Reversible Debugging

Ivan Lanese
Focus research group
Computer Science and Engineering Department
University of Bologna/INRIA
Bologna, Italy

Joint work with Elena Giachino
(Univ. Bologna/INRIA, Italy) and Claudio
Antares Mezzina (IMT Lucca, Italy)
Roadmap

- Reversible debugging
- Reversible debugging in practice
- An academic approach: causal-consistent reversible debugging
- Conclusion
Roadmap

- Reversible debugging
- Reversible debugging in practice
- An academic approach: causal-consistent reversible debugging
- Conclusion
Why debugging?

- Developers spend 50% of their programming time finding and fixing bugs.
- The global cost of debugging has been estimated in $312 billions annually.
- The cost of debugging is bound to increase with the increasing complexity of software:
  - Size
  - Concurrency, distribution
- Surprisingly, a very little amount of research concentrates on debugging.
Standard debugging strategy

- When a failure occurs, one has to re-execute the program with a breakpoint before the expected bug
- Then one executes step-by-step forward from the breakpoint, till the bug is found
- Limitations:
  - High cost of replaying
    » Time, use of the actual execution environment
  - Difficult to precisely replay the execution
    » Concurrency or non-determinism
  - Difficult to find the exact point where to put the breakpoint
    » If breakpoint too late, the execution needs to be redone
    » Frequently, many attempts are needed
Reversibility allows one to execute programs not only in the standard, forward, direction, but also backward, going to past states.

Reversibility in a sequential setting: recursively undo the last action.

Many actions lose information: one needs to keep information on the past states.

- The assignment $x=0$ deletes the old value of $x$, which should be saved.

Reversibility has applications in hardware design, biological modeling, simulation, quantum computing...
Reversibility for debugging

- Reversible debuggers extend standard debuggers with the ability to execute the program under analysis also backward
- Avoids the common “Damn, I put the breakpoint too late” error
  - Just execute backward from where the program stopped till the desired point is reached
  - You may even not need a breakpoint: if the program crashed, you can execute backward from where it crashed
- The overhead due to storing history information is a main limitation for reversible debuggers
Reversible debugging strategy

- 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.
  - Non-determinism is no more a problem.
- Some form of causal exploration is also possible.
  - If variable $i$ has not the desired value, run backward with a watchpoint telling where the value of $i$ changes.
Roadmap

- Reversible debugging
- Reversible debugging in practice
- An academic approach: causal-consistent reversible debugging
- Conclusion
A reversible debugger: GDB

- GDB supports reversible debugging since version 7.0 (2009)
- Uses record and replay
  - One activates the recording modality
  - Executes the program forward
  - Can explore the recorded execution backward and forward
  - When exploring, instructions are not re-executed
GDB reverse 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 the last stop event
- ...
- Breakpoints and watchpoints can be used also in backward direction
A commercial reversible debugger: UndoDB

- From UndoSoftware, Cambridge, UK
  http://undo-software.com/
- Built as an extension of GDB
- Available for Linux and Android
- Allows reversible debugging for programs in C/C++
UndoDB commands

- Recording enabled by default, but can be deferred
- Reversible commands from GDB (reimplemented)
- Commands for exploring the recorded execution, more high-level
  - Define and go to bookmark
  - Go back n “simulated nanoseconds”
- Various commands for configuration
- Commands to write a recorded execution to file, and reload it
  - Useful to record on client premises and explore at company premises
UndoDB winning feature

**Performance**

- Comparison with GDB, on recording gzipping a 16MB file

<table>
<thead>
<tr>
<th></th>
<th>Native</th>
<th>UndoDB</th>
<th>GDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>1.49 s</td>
<td>2.16 s (1.75 x)</td>
<td>21 h (50000 x)</td>
</tr>
<tr>
<td>Space</td>
<td>-</td>
<td>17.8 MB</td>
<td>63 GB</td>
</tr>
</tbody>
</table>
UndoDB: how it works

- Takes periodically snapshots of memory
  - Only changed pages are stored (copy-on-write)
  - The frequency of snapshots changes dynamically
- Nondeterministic events are stored during recording and simulated during replay
  - Including thread switches, shared memory accesses and system calls
  - The same thread scheduling is considered during replay
Roadmap

● Reversible debugging
● Reversible debugging in practice
● An academic approach: causal-consistent reversible debugging
● Conclusion
Causal-consistent reversible debugging

- We are interested in debugging concurrent/distributed programs
- The definition “recursively undo the last action” is no more valid: there are many possible last actions
- Since [Danos&Krivine, CONCUR 2004] the notion of reversibility for concurrent systems is causal-consistent reversibility
  - Any action can be undone, provided that its consequences (if any) have been undone
  - Concurrent actions can be undone in any order, but causal-dependent actions are undone in reverse order
Debugging and causality

- Causal-consistency relates backward computations with causality
- Debugging amounts to find the bug that caused a given misbehavior
- We propose the following debugging strategy: follow causality links backward from misbehavior to bug
- Which primitives do we need to enable such a strategy?
  - The details depend on the chosen language...
  - ...but the general idea carries over to different languages
Our target language: μOz

- A kernel language of Oz
- Oz is at the base of the Mozart language
- Thread-based concurrency
- Asynchronous communication via ports
- Shared memory
  - Variable names are sent, not their content
- Variables are always created fresh and never modified
- Higher-order language
  - Procedures can be communicated
The **roll** primitive

- The main primitive we propose is **roll t n**
- Undoes the last **n** actions of thread **t**...
- ... in a causal-consistent way
  - Before undoing an action one has to undo all (and only) the actions depending on it
- A single **roll** may cause undoing actions in many threads
Different interfaces for roll

- One interface for each possible misbehavior
- Wrong value in a variable: `rollvariable id` goes to the state just before the creation of variable `id`
- Wrong value in a queue element: `rollsend id n` undoes the last `n` sends to port `id`
- Thread blocked on a receive: `rollreceive id n` undoes the last `n` reads on the port `id`
- Unexpected thread: `rollthread t` undoes the creation of thread `t`
Using roll-like primitives

- The programmer can follow causality links backward
- No need for the programmer to know which thread or instruction originated the misbehavior
  - The primitives find them automatically
- The procedure can be iterated till the bug is found
- Only relevant actions are undone
  - Thanks to causal consistency
- Looking at which threads are involved gives useful information
  - If an unexpected thread is involved, an interference between the two threads has happened
Additional commands

- The roll-like commands are added on top of a standard debugger
  - Commands for forward execution, breakpoints, state exploration
- A few other commands are related to reversibility
  - Step backward
  - History exploration
CaReDeb: a causal-consistent debugger

- Only a prototype to test our ideas
- Debugs $\mu$Oz programs
- Written in Java
- Available at http://www.cs.unibo.it/carededeb
- Description and underlying theory in [Giachino, Lanese & Mezzina, FASE 2014]
Roadmap

- Reversible debugging
- Reversible debugging in practice
- An academic approach: causal-consistent reversible debugging
- Conclusion
## Research vs Industry

<table>
<thead>
<tr>
<th>Research</th>
<th>Industry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Innovative, complex techniques</td>
<td>Simpler, more standard techniques</td>
</tr>
<tr>
<td>Toy/limited languages</td>
<td>Real languages</td>
</tr>
<tr>
<td>No integration in the software development environment</td>
<td>Emphasis on integration in the software development environment</td>
</tr>
<tr>
<td>Performance not a key</td>
<td>Performance is the key</td>
</tr>
<tr>
<td>No interest in support and maintenance</td>
<td>Many companies live on support and maintenance</td>
</tr>
</tbody>
</table>
Future work

- Extending the set of language constructs we are able to tackle, towards real languages
- Find simplified mechanisms easier to implement efficiently
- Interact with UndoSoftware to bridge the gap
  - We want them to use our techniques, not our debugger
Some advertising

- We have COST Action IC1405 on Reversible computation – extending horizons of computing
- Not only debugging: reversible languages, reversible automata, reversible circuits, …
- Just preliminary results on verification for reversible systems
- Feel free to contact Irek Ulidowski or me if you want to join
Finally

Thanks!

Questions?
µOz syntax

- \( S ::= \)
  - skip \[Empty statement]\n  - \( S_1 \, S_2 \) \[Sequence]\n  - let \( x = v \) in \( S \) end \[Variable declaration]\n  - if \( x \) then \( S_1 \) else \( S_2 \) end \[Conditional]\n  - thread \( S \) end \[Thread creation]\n  - let \( x = c \) in \( S \) end \[Procedure declaration]\n  - \( \{x \, x_1 \ldots x_n\} \) \[Procedure call]\n  - let \( x = \text{Newport} \) in \( S \) end \[Port creation]\n  - \( \{\text{Send} \, x \, y\} \) \[Send]\n  - let \( x = \{\text{Receive} \, y\} \) in \( S \) end \[Receive]\n
- \( c ::= \text{proc} \{x_1 \ldots x_n\} \, S \) end
μOz semantics

- Semantics defined by a stack-based abstract machine
- The abstract machine exploits a run-time syntax
- Each thread is a stack of instructions
  - The starting program is inserted into a stack
  - Thread creation creates new stacks
- Procedures are stored as closures
- Ports are queues of variables
- Semantics closed under
  - Contexts (for both code and state)
  - Structural congruence
μOz semantics: rules

R:skp
\[ \frac{\langle \text{skip } T \rangle}{0} \parallel \frac{T}{0} \]

R:var
\[ \frac{\langle \text{let } x = v \text{ in } S \text{ end } T \rangle}{0} \parallel \frac{\langle S\{x'/x\} \ T \rangle}{x' = v} \quad \text{if } x' \text{ fresh} \]

R:npr
\[ \frac{\langle \text{let } x = c \text{ in } S \text{ end } T \rangle}{0} \parallel \frac{\langle S\{x'/x\} \ T \rangle}{x' = \xi \parallel \xi : c} \quad \text{if } x', \xi \text{ fresh} \]

R:npt
\[ \frac{\langle \text{let } x = \text{NewPort} \text{ in } S \text{ end } T \rangle}{0} \parallel \frac{\langle S\{x'/x\} \ T \rangle}{x' = \xi \parallel \xi : \bot} \quad \text{if } x', \xi \text{ fresh} \]

R:if1
\[ \frac{\langle \text{if } x \text{ then } S_1 \text{ else } S_2 \text{ end } T \rangle}{x = \text{true}} \parallel \frac{\langle S_1 \ T \rangle}{x = \text{true}} \]

R:nth
\[ \frac{\langle \text{thread } S \text{ end } T \rangle}{0} \parallel \frac{T \parallel \langle S \ \langle \rangle \rangle}{0} \]

R:pc
\[ \frac{\{ x \ x_1 \ldots x_n \} \ T}{x = \xi \parallel \xi : \text{proc} \{ y_1 \ldots y_n \} \ S \text{ end}} \parallel \frac{\langle S\{x_1/y_1\} \ldots \{x_n/y_n\} \ T \rangle}{x = \xi \parallel \xi : \text{proc} \{ y_1 \ldots y_n \} \ S \text{ end}} \]

R:snd
\[ \frac{\langle \{ \text{Send } x \ y \} \ T \rangle}{x = \xi \parallel \xi : Q} \parallel \frac{T}{x = \xi \parallel \xi : y; Q} \]

R:rcv
\[ \frac{\langle \text{let } x = \{ \text{Receive } y \} \text{ in } S \text{ end } T \rangle}{y = \xi \parallel \xi : Q; z \parallel z = w} \parallel \frac{\langle S\{x'/x\} \ T \rangle}{y = \xi \parallel \xi : Q \parallel z = w \parallel x' = w} \quad \text{if } x' \text{ fresh} \]
μOz reversible semantics

- We give unique names to threads
- We add histories to threads to remember past actions
- We add a delimiter to record when scopes end
  - For let
  - For procedure body
  - For if-then-else
- Ports have histories too
  - Should record also sender and receiver of each message
  - We do not want to change the order of communications
\[ \muOz \text{ reversible semantics: forward rules} \]

\[
\begin{array}{c}
\text{R:fw:skp} \\
\frac{t[H] \langle \text{skip } C \rangle}{0} \quad | \quad t[H \ \text{skip}]C
\end{array}
\]

\[
\begin{array}{c}
\text{R:fw:var} \\
\frac{t[H] \langle \text{let } x = v \ \text{in } S \ \text{end } C \rangle}{0} \quad | \quad t[H \ * x']\langle S\{x'/x\} \ \langle \text{esc } C \rangle\rangle \\
\text{if } x' \text{ fresh}
\end{array}
\]

\[
\begin{array}{c}
\text{R:fw:npr} \\
\frac{t[H] \langle \text{let } x = c \ \text{in } S \ \text{end } C \rangle}{0} \quad | \quad t[H \ * x']\langle S\{x'/x\} \ \langle \text{esc } C \rangle\rangle \\
x' = \xi \ \parallel \ \xi : c \\
\text{if } x', \xi \text{ fresh}
\end{array}
\]

\[
\begin{array}{c}
\text{R:fw:npt} \\
\frac{t[H] \langle \text{let } x = \text{NewPort} \ \text{in } S \ \text{end } C \rangle}{0} \quad | \quad t[H \ * x']\langle S\{x'/x\} \ \langle \text{esc } C \rangle\rangle \\
x' = \xi \ \parallel \ \xi : \bot\bot \\
\text{if } x', \xi \text{ fresh}
\end{array}
\]

\[
\begin{array}{c}
\text{R:fw:if1} \\
\frac{t[H] \langle \text{if } x \ \text{then } S_1 \ \text{else } S_2 \ \text{end } C \rangle}{x = \text{true}} \quad | \quad t[H \ \text{if}(x)S_2]\langle S_1 \ \langle \text{esc } C \rangle\rangle \\
x = \text{true}
\end{array}
\]

\[
\begin{array}{c}
\text{R:fw:nth} \\
\frac{t[H] \langle \text{thread } S \ \text{end } C \rangle}{0} \quad | \quad t[H \ * t']C \ \parallel \ t'[\bot]\langle S\ \langle \rangle \rangle \\
\text{if } t' \text{ fresh}
\end{array}
\]

\[
\begin{array}{c}
\text{R:fw:pc} \\
\frac{t[H] \langle \{ x \ (x_i)^n \} \ C \rangle}{x = \xi \ \parallel \ \xi : \text{proc} \ } \quad | \quad t[H \ \{ x \ (x_i)^n \} \langle S\{x/y_i\}^n \ \langle \text{esc } C \rangle\rangle \\
\text{S end}
\end{array}
\]

\[
\begin{array}{c}
\text{R:fw:snd} \\
\frac{t[H] \langle \{ \text{Send } x \ y \} \ C \rangle}{x = \xi \ \parallel \ \xi : K|K_{h}} \quad | \quad t[H \ \uparrow x]C \\
\text{R:fw:rev} \\
\frac{t[H] \langle \text{let } y = \{ \text{Receive } x \} \ \text{in } S \ \text{end } C \rangle}{\theta \ \parallel \ \xi : K; t':z|K_{h}} \quad | \quad t[H \ \downarrow x(y')]\langle S\{y'/y\} \ \langle \text{esc } C \rangle\rangle \\
\text{if } y' \text{ fresh} \land \ \theta \triangleq x = \xi \ \parallel \ z = w
\end{array}
\]

\[
\begin{array}{c}
\text{R:fw:scp} \\
\frac{t[H] \langle \text{esc } C \rangle}{0} \quad | \quad t[H \ \text{esc}]C
\end{array}
\]
μOz reversible semantics: backward rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Equation 1</th>
<th>Equation 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R:bk:skp</td>
<td>( t[H \text{ skip}]C ) 0 ( \parallel ) ( t[H] \langle \text{skip } C \rangle ) 0</td>
<td></td>
</tr>
<tr>
<td>R:bk:var</td>
<td>( t[H \ast x] \langle S \langle \text{esc } C \rangle \rangle ) ( x = v ) 0</td>
<td>( t[H] \langle \text{let } x = v \text{ in } S \text{ end } C \rangle ) 0</td>
</tr>
<tr>
<td>R:bk:npr</td>
<td>( t[H \ast x] \langle S \langle \text{esc } C \rangle \rangle ) ( x = \xi \parallel \xi : c ) 0</td>
<td>( t[H] \langle \text{let } x = c \text{ in } S \text{ end } C \rangle ) 0</td>
</tr>
<tr>
<td>R:bk:npt</td>
<td>( t[H \ast x] \langle S \langle \text{esc } C \rangle \rangle ) ( x = \xi \parallel \xi : \bot \bot ) 0</td>
<td>( t[H] \langle \text{let } x = \text{NewPort} \text{ in } S \text{ end } C \rangle ) 0</td>
</tr>
<tr>
<td>R:bk:if1</td>
<td>( t[H \text{ if}(x)S_2] \langle S_1 \langle \text{esc } C \rangle \rangle ) ( x = \text{true} ) ( x = \text{true} )</td>
<td>( t[H] \langle \text{if } x \text{ then } S_1 \text{ else } S_2 \text{ end } C \rangle ) 0</td>
</tr>
<tr>
<td>R:bk:nth</td>
<td>( t[H \ast t']C \parallel t'[\bot] \langle S \langle \rangle \rangle ) 0</td>
<td>( t[H] \langle \text{thread } S \text{ end } C \rangle ) 0</td>
</tr>
<tr>
<td>R:bk:pc</td>
<td>( t[H { x (x_i)^n }] \langle S \langle \text{esc } C \rangle \rangle ) 0</td>
<td>( t[H] \langle { x (x_i)^n } C \rangle ) 0</td>
</tr>
<tr>
<td>R:bk:snd</td>
<td>( t[H \uparrow x]C ) ( x = \xi \parallel \xi : t:y; K</td>
<td>K_h ) ( x = \xi \parallel \xi : K</td>
</tr>
<tr>
<td>R:bk:rcv</td>
<td>( t[H \downarrow x(z)] \langle S \langle \text{esc } C \rangle \rangle ) ( z = w \parallel x = \xi \parallel \xi : K</td>
<td>t':y,t; K_h ) ( x = \xi \parallel \xi : K; t':y</td>
</tr>
</tbody>
</table>