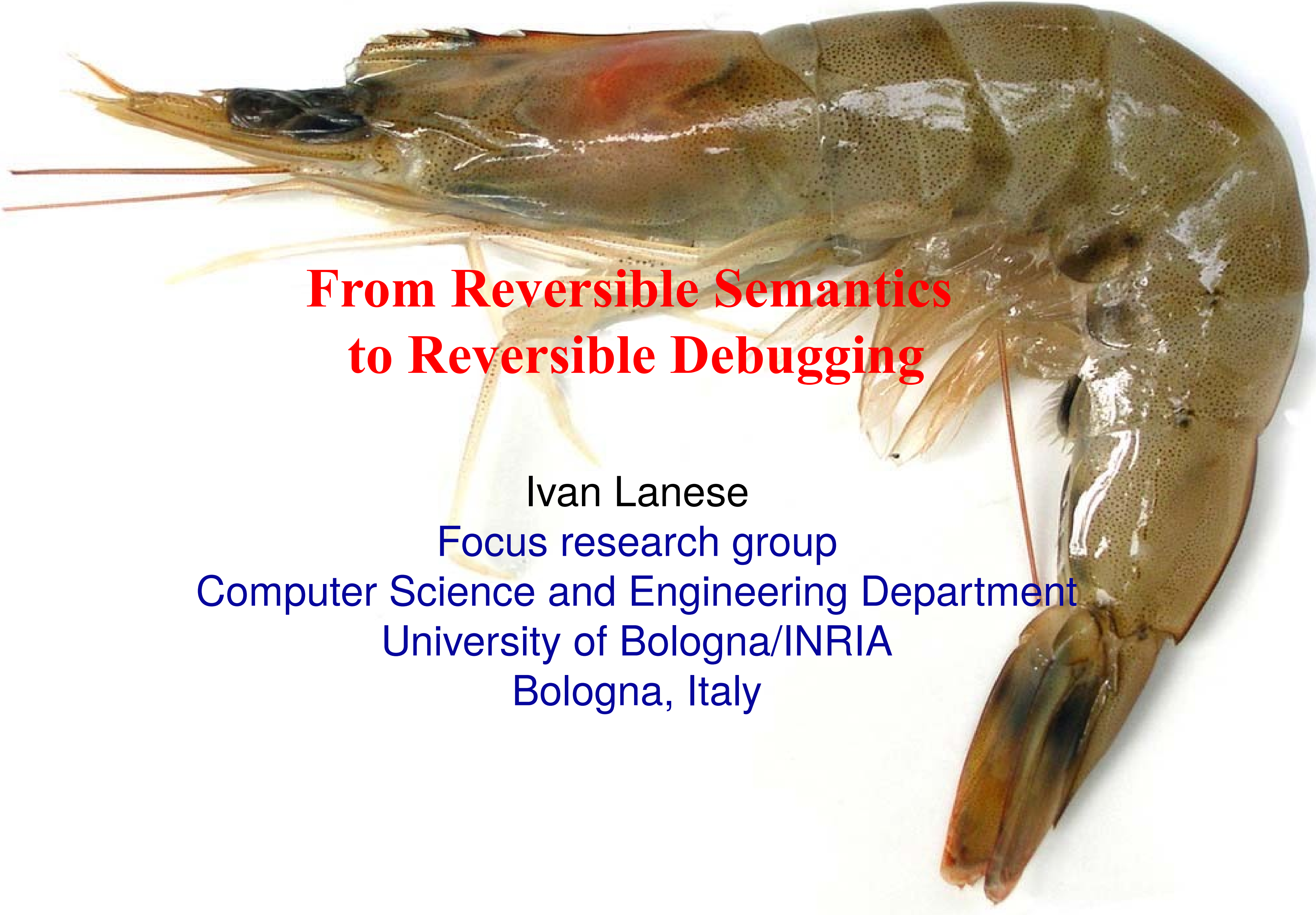


From Reversible Semantics to Reversible Debugging

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

Impact

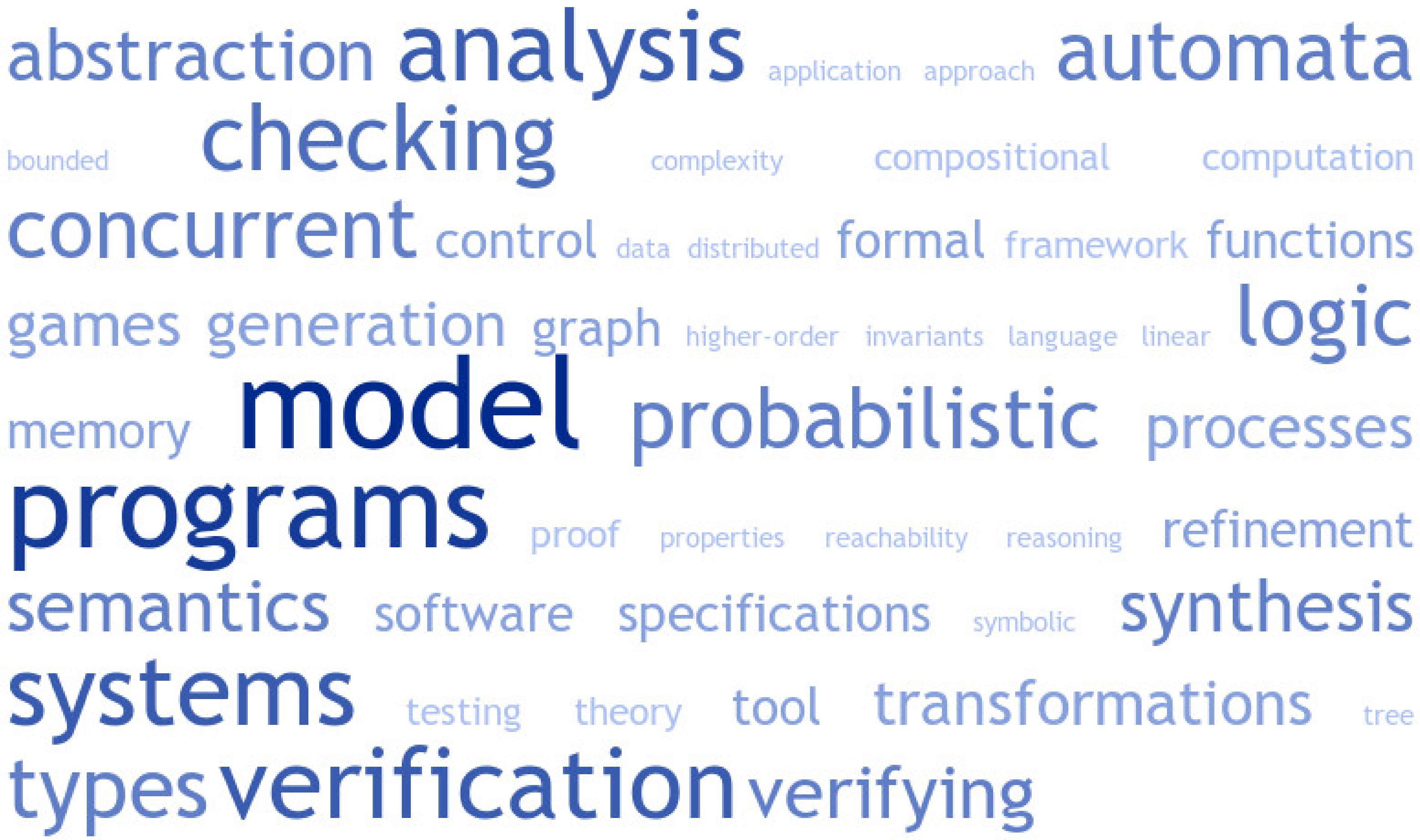
Research on debugging has impact

- 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
  - Cloud, IoT

How much research on debugging?

- We expect lot of research on debugging
- Let us set up an experiment
- Let us analyze the titles of papers accepted at the last 6 editions of main ETAPS conferences
  - ESOP, FASE, FOSSACS, TACAS

Result at a glance

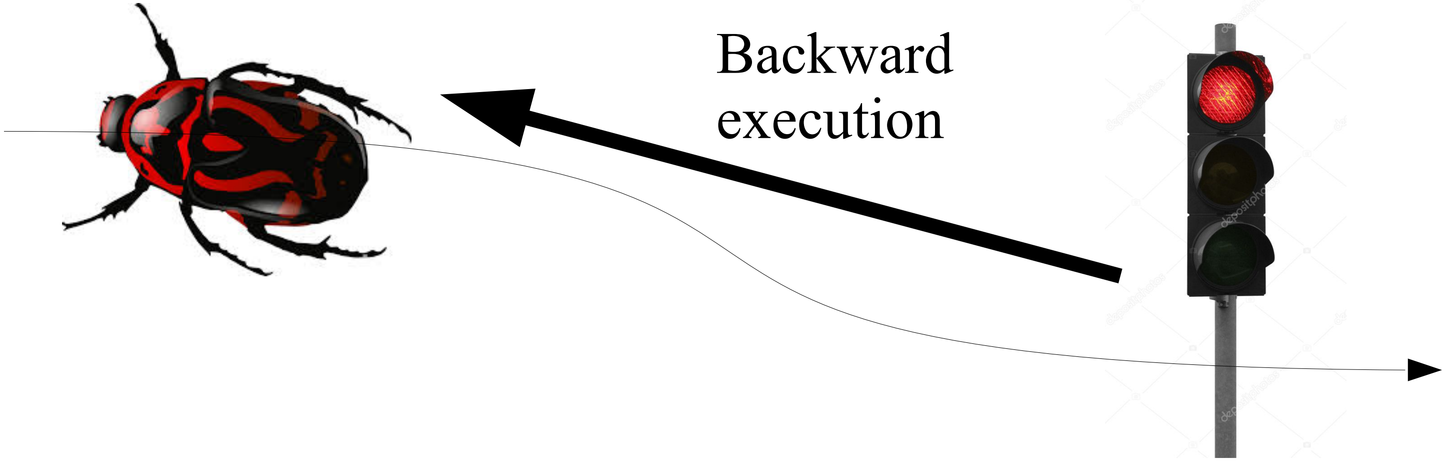


Highlights of the results

- Debugging is not in the wordle
  - Top 49 words, at least 15 occurrences each
- Actually, there were 4 occurrences of debug*
- As a comparison:
  - analisys: 66
  - verification: 63
  - type: 44
  - synthesis: 34
  - abstraction: 32
  - specification: 23
  - testing: 21

We want to study debugging

- Reversibility can play and plays a role
- In debugging given a misbehavior you have to find the bug causing it
- The bug precedes the misbehavior
- Reversible debugging enables to execute the program backward looking for the bug



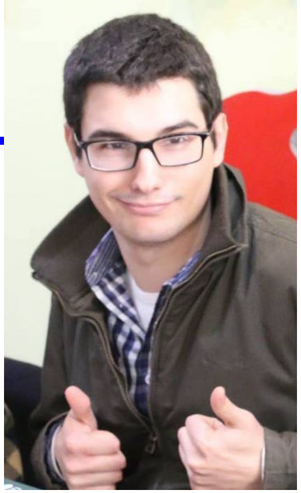
Status of reversible debugging

- Well understood in the sequential setting
- Actively used in open source community and industry
  - GDB, Microsoft WinDbg, UndoDB
- Very limited in the analysis of concurrent programs
  - Either not supported or executions are linearized
  - Inefficient, and loses relevant information
  - Causal relations between processes at the basis of concurrent bugs
    - Deadlocks, races, ...
- We tackled the challenge of reversible debugging of concurrent systems

Current status of our work: CauDEr

- Reversible debugger for a subset of Core Erlang
- Intermediate result in the compilation of Erlang
- Erlang is a functional language for concurrent and distributed systems from Ericsson
  - Based on asynchronous message passing following the actor model
  - Used in relevant projects such as Facebook chat

CauDEr approach



- CauDEr allows one to explore a concurrent computation back and forward
- Provides support to
  - find the direct causes of a visible (mis)behavior
  - go back when the corresponding instruction has been executed
- Then one can
  - analyze the instruction and the state
  - find the bug or find information on how to go backward again
- This may involve multiple processes because of interaction

Plan of the talk

- More than on the current status I want to present the journey that lead us there
- I will present it via 5 questions that had to be answered
- This covers a line of work starting in 2004, to which many people contributed

5 questions

1. Which is the correct notion of reversibility for concurrent systems?
2. Which history information needs to be stored?
3. How to control the basic reversibility mechanism?
4. How to exploit reversibility for debugging?
5. How to apply reversible debugging to real languages?

5 questions

1. Which is the correct notion of reversibility for concurrent systems?
2. Which history information needs to be stored?
3. How to control the basic reversibility mechanism?
4. How to exploit reversibility for debugging?
5. How to apply reversible debugging to real languages?

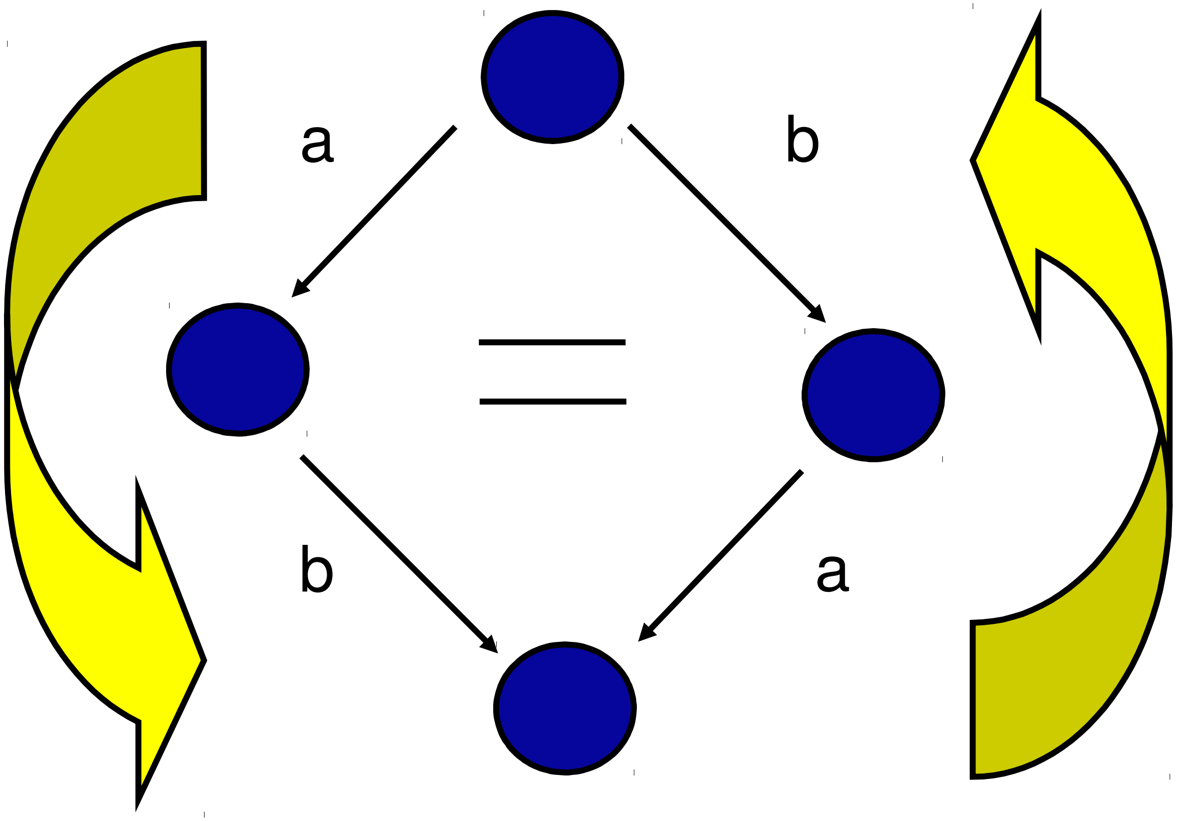
Reverse execution of a sequential program

- Recursively undo the last step
  - Computations are undone in reverse order
  - To reverse A;B reverse B, then reverse A
- Not suitable for concurrent systems: there is no clearly defined last step
  - Different steps may have overlapping time spans
  - In some distributed systems there is not even an agreed notion of time

Reversibility and concurrency

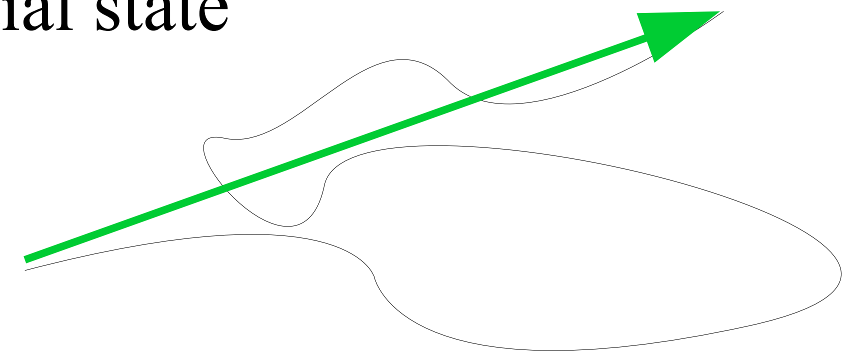
- What to do in a concurrent setting?
- For sure, if action A caused action B, A could not be the last one
- Causal-consistent reversibility: recursively undo any action whose consequences (if any) have already been undone
  [Danos & Krivine: CONCUR 2004]
- Relies on a notion of causality
- Conservative extension of the sequential definition

Causal-consistent reversibility



Why do we want causal consistency?

- If we are not causal consistent we may undo a cause without undoing the consequence
- We reach a state where the consequence is in place, without any cause justifying it
  - Possibly not reachable by forward execution
- In causal-consistent reversibility all states are reachable with a forward-only computation
  - When starting from an initial state
- Suitable for debugging



Undoing computational steps

- We need to undo steps of the computation
- Not necessarily easy
- Computation steps may cause loss of information
  - X=5 causes the loss of the past value of X
  - X=X+Y causes no loss of information
- Loss of information enemy of reversibility
- Sequential reversible languages never lose information and are backward deterministic
- Concurrent reversible languages lose selected information and are backward confluent
  - Concurrent actions can be undone in any order

Different approaches to undo

- Saving a past state and redoing the same computation from there (rollback-recovery)
- Undoing steps one by one
  - Limiting the language to constructs that are reversible
    - Featuring only actions that cause no loss of information
    - Approach of languages such as Janus, ROOPL and RFun
  - Taking a language which is not reversible and make it reversible
    - One should save information on the past configurations
    - X=5 becomes reversible by recording the old value of X
- We use this last approach

5 questions

1. Which is the correct notion of reversibility for concurrent systems?
2. Which history information needs to be stored?
3. How to control the basic reversibility mechanism?
4. How to exploit reversibility for debugging?
5. How to apply reversible debugging to real languages?

We need enough information, but not too much

- We need enough information to recover past states
- But there is also some information we do not want
- Two key results from causal-consistent reversibility theory limit the amount of information we do want

Loop Lemma

- Doing A and then undoing A should lead back to the original state
- Undoing A (if in the past) ad then redoing A should lead back to the original state
- When going backward we forget everything we learned
- E.g., we cannot count the number of backward steps
- Meaningful for debugging
  - we are interested in the state of the program
  - not in how we used the debugger commands to reach it

Causal equivalence

- Two computations are causal equivalent if they differ only for
  - removing/adding pairs of do A/undo A or undo A/redo A
  - swapping concurrent actions
- Equivalence relation on computations

Causal consistency theorem

- Two coinitial computations are cofinal iff they are causal equivalent
- The right to left implication is easy
  - Causal equivalence does not change the final state
- The left to right implication seems very strong
- None of the implications seems related to history information

- Pay attention: history information is part of the state

Causal consistency theorem explained

- Two coinitial computations are cofinal iff they are causal equivalent
- We cannot remember anything distinguishing causal equivalent computations
  - E.g., the order of concurrent actions
- We need to remember something about any other difference
  - Keep track of any nondeterministic choice

Example

-
  ```
  x = {0,1}
  if x=0 then
          x=2
  else
          x=2
  endif
  ```

  Nondeterministic assignment

- Two possible computations
- The two possible computations lead to the same state
- The causal consistency theorem states that we need history information to distinguish them
  - At least we should trace the chosen branch

Uncontrolled reversibility

- We know
  - which actions can be reversed
  - which history information is needed
  - (how to efficiently store it remains a difficult problem)
- We can define an uncontrolled reversible semantics
  - enables to execute actions back and forward
  - if causal consistency is satisfied
- It has been defined for many calculi and languages
- Enables to write a reversible simulator

Is uncontrolled reversibility enough?

- Good to fix the theoretical footing of a reversible language
  - Highlights concepts of causality and history information
  - Supported by the causal-consistency theory
  - Mostly application independent
- Too wild for real applications
- We need control mechanisms telling us when to go forward and when to go backward, and up to where

5 questions

1. Which is the correct notion of reversibility for concurrent systems?
2. Which history information needs to be stored?
3. How to control the basic reversibility mechanism?
4. How to exploit reversibility for debugging?
5. How to apply reversible debugging to real languages?

Separation of concerns

- We propose two layers
  - An uncontrolled layer specifying what can be undone and how
  - A controlled layer specifying when things need to be undone
- Many mechanisms exists
  - Irreversible actions, controller processes, ...
- The choice depends on the application area
- In debugging we want to undo a past action that is the possible cause of a given misbehavior

How to undo an arbitrary past action?

- We need to undo its consequences first
- We do not want to undo any other action
- We can apply this idea recursively
- We call such an operation a causal-consistent rollback

How to implement a rollback?

-
  ```
  Roll(A){
      for each B $\in$ {direct consequences of A}
          Roll(B)
      undo A}
  ```
- Maybe not the most efficient implementation but …
- … it is language independent
- … it is built on top of the uncontrolled semantics
  - Results can be lifted, e.g., only forward reachable states can be reached

5 questions

1. Which is the correct notion of reversibility for concurrent systems?
2. Which history information needs to be stored?
3. How to control the basic reversibility mechanism?
4. How to exploit reversibility for debugging?
5. How to apply reversible debugging to real languages?

Causal-consistent reversible debugging

- General debugging aim: given a visible misbehavior, finding the bug causing it
- The bug can be in a different process, yet there is always a causal chain connecting the two
- Main idea: follow the causal chain backward using rollback
- History information can be used to find the target for the rollback
- The details depend on the chosen language and on the kind of misbehavior

Example: wrong message

- Erlang is based on message passing
- A possible misbehavior is a wrong message
- By inspecting the history one can find the corresponding send instruction
- One can then undo the send instruction using roll
- By inspecting the code and the state before the send one gets useful information on where the bug may be
- If the wrong message is due to a wrong value in a variable one can roll the variable assignment, and so on

A causal-consistent debugger includes

- All commands for forward debugging
  - Step by step execution, breakpoints, state exploration
- Reversible simulation commands
- Causal-consistent rollback
- History exploration

5 questions

1. Which is the correct notion of reversibility for concurrent systems?
2. Which history information needs to be stored?
3. How to control the basic reversibility mechanism?
4. How to exploit reversibility for debugging?
5. How to apply reversible debugging to real languages?

We are currently working on this question

- Real languages are complex
  - We need a semantics for them
  - We need a concurrency model for them
  - We need to be able to undo any possible action
- Lot of work is needed
- Yet we know some possible simplifications

Separation of concerns

- Languages like Erlang are composed by
  - a sequential part
  - few constructs dealing with concurrency (send, receive, spawn)
- One can
  - leverage approaches for sequential reversibility for the sequential part
  - concentrate on the few concurrent constructs
- Not possible in languages with shared memory
  - Any assignment and variable access has impact on concurrency

Approximation

- The actual causality structure may be very complex
- E.g., are two processes reading from the same mailbox queue dependent?
  - Of course yes…
  - … but not if the two messages are equal
  - … or if the two messages are discarded
  - … or if they lead to the same effect
- Not easy to characterize and deal with
- Approximation: they are always dependent
- Drawback: fake dependences are created
  - May be misleading

Back to the future

- This is where we are now
- Feel free to play with CauDEr

  https://github.com/mistupv/cauder
- Lot of work remains
  - We have positions on this topic
  - If looking for a PhD or postdoc on this topic contact me

Future work

- On CauDEr
  - Support full Erlang
  - Enable to record a computation in the production environment and replay it inside the debugger (ongoing)
  - Improving efficiency
- More in general
  - Evaluate how much causal-consistent debugging helps the developer
  - Are there other useful debugger commands?
  - Applying the approach to other languages

Finally

Thanks!

Questions?