# Causal-Consistent Reversible Debugging

Elena Giachino, Ivan Lanese, Claudio Antares Mezzina

# Causal-consistent Reversible Debugging*

Elena Giachino[1], Ivan Lanese[1], and Claudio Antares Mezzina[2]

[1] Focus Team, University of Bologna/INRIA, Italy
[2] SOA Unit, FBK Trento, Italy
giachino@cs.unibo.it, lanese@cs.unibo.it, mezzina@fbk.eu

**Abstract.** Reversible debugging provides developers with a way to execute their applications both forward and backward, seeking the cause of an unexpected or undesired event. In a concurrent setting, reversing actions in the exact reverse order they have been executed may lead to undo many actions that were not related to the bug under analysis. On the other hand, undoing actions in some order that violates causal dependencies may lead to states that could not be reached in a forward execution. We propose an approach based on causal-consistent reversibility: each action can be reversed if all its consequences have already been reversed. The main feature of the approach is that it allows the programmer to easily individuate and undo exactly the actions that caused a given misbehavior till the corresponding bug is reached. This paper major contribution is the individuation of the appropriate primitives for causal-consistent reversible debugging and their prototype implementation in the CaReDeb tool. We also show how to apply CaReDeb to individuate common real-world concurrent bugs.

## 1 Introduction

Reversible debugging has been known for the last 40 years [8, 22], and gets all its interest and motivation from assisting the programmer in the search of possible bugs by exploring the computation both forward and backward. Retracing back the steps is very useful when investigating a misbehavior. In a sequential setting it is also very natural: steps are simply undone in reverse order of execution.

In a concurrent world, where multiple threads execute concurrently, there may not be a unique "last" action. Thus, the concept of reversibility has been interpreted and implemented in different ways, depending on the answer to the following question: When a misbehavior is encountered, how can one proceed in order to retrace the steps towards the bug?

We will describe the different approaches on a simple scenario. Train passengers are taking their seats on a train. Some of them have reserved a seat, some others have not. Those without reservation pick randomly a free seat. Those with reservation take their assigned seat unless they find it has been occupied by someone else, in which case they pick a free one. What should not happen is

* This work has been partially supported by the French National Research Agency (ANR), project REVER n. ANR 11 INSE 007.

that some passenger X with a reservation finds himself standing without a seat. If this happens, how do we find the problem and fix it? Following the approaches in the literature, we could:

**Record/reverse-execute debugging: [9, 4]** start asking people to stand up in the exact reverse order they occupied the seats. Then we risk making many innocent people leave their seats before finding the one who is occupying the seat of passenger X.

**Checkpoint/replay debugging: [20, 1, 19]** send everyone out of the train and start again the sitting algorithm. But this time passengers may choose seats in different orders, thus the problem may not occur, or a different passenger may be left without the seat he is entitled to.

What one would like to do is to undo the sitting of the "causal predecessor" of passenger X: the one who took his seat. If, in turn, he had another reservation, then we would undo the sitting of his causal predecessor and so on. In this way, we can possibly find a place for the passenger with reservation by undoing a limited number of seat actions.

This form of reversibility, called *causal-consistent* is quite natural in practice as the example above shows, but has never been applied to debugging as far as we know. The term causal-consistent highlights that actions are reversed by respecting causes: only actions that have caused no successive actions can be undone. In other words, concurrent actions can be reversed in any order, while dependent actions are reversed starting from the consequences. Causal-consistent reversibility has been studied mainly in the field of process calculi [6, 17, 12, 5]. The key contribution of this paper is to individuate the primitives enabling to apply the abstract theory of causal-consistent reversibility to help programmers to debug concurrent applications. In general, one finds a misbehavior in a concurrent program and has to find the instruction in the code that caused it. This instruction may be in any of possibly many threads, and far back in the code. We provide new primitives allowing the programmer to go back in the computation following the causes of the misbehavior till the bug is reached. For instance, if the fault is a wrong value of a variable, we provide a primitive to find and undo the (last) assignment to this variable. We will show how this and similar primitives can be applied to various categories of common bugs found in real-world concurrent applications. We also present CaReDeb, a prototype implementation of our approach.

While the idea of applying causal-consistent reversibility to debugging can be applied in principle to any concurrent language, and the debugging primitives needed to implement this idea do not change much from one language to the other, their actual implementation relies on the definition of a causal-consistent reversible semantics for the language. As far as we know, the only programming language equipped with a causal-consistent reversible semantics is $\mu$Oz [15]. Thus, we have chosen $\mu$Oz for our studies.

| | | |
|---|---|---|
| $S ::=$ | | Statements |
| | **skip** | Empty statement |
| $\mid$ | $S_1\ S_2$ | Sequential composition |
| $\mid$ | **let** $x = v$ **in** $S$ **end** | Variable declaration |
| $\mid$ | **if** $x$ **then** $S_1$ **else** $S_2$ **end** | Conditional statement |
| $\mid$ | **thread** $S$ **end** | Thread creation |
| $\mid$ | **let** $x = c$ **in** $S$ **end** | Procedure declaration |
| $\mid$ | $\{\ x\ x_1 \dots x_n\ \}$ | Procedure call |
| $\mid$ | **let** $x =$ `NewPort` **in** $S$ **end** | Port creation |
| $\mid$ | $\{$ `Send` $x\ y\ \}$ | Send on a port |
| $\mid$ | **let** $x = \{$ `Receive` $y\ \}$ **in** $S$ **end** | Receive from a port |
| $v ::=$ | **true** $\mid$ **false** $\mid$ 0 $\mid$ 1 $\dots$ | Simple values |
| $c ::=$ | **proc** $\{\ x_1 \dots x_n\ \}\ S$ **end** | Procedure |

**Fig. 1:** $\mu$Oz Syntax

## 2 The $\mu$Oz Language

In this section we informally present $\mu$Oz, a fragment of the Oz language, whose complete theoretical treatment can be found in [15]. Some details relevant for debugging are summarized in Section 6. $\mu$Oz is a higher-order language featuring thread-based concurrency and asynchronous communication via ports.

The syntax of $\mu$Oz is in Figure 1. Values in $\mu$Oz are booleans, natural numbers, (communication) ports and procedures. Variables are immutable, i.e. read-only variables that are initialized at the time of their declaration. Communication is asynchronous and is realized by means of send and receive actions on a port, to which a FIFO queue is associated. Variable declaration, procedure declaration, port creation and receive are binders. Specifically, $x$ is bound in $S$ in **let** $x = v$ **in** $S$ **end**, **let** $x = c$ **in** $S$ **end**, **let** $x =$ `NewPort` **in** $S$ **end**, and **let** $x = \{$ `Receive` $y\ \}$ **in** $S$ **end**.

## 3 Causal consistent debugging

In this section we describe the commands enabling causal-consistent reversible debugging, which are also implemented in our causal-consistent reversible debugger prototype CaReDeb [2]. Clearly, some of the available commands are standard for (reversible) debuggers, while others are peculiar of our causal-consistent approach. We will give more emphasis to the last ones. For simplicity, we also distinguish commands for controlling the execution (labeled with "control" in Table 1) from those for exploring the configuration of the program under debugging (labeled with "explore" in Table 1). Most of the commands can be abbreviated: abbreviations are in parenthesis after the command name in Table 1.

| | | |
|---|---|---|
| control | **forth (f) t** | (forward execution of one step of thread `t`) |
| | **run** | (runs the program) |
| | **rollvariable (rv) id** | (causal consistent undo of the creation of variable id) |
| | **rollsend (rs) id n** | (causal consistent undo of last n send to port `id`) |
| | **rollreceive (rr) id n** | (causal consistent undo of last n receive from port `id`) |
| | **rollthread (rt) t** | (causal consistent undo of the creation of thread `t`) |
| | **roll (r) t n** | (causal-consistent undo of n steps of thread `t`) |
| | **back (b) t** | (backward execution of one step of thread `t` (if possible)) |
| explore | **list (l)** | (displays all the available threads) |
| | **store (s)** | (displays all the ids contained in the store) |
| | **print (p) id** | (shows the state of a thread, channel, or variable) |
| | **history (h) id** | (shows thread/channel computational history) |

**Table 1.** CaReDeb main commands

Commands for forward execution are standard: command **forth t** executes a single step in a given thread **t**, while command **run** executes the program under a round-robin scheduler (breakpoints may be used to stop the execution).

Commands for backward execution are more peculiar. The main feature of our debugger is a suite of commands that undo the last action that produced some unexpected visible behavior (the behavior is visible by analyzing the state of the application). We present these commands by listing the possible visible bad behaviors:

**Wrong value in a variable:** if a variable **id** has an unexpected value, command **rollvariable id** allows the programmer to go to the state just before the creation of variable **id**;

**Wrong value in a queue element:** if an element of the queue associated to port **id** has an unexpected value, command **rollsend id n** allows the programmer to undo the last **n** sends to this port. If **n** is unspecified, the last send is undone;

**Thread blocked on a receive:** if a thread is blocked on a receive on an empty queue, it may be the case that the desired message has been read by another thread. This can be checked by looking at the history of the queue, which contains the messages that were in the queue in the past. In this case, command **rollreceive id n** allows the programmer to undo the last **n** receives on the port **id**. If **n** is unspecified, the last receive is undone;

**Unexpected thread:** if an unexpected thread **t** is found, command **rollthread t** allows the programmer to undo the creation of the thread.

All these commands are causal-consistent, i.e. they undo all the actions which depend on the target action, while not undoing concurrent actions. For instance, undoing the send of a value requires to undo the reception of the same value, if it has been performed and not yet undone. Similarly, undoing the creation of a thread requires to undo all the actions performed by the created thread. This is fundamental to ensure causal consistency: on one side this ensures we go back to a past state that could have been reached by a forward execution, on the other

side we undo the minimal number of actions needed to reach this aim. These commands also print information on which actions have been undone, and in which order.

While these commands are the ones more in line with our philosophy of going back following the causes of misbehavior, other commands can be used by the programmer to go back following his intuition. In particular, we provide command **roll t n** which undoes (in a causal-consistent way) the last **n** steps done by thread **t**. Command **back t** is the symmetric of **forth t**, and undoes a single step of thread **t**. Notably, this command is enabled only if all the consequences of the step (if any) have already been undone.

Commands for exploring the configuration can be divided in two categories: commands for exploring the standard information (state, code, ports), and commands for exploring the history of the computation and of ports. Note that variables have no associated history information, since their values cannot be changed.

Among standard commands, we have the command **list** to display the list of threads (including whether they are active or terminated), and the command **store** to display the identifiers in the store. The content of a given identifier **id** can be printed by command **print id**. According to what **id** is, it may print the value of a variable (possibly a procedure), the queue associated to a port, or the code still to be executed by a thread. History information is displayed by command **history id**. Here **id** may refer to a thread, and in this case the history is the list of the past actions executed by the thread, or to a queue, and in this case the history is the list of messages that were in the queue and have been read.

## 4 Assessment: real-world concurrency bugs

In this section we evaluate our debugging techniques against real world concurrency bugs as described in [16]. Amazingly, all real world concurrency bugs reported have a simple pattern, involving a small number of variables, threads, and resources. They are however small snippets immersed in thousands of lines of code of huge applications such as Mozilla, Apache, MySQL and OpenOffice. This tells us that it is not necessary to find complex bug examples in order to reason about real world.

According to [16], real world concurrency bugs are mainly of three kinds: *order violation*, *atomicity violation*, or *deadlock*. We show below an example for each class of bug, and apply our debugging primitives to isolate them. The bugs were originally in C/C++ programs, but we recast them here in $\mu$Oz.

An order violation bug occurs when the programmer assumes a given order among two actions, but those actions may actually occur also in a different order. A simple example of order violation bug in $\mu$Oz follows:

```
let one = 1 in
let two = 2 in
let k = port in
```

```
thread {send k one} end; // t_1
thread  {send k two} end; // t_2
thread let x = {receive k} in skip end end //t_3
end end end
```

Here the programmer assumed that value `one` would be sent before value `two` but did not enforce this property. In fact, even if thread *t_2* is created after thread *t_1*, it may run faster and execute its sending of value `two` before the sending of value `one` from thread *t_1*. When this happens, the programmer may note two possible misbehaviors: (i) variable `x` is 2, while 1 was expected, or (ii) the port contains value 1, while 2 was expected.

In the first case, the most natural thing to do is to execute **rollvariable x**. This would put back the variable in the queue. Notice that one can do this without knowing where in a possibly huge code the receive was. One can see by inspecting the queue that the two values are not in the expected order. Using the command **rollsend k** twice one can put back the two messages, thus finding the send which caused the misbehavior. Also, the fact that, when undoing the send of `one`, the send of `two` is not undone by the causal-consistent mechanism confirms that the expected dependency was not enforced.

In the second case, one can inspect the history of port `k` using command **history k** and see that value `two` was indeed in the queue but has been read. Using command **rollreceive k** the value is put back in the queue. From here the same technique used above can be applied.

We stress here the fact that it may seem easy to catch this kind of bugs in this simple example, but actually these threads may not be so much distinguishable when immersed in the whole program, and our debugging techniques can be applied in the exact same way, since they require only to know the misbehavior, not of being aware of the involved instructions: these are highlighted by the debugging commands.

An atomicity violation bug occurs when the programmer assumes that two actions are executed in an atomic way, but does not enforce this atomicity constraint. A simple example of atomicity violation bug in $\mu$Oz follows:

```
let t = true in
let f = false in
let k = port in
thread {send k t} ; let x = {receive k} in skip end end; // t_4
thread  {send k f}  ; let y = {receive k} in skip end end // t_5
end end end
```

Here the programmer assumed the pairs of send and receive on channel `k` in threads *t_4* and *t_5* to be atomic, but does not enforced this property. In fact, it is possible that thread *t_5* receives the value intended for thread *t_4* and vice versa. One can see as misbehavior the fact that `x` and `y` have not the expected value. Without knowing where this value has been assigned, one can use commands **rollvariable x** and **rollvariable y** to undo the corresponding assignments. From the output of the debugger one immediately discovers which thread is responsible of the assignment. When the two values are back in the queue of

port `k` one can use command **rollsend k** and immediately discover that the send has not been performed by the expected thread, thus finding the bug.

A simple example of *deadlock* in $\mu$Oz is the following:

```
let t = true in
let k = port in
thread {send k t} end;  // t_7
thread let x = {receive k} in {send k t} end end; // t_8
thread let y = {receive k} in skip end  end// t_9
end end end
```

Here, if the send from thread *t_7* is received by thread *t_9* instead of *t_8*, then the execution of thread *t_8* blocks indefinitely because its receive precedes its send. By looking at the list of threads using command **list**, one can see that thread *t_8* has not terminated its computation. One can look at the code of the thread using command **print** $t_8$ and see that it is waiting on a receive on port `k`. The first thing that one can do is to look into the history of the port in order to see if some message was ever put into the queue. By executing command **history k** one finds that a message was actually inside the queue in the past, and was picked up by another thread. Then, by means of command **rollreceive k**, one can undo the receive of the message. One can see from the output of the debugger that this requires to undo actions of thread *t_9*. This highlights the fact that thread *t_9* is also involved in the deadlock. Even more, one has discovered exactly which are the two receive actions and the unique send action critical for the deadlock to occur, and has all the needed information to fix the bug.

An interesting fact reported in [16] is that most of the bugs (101 out of the 105 reported) involve no more than 2 threads. This highlights the importance of the fact that the techniques above allow the programmer to immediately find the involved threads starting from the unexpected behaviors, and to avoid to undo all the actions of the many unrelated threads.

## 5 Debugging a concurrent application

In this section we describe a use case for CaReDeb [2], more complex than the paradigmatic examples discussed in the previous section. Differently from those bugs, the bug considered here is not concurrent per se. Nevertheless, the concurrent nature of the application makes it more difficult to individuate.

Our sample program (inspired by [18]) implements a procedure for handling purchase orders. Figure 2 (without the dashed part) depicts its intended behavior. Before an order is placed, two conditions must be verified: the availability of the customer's credit, and the completeness of the delivery details. The two independent checks `CheckCredit` and `CheckAddress` are performed concurrently. If the credit is insufficient, `CheckCredit` invokes the procedure `ProposeLoan` that offers to the client a loan of 20% of his credit for the purchase. A positive answer is sent as a result if the updated credit is enough to match the price, a negative response otherwise. The results of the checks are sent to the `asynchAnd` procedure

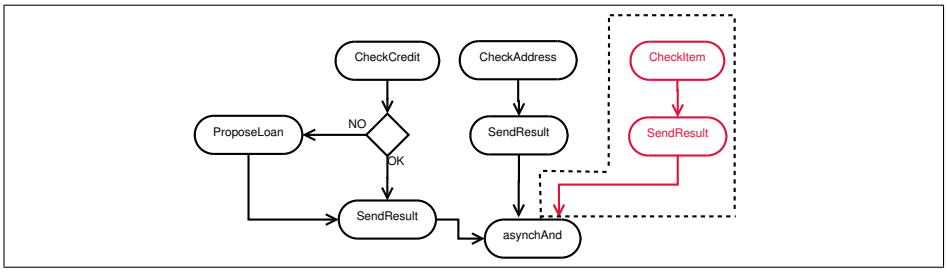


**Fig. 2:** The purchase workflow.

as soon as they are available. The `asynchAnd` procedure performs a short-circuit evaluation of n-ary AND: if it receives the value `false`, then it immediately produces `false` as a result, otherwise it waits for another value. If no more values are expected, then it sends the value `true` as a result.

The definition of the procedure `asynchAnd` is as follows, where `n`, `inp` and `out` are the formal parameters corresponding to the number of expected operands, the input and the output ports, respectively.

**Listing 1.1.** The asynchronous AND procedure

```
let asynchAnd = proc {n inp out}
                if (n>0) then let k={receive inp} in
                              let v={receive k} in
                              let m = n-1 in
                              if v then {asynchAnd m inp out}
                              else {send out false} end
                              end end end
                else {send out true} end
                end in
```

Since the operands are computed independently, they are sent to different ports. The statement `k={receive inp}` receives the id of the port `k` on which to wait for the operand `v`. If `v` is true, then the procedure is recursively called waiting for the `n-1` values left. Otherwise, the conjunction fails and `false` is sent over the result port `out`. When all the expected values have been gathered (`n=0`) the item can be delivered (i.e., `true` is sent over `out`).

The program defines three concurrent threads:

```
thread {asynchAnd 2 input result} end; //t_1
thread {send input outCr};{checkCredit outCr inCr} end; //t_2
thread {send input outAdd};{checkAddress outAdd} end; //t_3
```

The first thread invokes the asynchronous AND for two values expected on ports whose names are sent over port `input`. The result is sent on port `result`. The other two threads send on `input` the ports `outCr` and `outAdd`, respectively, on which the value will be communicated, and then invoke the corresponding check procedure.

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 Figure 2). A new port `outIt` and a new parallel thread are created:

```
thread {send input outIt}; {checkItem outIt} end //t_4
```

Hence we have three concurrent `send`s over port `input`. The order in which `asynchAnd` is going to process the results of the checks depends on the scheduling order of those `send`s.

Before putting the upgraded version of the program to work (the code is available on CaReDeb web site [2]), we want to test its behavior. We consider a test case where the price of the item is 15 euros and the credit amount is 10 euros. We also assume that the item is in stock and that the delivery details are fine. We execute the test using the **run** command of CaReDeb, which executes the program till the end. We know that, in this test case, the purchase should be rejected. Instead, within our test run, the procedure `asynchAnd` returns `true`. A bug is somewhere in our program and we need to find it. The problem may lie either in a wrong assessment of the credit, or in a wrong assessment of the loan, or in a bug within the asynchronous AND. Even in this simple example we cannot say a priori where the bug can be found.

Since something is clearly wrong with the value received on port `result`, we jump back just before that value was sent, by performing **rollsend result**. This points us back to thread *t_1* executing the else branch in the body of `asynchAnd`. Unfortunately, in this case, this does not give us much insight on the source of the bug, because it happens to roll back of one step only thread *t_1*. Anyway, we are now sure that the problem was not related to conflicting sends to port `result`.

At this point, following the intuition that something wrong may be within the control of the credit performed by the procedures in thread *t_2*, we execute **roll t_2 1** and we cause the reversing of the last action within procedure `proposeLoan`, namely the sending of the result to `asynchAnd`. Somehow surprisingly, this does not cause the undo of other actions. The fact that no `receive` inside the `asynchAnd` procedure needed to be undone means that the result of `proposeLoan` has not been considered by `asynchAnd`. By looking at the `send` just undone we also notice that the sent value is correct. Therefore we can exclude that the problem is in the check of the credit availability, which is performed properly.

Notably, if we would have undone the execution of thread *t_4*, managing the checking of the item availability, we would have seen a very different behavior. This could have been done, for instance, by using command **rollthread t_4**. As a result, also part of the computation of thread *t_1* performing the `asynchAnd` would have been undone, showing an actual dependence between the two threads.

Let us go back to our debugging strategy. Since we now think that the problem is due to the `asynchAnd` procedure, we can undo its execution step by step looking for the bug, using command **back t_1**. When we reach the beginning of

the thread we notice that `asynchAnd` was invoked with 2 as number of operands, while we would have expected 3 operands. This is the bug we were looking for. This was due to the fact that the invocation was not updated after the check of the item had been introduced.

The commands **roll** and **back** allowed us to check the guesses about the location of the bug, independently from the possible interleavings of thread execution. With the checkpoint/replay debugging one could have experienced a different scheduling of the threads at every attempt of action reversing, thus the bug could have showed or hidden itself in an unpredictable way. With the record/reverse-execute debugging, instead, one would have needed to reverse the actions following a strict chronological order, possibly needing to reverse many unrelated actions before finding the bug.

## 6 Underlying theory

We summarize here a few theoretical notions, mainly adapted from [15], which ensure the soundness of our debugging strategy.

### 6.1 Causality Relation

We present below the notion of causality for $\mu$Oz we rely on to define causal consistency.

**Definition 1 (Dependent reductions).** *We define below when two forward reductions are (causally) dependent:*

1. *a reduction of a thread depends on the previous reductions of the same thread;*
2. *all the reductions of a thread depend on the thread creation;*
3. *all the uses of a variable depend on its creation;*
4. *a receive of an element from a queue depends on the send of this element;*
5. *a send on a queue depends on the previous sends to the same queue;*
6. *a receive from a queue depends on the previous receives from the same queue.*

*Remark 1.* The conditions $1-4$ above correspond to the conditions defining the well-known *happens-before* relation [11] provided by Lamport. Conditions *5* and *6* above instead have no correspondence in the happens-before, and formalize the fact that queues are order preserving. Thus our causality model is stricter than Lamport's.

### 6.2 $\mu$Oz semantics

The debugger relies on the definition of a causal-consistent semantics for $\mu$Oz, whose main features are described below.

The operational semantics of $\mu$Oz is defined in [15] by a simple stack-based abstract machine. It exploits an extended syntax featuring also tasks and threads (used for statement execution), port queues (for communication), and the store.

Tasks are a parallel composition of threads. Threads are stacks of statements. The store is a conjunction of bindings, procedures, and ports (essentially implemented as named FIFO queues).

The standard $\mu$Oz semantics is defined as a reduction relation, denoted $\rightarrow$, between configurations of the form $(U, \sigma)$, where $U$ is a task and $\sigma$ is a store (0 is the empty store).

Let us comment a few sample reduction rules. Rule R:npt creates a new port $x'$, by putting a new binding $x' = \xi$ in the store. Also, $x'$ is substituted for $x$ in the scope $S$ to avoid variable capture. A queue is associated to $\xi$ and initialized to $\bot$ (the empty queue). Task $T$ is the continuation of task $S$.

$$[\text{R:npt}](\langle \textbf{let } x = \texttt{NewPort } \textbf{in } S \textbf{ end } T\rangle,\ 0) \rightarrow (\langle S\{^{x'}/_{x}\}\ T\rangle,\ x' = \xi \parallel \xi : \bot),$$

with $x', \xi$ fresh.

Rule R:snd performs a send, by enqueuing variable $y$ in the queue of port $x$.

$$[\text{R:snd}](\langle \{\ \texttt{Send}\ x\ y\ \}\ T\rangle,\ x = \xi \parallel \xi : Q) \rightarrow (T,\ x = \xi \parallel \xi : y; Q)$$

Rule R:rcv performs a receive, dequeuing the corresponding element $z$ and fetching its value $w$. The value $w$ is assigned to the fresh variable $x'$ that substitutes the formal variable $x$.

$$\begin{aligned}[\text{R:rcv}](\langle \textbf{let } x = \{\ \texttt{Receive}\ y\ \} \textbf{ in } S \textbf{ end } T\rangle,\ y = \xi \parallel \xi : Q; z \parallel z = w)\\ \rightarrow (\langle S\{^{x'}/_{x}\}\ T\rangle,\ y = \xi \parallel \xi : Q \parallel z = w \parallel x' = w)\end{aligned}$$

with $x'$ fresh.

The reduction relation $\rightarrow$ is closed under evaluation contexts (and structural congruence). We refer to [15] for further details.

*Debug-mode semantics.* The debugger relies on a causal-consistent reversible semantics, which keeps track of history and causality information. This semantics is proved to be a conservative extension of $\mu$Oz semantics, i.e. forward computations during debugging are indeed a decorated version of the $\mu$Oz reductions [15].

In the reversible semantics, threads have a name $t$ (which is unique) and a history $H$, and execute an extended statement stack $C$. The history stores information about executed statements. Sent variables are stored in the queue, not in the history. Also, for an if statement just the discarded branch has to be stored, since the other one is available in the thread code. History is needed also inside ports, to remember the order of communications.

The semantics is defined by means of two reduction relations, a forward relation $\twoheadrightarrow$ and a backward relation $\rightsquigarrow$.

Let us see, as an example, how the instrumented forward reduction rules for communication and the corresponding backward rules are defined. Rule R:fw:npt stores $*x'$ in the history, meaning that $x'$ has been used as fresh port, and uses the scope delimiter **esc** to recall the scope of the binding. The created queue comes with an empty history $\bot$.

$$\begin{aligned}[\text{R:fw:npt}]\ &(t[H]\langle \textbf{let } x = \texttt{NewPort } \textbf{in } S \textbf{ end } C\rangle,\ 0)\\ &\twoheadrightarrow (t[H\ *x']\langle S\{^{x'}/_{x}\}\ \langle \textbf{esc } C\rangle\rangle,\ x' = \xi \parallel \xi : \bot|\bot)\end{aligned} \qquad x', \xi \text{ fresh}$$

Rule R:fw:snd stores $\uparrow x$ in the history, to record the sending on port $x$. Also, the name $t$ of the thread sending the value is stored in the queue together with the variable $y$, to avoid that a different thread takes the value when rolling back.

$$\begin{aligned}[\text{R:fw:snd}]\ (t[H]\langle\{\ \texttt{Send}\ x\ y\ \}\ C\rangle,\ x = \xi \parallel \xi : K|K_h) \\ \twoheadrightarrow (t[H\ \ \uparrow x]C,\ x = \xi \parallel \xi : t{:}y; K|K_h)\end{aligned}$$

Rule R:fw:rcv stores $\downarrow x(y')$ in the history, to record that $y'$ has been received from port $x$. The read value is also kept in the queue history, with information on which thread read it.

$$\begin{aligned}[\text{R:fw:rcv}]\ (t[H]\langle\textbf{let}\ y = \{\ \texttt{Receive}\ x\ \}\ \textbf{in}\ S\ \textbf{end}\ C\rangle,\ \theta \parallel \xi : K; t' : z|K_h) \\ \twoheadrightarrow (t[H\ \ \downarrow x(y')]\langle S\{^{y'}/_{y}\}\ \langle\textbf{esc}\ C\rangle\rangle,\ \theta \parallel \xi : K|t' : z, t; K_h \parallel y' = w) \\ \text{if} \quad y' \text{ fresh } \wedge\ \theta \triangleq x = \xi \parallel z = w\end{aligned}$$

The backward rules are in one to one correspondence with the forward ones, and use the stored information to get back to the original state. Notably, rules R:bk:npt and R:bk:rcv below go back to a term which is not the starting one, but which is equivalent up to $\alpha$-conversion. Also, they exploit the scope delimiter **esc** to identify the scope of the statement to be reversed. The occurrence of **esc** in the rule is always matched by the nearest occurrence in the term. In rule R:bk:npt the $\perp$ in the history ensures that the actions on the port are rolled back before its creation is rolled back.

$$\begin{aligned}[\text{R:bk:npt}]\ (t[H\ \ *x]\langle S\ \langle\textbf{esc}\ C\rangle\rangle,\ x = \xi \parallel \xi : \perp|\perp) \\ \rightsquigarrow (t[H]\langle\textbf{let}\ x = \texttt{NewPort}\ \textbf{in}\ S\ \textbf{end}\ C\rangle,\ 0)\end{aligned}$$

$$\begin{aligned}[\text{R:bk:rcv}]\ (t[H\ \ \downarrow x(z)]\langle S\ \langle\textbf{esc}\ C\rangle\rangle,\ z = w \parallel x = \xi \parallel \xi : K|t' : y, t; K_h) \\ \rightsquigarrow (t[H]\langle\textbf{let}\ z = \{\ \texttt{Receive}\ x\ \}\ \textbf{in}\ S\ \textbf{end}\ C\rangle,\ x = \xi \parallel \xi : K; t' : y|K_h)\end{aligned}$$

### 6.3 Properties of debugging

The soundness of our debugging follows from two results from the theory of causal-consistent reversibility [6]. Without going into the technical details, we just want to emphasize their relation with the debugging.

**Proposition 1 (Debugging Soundness).**

1. *Every reduction step can be reversed.*
2. *Every state reached during debugging could have been reached by a forward-only execution from the initial state.*

The first item ensures that the debugger can undo every forward step, and, viceversa, it can re-execute every step previously undone. This property is known as Loop Lemma [6, Lemma 6], and has been proved for $\mu$Oz in [15, Lemma 3].

The second item ensures that any sequence of forward and backward debugging commands can only reach states which are part of normal forward-only computations. This property is known as Parabolic Lemma [6, Lemma 10], and its proof for $\mu$Oz is analogous to the one in [6].

## 7 Implementation aspects

We have implemented a prototype of our causal-consistent debugger, CaReDeb [2], to test it in practice. CaReDeb is implemented in Java. It provides all the primitives presented in Section 3. The most interesting aspects of the implementation lie in the treatment of the **roll** command, and its variants **rollvariable**, **rollsend**, **rollreceive**, and **rollthread**. In fact, the step-by-step backward command **back** follows strictly the semantics presented in Section 6, and it is guaranteed to be correct by Proposition 1.

The command **roll** actually undoes actions in the target thread following the semantics in Section 6, and thus the correctness result holds also for it. However, if one such action has dependences, these dependences must be retrieved and rolled back beforehand. This requires to find both the dependent thread and the dependent action inside it. One could in principle find these actions by inspecting the histories of the threads in the system, but for efficiency reasons we explicitly write this information near to the action having the dependency as a pair `(thread_id,`$\gamma$`)` where $\gamma$ is a natural number pointing to a specific action of thread `thread_id`. The only actions that may have dependencies are communication actions and thread creation actions.

When reversing a thread creation action, if the child thread memory is not empty, then a `ChildMissingException` is thrown containing the name of the child thread: this thread will be fully reversed before continuing.

When a `send` operation on some port `k` is being reversed, the corresponding message must be removed from the queue of `k`. However, this can be done immediately only if it was the last message sent on port `k`. Otherwise, either the message has been already read or other messages have been sent afterwards. In the first case also the `receive` of the message (and all its dependencies) must be reversed. In the second case, the sending of the other messages have to be reversed in order to maintain causal consistency. In both the cases a `WrongElementChannel` exception is raised with either a list of reading threads or of sending threads to be reversed as arguments. Actually, the list contains pairs of the form `(thread_id,`$\gamma$`)`, so that each `thread_id` is reversed back to its $\gamma$-th instruction.

One may think that also port creation should check for dependencies. This is not the case: undoing a port creation simply removes the port from the store. Actually, this is safe. In fact, assume that other threads interacted with this port. Then, they must know the name of the port, either by receiving it by a communication with the port creator, or by having been created by the port creator. In both the cases, these actions are undone before the port creation can be undone, thus all the operations on the port have been undone before the undo of the port creation.

## 8 Related work and conclusion

We presented a debugging technique allowing the programmer to look for bugs in a concurrent program by following backward causal dependencies from the

misbehavior to the bug. We also presented CaReDeb [2], a prototype debugger for $\mu$Oz, a fragment of the Oz language, enabling such an approach. The approach relies on storing history and causality information, and is based on the solid theory of causal-consistent reversibility.

Reversibility for debugging for sequential programs has been quite extensively explored [14, 7, 10, 3]. The interplay between reversibility and concurrency makes things more complex: concurrent reversible debugging is a less explored world. All the approaches to concurrent reversible debugging in the literature fall in the two categories below:

**Record/reverse-execute debugging:** A log is kept while executing, and when going back thread activities are undone in the exact order they were executed, preserving the particular interleaving [9, 4].

**Checkpoint/replay debugging:** No log is kept[3], and in order to go back to a previous step, the execution is replayed from the start (or from a previous checkpoint) until that step [20, 1, 19].

Both approaches present drawbacks. In the first case, if the error was due to one among a million of independent parallel threads, and that one was the first one to execute, one needs to undo all the program execution before finding the bug. Even more, one does not understand which threads are related to a given misbehavior, since there is no information on the relations among them. Following the second approach, actions could get scheduled in a different order at every replay, and the error may not get reproduced. Even if it does, one may not get any insight on the causes of the error.

Causality in the context of non-reversible concurrent debugging has been addressed in different works [13, 23, 21], which mainly rely on the Lamport's *happens-before* causality relation [11] (we sketched a comparison between our causality notion and Lamport's in Section 6.1). In all these works causality is used to support determinism in replaying techniques and to define efficient dynamic slicing. In contrast, we use causality as a support for rollback primitives allowing the programmer to find the causes of a misbehavior.

As far as we know, the only work that addresses reversibility and causality together is about Causeway [18]. However, Causeway is not a full-fledged debugger, but just a post-mortem traces analyzer. It exploits a causality notion, based on the Lamport's *happens-before*, more liberal than ours.

Up to now, we mainly focused on the design of the right primitives to support our causal-consistent reversible debugging strategy. We are aware that causality can be exploited also for efficiency reasons. As future work, we plan to exploit the causality information we have also for improving the efficiency, possibly integrating our techniques with others from the literature.

The debugger CaReDeb we presented is just a prototype. The interaction with it is console-based, but we are in the process of upgrading it to an Eclipse plugin. Moreover, it targets the toy language $\mu$Oz. Our main goal is to develop

[3] The replay approaches that log the chosen interleaving and replay actions in the same order, avoiding non-determinism, fall in the record/reverse-execute class.

a causal-consistent reversible semantics and a causal-consistent debugger for a mainstream language, such as Java, Erlang or C++.

## References


1. K. Arya, T. Denniston, A. M. Visan, and G. Cooperman. Fred: Automated debugging via binary search through a process lifetime. *CoRR*, abs/1212.5204, 2012.
2. CaReDeb 0.4.0, a causal-consistent reversible debugger. `http://proton.inrialpes.fr/~mezzina/deb/`, 2013.
3. S.-K. Chen, W. K. Fuchs, and J.-Y. Chung. Reversible debugging using program instrumentation. *IEEE Trans. Software Eng.*, 27(8):715–727, 2001.
4. Chronon Systems. Commercial reversible debugger. `http://chrononsystems.com/`.
5. I. Cristescu, J. Krivine, and D. Varacca. A compositional semantics for the reversible p-calculus. In *LICS*, pages 388–397. IEEE Computer Society, 2013.
6. V. Danos and J. Krivine. Reversible communicating systems. In *Proc. of CONCUR'04*, volume 3170 of *LNCS*. Springer, 2004.
7. S. I. Feldman and C. B. Brown. Igor: A system for program debugging via reversible execution. In *Workshop on Parallel and Distributed Debugging*, pages 112–123, 1988.
8. R. Grishman. The debugging system aids. In *Proc. of the May 5-7, 1970, spring joint computer conference*, AFIPS '70 (Spring), pages 59–64. ACM, 1970.
9. S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 1–15, 2005.
10. T. Koju, S. Takada, and N. Doi. An efficient and generic reversible debugger using the virtual machine based approach. In *VEE*, pages 79–88, 2005.
11. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
12. I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversing higher-order pi. In *Proc. of CONCUR 2010*, volume 6269 of *LNCS*. Springer, 2010.
13. T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, Apr. 1987.
14. B. Lewis. Debugging backwards in time. *CoRR*, cs.SE/0310016, 2003.
15. M. Lienhardt, I. Lanese, C. A. Mezzina, and J.-B. Stefani. A reversible abstract machine and its space overhead. In *FMOODS/FORTE*, pages 1–17, 2012.
16. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339. ACM, 2008.
17. I. Phillips and I. Ulidowski. Reversing algebraic process calculi. *J. Log. Algebr. Program.*, 73(1-2), 2007.
18. T. Stanley, T. Close, and M. S. Miller. Causeway: a message-oriented distributed debugger. Technical report, HPL-2009-78, 2009. Available at `http://www.hpl.hp.com/techreports/2009/HPL-2009-78.html`.
19. Undo Software. Commercial reversible debugger. `http://undo-software.com/`.
20. A. M. Visan et al. Temporal debugging using urdb. *CoRR*, abs/0910.5046, 2009.
21. G. Xu et al. Efficient checkpointing of java software using context-sensitive capture and replay. In *ESEC-FSE '07*, pages 85–94. ACM, 2007.
22. M. V. Zelkowitz. Reversible execution. *Commun. ACM*, 16(9):566, 1973.

23. X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *SIGSOFT '06/FSE-14*, pages 81–91. ACM, 2006.