Reversible Debugging

International Training School on Reversible Computation
Toruń, Poland

Claudio Antares Mezzina
IMT School for Advanced Studies Lucca, Italy

August 30, 2017

Roadmap

Debugging 1/2

Definition

Debugging is the process of finding and resolving misbehaviours of a software system.

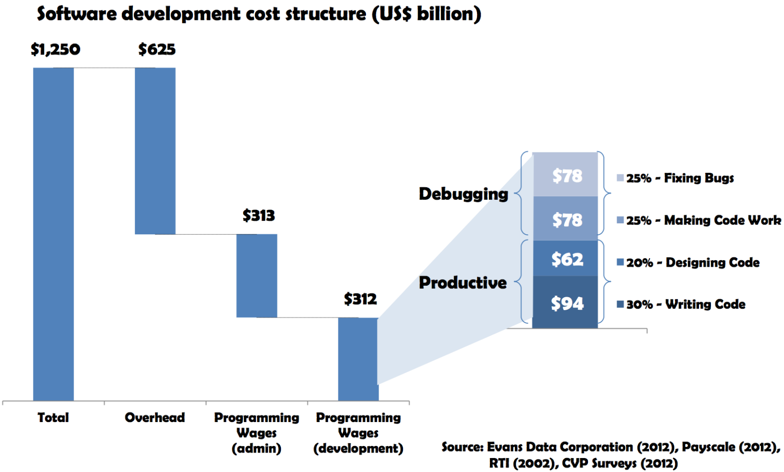
It is a well known technique used in software industry at many stages:

- prototipying
- maintenance after release (e.g., security patches)

Debuggers are integrated in modern IDEs and frameworks

- Eclipse
- .Net
- gdb part of the GNU project, along with gcc

Cost of Debugging



Study conducted by the Judge Business School of the Cambridge University [1]

Debugging 2/2

50% of programming time is spent in debugging.

- debugging is a costly and time consuming activity

Debugging 2/2

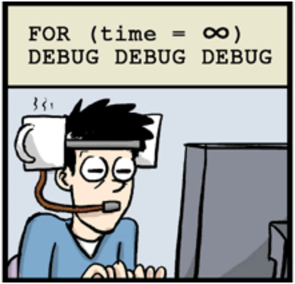
50% of programming time is spent in debugging.

- debugging is a costly and time consuming activity

Classical debugging steps

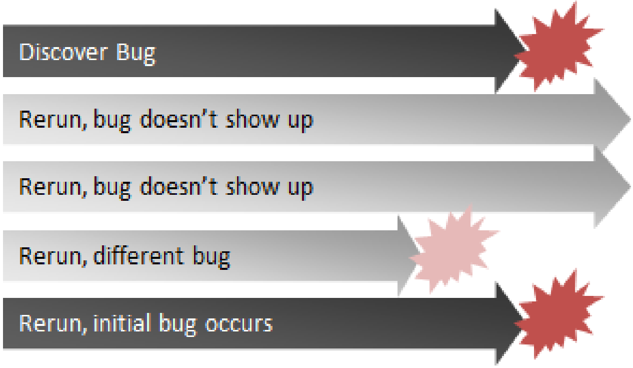
1. Trying to **reproduce** the problem
2. Trying to understand the **causes** (adding printings)
3. Trying to highlight the **exact** line of code which generates the error (breakpoints)

Debugging life[1]



[1] http://phdcomics.com/

Cyclic Debugging

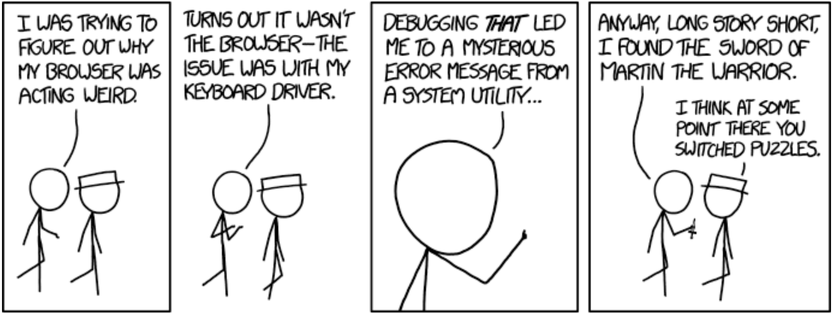


Cyclic Debugging

- Cumbersome and time consuming activity
- Difficult to understand the real reason of a bug
- Easy to *get lost and forget* the reason why you are debugging

Cyclic Debugging

- Cumbersome and time consuming activity
- Difficult to understand the real reason of a bug
- Easy to *get lost and forget* the reason why you are debugging



Record-Replay Debugging

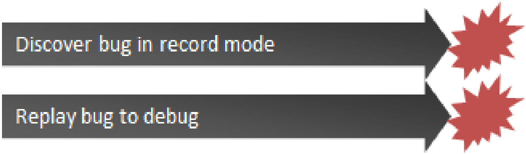
How to reproduce a bug?

One technique to reproduce the bug is to use record-replay

Record-Replay Debugging

How to reproduce a bug?

One technique to reproduce the bug is to use record-replay

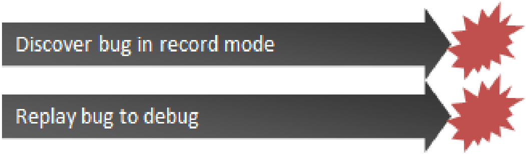


- The execution is recorded and replayed
- Execution is still forward
- Cannot step back the execution

Record-Replay Debugging

How to reproduce a bug?

One technique to reproduce the bug is to use record-replay



- The execution is recorded and replayed
- Execution is still forward
- Cannot step back the execution

Solution?

reversible debugging

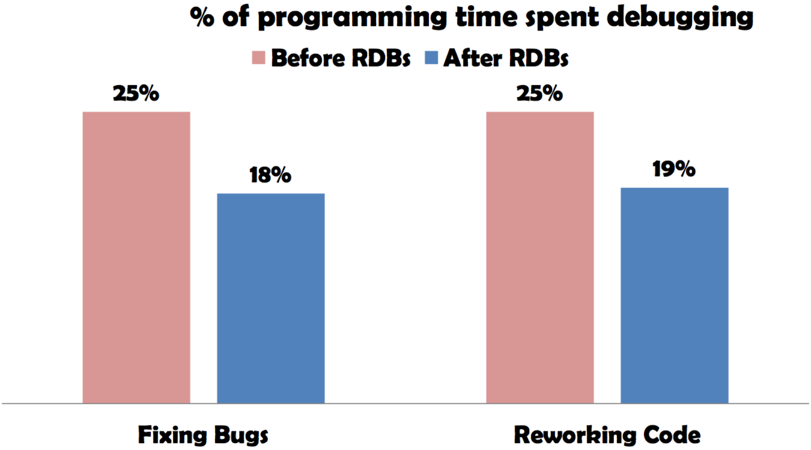
What is Reversible Debugging?

Definition [2]

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 a program both in forward and backward way.
- Reproduce or keep track of the past of an execution.



- stats taken from [1]
- 26% decrease in debugging time
- $81.1 billion software development costs saved per year

History

Reversible debugging has a long history

1973 Reversible Execution [3]

- first mention of Reversible Debugging
- forward-only record-replay for single-threaded progs

1987 debugging parallel programs with instant replay [4]

- assumes deterministic computation by enforcing the same execution order

1988 IGOR: a system for program debug via reversible execution [5]

- debugging single threaded programs
- record-replay

History

1996 Replay for Concurrent Non-Deterministic Shared-Memory Applications [6]

- record-replay of **multi-threaded** programs

2000 Efficient Algorithms for Bidirectional Debugging [7]

- single-threaded programs
- first to introduce backward breakpoints

2003 Using events to debug Java programs backwards in time [8]

- first (and only one) reversible degugger for Java
- first working debugger for a VM
- same techniques applied later for MS .Net IntelliTraces

2007 Reversible Debugging (added to gdb in 2009) [9]

State of the art: sequential debugging

- Reversible debuggers exist: GDB, UndoDB
- Many reversible debuggers deal only with sequential programs
- Some of them allow one to debug concurrent programs
  - They register scheduler events
  - The same scheduling is used when the program is replayed
  - Program events are linearized
  - Linearized execution can be wind and unwind like a tape

Sequential Reversible Debugging

- Take an execution containing a failure and move backward and forward along it looking for the bug
- The exact same execution can be explored many times forward and backward
  - Even bugs related to concurrency can always be replayed

GDB

- is the GNU debugger
- since version 7 (2009) includes reversible debugging
- uses record and replay technique
  1. one has to enable the recording phase (and decide the tape length)
  2. executes forward
  3. can explored the recorded session back and forth
  4. when going back and forth action are not re-executed

GDB Commands

Like the forward commands (step, next, continue), but in the backward direction

- reverse-step: goes back to the last instruction
- reverse-next: goes back to the last instruction, does not go inside functions
- reverse-continue: runs back till a breakpoint/watchpoint triggers
- breakpoints and watchpoints can be used also in the backward direction

gcc file.c **-ggdb**

Why RDs are not widely used[2]?

Why is reverse debugging rarely used? [closed]

49

gdb implemented support for reverse debugging in 2009 (with gdb 7.0). I never heard about it until 2012. Now I find it extremely useful for certain types of debugging problems. I wished that I heard of it before.

Correct me if I'm wrong but my impression is that the technique is still rarely used and most people don't know that it exists. Why?

For one, running in debug mode with recording on is **very** expensive compared to even normal debug mode; it also consumes a lot more memory.

It is easier to decrease the granularity from line level to function call level. For example, the standard debugger in eclipse allows you to "drop to frame," which is essentially a jump back to the start of the function with a reset of all the parameters (nothing done on the heap is reverted, and `finally` blocks are not executed, so it is not a true reverse debugger; be careful about that).

[2] Taken from stackowerflow

Why RDs are not widely used: the case of GDB

- Recording phase in GDB is not optimized and adds a huge overehead in terms of both space and time
- Overhead in times may hide under the carpet synchronisation/race condition bugs

UndoDB a commercial debugger

- From UndoSoftware, Cambridge, UK
  `http://undo-software.com/`
- A main company in the field of reversible debugging
- Improves GDB recording by using efficient incremental check-poiting techniques
- Available for Linux (intel/amd processors) and Android (ARM processors)
- Allows reversible debugging for programs in C/C++
- Allows to write on a file a recording session
  - useful for replaying the session on different machines
  - a client can send the recorded version to the developer

UndoDB success

Comparison with GDB, on recording gzipping a 16MB file

	Native	UndoDB	GDB
Time	1.49 s	2.16 s (1.75 x)	21 h (50000 x)
Space	-	17.8 MB	63 GB

Roadmap

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.
- Techniques to undo a concurrent execution in literature:
  - non-deterministic replay
  - deterministic replay/reverse

Non-deterministic replay

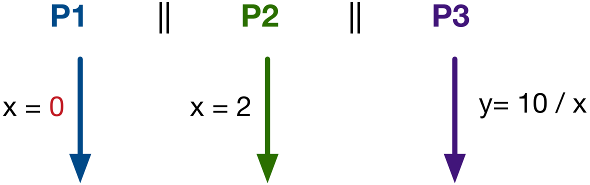
Non-deterministic replay

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

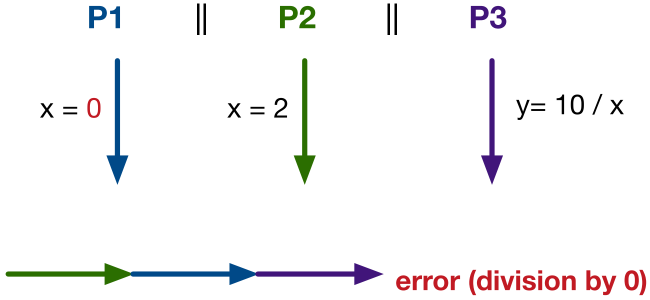
Non-deterministic replay:

- 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).

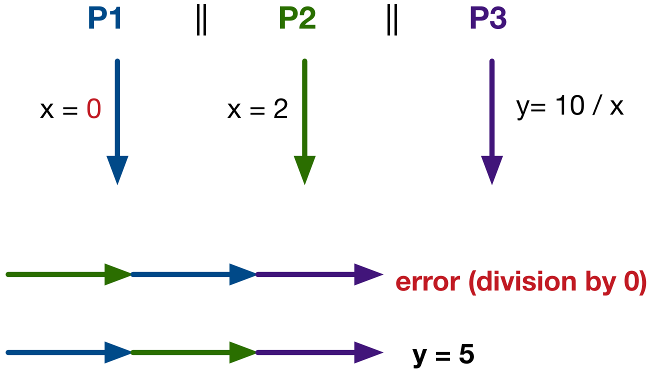
Non-deterministic replay: example



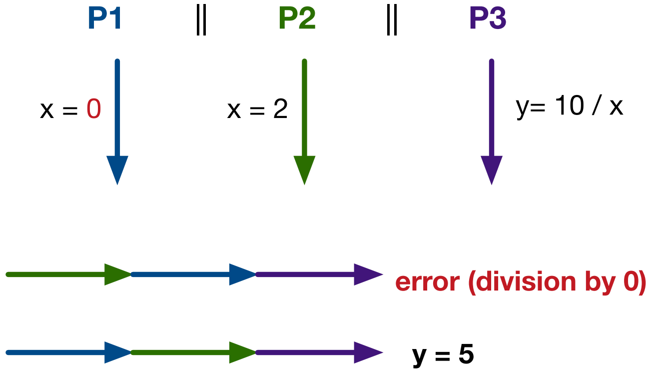
Non-deterministic replay: example



Non-deterministic replay: example



Non-deterministic replay: example



We need to store information about the threads execution

Replay/Reverse

replay/reverse execution

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

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.

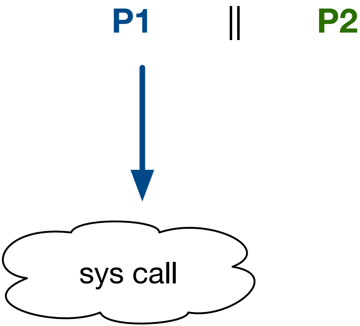
Deterministic Replay: example

P1 $\parallel$ **P2**

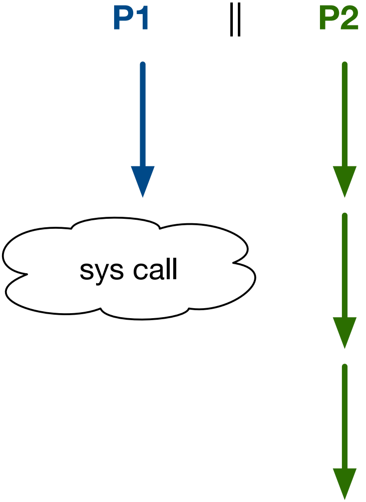
Replay

P1 $\|$ **P2**

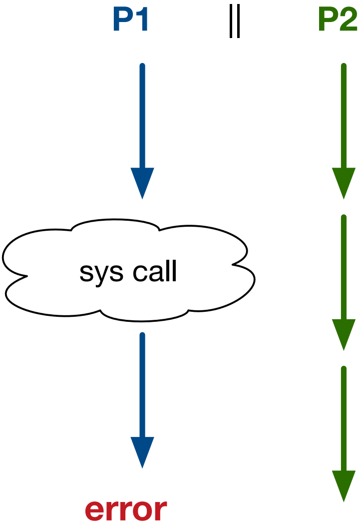
Deterministic Replay: example



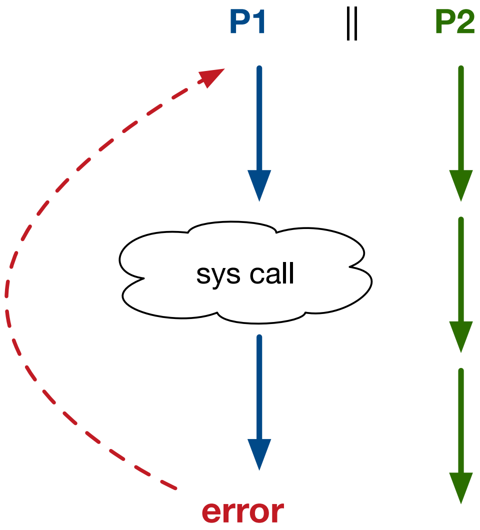
Deterministic Replay: example



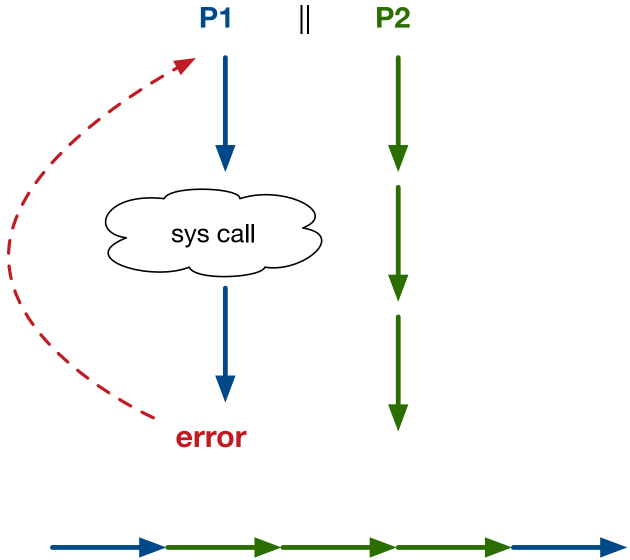
Deterministic Replay: example



Deterministic Replay: example



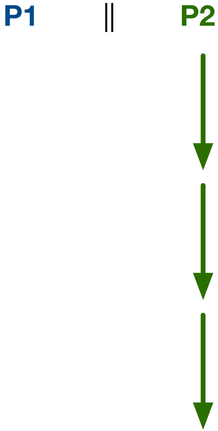
Deterministic Replay: example



Deterministic Replay: example

P1 $\parallel$ **P2**

Wanted execution



We need to **less** information about the threads execution

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.

A simple debugger for a simple language

- a subset of the Oz language [10]
- Functional Language
  - thread-based concurrency
  - asynchronous communication via channels (ports)
- well-know stack-based abstract machine

Syntax

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
	$\mid$	$\{\ x\ x_1 \ldots x_n\ \}$	Procedure call
	$\mid$	**let** $x = \texttt{NewPort}$ **in** S **end**	Port creation
	$\mid$	$\{\ \texttt{Send}\ x\ y\ \}$	Send on a port
	$\mid$	**let** $x = \{\ \texttt{Receive}\ y\ \}$ **in** S **end**	Receive from a port
v	$::=$	**true** $\mid$ **false** $\mid$ 0 $\mid$ $1 \ldots$	Simple values
c	$::=$	**proc** $\{\ x_1 \ldots x_n\ \}$ S **end**	Procedure

Semantics

- The semantics is defined as a reduction relation, noted $\rightarrow$, between configurations of the form (U, σ).
- To follow Oz notation, the relation $\rightarrow$ is defined by a set of rules of the form below, specifying that (U, σ) reduces to (U', σ') if condition G is satisfied

$$\frac{U \;\Big\|\; U'}{\sigma \;\Big\|\; \sigma'} \text{ if } G$$

- programs written as stacks of instructions
- variables are **always** created fresh and never modified
- sent values are variables names, not their contents

Semantics 1/2

skp $\dfrac{\langle \textbf{skip}\ T\rangle}{0} \;\Big\|\; \dfrac{T}{0}$

var $\dfrac{\langle \textbf{let}\ x = v\ \textbf{in}\ S\ \textbf{end}\ T\rangle}{0} \;\Big\|\; \dfrac{\langle S\{^{x'}/_{x}\}\ T\rangle}{x' = v}$ if x' fresh

npr $\dfrac{\langle \textbf{let}\ x = c\ \textbf{in}\ S\ \textbf{end}\ T\rangle}{0} \;\Big\|\; \dfrac{\langle S\{^{x'}/_{x}\}\ T\rangle}{x' = \xi \parallel \xi : c}$ if x', ξ fresh

npt $\dfrac{\langle \textbf{let}\ x = \texttt{NewPort}\ \textbf{in}\ S\ \textbf{end}\ T\rangle}{0} \;\Big\|\; \dfrac{\langle S\{^{x'}/_{x}\}\ T\rangle}{x' = \xi \parallel \xi : \perp}$ if x', ξ fresh

if1 $\dfrac{\langle \textbf{if}\ x\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}\ T\rangle}{x = \textbf{true}} \;\Big\|\; \dfrac{\langle S_1\ T\rangle}{x = \textbf{true}}$

Semantics 2/2

nth $$\frac{\langle \textbf{thread } S \textbf{ end } T\rangle}{0} \;\Bigg\|\; \frac{T \parallel \langle S\ \langle\rangle\rangle}{0}$$

pc $$\frac{\langle\{\ x\ x_1 \ldots x_n\ \}\ T\rangle}{\begin{array}{c} x = \xi \parallel \\ \xi : \textbf{proc } \{\ y_1 \ldots y_n\ \}\ S \textbf{ end}\end{array}} \;\Bigg\|\; \frac{\langle S\{^{x_1}/_{y_1}\}\ldots\{^{x_n}/_{y_n}\}\ T\rangle}{\begin{array}{c} x = \xi \parallel \\ \xi : \textbf{proc } \{\ y_1 \ldots y_n\ \}\ S \textbf{ end}\end{array}}$$

snd $$\frac{\langle\{\ \texttt{Send}\ x\ y\ \}\ T\rangle}{x = \xi \parallel \xi : Q} \;\Bigg\|\; \frac{T}{x = \xi \parallel \xi : y; Q}$$

rcv $$\frac{\langle \textbf{let } x = \{\ \texttt{Receive}\ y\ \} \textbf{ in } S \textbf{ end } T\rangle}{\begin{array}{c} y = \xi \parallel \\ \xi : Q; z \parallel z = w\end{array}} \;\Bigg\|\; \frac{\langle S\{^{x'}/_{x}\}\ T\rangle}{\begin{array}{c} y = \xi \parallel \\ \xi : Q \parallel z = w \parallel \\ x' = w\end{array}} \quad \text{if } x' \text{ fresh}$$

Reversing the language

- unique thread identifiers
- threads endowed with a history
- syntactic delimiters to statements, to delimit their scope
- queues with histories

Syntax Modifications

M, N	$::=$		Task
		0	No task
	$\mid$	$t[H]C$	Thread
	$\mid$	$M \parallel N$	Parallel composition
H	$::=$		History
		$\perp$	Empty history
	$\mid$	**skip**	Executed a skip
	$\mid$	$H\ H'$	Sequential composition
	$\mid$	$\uparrow x$	Sent on port x
	$\mid$	$\downarrow x(y)$	Received y from port x
	$\mid$	$*x$	Created variable x
	$\mid$	$\{\ x\ x_1 \ldots x_n\ \}$	Called procedure x
	$\mid$	$*t$	Created thread t
	$\mid$	**if**$(x)S$	Executed an if statement
	$\mid$	**esc**	Scope statement

$\theta \ ::=$	Store
$\quad 0$	Empty store
$\quad \mid \ x = w$	Binding
$\quad \mid \ \xi : c$	Closure
$\quad \mid \ \xi : K \vert K_h$	Port
$\quad \mid \ \theta \parallel \theta'$	Parallel composition
$K \ ::=$	Message queue
$\quad \perp$	Empty queue
$\quad \mid \ t\,{:}\,x \quad \mid \quad K; K'$	Messages
$K_h \ ::=$	Queue history
$\quad \perp$	Empty queue history
$\quad \mid \ t\,{:}\,x, t'; K_h$	Message in queue history

Making let reversible

$$\frac{\langle \textbf{let } x = v \textbf{ in } S \textbf{ end } T\rangle}{0} \;\Bigg\|\; \frac{\langle S\{^{x}/_{x'}\}\ \langle \qquad T\rangle\rangle}{x' = v} \text{ if } x' \text{ fresh}$$

Making let reversible

$$\frac{t[H]\langle \textbf{let } x = v \textbf{ in } S \textbf{ end } T\rangle}{0} \;\Bigg\|\; \frac{\langle S\{^{x}/_{x'}\}\ \langle \quad T\rangle\rangle}{x' = v} \text{ if } x' \text{ fresh}$$

- unique thread id and past history

Making let reversible

$$\frac{\color{red}{t[H]}\langle \textbf{let } x = v \textbf{ in } S \textbf{ end } T\rangle}{0} \;\Big\|\; \frac{\color{red}{t[H \quad * x']}\langle S\{^{x}/_{x'}\} \;\langle \quad T\rangle\rangle}{x' = v} \text{ if } x' \text{ fresh}$$

- unique thread id and past history
- history include the new action

Making let reversible

$$\frac{t[H]\langle \textbf{let } x = v \textbf{ in } S \textbf{ end } T\rangle}{0} \;\Big\|\; \frac{t[H \;* x']\langle S\{^x/_{x'}\}\ \langle \textbf{esc } T\rangle\rangle}{x' = v} \text{ if } x' \text{ fresh}$$

- unique thread id and past history
- history include the new action
- scope delimiter
  - identify the scope of the statement to be reversed
  - usefull to reverse procedure calls

Forward Semantics

skp $\dfrac{t[H]\langle \textbf{let } x = v \textbf{ in } S \textbf{ end } C\rangle \;\Big\|\; t[H \;\; * x']\langle S\{^{x'}/_{x}\} \langle \textbf{esc } C\rangle\rangle}{0 \;\Big\|\; x' = v}$ if x' fresh

var $\dfrac{t[H]\langle \textbf{let } x = c \textbf{ in end } S\ C\rangle \;\Big\|\; t[H \;\; * x']\langle S\{^{x'}/_{x}\} \langle \textbf{esc } C\rangle\rangle}{0 \;\Big\|\; x' = \xi \parallel \xi : c}$ if x', ξ fresh

npr $\dfrac{t[H]\langle \textbf{let } x = \texttt{NewPort} \textbf{ in end } S\ C\rangle \;\Big\|\; t[H \;\; * x']\langle S\{^{x'}/_{x}\} \langle \textbf{esc } C\rangle\rangle}{0 \;\Big\|\; x' = \xi \parallel \xi : \bot | \bot}$ if x', ξ fresh

npc $\dfrac{t[H]\langle \textbf{if } x \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end } C\rangle \;\Big\|\; t[H \; \textbf{if}(x)S_2]\langle S_1 \langle \textbf{esc } C\rangle\rangle}{x = \textbf{true} \;\Big\|\; x = \textbf{true}}$

Forward Semantics

nth $\dfrac{t[H]\langle \textbf{thread } S \textbf{ end } C\rangle}{0} \;\Big\|\; \dfrac{t[H \ \ast t']C \parallel t'[\bot]\langle S\ \langle\rangle\rangle}{0}$ if t' fresh

pc $\dfrac{t[H]\langle\{\ x\ (x_i)_1^n\ \}\ C\rangle}{x = \xi \parallel \\ \xi : \textbf{proc } \{\ (y_i)_1^n\ \}\ S \textbf{ end}} \;\Big\|\; \dfrac{t[H\ \{\ x\ (x_i)_1^n\ \}]\langle S(\{^{x_i}/_{y_i}\})_1^n\ \langle \textbf{esc } C\rangle\rangle}{x = \xi \parallel \\ \xi : \textbf{proc } \{\ (y_i)_1^n\ \}\ S \textbf{ end}}$

snd $\dfrac{t[H]\langle\{\ \texttt{Send}\ x\ y\ \}\ C\rangle}{x = \xi \parallel \xi : K|K_h} \;\Big\|\; \dfrac{t[H\ \ \uparrow x]C}{x = \xi \parallel \xi : t\!:\!y; K|K_h}$

rcv $\dfrac{t[H]\langle \textbf{let } y = \{\ \texttt{Receive}\ x\ \}\ \textbf{in } S \textbf{ end } C\rangle}{\theta \parallel \xi : K; t'\!:\!z|K_h} \;\Big\|\; \dfrac{t[H\ \ \downarrow x(y')]\langle S\{^{y'}/_y\} \\ \langle \textbf{esc } C\rangle\rangle}{\theta \parallel \xi : K|t'\!:\!z, t; K_h \parallel \\ y' = w}$

if y' fresh $\wedge\ \theta \triangleq x = \xi \parallel z = w$

$$\dfrac{t[H]\langle \textbf{esc } C\rangle}{0} \;\Big\|\; \dfrac{t[H\ \textbf{esc}]C}{0}$$

Backward Semantics (excerpt)

snd
$$\frac{t[H]\langle \{\ \mathtt{Send}\ x\ y\ \}\ C\rangle}{x = \xi \parallel \xi : K|K_h} \;\Big\|\; \frac{t[H\ \ \uparrow x]C}{x = \xi \parallel \xi : t\!:\!y; K|K_h}$$

snd^{-1}
$$\frac{t[H\ \ \uparrow x]C}{x = \xi \parallel \xi : t\!:\!y; K|K_h} \;\Big\|\; \frac{t[H]\langle \{\ \mathtt{Send}\ x\ y\ \}\ C\rangle}{x = \xi \parallel \xi : K|K_h}$$

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

rcv^{-1}
$$\frac{t[H\ \ \downarrow x(z)]\langle S\ \langle \textbf{esc}\ C\rangle\rangle}{z = w \parallel x = \xi \parallel \xi : K|t'\!:\!y, t; K_h} \;\Big\|\; \frac{t[H]\langle \textbf{let}\ z = \{\ \mathtt{Receive}\ x\ \}\ \textbf{in}\ S\ \textbf{end}\ C\rangle}{x = \xi \parallel \xi : K; t'\!:\!y|K_h}$$

more details in: *A reversible abstract machine and its space overhead* [11]

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 `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 t, c, or v)
	history (h) id	(shows thread/channel computational history)

Example of execution

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

- 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?

Desired execution

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

- t_0 is rolled-back enough in order to free the read value
- No domino effect, causing t_0 to fully roll-back

Building up a debugger [12]

- Java based Interpreter of the Oz 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

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

Implementations Details

Computation information:

- The history of each thread
- The history of each channel, containing:
  - elements of the form $(\mathrm{t0}$, i, a, $\mathrm{t1}$, j$)$
  - t0 sent a value a which has been received by t1
  - i and j are pointers to $\mathrm{t0}$ and $\mathrm{t1}$ send/receive instructions

Also the debugger maintains the following mappings:

- $var\_name \rightarrow (\mathrm{thread\_name}, i)$ pointing to the variable creator (for rollvar)
- $\mathrm{thread\_name} \rightarrow (\mathrm{thread\_name}, i)$ 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

Conclusions

Conclusions

We are not re-inventing the wheel, just improving it!

- causality information helps to find out the root of a bug
- causality information helps in saving the **right** order of events

Conclusions

We are not re-inventing the wheel, just improving it!

- causality information helps to find out the root of a bug
- causality information helps in saving the **right** order of events

Cakewalk? (Problems)

- Increasing the expressiveness of the language will modify the notion of causality
- system calls, interrupts, shared memory, file descriptors will make the entire setting more complex
  - some of them cannot be reverted
  - should be recorder and replayed as they are

References I

Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak.
Reversible debugging software "quantify the time and cost saved using reversible debuggers".

J. Engblom.
A review of reverse debugging.
In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–6, 2012.

M. V. Zelkowitz.
Reversible execution.
Commun. ACM, 16(9):566–, September 1973.

Thomas J. LeBlanc and John M. Mellor-Crummey.
Debugging parallel programs with instant replay.
IEEE Trans. Computers, 36(4):471–482, 1987.

References II

Stuart I. Feldman and Channing B. Brown.
Igor: A system for program debugging via reversible execution.
In *WPDD 98*, pages 112–123. ACM, 1988.

Mark Russinovich and Bryce Cogswell.
Replay for concurrent non-deterministic shared-memory applications.
SIGPLAN Not., 31(5):258–266, May 1996.

Bob Boothe.
A fully capable bidirectional debugger.
ACM SIGSOFT Software Engineering Notes, 25(1):36–37, 2000.

Bil Lewis and Mireille Ducassé.
Using events to debug java programs backwards in time.
In *OOPSLA 2003,*, pages 96–97. ACM, 2003.

References III

P. Brook and D. Jacobowitz.
Reversible debugging.
In *GCC Developers' Summit*, 2007.

Peter Van Roy and Seif Haridi.
Concepts, Techniques, and Models of Computer Programming.
MIT Press, 2004.

Michael Lienhardt, Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani.
A reversible abstract machine and its space overhead.
In *FMOODS / FORTE*, volume 7273 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2012.

References IV

Elena Giachino, Ivan Lanese, and Claudio Antares Mezzina.
Causal-consistent reversible debugging.
In *FASE 2014*, volume 8411 of *LNCS*, pages 370–384.
Springer, 2014.