen, S.: Reversing imperative parallel programs. In: EXPRESS/SOS. EPTCS, vol. 255, pp. 51–66 (2017)
23. Huang, J., Zhang, C.: Debugging concurrent software: Advances and challenges. J. Comput. Sci. Technol. 31(5), 861–868 (2016)
24. Landauer, R.: Irreversibility and heat generated in the computing process. IBM Journal of Research and Development 5, 183 –191 (1961)
25. Lanese, I., Lienhardt, M., Mezzina, C.A., Schmitt, A., Stefani, J.B.: Concurrent flexible reversibility. In: ESOP. Lecture Notes in Computer Science, vol. 7792, pp. 370–390. Springer (2013)
26. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.B.: Controlling reversibility in higher-order pi. In: CONCUR. Lecture Notes in Computer Science, vol. 6901, pp. 297–311. Springer (2011)
27. Lanese, I., Mezzina, C.A., Stefani, J.: Controlled reversibility and compensations. In: RC. Lecture Notes in Computer Science, vol. 7581, pp. 233–240. Springer (2012)
28. Lanese, I., Mezzina, C.A., Stefani, J.B.: Reversibility in the higher-order π-calculus. Theor. Comput. Sci. 625, 25–84 (2016)
29. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr website. URL: `https://github.com/mistupv/cauder`
30. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: A causal-consistent reversible debugger for Erlang. In: FLOPS. Lecture Notes in Computer Science, vol. 10818, pp. 247–263. Springer (2018)
31. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. Journal of Logical and Algebraic Methods in Programming (2018), to appear. Available at `http://users.dsic.upv.es/~gvidal/lnpv17/paper.pdf`
32. Laursen, J.S., Schultz, U.P., Ellekilde, L.: Automatic error recovery in robot assembly operations using reverse execution. In: IROS. pp. 1785–1792. IEEE (2015)
33. Lienhardt, M., Lanese, I., Mezzina, C.A., Stefani, J.B.: A reversible abstract machine and its space overhead. In: FMOODS/FORTE. Lecture Notes in Computer Science, vol. 7273, pp. 1–17. Springer (2012)
34. McNellis, J., Mola, J., Sykes, K.: Time travel debugging: Root causing bugs in commercial scale software. CppCon talk, `https://www.youtube.com/watch?v=l1YJTg_A914` (2017)
35. Mezzina, C.A.: On reversibility and broadcast. In: RC. Lecture Notes in Computer Science, Springer (2018), to appear
36. Phillips, I., Ulidowski, I.: Reversing algebraic process calculi. Journal of Logic and Algebraic Programming 73(1-2), 70–96 (2007)

37. Phillips, I., Ulidowski, I., Yuen, S.: A reversible process calculus and the modelling of the ERK signalling pathway. In: RC. Lecture Notes in Computer Science, vol. 7581, pp. 218–232. Springer (2012)
38. Schordan, M., Jefferson, D.R., Jr., P.D.B., Oppelstrup, T., Quinlan, D.J.: Reverse code generation for parallel discrete event simulation. In: RC. Lecture Notes in Computer Science, vol. 9138, pp. 95–110. Springer (2015)
39. Shibanai, K., Watanabe, T.: Actoverse: a reversible debugger for actors. In: AGERE@SPLASH. pp. 50–57. ACM (2017)
40. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobbs Journal 30(3) (2005)
41. TCP example from EDD github repository. `https://github.com/tamarit/edd/tree/master/examples/Concurrency/tcp`
42. Undo Software: Undodb, commercial reversible debugger. `http://undo-software.com/`
43. Undo Software: Increasing software development productivity with reversible debugging (2014), `http://undo-software.com/wp-content/uploads/2014/10/Increasing-software-development-productivity-with-reversible-debugging.pdf`
44. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: RC. Lecture Notes in Computer Science, vol. 7165, pp. 14–29. Springer (2011)
45. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: PEPM. pp. 144–153. ACM (2007)