INSTITUTE
FOR ADVANCED
STUDIES
LUCCA

Causal-Consistent Reversible Debugging

3rd Cina Meeting, Torino

Claudio Antares Mezzina
(in collaboration with Elena Giachino and Ivan Lanese)

February 11, 2015

IMT INSTITUTE FOR ADVANCED STUDIES LUCCA

Contents

Reversible Debugging

Jakob Englom, S4D 2012

Reverse debugging is the ability of a debugger to stop **after** a failure in a program has been observed and to **go back** into the history of the execution to uncover the reason of failure.

Implications

- Ability to execute an application both in forward and backward way.
- Reproduce or keep track of the past of an execution.

Reversibility and Debugging

Question

When a misbehaviour is detected, how one should proceed in order to retrace the steps that led to the bug?

- **Sequential setting**: recursively undo the last action.
- **Concurrent setting**: there is not a clear understanding of which is the last action.

State of the Art for Concurrent Reversible Debugging

Non-deterministic replay

The execution is replayed non deterministically from the start (or from the a previous checkpoint) till the desired point.

Deterministic replay/ reverse-execute debugging

A log of the scheduling among threads is kept and then actions are reversed or replayed accordingly.

Drawbacks

Non-deterministic replay:

- Actions could get scheduled in a different order and hence the bug may not be reproduced.
- Particularly difficult to reproduce concurrency problems (e.g. race conditions).

Deterministic replay / reverse execution:

- Also action in threads not related to the bug may be undone.
- If one among several indipendent threads causes the bug, and this thread has been scheduled as first, then one has to undo the entire execution to find the bug.

Our Approach: Causal-Consistent Reversibility

Actions are reversed respecting the *causes*:

- only actions that have caused no successive actions can be undone;
- concurrent actions can be reversed in **any** order;
- dependent actions are reversed starting from the consequences.

Benefits

The programmer can easily individuate and undo the actions that caused a given misbehaviour.

Chosen language

- μOz: subset of the Oz language [Van Roy et al.]
- Functional language
  - thread-based concurrency
  - asynchronous communication via ports (channels)
- μOz advantages:
  - well-know stack-based abstract machine
  - equipped with a **causal-consistent reversible semantics** (from previous work)

Syntax

$S ::=$

$\mathbf{skip}$	empty stm
$S_1\ S_2$	sequence
$\mathbf{let}\ x = v\ \mathbf{in}\ S\ \mathbf{end}$	var declaration
$\mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}$	conditional
$\mathbf{thread}\ S\ \mathbf{end}$	thread creation
$\mathbf{let}\ x = c\ \mathbf{in}\ S\ \mathbf{end}$	procedure declaration
$\{x\ \tilde{y}\}$	procedure call
$\mathbf{let}\ x = \ \mathbf{NewPort}\ \ \mathbf{in}\ S\ \mathbf{end}$	port creation
$\{\mathbf{Send}\ x\ y\}$	send on a port
$\mathbf{let}\ x = \{\ \mathbf{Receive\ y}\ \}\ \mathbf{in}\ S\ \mathbf{end}$	receive from a port

$v ::= \quad \mathbf{true} \mid \mathbf{false} \mid \mathbb{N}$	simple values
$c ::= \quad \mathbf{proc}\ \{\tilde{x}\}\ S\ \mathbf{end}$	procedures

The Debugger Commands 1/2

control	**forth (f) t**	(forward execution of one step of thread t)
	run	(runs the program)
	rollvariable (rv) id	(c-c undo of the creation of variable id)
	rollsend (rs) id n	(c-c undo of last n send to port id)
	rollreceive (rr) id n	(c-c undo of last n receive from port id)
	rollthread (rt) t	(c-c undo of the creation of thread t)
	roll (r) t n	(c-c undo of n steps of thread t)
	back (b) t	(backward execution of one step of thread t (if possible))

The Debugger Commands 2/2

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)

Example of execution

```
let a = true  in                                                    (1)
   let b = false  in                                                (2)
      let chan = port in                                            (3)
         thread {send chan a}; skip; {send chan b} end;             (4)
         let y = {receive chan} in skip end                         (5)
      end                                                           (6)
   end                                                              (7)
end                                                                 (8)
```

- at line (4) thread t_1 is created from thread t_0
- t_1 fully executes, then t_0 fully executes
- what should be the shape of t_0 (and of the port) if t_1 rolls of 3 steps?

$$
\begin{array}{ll}
t_0 & \textbf{let } y = \{\textbf{receive } chan\} \textbf{ in skip end} \\
t_1 & \{\textbf{send } chan\ a\}; \textbf{skip}; \{\textbf{send } chan\ b\} \\
x & \bot
\end{array}
$$

- t_0 is automatically rolled-back enogh in order to release the read value a
- t_0 rolled-back as little as possible (no domino effect)

Debugging Soundness

Properties

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

Prop 1 ensures that the debugger can undo every forwar step, and, vice-versa, it can re-execute every step previously undone.

Prop 2 ensures that any sequence of debugging commands can only lead to states which are part of the normal forward-only computations.

Implementation

- Java based
- Interpreter of the μOz reversible semantics:
  - forward and backward steps
  - **roll** as controlled sequence of the backward steps
  - **rollvarialbe**, **rollthread**, **rollsend** and **rollreceive** are based on **roll**
- It keeps history and causality information to enable reversibiliby

`http://www.cs.unibo.it/caredeb`

History and Causality Information

- The history of each thread.
- The history of each channel, containing:
  - elements of the form (t_a, i, v, t_b, j)
  - t_a sent a value v which has been received by t_b
  - i and j are pointes to t_a and t_b send/receive instructions
- We also maintain the following mappings:
  - var_name $\rightarrow$ (thread_name, i) pointing to the variable creator (for **rollvar**)
  - thread_name $\rightarrow$ (thread_name, i) pointing to the thread creator (for **rollvar**)
  - could be retrieved by inspecting histories, but storing them is much more efficient

Reversing: code snippet

```
private static void rollTill(HashMap <String , Integer > map) {
  //map contains pairs <thread_name ,i>
  Iterator<String> it = map.keySet().iterator();
  while(it.hasNext())
  {
    String id = it.next();
    int gamma = map.get(id);
    //getGamma retrieves the next gamma in the history
    while(gamma <= getGamma(id)){
      try {
        stepBack(id);
      }
      catch (WrongElementChannel e) {
        rollTill(e.getDependencies());}
      catch (ChildMissingException e) {
        rollEnd(e.getChild());}
      }
    }
 }
```

Demo Time

Future Work

- Improve the debugges user experience
  - GUI
  - Eclipse plug-in
- Other form of causality analisys
- Move to more popular programing languages / models
  - e.g. Java with Actors
- Causal Consistent Replay

Thank you!

Any Questions?