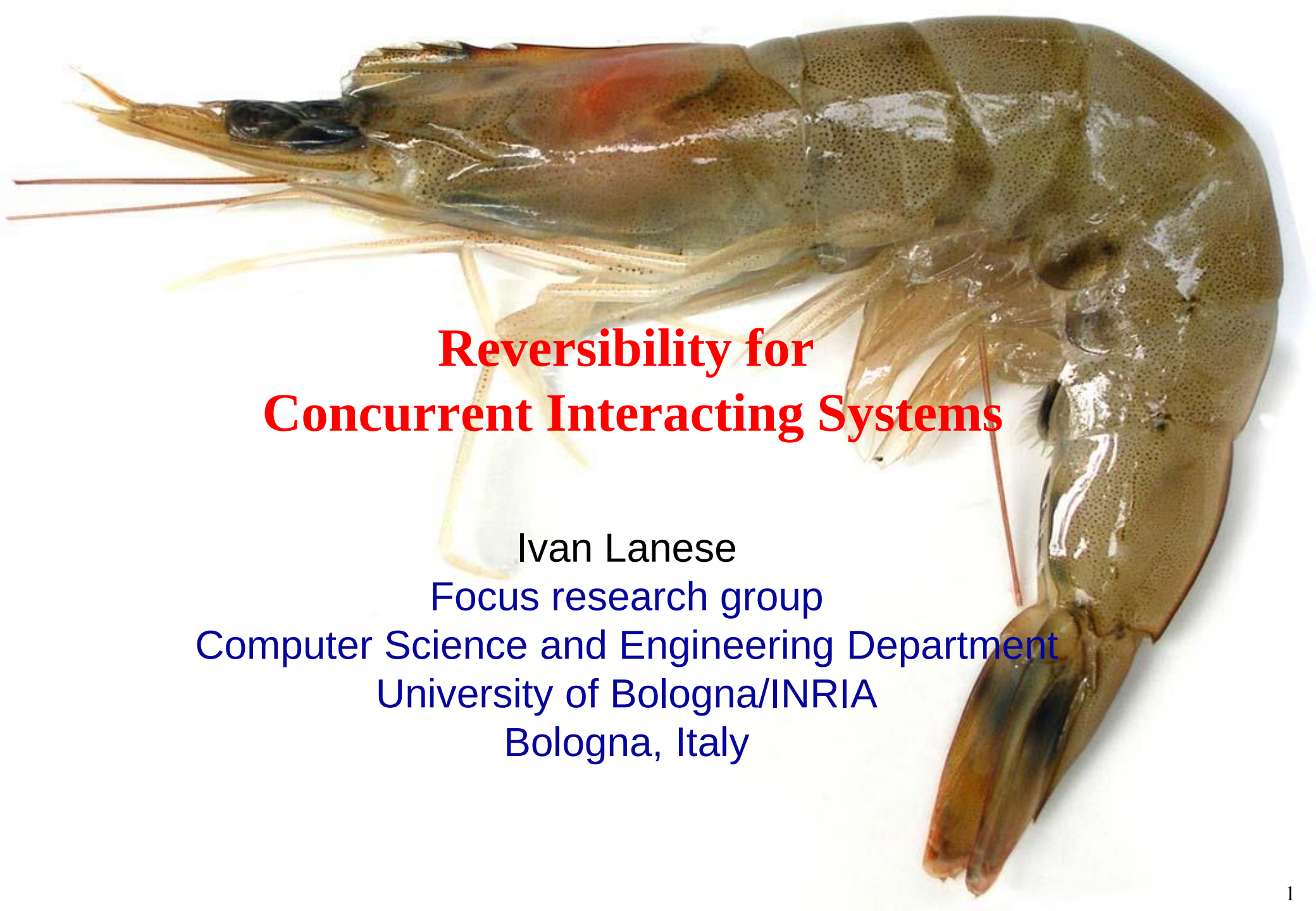


Reversibility for Concurrent Interacting Systems

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

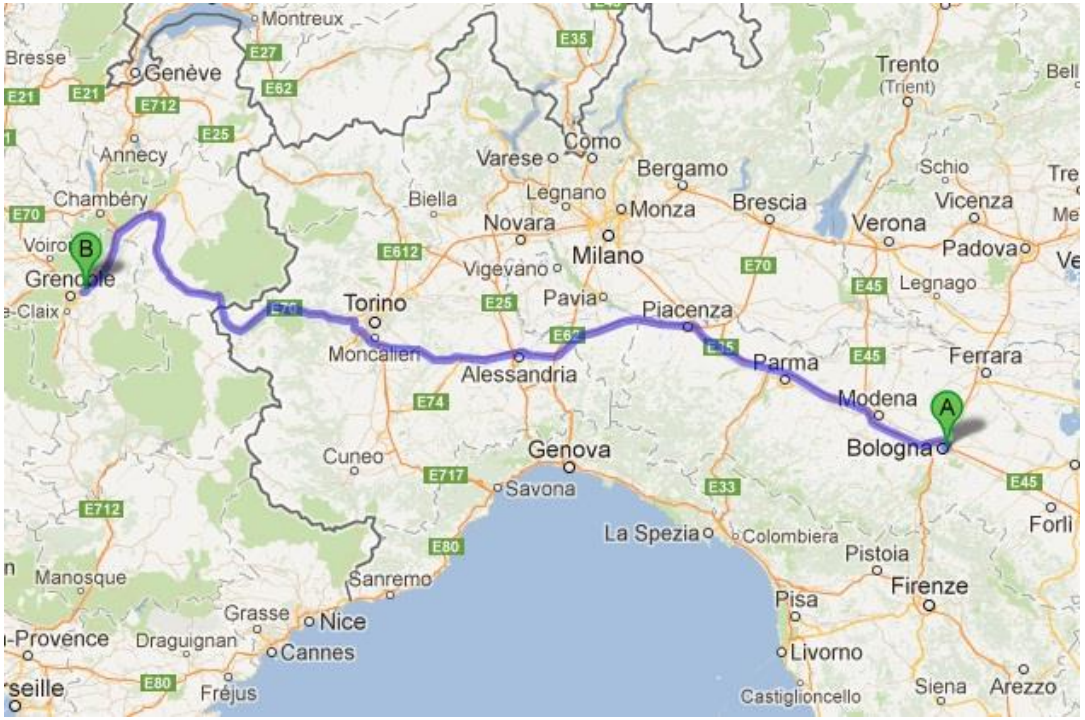
Contributors

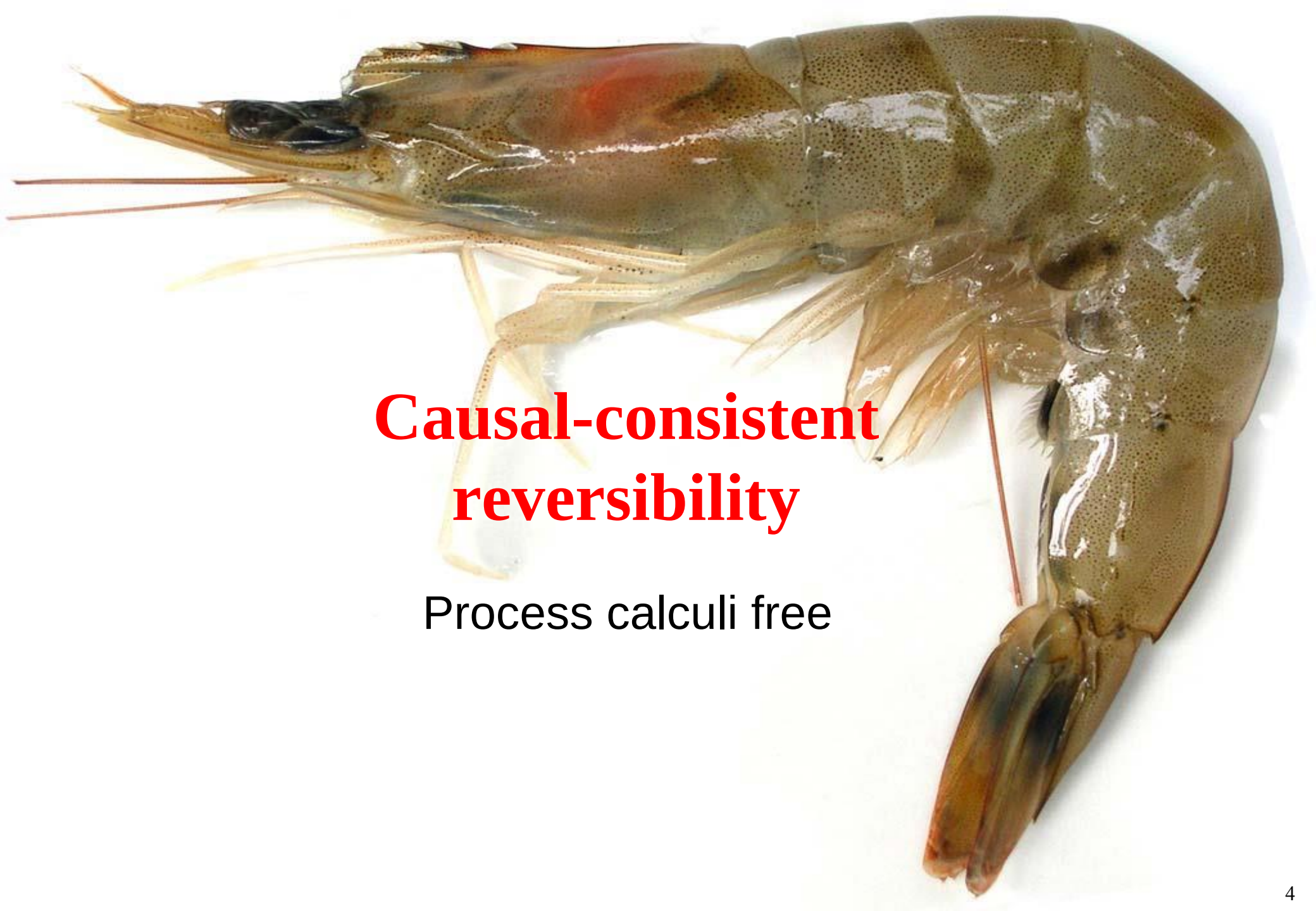
- My coauthors: Elena Giachino (Italy), Michael Lienhardt (Italy), Claudio Antares Mezzina (Italy), Jean-Bernard Stefani (France), Alan Schmitt (France), Francesco Tiezzi (Italy)
- Other groups working on similar topics
  - In France: Jean Krivine, Daniele Varacca, Ioana Cristescu, ...
  - In UK: Irek Ulidowski, Iain Phillips, ...

- Part of the work has been done inside the French ANR Project REVER
  - INRIA Grenoble, PPS, CEA, Bologna

Map of the talk

1. Causal-consistent reversibility
2. Controlling reversibility
3. Specifying alternatives
4. Conclusion





Causal-consistent reversibility

Process calculi free

What is reversibility for us?

The possibility of executing a computation both in the standard, forward direction, and in the backward direction, going back to a past state

- What does it mean to go backward?
- If from state S_1 we go forward to state S_2, then from state S_2 we should be able to go back to state S_1

Reversibility everywhere

- Reversibility widespread in the world
  - Chemistry/biology
  - Quantum phenomena
  - Circuits
  - Robots
  - ...

Why reversibility for concurrent systems?

- Modelling concurrent systems
  - Suitable for systems which are naturally reversible
  - Biological, chemical, ...
- Programming concurrent systems
  - State space exploration, such as in Prolog
  - Define reversible functions
  - Build reliable systems
- Debugging concurrent systems
  - Avoid the “Gosh, I should have put the breakpoint at an earlier command” problem

Reversibility for reliability: the idea

- To make a system reliable we want to avoid “bad” states
- If a bad state is reached, reversibility allows one to go back to some past state
- Far enough, so that the decisions leading to the bad state has not been taken yet
- When we restart computing forward, we should try new directions

Reversibility and patterns for reliability

- Reversibility seems related to some patterns for programming reliable systems
- Checkpointing
  - We save the state of a program to restore it in case of errors
- Rollback-recovery
  - We combine checkpointing with logs to recover a program state
- Transactions [see Rudi's talk]
  - Computations which are executed all or nothing
  - In case of error their effect should be undone
  - Both in database systems (ACID transactions) and in service oriented computing (long running transactions)

Some other manifestation of reversibility?

- Application undo
  - Allows you to undo a wrong command in your favorite editor
- Backup
  - Allows one to go back to a past version of a file
- SVN and the like
  - More refined techniques to go back to past versions of files
- Understanding reversibility will shed some light on these mechanisms too?

What is the status of approaches to reliability?

- A lot of approaches
- A bag of tricks to face different problems
- No clue on whether and how the different tricks compose
- No unifying theory for them

- Understanding reversibility is the key to
  - Understand existing patterns for programming reliable systems
  - Combine and improve them
  - Develop new patterns

Reverse execution of a sequential program

- Recursively undo the last step
  - Computations are undone in reverse order
  - To reverse A;B reverse first B, then reverse A
- First we need to undo single computation steps
- We want the Loop Lemma to hold
  - From state S, doing A and then undoing A should lead back to S
  - From state S, undoing A (if A is in the past) and then redoing A should lead back to S
  - [Danos, Krivine: Reversible Communicating Systems. CONCUR 2004]

Undoing computational steps

- 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
  - Old value of X can be retrieved by doing X=X-Y

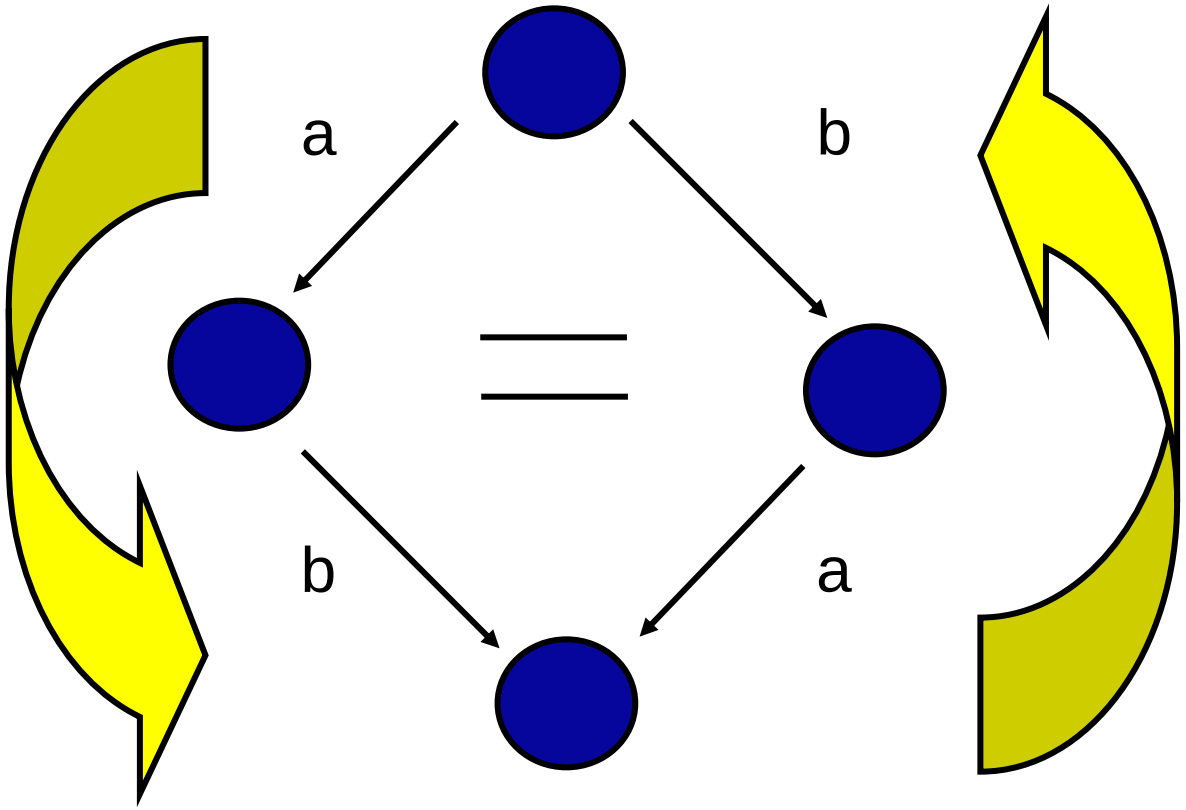
Different approaches to reversibility

- Saving a past state and redoing the same computation from there
- Undoing steps one by one
  - Considering languages which are reversible
    - » Featuring only actions that cause no loss of information
  - 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

Reversibility and concurrency

- In a sequential setting, recursively undo the last step
- Which is the last step in a concurrent setting?
- Many possibilities
- For sure, if an action A caused an action B, A could not be the last one
- Causal-consistent reversibility: recursively undo any action whose consequences (if any) have already been undone
- Proposed in [Danos, Krivine: Reversible Communicating Systems. CONCUR 2004]

Causal-consistent reversibility



Causal-consistent reversibility: advantages

- No need to understand timing of actions
  - Difficult since a unique notion of time may not exist
- Only causality has to be analyzed
  - Easier since causality has a local effect

Causal history information

- Remembering history information is not enough
- We need to remember also causality information
- Actions performed by the same thread are always totally ordered by causality
- Actions in different threads may be related if the threads interact
- If thread T_1 sent a message to thread T_2 then
  - T_2 depends on T_1
  - T_1 cannot reverse the send before T_2 reverses the receive
- We need to remember information on communication between threads

Causal equivalence

- According to causal-consistent reversibility
  - Changing the order of execution of concurrent actions should not make a difference
  - Doing an action and then undoing it (or undoing and redoing) should not make adifference (Loop Lemma)
- Two computations are causal equivalent if they are equal up to the transformations above

Causal consistency theorem

- Causal equivalent computations should
  - Lead to the same state
  - Produce the same history information
- Computations which are not causal equivalent
  - Should not lead to the same state
  - Otherwise one would wrongly reverse them in the same way
  - If in a non reversible setting they would lead to the same state, we should add history information to distinguish the states

Example

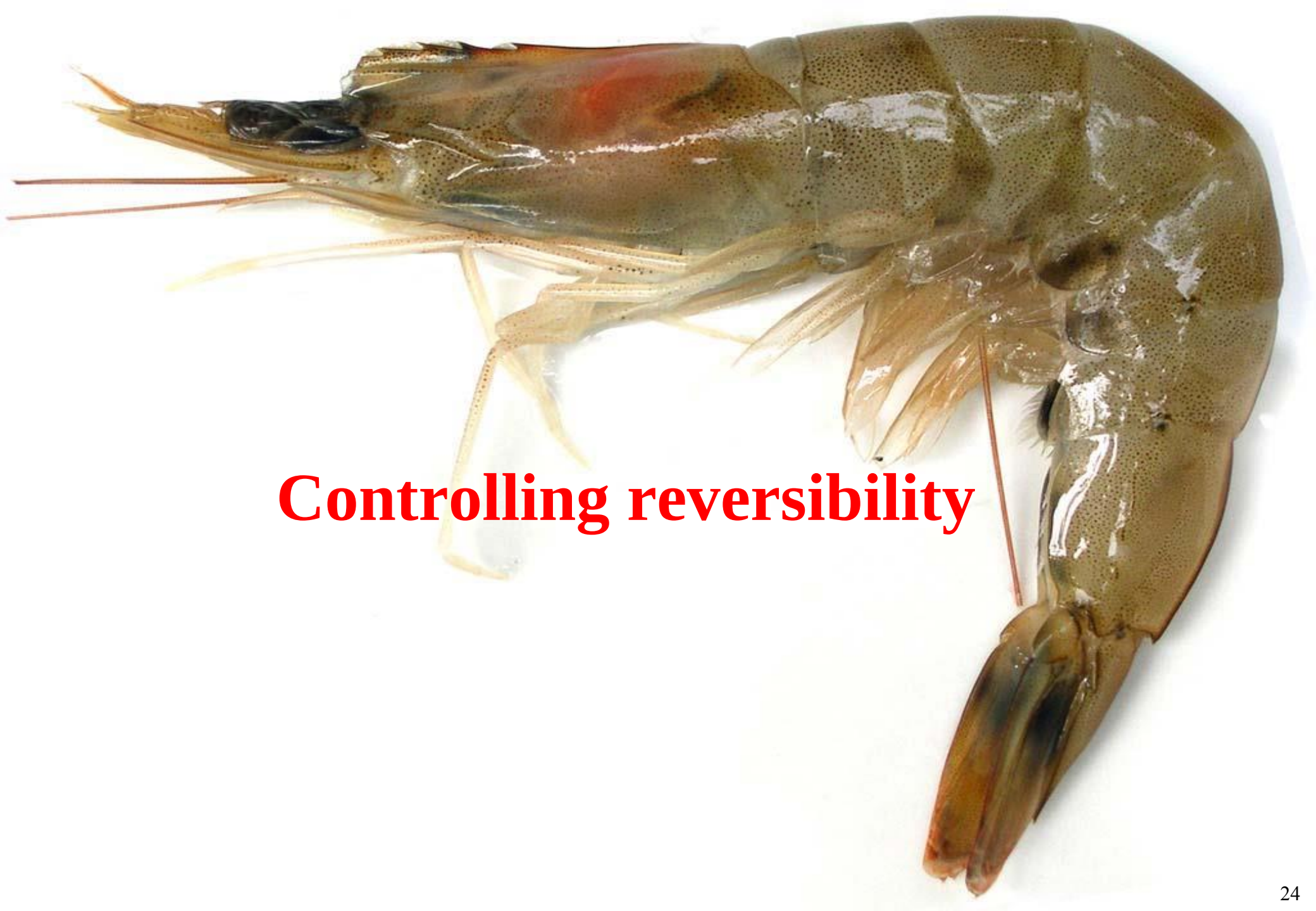
- If x>5 then y=2 else y=7 endif;y=0
- Two possible computations, leading to the same state
- From the causal consistency theorem we know that we need history information to distinguish them
  - At least we should trace the chosen branch
- The amount of information to be stored in the worst case is linear in the number of steps
  [Lienhardt, Lanese, Mezzina, Stefani: A Reversible Abstract Machine and Its Space Overhead. FMOODS/FORTE 2012]

Many reversible calculi

- Reversible variants of many calculi have been studied
  - CCS: Danos & Krivine [CONCUR 2004]
  - CCS-like calculi: Phillips & Ulidowski [FoSSaCS 2006, JLAP 2007]
  - HOπ: Lanese, Mezzina & Stefani [CONCUR 2010]
  - μOz: Lienhardt, Lanese, Mezzina & Stefani [FMOODS&FORTE 2012]
  - π-calculus: Cristescu, Krivine, Varacca [LICS 2013, see Daniele's talk]
  - Klaim: Giachino, Lanese, Mezzina, Tiezzi [PDP 2015]
- All applying the ideas we discussed
- With different technical solutions

This is just uncontrolled reversibility

- The works above describe how to go back and forward, but not when to go back and when to go forward
- Non-deterministic is not enough
  - The program may go back and forward between the same states forever
  - If a good state is reached, the program may go back and lose the computed result
- We need some form of control for reversibility
  - Different possible ways to do it
  - Which one is better depends on the intended application



Controlling reversibility

A taxonomy for reversibility control

- Categorization according to who controls the reversibility
- Three different possibilities
  - Internal control: reversibility is controlled by the programmer
  - External control: reversibility is controlled by the environment
  - Semantic control: reversibility control is embedded in the semantics of the language

Internal control

- Reversibility is controlled by the programmer
- Explicit operators to specify whether to go backward and whether to go forward
- A few possibilities have been explored
  - Irreversible actions
    [Danos, Krivine: Transactions in RCCS. CONCUR 2005]
  - **Roll** operator
    [Lanese, Mezzina, Schmitt, Stefani: Controlling Reversibility in Higher-Order Pi. CONCUR 2011]

Irreversible actions

- Execution is non-deterministically backward or forward
- Some actions, once done, cannot be undone
  - This allows to make a computed result permanent
  - They are a form of commit
- Still most programs are divergent
- Suitable to model biological systems
  - Most reactions are reversible
  - Some are not

Roll operator

- Normal execution is forward
- Backward computations are explicitly required using a dedicated command
- **Roll** γ, where γ is a reference to a past action
  - Undoes action pointed by γ, and all its consequences
  - Go back n steps not meaningful in a concurrent setting
- γ is a form of checkpoint
- This allows to make a computed result permanent
  - If there is no **roll** pointing back past a given action, then the action is never undone
- Still most programs are divergent

External control

- Reversibility is controlled by something outside the program
- Again a few possibilities have been explored
  - Controller processes
    [Phillips, Ulidowski, Yuen: A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway. RC 2012]
  - Causal-consistent reversible debugger
    [Giachino, Lanese, Mezzina: Causal-Consistent Reversible Debugging. FASE 2014]

Controller processes

- Two layered system
- A reversible slave process and a forward master process
- The slave process may execute only
  - Actions allowed by the master
  - In the direction allowed by the master
- Used to model biological systems
- Allows for non causal-consistent reversibility

Reversible debugger

- The user controls the direction of execution via the debugger commands
- In standard debuggers: step, run, ...
- A reversible debugger also provides the command "step back"
  - In a concurrent setting one should specify which thread should step back (or forward)
  - Some threads may be blocked and unable to step back (or forward)
- Reversible debuggers exist (e.g, gdb, UndoDB[see Greg's talk])

Causal-consistent reversible debugger

- We exploit the causal information to help debugging concurrent applications
- We provide a debugger command like the **roll**
- Undo a given past action and all its consequences
- Different possible interfaces for **roll**
  - The last assignment to a given variable
  - The last send to a given channel
  - The last read from a given channel
  - The creation of a given thread
- http://www.cs.unibo.it/caredeb/index.html

Semantic control

- Reversibility policy embedded in the language
- Again a few possibilities have been explored
  - Prolog
  - State-space exploration via heuristics
  - Energy-based control
    [Bacci, Danos, Kammar: On the Statistical Thermodynamics of Reversible Communicating Processes. CALCO 2011]

Prolog backtracking

- Prolog tries to satisfy a given goal
- It explores deep-first the possible solutions
- When it reaches a dead end, it rollbacks and tries a different path
- The search is normally quite efficient
- Relies on the programmer expertise to avoid divergence

State-space exploration via heuristics

- In general, there are different ways to explore a state space looking for a solution
- Strategy normally composed by a standard algorithm plus some heuristics driving it
- As before, if the algorithm reaches a dead end, it rollbacks and tries a different path
- Sample algorithm
  - Count how many times each action has been done and undone
  - Choose paths which have been tried less times
  - Use heuristics in case of ties

Energy-based control

- Assumes a world with a given amount of energy
- Forward and backward steps are taken subject to some probability
- The rates depend on the available amount of energy
- Under suitable conditions on the energy a computation is guaranteed to commit in finite average time

Back to our **roll**

- Our application field: programming reliable concurrent/distributed systems
- Normal computation should go forward
  - No backward computation without errors
- In case of error we should go back to a past state
  - We assume to be able to detect errors
- We should go to a state where the decision leading to the error has not been taken yet
  - The programmer should be able to find such a state

The kind of algorithm we want to write

- γ: take some choice

  ....

  if we reached a bad state

  roll γ

  else

  output the result
- The approach based on the **roll** operator is suitable to our aims
- Not necessarily the best in all the cases

Roll and loop

- With the **roll** approach
- We reach a bad state
- We go back to a past state
- We may choose again the same path
- We reach the same bad state again
- We go back again to the same past state
- We may choose again the same path
- …

Permanent and transient errors

- Going back to a past state forces us to forget everything we learned in the forward computation
  - We forget that a given path was not good
  - We may retry again and again the same path
- The approach is fine for transient errors
  - Errors that may disappear by retrying
  - E.g., message loss on the Internet
- The approach is less suited for permanent errors
  - Errors that occur every time a state is reached
  - E.g., division by zero, null pointer exception
  - We can only hope to take a different branch in a choice

We should break the Loop Lemma

- In case of error we want to change path
  - Not possible with the **roll** alone
  - The programmer cannot avoid to take the same path again and again
- We need to remember something from the past try
  - Not allowed by the Loop Lemma

Specifying alternatives

Alternatives

- [Lanese, Lienhardt, Mezzina, Schmitt, Stefani: Concurrent Flexible Reversibility. ESOP 2013]
- The programmer may declare different ordered alternatives to solve a problem
- The first time the first alternative is chosen
- Undoing the choice causes the selection of the next alternative
  - Like in Prolog
  - We rely on the programmer for a good definition and ordering of alternatives

Specifying alternatives

- Actions A%B
- Normally, A%B behaves like A
- If A%B is the target of a **roll**, it becomes B
- Intuitive meaning: try A, then try B
- B may have alternatives too

Programming with alternatives

- We should find the actions that may lead to bad states
- We should replace them with actions with alternatives
- We need to find suitable alternatives
  - Retry
  - Retry with different resources
  - Give up and notify the user
  - Trace the outcome to drive future choices

Example

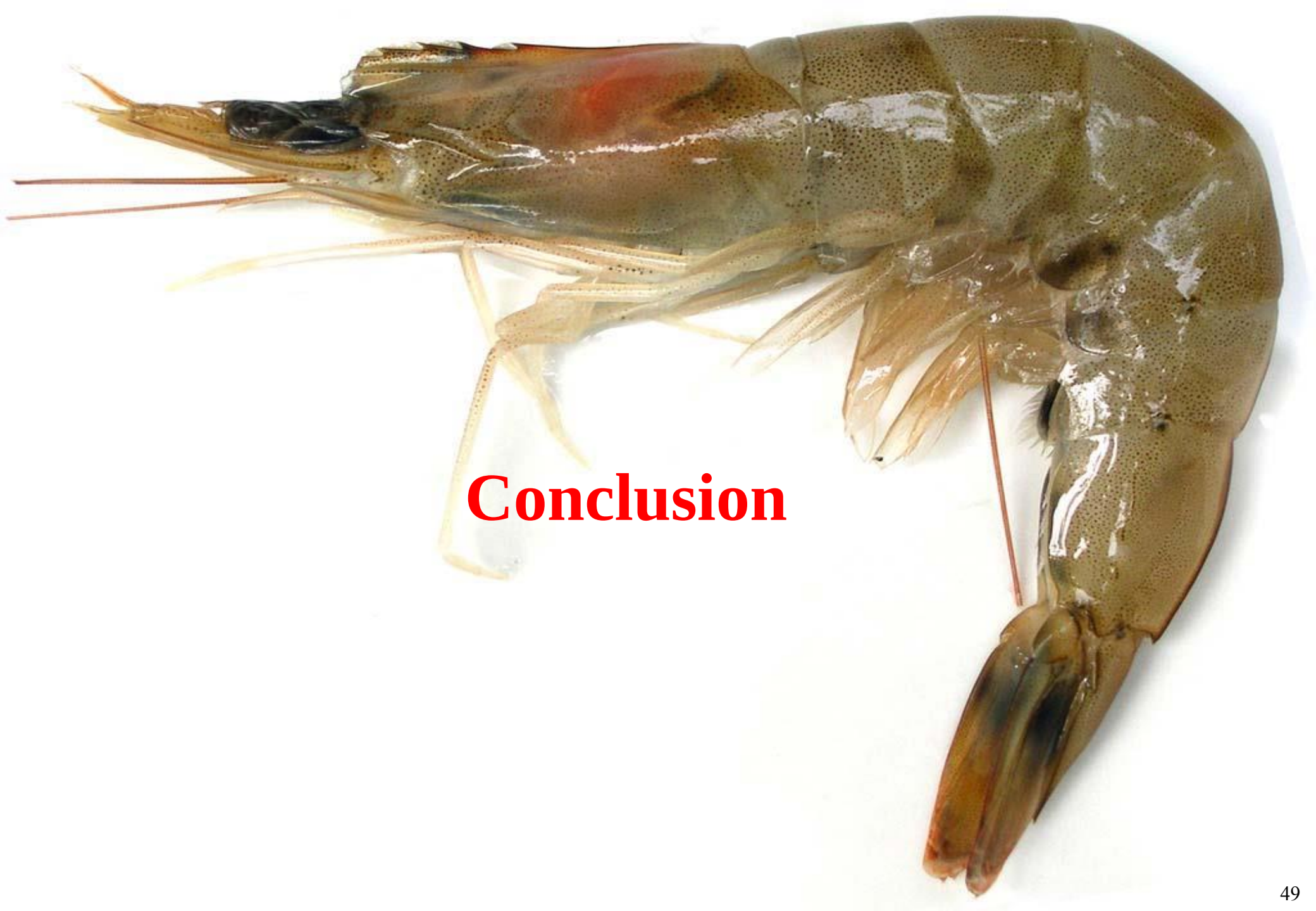
- Try to book a flight to Grenoble with Airfrance
- A Airfrance website error makes the booking fail
  - Retry: try again to book with Airfrance
  - Retry with different resources: try to book with Alitalia
  - Give up and notify the user: no possible booking, sorry
  - Trace the outcome to drive future choices: remember that Airfrance web site is prone to failure, next time try a different company first

Application: Communicating transactions

- [de Vries, Koutavas, Hennessy: Communicating Transactions. CONCUR 2010]
- Transactions that may communicate with the environment and with other transactions while computing
- In case of abort one has to undo all the effects on the environment and on other transactions
  - To ensure atomicity

Communicating transactions via reversibility

- We can encode communicating transactions
  - We label the start of the transaction with γ
  - An abort is a **roll** γ
  - The **roll** γ undoes all the effects of the transaction
  - A commit simply disables the **roll** γ
- The mapping is simple, the resulting code quite complex
  - We also need all the technical machinery for reversibility
- The encoding is more precise than the original semantics
  - We avoid some useless undo
  - Since our treatment of causality is more refined



Conclusion

Summary

- Uncontrolled reversibility for concurrent systems
- Mechanisms for controlling reversibility
- How to avoid looping using alternatives

Future work: language

- Can we make mainstram concurrent languages reversible?
  - Concurrent ML, Erlang, Java, ...
  - How to deal with data structures, modularity, type systems, ...

Future work: reasoning

- How to reason on reversible programs?
- Initial works based on
  - Behavioural equivalences
    [Krivine: A Verification Technique for Reversible Process Algebra. RC 2012]
    [Aubert, Cristescu: Reversible Barbed Congruence on Configuration Structures. ICE 2015]
  - Logic [Abadi: The Prophecy of Undo. FASE 2015]
  - Session types
    [Barbanera, Dezani-Ciancaglini, de'Liguoro: Compliance for reversible client/server interactions. BEAT 2014]
    [Barbanera, Dezani-Ciancaglini, Lanese, De’Liguoro: Retractable contracts. PLACES 2015]

Future work: applications

- Can we find some killer applications?
  - Software transactional memories
  - Existing algorithms for distributed checkpointing
  - Debugging

Questions

- Do you use sequential reversibility, causal-consistent reversibility, or something different?
- Do you think that causal-consistent reversibility is meaningful in your setting?
- Do you need techniques to control reversibility?
- Which ones do you use?

Finally