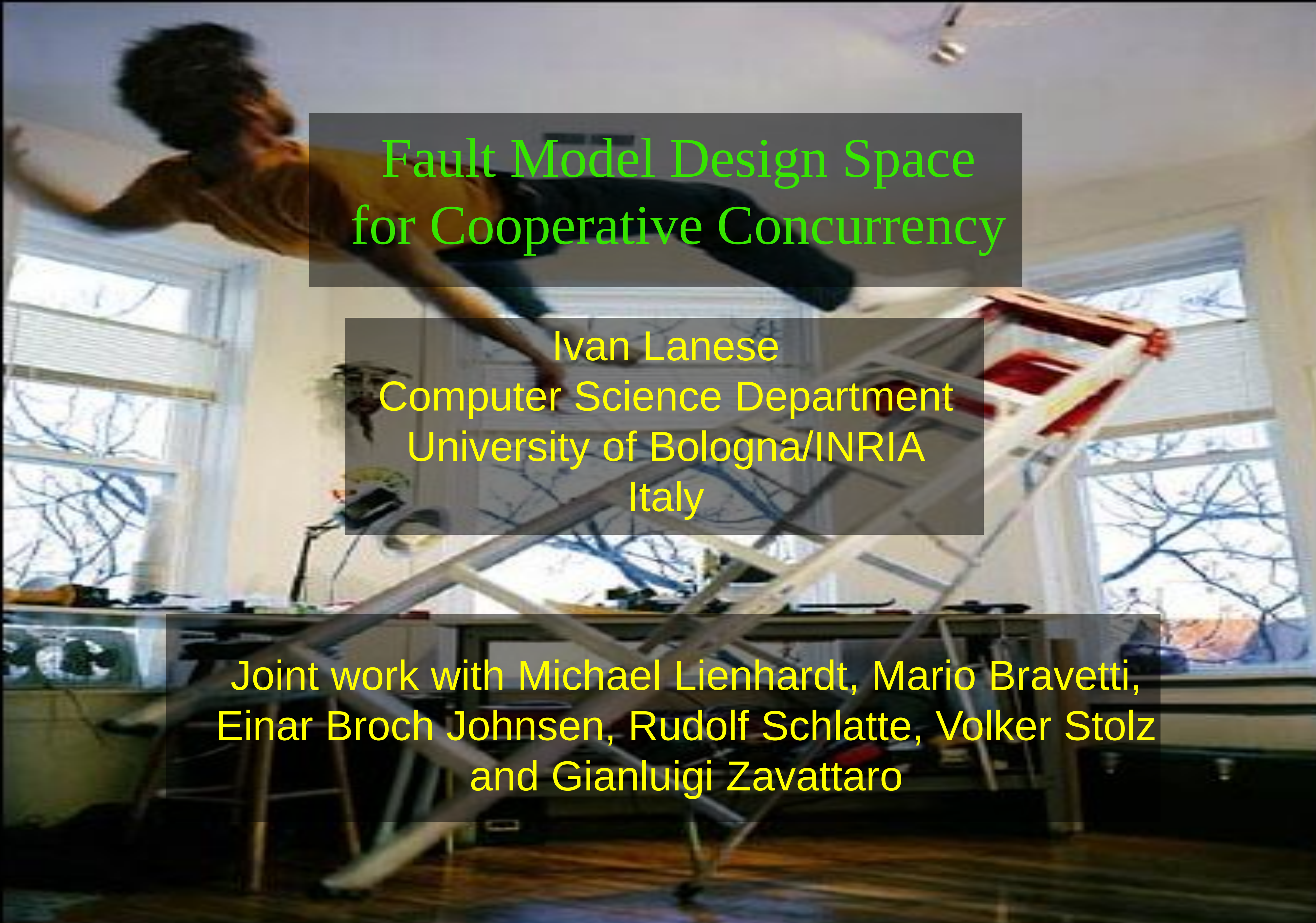


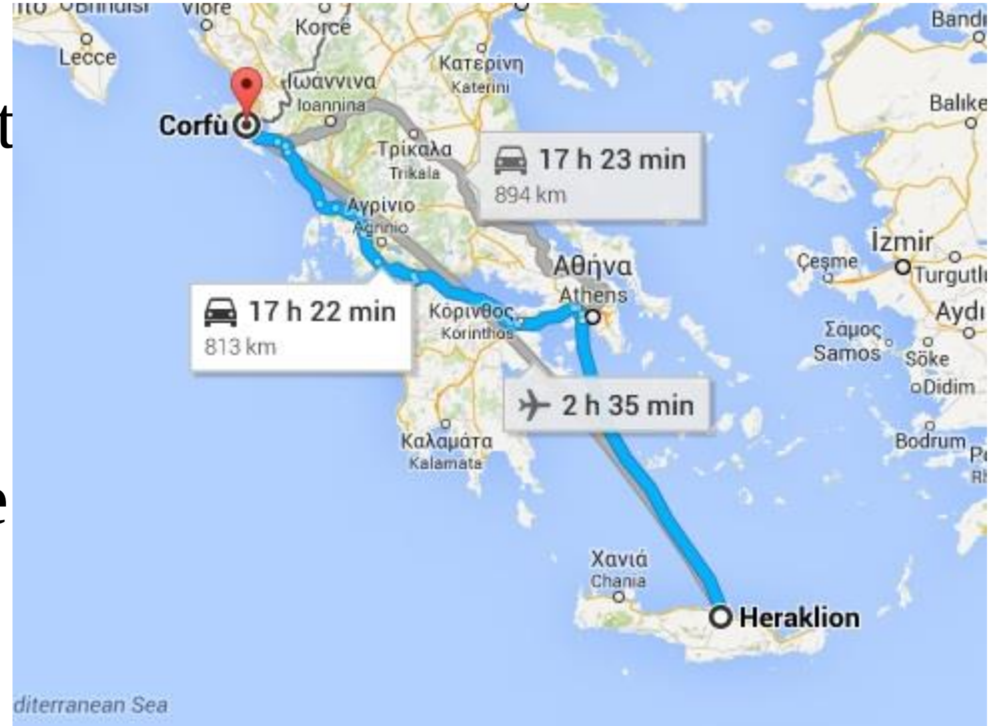
Fault Model Design Space for Cooperative Concurrency

Ivan Lanese
Computer Science Department
University of Bologna/INRIA
Italy

Joint work with Michael Lienhardt, Mario Bravetti, Einar Broch Johnsen, Rudolf Schlatte, Volker Stolz and Gianluigi Zavattaro

From one ISoLA to another

- At ISoLA 2010
  Which properties a good fault model should satisfy
- At ISoLA 2014
  The design space of fault models for the ABS language
- Some of the properties discussed in 2010 will turn out to be relevant



Roadmap

- Cooperative concurrency
- What a fault is?
- What a fault does?
- Where a fault goes?
- Conclusion

Roadmap

- Cooperative concurrency
- What a fault is?
- What a fault does?
- Where a fault goes?
- Conclusion

Our aim

- A fault model is a main component of a language design
- Refined fault models have been developed for mainstream languages
  - Such as Java, C++ or Haskell
- Can we say something more?
- Yes, since the fault model is related to the other constructs of the language!
- ABS uses cooperative concurrency, while Java, C++ and Haskell do not

ABS language

- A concurrent object-oriented language
- Chosen as specification language inside the HATS and ENVISAGE European projects
- Based on asynchronous method calls and futures
- Providing cooperative concurrency
- Equipped with a full formal semantics based on rewriting logic
- Suitable for static analysis

Futures

- A standard mechanism for allowing asynchronous method calls (cfr. java.util.concurrent)
- An asynchronous method call spawns a remote computation but the local computation can proceed
- The result of the remote computation is stored in a future
- The caller checks the future to get the result
  - It blocks only when it needs the result
- In ABS, futures are first-class entities

ABS basics

- s ::= … (standard o-o constructs)
  - f := o!m(p_1,…p_n) (asynchronous invocation)
  - x := f.**get** (read future)
  - **await** g **do** {s} (await)
  - **suspend** (processor release)

- g is a guard including:
  - Checks for future availability: ?f
  - Boolean expressions

Cooperative concurrency and invariants

- ABS suitable for reasoning based on invariants on the state of objects
- Invariants should hold at all points where the processor may be released
  - **suspend** and **await**
- No interference is possible at other program points
- Verification of concurrent programs using sequential reasoning

Aspects of faults

- We will consider 3 aspects of faults
- What a fault is?
  - How is it represented inside the language?
- What a fault does?
  - How is it generated?
  - How is it managed?
  - What happens if it is not?
- Where a fault goes?
  - How does a fault propagate?
  - Who is notified about it?

Roadmap

- Cooperative concurrency
- What a fault is?
- What a fault does?
- Where a fault goes?
- Conclusion

What a fault is?

- Exceptions are the language concept corresponding to faults
- ABS features two concepts that may be suitable to represent exceptions
  - Objects: have both a mutable state and a behavior
  - Datatypes: represent simple structures such as lists and sets
- Exceptions need only to be generated and consumed
- This pushes towards datatypes
  - To preserve simplicity of language use and analisys

Is exception a datatype?

- Programmers need two kinds of exceptions:
  - system-defined: Division by Zero, Array out of Bounds
  - user-defined: ...
- Programmers of modules need to define local exceptions
- There is the need for an open datatype
  - not available in ABS

Introducing open datatypes

- Introducing open datatypes in ABS
  - major change of a main feature of ABS
  - in contrast with the fact that ABS has a nominal type system
- Allowing any datatype to be an exception
  - new exceptions can be added by defining new datatypes
  - to understand an exception, a case on the type and on the constructor needs to be performed
  - information on types should be kept at run-time
- Exceptions are the unique extensible datatype
  - the solution with minimal impact on ABS

Exception as unique extensible datatype

- Declaration:
  exception NullPointerException
  exception RemoteAccessException
  exception MyUserDefinedException(Int, String)
- Use:

```
try { ... }
catch (e)
   { NullPointerException => ...
     MyUserDefinedException(n,s) => ...
     e2 => ...
    }
```

Roadmap

- Cooperative concurrency
- What a fault is?
- What a fault does?
- Where a fault goes?
- Conclusion

What an exception does?

- Exceptions are managed using **try** ... **catch** ...
- Exceptions may be generated
  - By the **throw** e command
  - By normal commands, such as x:=y/0;
  - By errors in a remote communication
    - » Asynchronous method invocation
- Where are exceptions due to asynchonous communication raised?
  - Not by the asynchronous method invocation
  - By the **get** on the corresponding future
  - Possibly by the **await**

What an unmanaged exception does?

- An unmanaged exception kills the current process, releasing the lock
- To enable proofs by invariants, the invariant should hold whenever an exception may be raised
  - Not easy
  - Rollback may be an option
- If the invariant cannot be guarenteed, the whole object needs to be killed
- If invariants would involve more than one object, all of them would need to be killed

Exception properties

- Not all exceptions need to kill the object if unmanaged
- Only the ones for which it is not possible to guarentee the invariant
- Can be specified by a property **deadly**
- Exceptions may also have other kinds of properties
  - E.g., **catchable** or not

Where are exception properties declared?

- In the exception declaration:
  deadly exception ArrayOutOfBound
  – The exception will behave the same in all contexts
- In the construct raising it: **throw deadly** ArrayOutOfBound
  – Also, x:=a[i] **deadly**
  – Behavior of exception not visible in the declarations
- In the method: Int calc(Int x) **deadly** ArrayOutOfBound
  – Enough to look at method declaration
  – More compositional
  – Not suitable for properties relevant inside the method (e.g., **catchable**)

Roadmap

- Cooperative concurrency
- What a fault is?
- What a fault does?
- Where a fault goes?
- Conclusion

Where an exception goes?

- Information about raised exceptions should be propagated
  - Other methods/objects may be influenced by the exception
  - In particular, for uncaught exceptions
- Possible targets are:
  - Methods using the result of the computation
  - Methods invoked by the failed one
  - Other methods in the same object
  - Methods trying to access the object after it died

Propagation through futures

- In synchronous method calls, only the caller has direct access to the result of a computation
- With asynchronous method calls, all the methods receiving the future have access to the result
  - the caller may be one of them or not
  - may even terminate before
- The future is accessed via **await** and **get**
  - The **get** should raise the exception
    - There is no value in the future
  - Less clear for the **await**

Propagation through futures: concurrency

- Different processes can perform a **get** on the same future concurrently
- All of them should raise an exception
  - Avoids races
  - Ensures the behaviour of the future changes at most once

Propagation through method invocations

- When a method raises an exception, many invocations from it to other methods may be running or pending
  - because of asynchronicity
- Those invocations may become useless or even undesired because of the exception
- A mechanism to **cancel** them seems useful
- There are different possible timings:
  - if they have not started yet, the invocation can be removed
  - if they are running, an exception may be raised in them
  - if they have completed, they may be compensated

How to cancel method invocations?

- Programmed propagation
  - An explicit primitive f1 := f.**cancel**(e) is used
    - f is the future individuating the invocation
    - e is the exception to be thrown
    - f1 will contain the result of the cancellation
  - Suitable for caught exceptions
- Automatic propagation
  - The exception is sent to all the methods invoked whose future has not been checked yet
  - One may also want to consider futures received/passed as parameters
  - Suitable for uncaught exceptions

Propagation to processes in the same object

- If the invariant cannot be restored, killing the object may be too extreme
- The exception may be propagated to the next method invocations to be executed
  - they may be able to manage it
- Exception may be raised
  - at the beginning of a method invocation
  - when it resumes after an **await** or **suspend**
- A dedicated system exception should be used

Propagation through dead objects

- If an object is dead, the invocation of one of its methods should raise an exception
- Surely in case of **get**, not clear for **await**
- A dedicated system exception should be raised

Roadmap

- Cooperative concurrency
- What a fault is?
- What a fault does?
- Where a fault goes?
- Conclusion

Conclusion

- Designing a fault model for ABS involves many, non trivial choices
  - the best interplay between exceptions and **await** is not yet clear
- Asynchronous method calls, futures, cooperative concurrency and invariants have an impact on the choices
  - the choices good for Java may not be good for ABS

What about ISoLA 2010 properties?

- For theoretical models
  - Full specification: quite tricky to ensure without a formal semantics
  - Expressiveness, minimality: the main trade-off we considered
  - Intuitiveness: we tried to address it
- For programming languages
  - Usability: more validation needed
  - Robustness: we provided constructs to deal with network failures
  - Compatibility: not addressed
- Most properties are relevant, mainly the ones for theoretical models

Future/related work

- Some of the alternatives we propose have never been fully explored
- A full exploration will require
  - the precise definition of their semantics
  - their introduction in the ABS implementation
- A few (complex) alternatives have been partially explored
  - Compensations [COORDINATION 2011]
  - Rollback [previous talk]

End of talk

Thanks!

Any question?