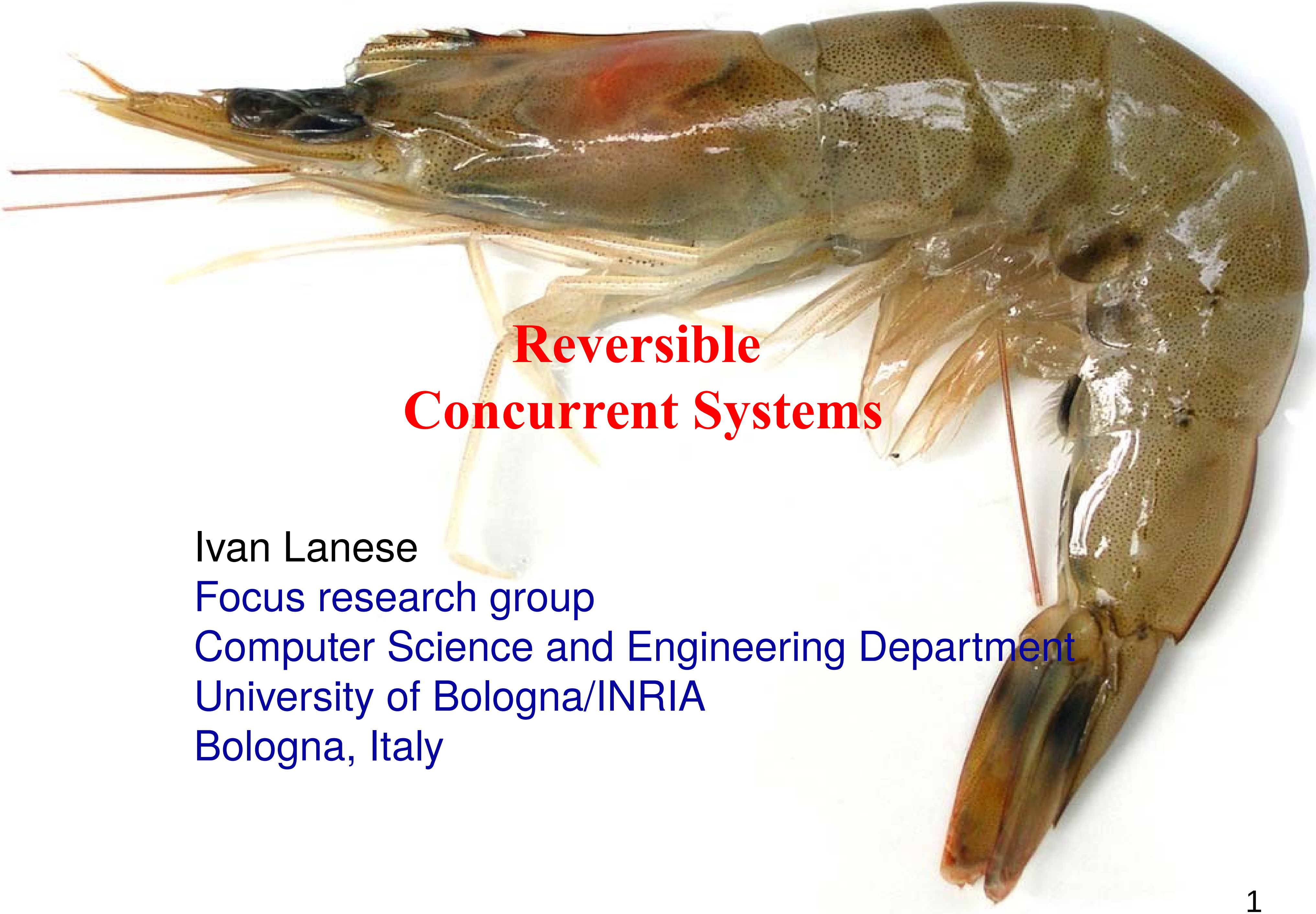


Reversible Concurrent Systems

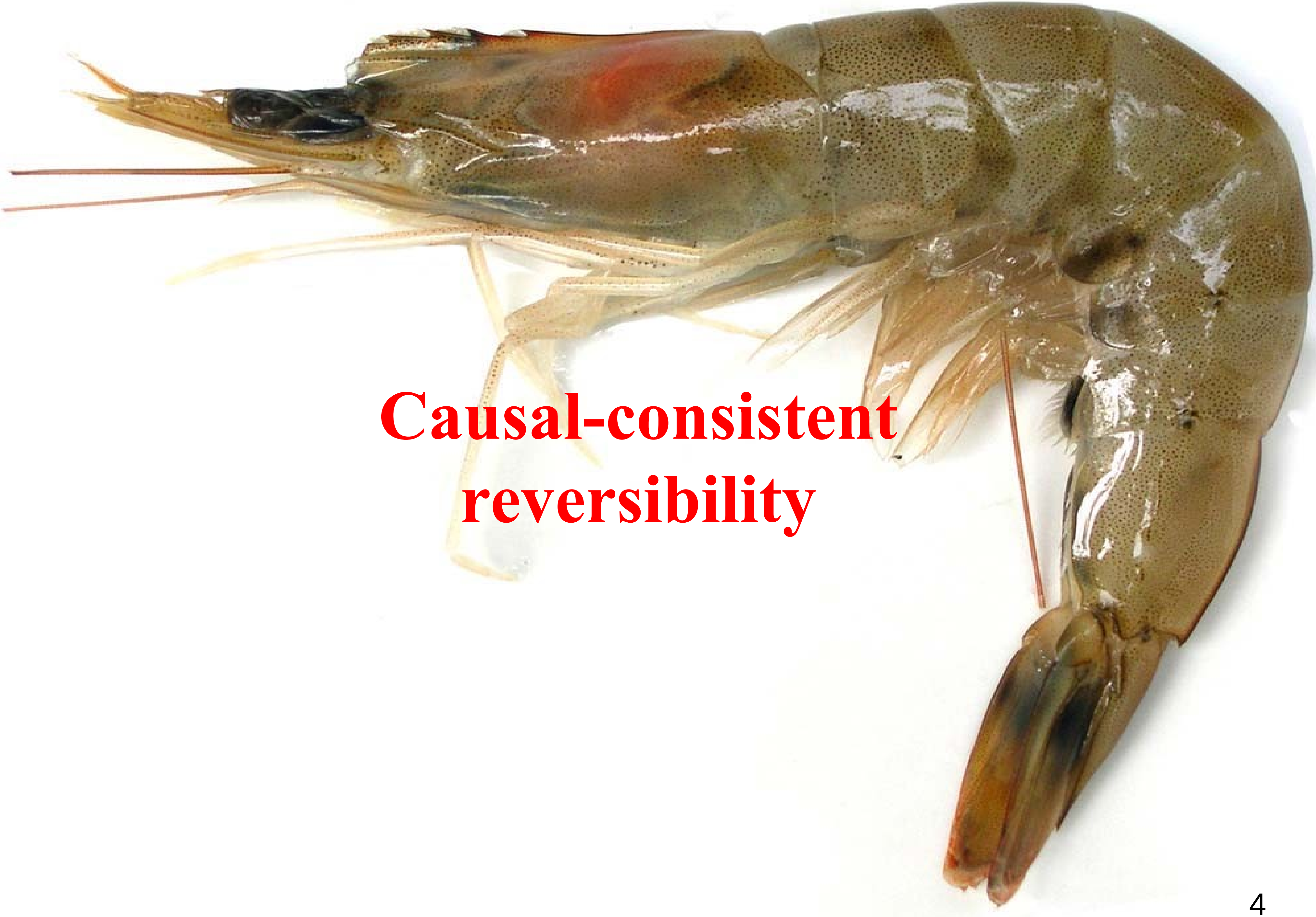
Ivan Lanese
Focus research group
Computer Science and Engineering Department
University of Bologna/INRIA
Bologna, Italy

Contributors

- Elena Giachino (University of Bologna/INRIA, Italy)
- Michael Lienhardt (University of Turin, Italy)
- Claudio Antares Mezzina (IMT Lucca, Italy)
- Jean-Bernard Stefani (INRIA, France)
- Alan Schmitt (INRIA, France)

Map of the talk

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



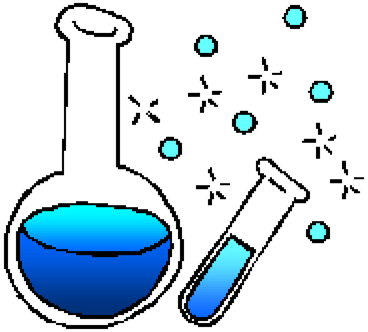
Causal-consistent reversibility

What is reversibility?

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

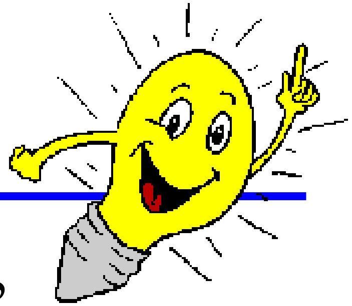


- Reversibility widespread in the world
  - Undo button in editors
  - Backup, svn
  - Chemistry/biology
  - Quantum phenomena
  - Optimistic simulation
  - ...

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 line” 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
  - Similar to what is done in many approaches, such as transactions and checkpointing
- 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

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

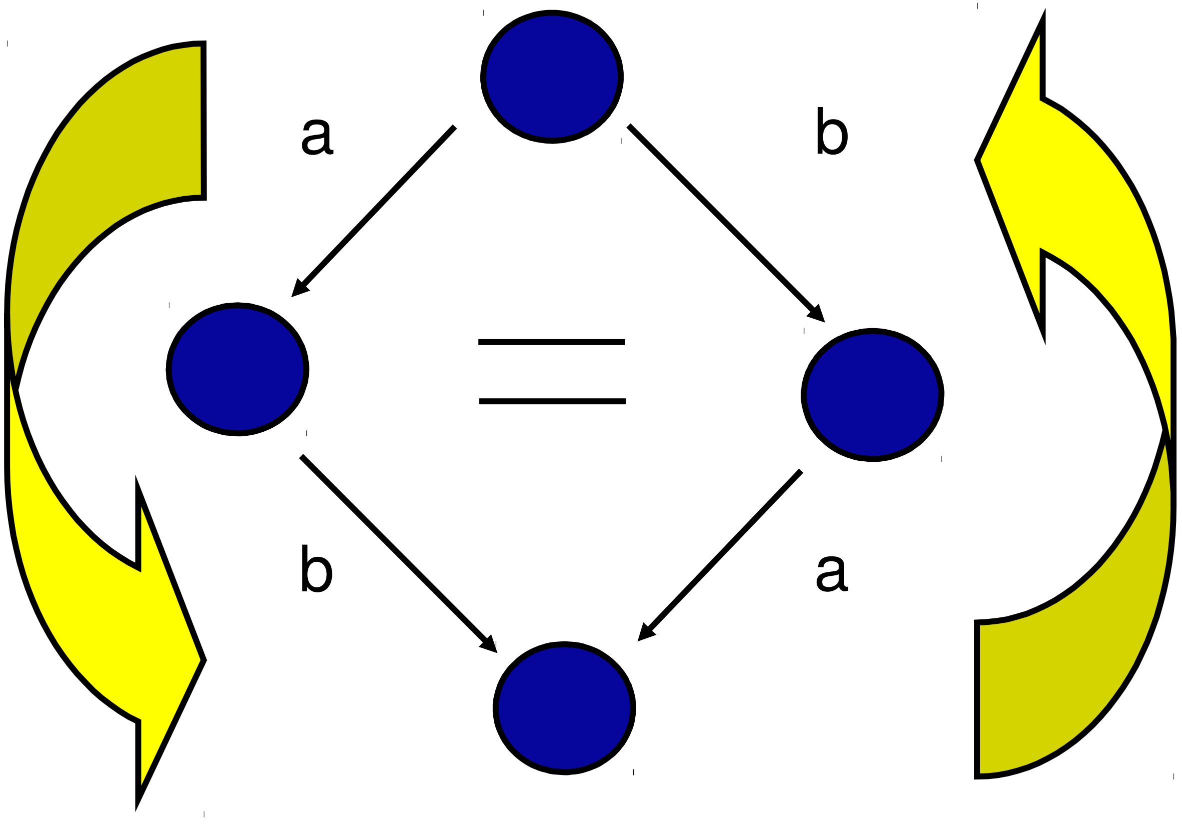
Different approaches to reversibility

- Saving a past state and redoing the same computation from there (checkpoint & replay)
- Undoing steps one by one
  - Restricting the language to commands which are naturally reversible
    - Cause no loss of information
  - Keeping the whole language (non 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 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 a difference (Loop Lemma)
- Two computations are causal equivalent if they are equal up to the transformations above

Causal consistency theorem

- Two computations from the same state should lead to the same state iff they are causal equivalent
- Causal equivalent computations
  - Produce the same history information
  - Can be undone in the same ways
- 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 differentiate 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

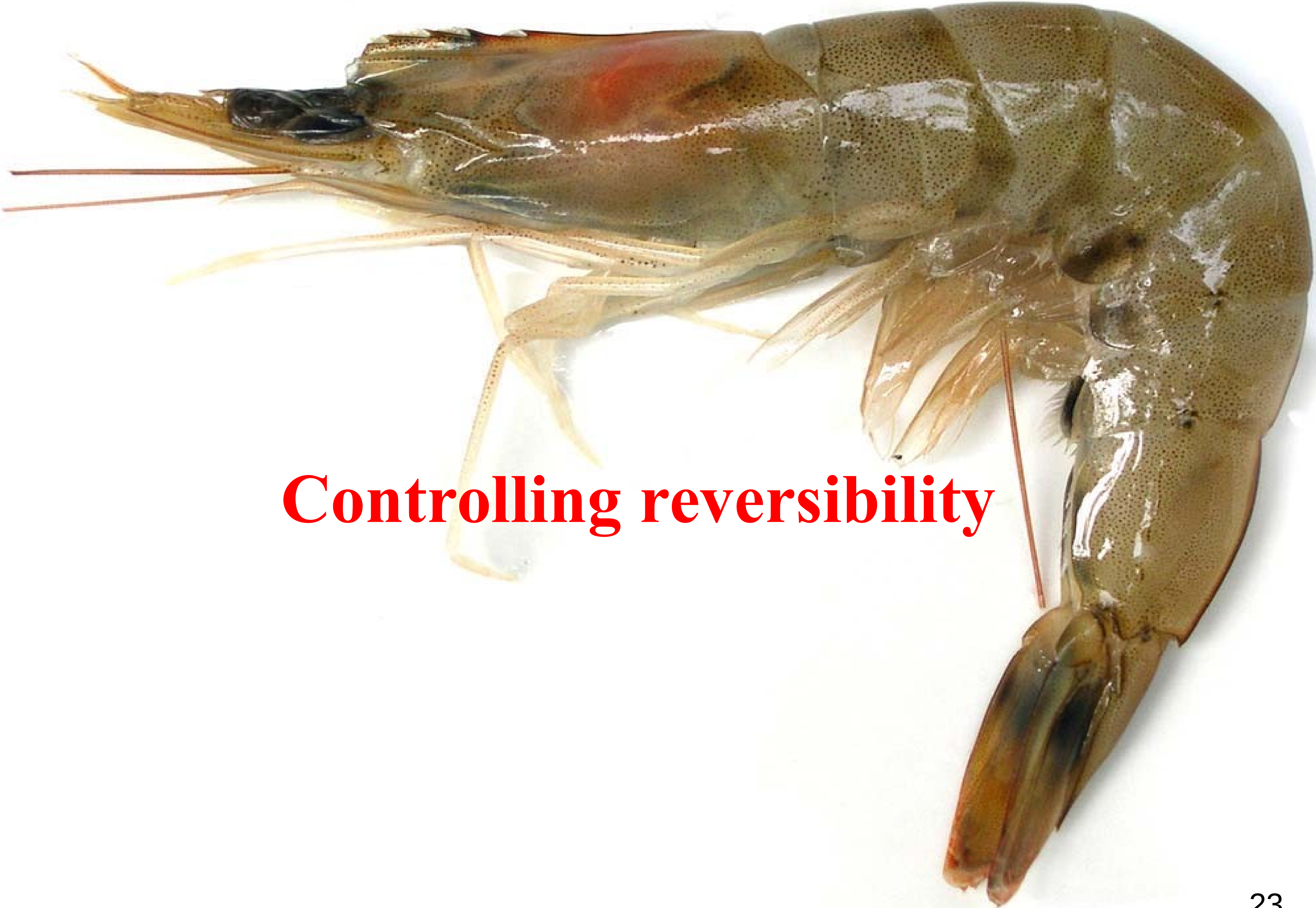
- Causal-consistent reversible extensions of many calculi have been defined and studied
  - CCS: Danos & Krivine [CONCUR 2004]
  - CCS-like calculi: Phillips & Ulidowski [FoSSaCS 2006, JLAP 2007]
  - HOπ: Lanese, Mezzina & Stefani [CONCUR 2010]
  - μOz: Lienhardt, Lanese, Mezzina & Stefani [FMOODS&FORTE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

Example

- In CCS:
  $\overline{a}.P+Q \mid a.P_1+Q_1 \rightarrow P \mid P_1$
- In (a) reversible CCS
  $k:\overline{a}.P+Q \mid k_1:a.P_1+Q_1 \leftrightarrow$
  $\nu k_1,k_2\ [a, k:Q, k_1:Q_1, k_2, k_3] \mid k_2:P \mid k_3:P_1$

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
  - We show one approach as example



Controlling reversibility

Do you remember our aim?

- 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

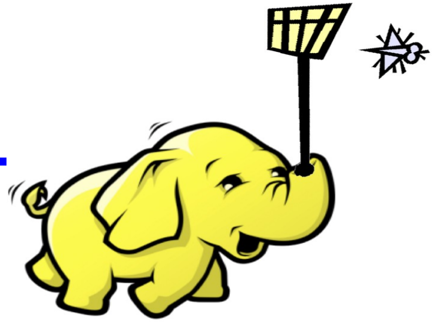
Roll operator

- 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
  - Undo the last n steps not meaningful in a concurrent setting
- γ is a form of checkpoint
- This allows one to make a computed result permanent
  - If there is no **roll** pointing back past a given action, then the action is never undone

The kind of algorithms we want to write

-
  ```
  $\gamma$: take some choice
  ....
  if we reached a bad state
      roll $\gamma$
  else
      output the result
  ```
- The **roll** operator is suitable for our aims
- Not necessarily the best in all the cases
- Most programs are divergent

Reversible debugger



- The user controls the direction of execution via the debugger commands
- In standard debuggers: step, run, ...
- A reversible debugger also provides commands such as “step back”
- Reversible debuggers for sequential programs exist (e.g, gdb, UndoDB)

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

Roll and loop

- Let us go back to **roll** as a programming construct
- 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 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

- 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

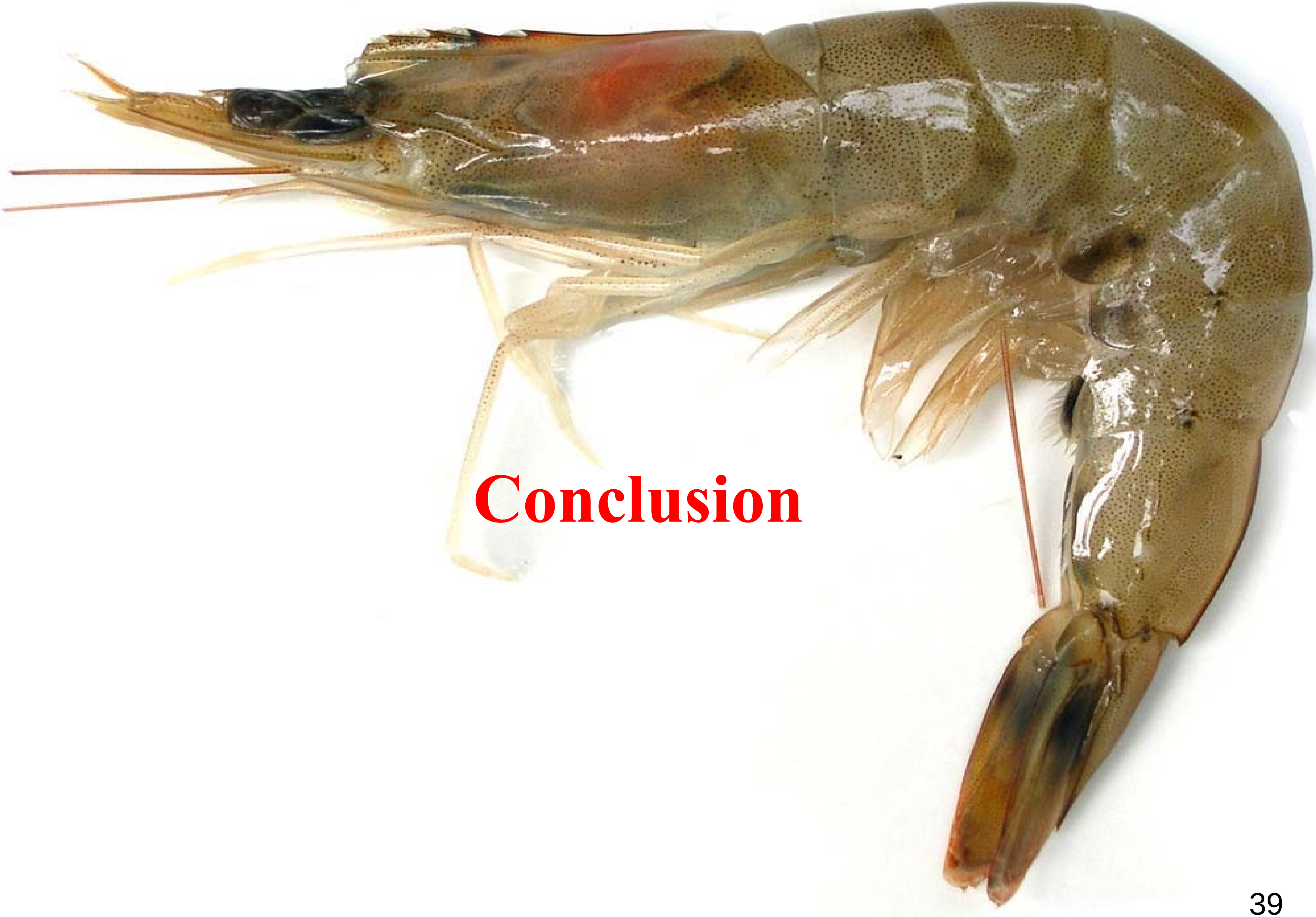
- Try to book a flight to Frankfurt with Lufthansa
- A Lufthansa website error makes the booking fail
  - Retry: try again to book with Lufthansa
  - Retry with different resources: try to book with Alitalia
  - Give up and notify the user: no possible booking, sorry
  - Trace the outcome to drive future choices: remember that Lufthansa 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
- A sample mechanism for controlling reversibility
- How to avoid looping using alternatives

Future work

- Can we make mainstram concurrent languages reversible?
  - Concurrent ML, Erlang, Java, ...
  - How to deal with data structures, modularity, type systems, …
  - First step: arbitrary sequential language + simple concurrency model
- Can we find some killer applications?
  - Software transactional memories
  - Existing algorithms for distributed checkpointing
  - Debugging

Finally

Thanks!

Questions?