Causal-Consistent Reversible Debugging

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

Joint work with Elena Giachino (FOCUS) and Claudio Antares Mezzina (FBK Trento)
Our claim: Causal-consistent reversibility can help the programmer to debug concurrent applications.

Reversibility helpful also in a sequential setting:
- Some commercial debuggers provide the command “step back” in a sequential setting
  - For instance, gdb

Can we apply similar techniques in a concurrent setting?

Can we do better?
Debugging and causality

- Debugging amounts to find the bug that caused a given misbehavior
- Debugging strategy: follow causality links backward from misbehavior to bug
- Which primitives do we need?
- Mainly, primitives that given a misbehavior go back towards its causes
- We define them in the context of $\mu$Oz
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
Main primitives

- One primitive 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`
  - If `n` is unspecified, the last send is undone
- **Thread blocked on a receive**: `rollreceive id n` undoes the last `n` reads on the port `id`
  - If `n` is unspecified, the last receive is undone
- **Unexpected thread**: `rollthread t` undoes the creation of thread `t`
Using causal-consistent 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
- The procedure can be iterated till the bug is found
- Only relevant actions are undone
  - All the primitives in the last slide are causal consistent
Additional commands

- A debugger needs two kinds of commands
  - Commands to control execution
  - Commands to explore the configuration
    » Both code and state
- Some of the commands are standard, others related to reversibility or causal consistency
Execution control commands: standard

- Step forward
  - The user specifies the target thread
- Run
  - Round robin scheduler
- Breakpoints
Execution control commands: non standard

- **Step backward**
  - Goes back one step
  - The user specifies the target thread
  - Not enabled if waiting for dependencies to be undone
  - E.g., cannot step back the creation of a thread with non-empty history
  - Only in reversible debuggers

- **Roll t n**
  - Undoes the last n actions of thread t
  - Causal consistent
  - Only in causal-consistent reversible debuggers
Configuration exploration commands

- List of threads
  - Only in concurrent debuggers
- Display the code of a thread
- Display the history of a thread
  - Only in reversible debuggers
- Display the store
- Display the history of a queue
  - Only in reversible debuggers
Testing our approach

- According to [Lu et al., ASPLOS 08] most of the concurrency bugs
  - Involve only 2 threads, and a few variables
  - Are order violation, atomicity violation or deadlock
- The causal-consistent approach allows the programmer to concentrate on the involved threads
- We put our debugger at work on three paradigmatic examples, one for each class of common bugs
CaReDeb: a causal-consistent debugger

- Only a prototype to test our ideas
- Debugs μOz programs
- Written in Java
- Based on the reversible semantics of μOz
- Available at http://proton.inrialpes.fr/~mezzina/deb/
- Starts with java -jar deb.jar inputfile
CaReDeb: a causal-consistent debugger
Future work

- Improving the prototype
  - Usability: integration with eclipse
  - Performance: both in time and in space

- Extending the language
  - Additional constructs
  - What about applying the approach to a mainstream language?

- Assessment
  - Can we improve the rate of bug discovery thanks to our approach?
Finally

Thanks!

Questions?
μOz syntax

- $S ::= \begin{align*}
  \text{skip} & \quad \text{[Empty statement]} \\
  S_1 \ S_2 & \quad \text{[Sequence]} \\
  \text{let } x = v \ \text{in } S \ \text{end} & \quad \text{[Variable declaration]} \\
  \text{if } x \ \text{then } S_1 \ \text{else } S_2 \ \text{end} & \quad \text{[Conditional]} \\
  \text{thread } S \ \text{end} & \quad \text{[Thread creation]} \\
  \text{let } x = c \ \text{in } S \ \text{end} & \quad \text{[Procedure declaration]} \\
  \{x \ x_1 \ldots x_n\} & \quad \text{[Procedure call]} \\
  \text{let } x = \text{Newport} \ \text{in } S \ \text{end} & \quad \text{[Port creation]} \\
  \{\text{Send } x \ y\} & \quad \text{[Send]} \\
  \text{let } x = \{\text{Receive } y\} \ \text{in } S \ \text{end} & \quad \text{[Receive]} \\
\end{align*}

- $c ::= \text{proc } \{x_1 \ldots x_n\} \ S \ \text{end}$
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
\[
\begin{align*}
\langle \text{skip } T \rangle & \quad \parallel \quad T \\
0 & \quad \parallel \quad 0
\end{align*}
\]

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

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

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

R:ifl
\[
\begin{align*}
\langle \text{if } x \text{ then } S_1 \text{ else } S_2 \text{ end } T \rangle & \quad \parallel \quad \langle S_1 T \rangle \\
x = \text{true} & \quad \parallel \quad x = \text{true}
\end{align*}
\]

R:nth
\[
\begin{align*}
\langle \text{thread } S \text{ end } T \rangle & \quad \parallel \quad T \parallel \langle S \langle \rangle \rangle \\
0 & \quad \parallel \quad 0
\end{align*}
\]

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

R:send
\[
\begin{align*}
\langle \{ \text{Send } x \ y \} T \rangle & \quad \parallel \quad T \\
x = \xi \parallel \xi : Q & \quad \parallel \quad x = \xi \parallel \xi : y; Q
\end{align*}
\]

R:rcv
\[
\begin{align*}
\langle \text{let } x = \{ \text{Receive } y \} \text{ in } S \text{ end } T \rangle & \quad \parallel \quad \langle S\{x'/x\} T \rangle \\
y = \xi \parallel \xi : Q; z \parallel z = w & \quad \parallel \quad y = \xi \parallel \xi : Q \parallel z = w \parallel x' = w
\end{align*}
\quad \text{if } x' \text{ fresh}
\]
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
### μOz Reversible Semantics: Forward Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>R:fw:skp</td>
<td>$t[H](\text{skip } C)$</td>
<td>$t[H \text{ skip}] C$</td>
<td></td>
</tr>
<tr>
<td>R:fw:var</td>
<td>$t[H](\text{let } x = v \text{ in } S \text{ end } C)$</td>
<td>$t[H \ast x'] S{x'/x} \langle\text{esc } C\rangle$</td>
<td>$x'$ fresh</td>
</tr>
<tr>
<td>R:fw:npr</td>
<td>$t[H](\text{let } x = c \text{ in } S \text{ end } C)$</td>
<td>$t[H \ast x'] S{x'/x} \langle\text{esc } C\rangle$</td>
<td>$x'$, $\xi$ fresh</td>
</tr>
<tr>
<td>R:fw:npt</td>
<td>$t[H](\text{let } x = \text{NewPort} \text{ in } S \text{ end } C)$</td>
<td>$t[H \ast x'] S{x'/x} \langle\text{esc } C\rangle$</td>
<td>$x'$, $\xi$ fresh</td>
</tr>
<tr>
<td>R:fw:if1</td>
<td>$t[H](\text{if } x \text{ then } S_1 \text{ else } S_2 \text{ end } C)$</td>
<td>$t[H \text{ if}(x)S_2] S_1 \langle\text{esc } C\rangle$</td>
<td>$x = \text{true}$</td>
</tr>
<tr>
<td>R:fw:nth</td>
<td>$t[H](\text{thread } S \text{ end } C)$</td>
<td>$t[H \ast t'] C \parallel t'[\bot] S$</td>
<td>$t'$ fresh</td>
</tr>
<tr>
<td>R:fw:pc</td>
<td>$t[H]\langle{ x \ (x_i)^n } C \rangle$</td>
<td>$t[H { x \ (x_i)^n }\rangle S{y_i/y_i}^n \langle\text{esc } C\rangle)$</td>
<td>$x = \xi \parallel \xi : \text{proc} { (y_i)^n } S \text{ end}$</td>
</tr>
<tr>
<td>R:fw:snd</td>
<td>$t[H]\langle{ \text{Send } x \ y } C \rangle$</td>
<td>$t[H \uparrow x] C$</td>
<td>$x = \xi \parallel \xi : K</td>
</tr>
<tr>
<td>R:fw:rev</td>
<td>$t[H]\langle{ \text{Receive } x } \text{ in } S \text{ end } C \rangle$</td>
<td>$t[H \downarrow (x'(y')) S{y'/y} \langle\text{esc } C\rangle)$</td>
<td>$\theta \parallel \xi : K; t':z</td>
</tr>
<tr>
<td>R:fw:scp</td>
<td>$t[H]\langle\text{esc } C\rangle$</td>
<td>$t[H \text{ esc}] C$</td>
<td></td>
</tr>
</tbody>
</table>
μOz reversible semantics: backward rules

R:bk:skp

\[
\begin{array}{c|c}
 t[H \, \text{skip}] C & t[H] \langle \text{skip } C \rangle \\
\hline
0 & 0 \\
\end{array}
\]

R:bk:var

\[
\begin{array}{c|c}
 t[H \, \ast x] \langle S \, \langle \text{esc } C \rangle \rangle & t[H] \langle \text{let } x = v \text{ in } S \text{ end } C \rangle \\
\hline
x = v & 0 \\
\end{array}
\]

R:bk:npr

\[
\begin{array}{c|c}
 t[H \, \ast x] \langle S \, \langle \text{esc } C \rangle \rangle & t[H] \langle \text{let } x = c \text{ in } S \text{ end } C \rangle \\
\hline
x = \xi \parallel \xi : c & 0 \\
\end{array}
\]

R:bk:npt

\[
\begin{array}{c|c}
 t[H \, \ast x] \langle S \, \langle \text{esc } C \rangle \rangle & t[H] \langle \text{let } x = \text{NewPort} \text{ in } S \text{ end } C \rangle \\
\hline
x = \xi \parallel \xi : \bot \bot & 0 \\
\end{array}
\]

R:bk:ifl

\[
\begin{array}{c|c}
 t[H \, \text{if}(x)S_2] \langle S_1 \, \langle \text{esc } C \rangle \rangle & t[H] \langle \text{if } x \text{ then } S_1 \text{ else } S_2 \text{ end } C \rangle \\
\hline
x = \text{true} & x = \text{true} \\
\end{array}
\]

R:bk:nth

\[
\begin{array}{c|c}
 t[H \, \ast t'] C \parallel t'[\bot] \langle S \, \langle \rangle \rangle & t[H] \langle \text{thread } S \text{ end } C \rangle \\
\hline
0 & 0 \\
\end{array}
\]

R:bk:pc

\[
\begin{array}{c|c}
 t[H \, \{ \, x \, (x_i)_1^n \, \} ] \langle S \, \langle \text{esc } C \rangle \rangle & t[H] \langle \{ \, x \, (x_i)_1^n \, \} \, C \rangle \\
\hline
0 & 0 \\
\end{array}
\]

R:bk:snd

\[
\begin{array}{c|c}
 t[H \, \uparrow x] C & t[H] \langle \{ \, \text{Send } x \, y \, \} \, C \rangle \\
\hline
x = \xi \parallel \xi : t : y ; K | K_h & x = \xi \parallel \xi : K | K_h \\
\end{array}
\]

R:bk:rcv

\[
\begin{array}{c|c}
 t[H \, \downarrow x(z)] \langle S \, \langle \text{esc } C \rangle \rangle & t[H] \langle \text{let } z = \{ \, \text{Receive } x \, \} \text{ in } S \text{ end } C \rangle \\
\hline
z = w \parallel x = \xi \parallel \xi : K | t' : y ; t ; K_h & x = \xi \parallel \xi : K ; t' : y | K_h \\
\end{array}
\]

R:bk:scp

\[
\begin{array}{c|c}
 t[H \, \text{esc}] C & t[H] \langle \text{esc } C \rangle \\
\hline
0 & 0 \\
\end{array}
\]