Causal-consistent Reversible Debugging

Elena Giachino[1] Ivan Lanese[1] **Claudio Antares Mezzina**[2]

[1]Focus Team, University of Bologna/INRIA, Italy

[2]**SOA Unit, FBK Trento, Italy**

April 11, 2014

FASE 2014

Roadmap

Reversible Debugging

Definition [Jakob Engblom, S4D 2012]

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

State of the Art for Concurrent Reversible Debugging

Non-deterministic replay

The execution is replayed non deterministically from the start (or from 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 execute:

- Also actions in threads not related to the bug may be undone.
- If one among several independent threads causes the bug, and this thread has been scheduled 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.

Roadmap

Chosen language

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

Syntax

$S ::=$		
	skip	empty stm
	$S_1\ S_2$	sequence
	let $x = v$ **in** S **end**	var declaration
	if x **then** S_1 **else** S_2 **end**	conditional
	thread S **end**	thread creation
	let $x = c$ **in** S **end**	procedure declaration
	$\{x\ \tilde{y}\}$	procedure call
	let $x =$ **NewPort** **in** S **end**	port creation
	$\{$**Send** $x\ y\}$	send on a port
	let $x = \{$ **Receive** y $\}$ **in** S **end**	receive from a port
$v ::=$	**true** $\mid$ **false** $\mid 0 \mid 1 \mid \ldots$	simple values
$c ::=$	**proc** $\{x_1 \ldots x_n\}$ S **end**	procedures

The Debugger Commands

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	(bk 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)

Example of execution

$$\textbf{let } a = \textbf{true} \;\; \textbf{in} \tag{1}$$
$$\quad \textbf{let } b = \textbf{false} \;\; \textbf{in} \tag{2}$$
$$\qquad \textbf{let } x = \textbf{port in} \tag{3}$$
$$\qquad\quad \textbf{thread } \{\textbf{send x a}\}; \{\textbf{send x b}\} \textbf{ end}; \tag{4}$$
$$\qquad\quad \textbf{let } y = \{\textbf{receive } x\} \textbf{ in skip end} \tag{5}$$
$$\qquad \textbf{end} \tag{6}$$
$$\quad \textbf{end} \tag{7}$$
$$\textbf{end} \tag{8}$$

- At line (4), thread t_1 is created by thread t_0
- Assume t_1 fully executes, then t_0 completes its execution
- Both the threads are now terminated
- What should happen if **roll $\mathbf{t_1}$ 2** is issued in debugging mode?

State after rollback

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

- t_0 is automatically rolled-back enough 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 forward 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.

Roadmap

Implementation

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

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

History and Causal Information

- The history of each thread
- The history of each channel, containing:
  - elements of the form (t_0, i, a, t_1, j)
  - t_0 sent a value a which has been received by t_1
  - i and j are pointers to t_0 and t_1 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 **rollthread**)
  - 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

Roadmap

Future work

- Improve the debugger user experience:
  - GUI
  - Eclipse plug-in
- Other forms of causality analysis
- Move to more popular programming languages / models
  - e.g. Java with actors
- Causal-Consistent Replay

Thank you!

Questions?