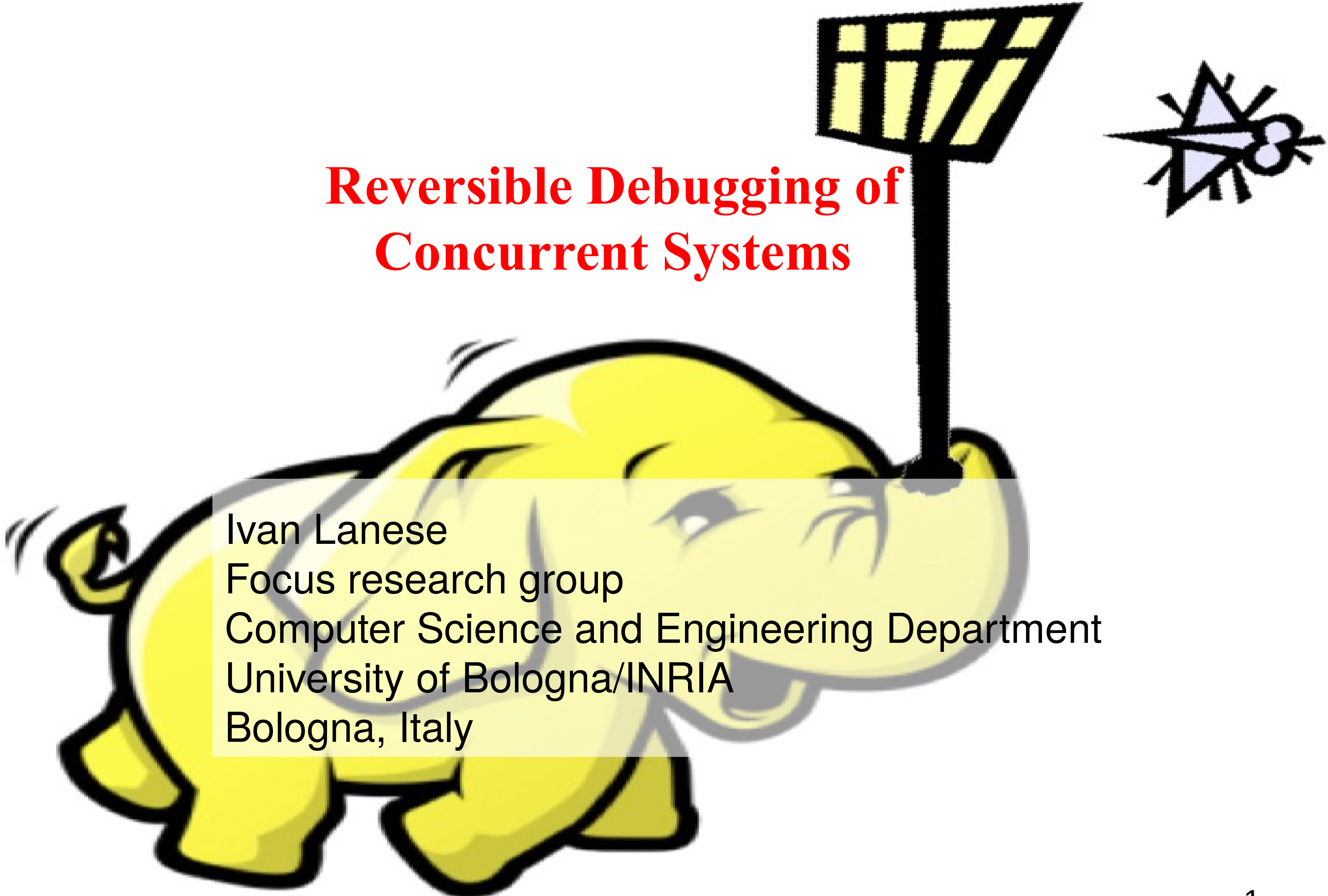
Reversible Debugging of Concurrent Systems



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

Roadmap

- Motivation
- State of the art: Sequential reversible debugging
- Causal-consistent reversible debugging
- Future directions

Roadmap

- Motivation
- State of the art: Sequential reversible debugging
- Causal-consistent reversible debugging
- Future directions

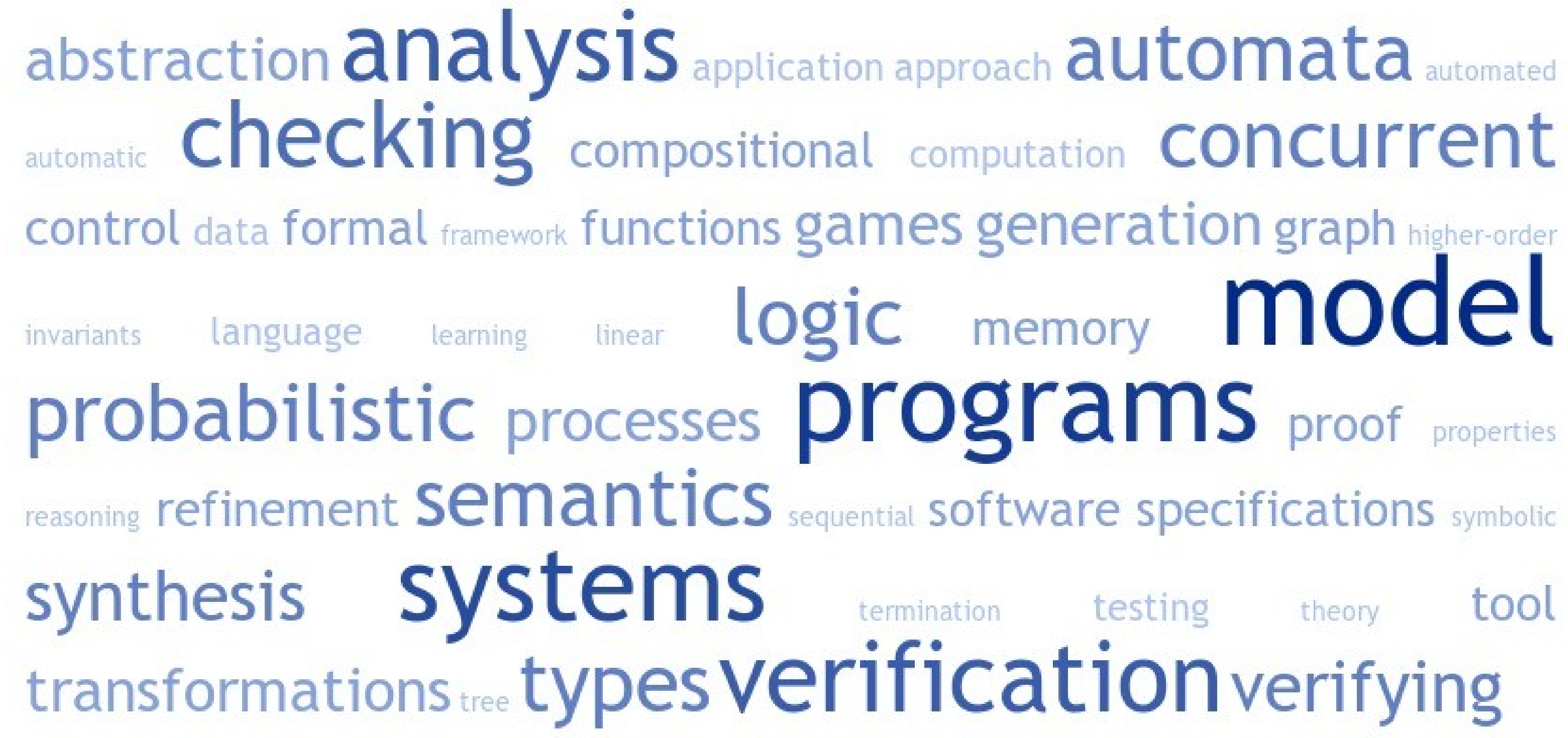
Why debugging?

- 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, heterogeneity
  - Cloud, IoT

Debugging is neglected?

- There should be lot of research on debugging
  - In particular from our community
- Let us set up an experiment
- Let us analyze the titles of papers accepted at the last 5 editions of main ETAPS conferences
  - ESOP, FASE, FOSSACS, TACAS

Result at a glance



Highlights of the results

- Debugging is not in the wordle
  - Top 50 words, at least 12 occurrences each
- Actually, there were 4 occurrences of debug*
- By comparison:
  - analisys: 56
  - verification: 51
  - type: 44
  - synthesis: 29
  - refinement: 20
  - specification: 20
  - test: 19
  - PRISM: 4

Roadmap

- Motivation
- State of the art: Sequential reversible debugging
- Causal-consistent reversible debugging
- Future directions

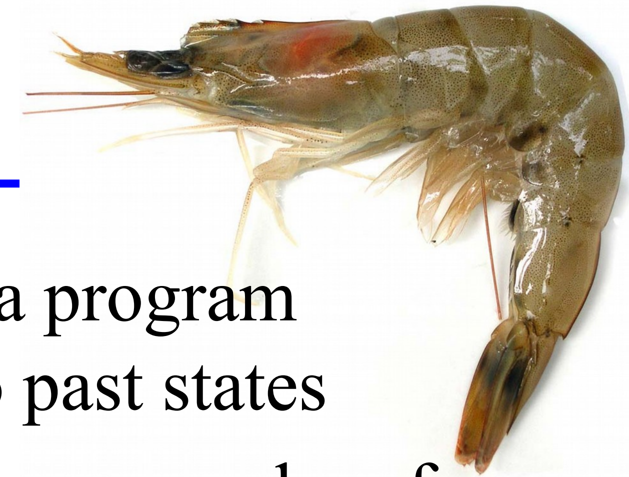
Standard debugging strategy

- 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 of standard debugging

- 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 the breakpoint is too late, the execution needs to be redone with an earlier breakpoint
  - Frequently many attempts are needed
  - Watchpoints do not help either

Reversibility to the rescue



- Reversibility: the possibility of executing a program both forward and backward, going back to past states
- Backward execution: undoing actions in reverse order of execution
- Requires history information since normal computation loses information
  - x=0 loses the old value of x

Reversibility for debugging

- Reversible debuggers extend standard debuggers
- Can execute the program under analysis both forward and backward
- Avoids the common “Damn, I put the breakpoint too late” exclamation
  - Just execute backward from where the program stopped till the desired point is reached

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 explored like a movie

Sequential 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
  - Even bugs related to concurrency can always be replayed

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 a breakpoint/watchpoint triggers
  - Breakpoints and watchpoints can be used also in the backward direction

A commercial reversible debugger: UndoDB

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

UndoDB commands

- Close to GDB commands
- Some more high-level commands and configuration commands
- 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

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

- Memory and time overheads are a relevant issue

Roadmap

- Motivation
- State of the art: Sequential reversible debugging
- Causal-consistent reversible debugging
- Future directions

Reversible debugging of concurrent systems

- We are interested in reversible debugging of concurrent systems
- Current approaches work on a linearization of the execution
- Causal information is lost by linearization
  - Can we exploit this information for improving debugging?
  - E.g., in model checking this information is exploited by partial-order reduction

Causal-consistent reversibility


- Since [Danos&Krivine, CONCUR 2004] the main 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
- At any point, many actions can be undone

Causal-consistent reversibility: rationale

- Execution order of concurrent actions should not have an impact
  - Not relevant
  - A full order not always exists
- Causal dependences instead are important
- This ensures that only states that could have been reached in the forward computation are reachable
- How to apply this definition to debugging?

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?

A proposal: 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
  - This depends on the language
- Examples are:
  - Wrong value in a variable: **rollvariable id** goes to the state just before the last change to variable **id**
  - Unexpected thread: **rollthread t** undoes the creation of thread **t**

Causal-consistent debugging strategy, refined

- 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

Exploiting causality information

- Some non-trivial errors become immediately visible
- Interference: if thread **t1** and thread **t2** should be independent but rollbacking **t1** makes also **t2** rollback
- Missing synchronization: if thread **t1** and thread **t2** should be dependent but rollbacking **t1** has no impact on **t2**
  - Dual of interference

Paradigmatic example: deadlock

- A thread **t1** is blocked but not terminated
- Inspecting **t1** one can find the resource which is not available
- By rollbacking the last grant of the resource one can find which thread holds the resource
- One can explore this thread backward to understand why it holds the resource

CaReDeb: a causal-consistent debugger

- Only a prototype to test our ideas
- Debugs programs in the μOz language
  - Toy functional language with threads and asynchronous communication via ports
- Written in Java
- Available at http://www.cs.unibo.it/caredeb
- Description and underlying theory in [Giachino, Lanese & Mezzina, FASE 2014]
- Interface not much user friendly...

Roadmap

- Motivation
- State of the art: Sequential reversible debugging
- Causal-consistent reversible debugging
- Future directions

Summary

- Debugging is a relevant but neglected topic
- Our community should be able to provide contributions in this area
- Causal-consistent reversible debugging is one possible direction
- Even for this direction we are just at the beginning

Future directions: making the approach practical

- Enable causal-consistent debugging of real languages
  - Need to understand the causal semantics of all constructs
  - Interplay with memory management
  - Large theoretical and implementation work
- Current work on a subset of Erlang
  - Actor-based concurrency easier than shared-memory concurrency
- We also plan to tackle Java + Akka
- Efficiency (time and size of history information)
- Integration in standard tool-chain
  - Building on top of GDB and integration into Eclipse

Future directions: to **roll** or not to **roll**?

- Is the **roll** primitive good?
  - Which is the impact on actual debugging?
  - It would be interesting to setup an experiment
- Is the **roll** primitive helpful for all kinds of bugs?
- Are there other useful primitives?

Finally

Thanks!

Questions?

Our target language: μOz

- A kernel language of Oz
  [P. Van Roy and S. Haridi. Concepts, Techniques and Models of Computer Programming. MIT Press, 2004]
- 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