Causal-Consistent Reversible Debugging for Message Passing Programs

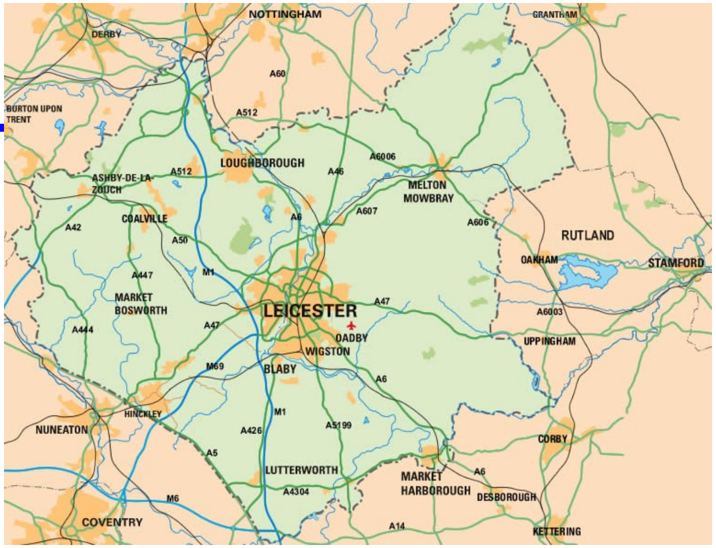
Ivan Lanese
Focus research group
Computer Science and Engineering Department
University of Bologna/INRIA
Bologna, Italy

Joint work with Adrian Palacios and German Vidal

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 778233

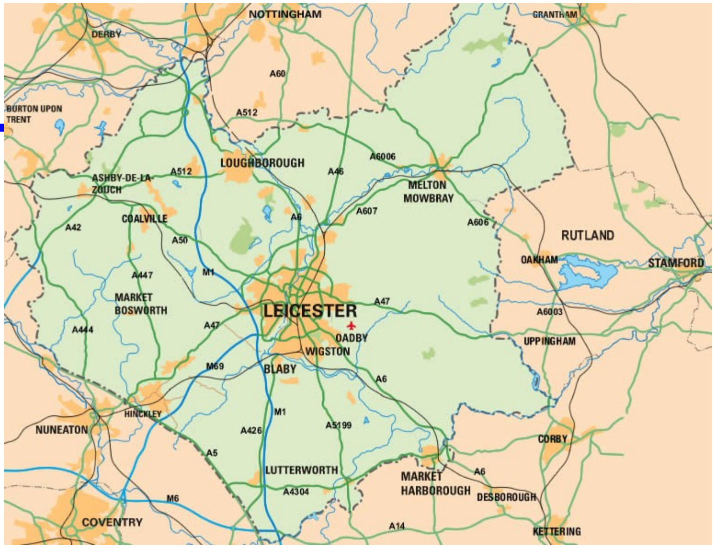
BEHAPI
Behavioural Application Program Interfaces

Roadmap



- Motivation
- Causal-consistent rollback and replay
- Suitability for debugging
- Demo

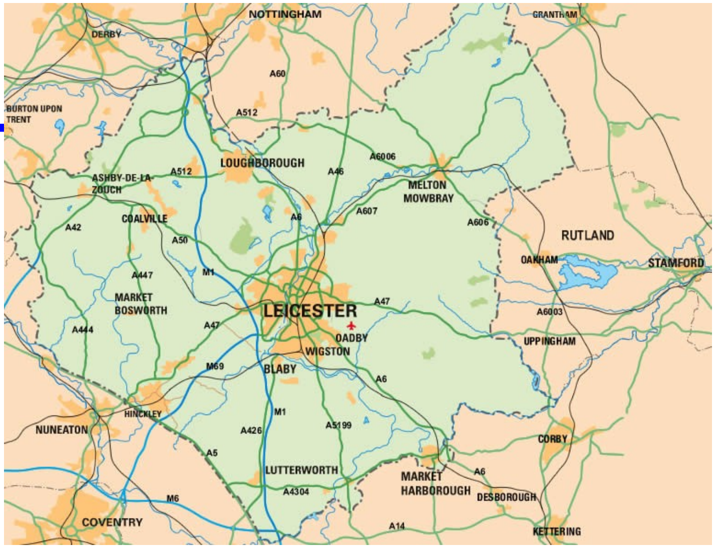
Roadmap



- Motivation
- Causal-consistent rollback and replay
- Suitability for debugging
- Demo

- … but this talk is also reversible, hence we will start from the demo and go backward

Roadmap



- Motivation
- Causal-consistent rollback and replay
- Suitability for debugging
- Demo

Case study

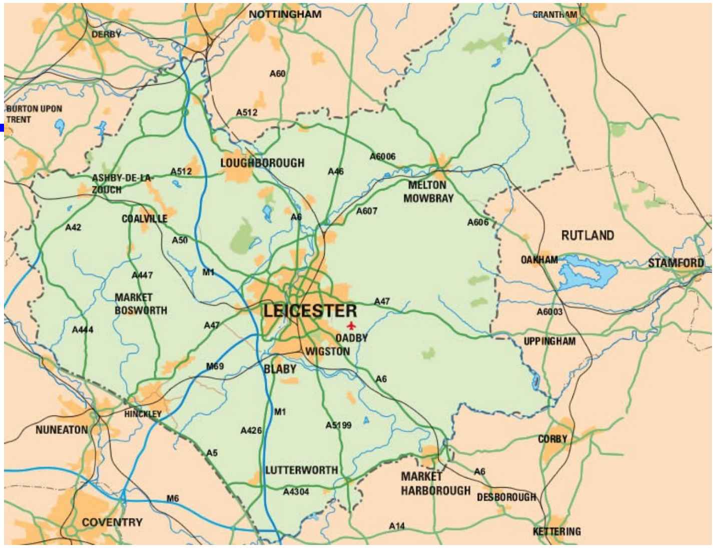
- A (very simple) online purchase system
- An item can be purchased if
  - the customer has enough credit
  - the address of the customer is correct
- The checks are performed by different processes, and another process computes short-circuit AND on them
- During perfective maintenance, a check for availability of the good in store is added
- Everything worked fine for a while, but at some point an item was sold to a client that had not enough credit
- What happened?

Demo

What we have seen

- We traced an execution in the standard Erlang execution environment
- We used the trace to replay it inside the debugger
- We explored it looking for the bug causing a visible misbehavior
  - We moved forward redoing events of interest, including all and only their causes
  - We moved back undoing events of interest, including all and only their consequences
- These are called causal-consistent replay and causal-consistent rollback

Roadmap



- Motivation
- Causal-consistent rollback and replay
- Suitability for debugging
- Demo

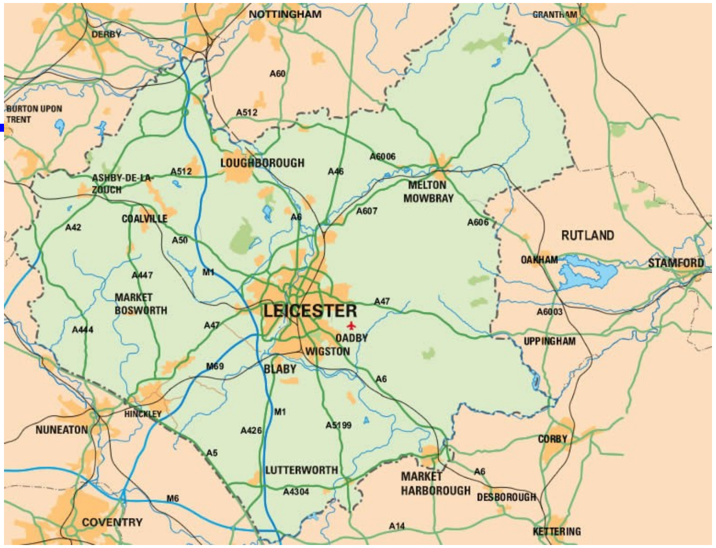
Suitability: correctness and completeness

- Replay is correct and complete
- A misbehavior occurs in the original computation if and only if it occurs in the replay
  - Actually, in all possible replays (provided we go till the end)

Suitability: minimality

- Replay and rollback are minimal
- Replaying an action A redoes the minimal amount of actions needed to reach a consistent state where A has been performed
  - A state is consistent if it can be reached in a forward computation
- Dual for rollback
- The user can concentrate on actions of interest

Roadmap



- Motivation
- Causal-consistent rollback and replay
- Suitability for debugging
- Demo

Causal-consistent replay/rollback

- These are the main commands provided by the debugger
- One can replay/rollback
  - n steps on a given process
  - the execution of a relevant (i.e., concurrent) action
    - message send
    - message receive
    - process spawn
  - (one can also rollback variable creations)

Causal-consistent rollback

- It allows one to undo a selected past action
- In the undo procedure any action can be undone, provided that its consequences (if any) are undone beforehand
- Concurrent actions can be undone in any order, but causal-dependent actions are undone in reverse order

Causal-consistent replay

- It allows one to redo a selected future action
- In the redo procedure any action can be redone, provided that its causes (if any) are redone beforehand
- Concurrent actions can be redone in any order, but causal-dependent actions are redone in original order

Causal-consistent replay

- It allows one to redo a selected future action
- In the redo procedure any action can be redone, provided that its causes (if any) are redone beforehand
- Concurrent actions can be redone in any order, but causal-dependent actions are redone in original order

- Rollback and replay are dual
- Note: definitions based on causality, not time
  - Work also if no unique notion of time exists
- Which traces can we get by using them?

Causal equivalence

- Two computations are causal equivalent if they differ only for
  - swaps of consecutive independent actions
  - introduction or removal of do A/undo A or undo A/redo A pairs
- The original computation and any computation obtained using causal-consistent rollback and replay are causal equivalent
  - provided that we replay till the end
- Two causal equivalent computations contain the same misbehaviors

How is logging performed?

- We built a tracer that instruments an Erlang program so to produce a log for each process
- We log only concurrency-related actions
- Unique identifiers are attached to messages to match sends with receives
- The log has the form

  {73,spawn,74} pid

  {73,send,5}

  {75,receive,7} unique message identifier

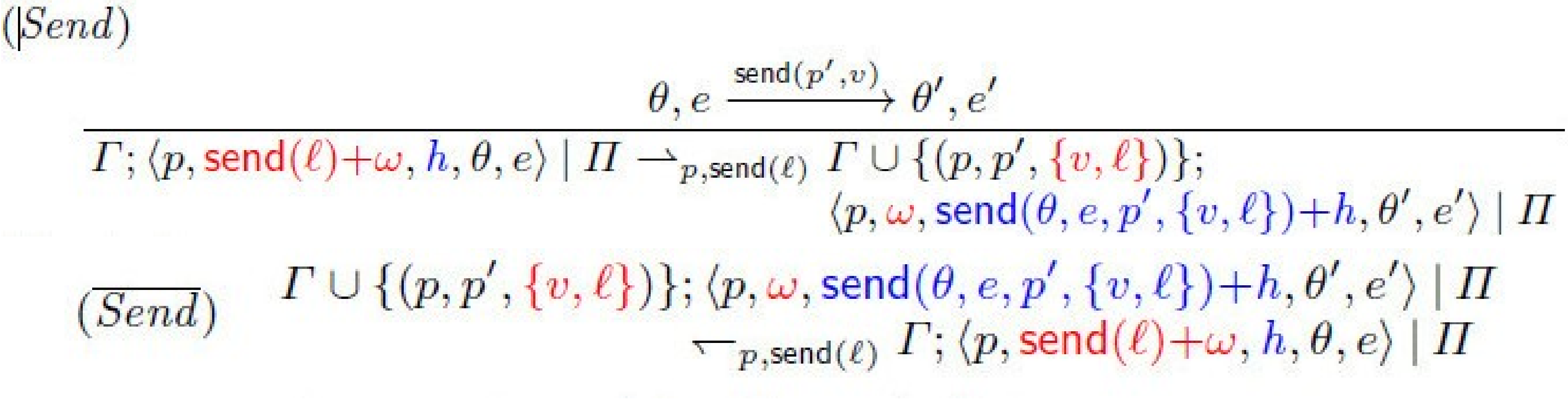
  …
- This is enough to replay a causal equivalent computation

Formal specification of replay and rollback

- Specification is needed to prove the properties we discussed before
- Both replay and rollback are specified in two steps
- Uncontrolled semantics: which forward/backward steps are legal at any given point, and how to execute them
  - extends the semantics of Core Erlang with logs (for replay) and histories (for backward execution)
  - ensures that causality and log are not violated
- Controlled semantics: which forward/backward steps are needed to perform a causal-consistent rollback/replay
  - explores the tree of causes/consequences

Uncontrolled semantics: structure

- The syntax of processes also includes their log and history
- Both forward and backward steps are possible
- The log is consumed going forward and recreated going backward, the history is consumed going backward and recreated going forward



$(|Send)$

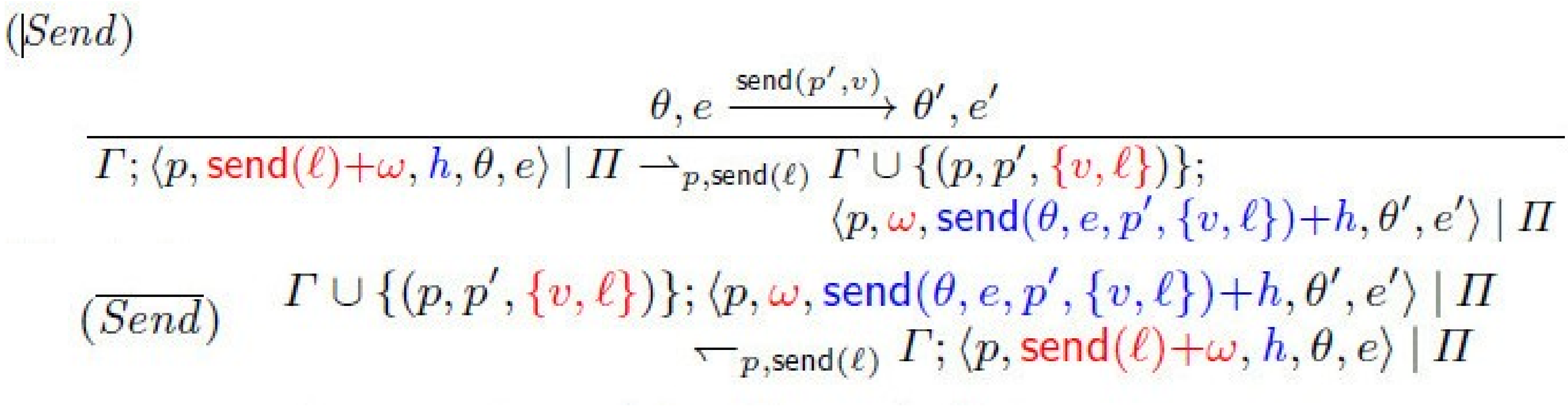
$$\frac{\theta, e \xrightarrow{\mathsf{send}(p',v)} \theta', e'}{\Gamma; \langle p, \mathsf{send}(\ell){+}\omega, h, \theta, e\rangle \mid \Pi \rightharpoonup_{p,\mathsf{send}(\ell)} \Gamma \cup \{(p,p',\{v,\ell\})\}; \langle p, \omega, \mathsf{send}(\theta,e,p',\{v,\ell\}){+}h, \theta', e'\rangle \mid \Pi}$$

$(\overline{Send})$

$$\Gamma \cup \{(p,p',\{v,\ell\})\}; \langle p, \omega, \mathsf{send}(\theta,e,p',\{v,\ell\}){+}h, \theta', e'\rangle \mid \Pi \leftharpoondown_{p,\mathsf{send}(\ell)} \Gamma; \langle p, \mathsf{send}(\ell){+}\omega, h, \theta, e\rangle \mid \Pi$$

Uncontrolled semantics: constraints

- Causality violations are avoided
  - Cannot undo a send before the corresponding receive
  - Cannot redo a receive before the corresponding send
- Only forward steps compatible with the log are allowed
  - A receive can only take the expected message



$(|\mathit{Send})$

$$\frac{\theta, e \xrightarrow{\mathsf{send}(p',v)} \theta', e'}{\Gamma; \langle p, \mathsf{send}(\ell)+\omega, h, \theta, e\rangle \mid \Pi \rightharpoonup_{p,\mathsf{send}(\ell)} \Gamma \cup \{(p,p',\{v,\ell\})\}; \langle p, \omega, \mathsf{send}(\theta,e,p',\{v,\ell\})+h, \theta', e'\rangle \mid \Pi}$$

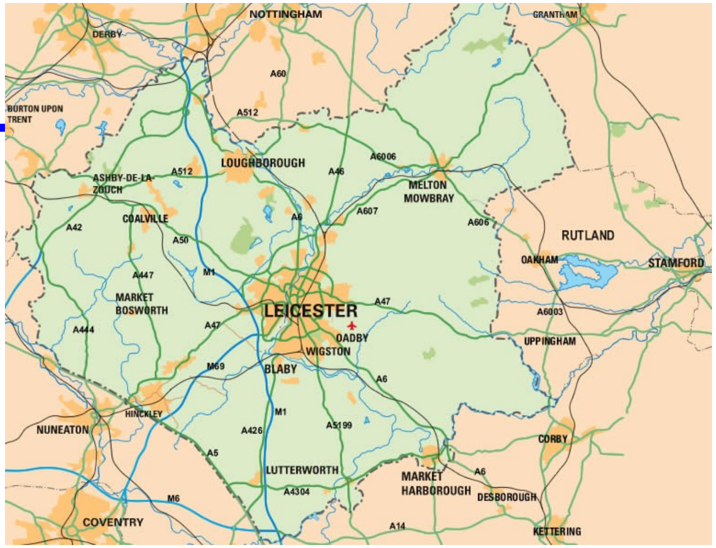
$(\overline{\mathit{Send}})$

$$\Gamma \cup \{(p,p',\{v,\ell\})\}; \langle p,\omega,\mathsf{send}(\theta,e,p',\{v,\ell\})+h,\theta',e'\rangle \mid \Pi \leftharpoondown_{p,\mathsf{send}(\ell)} \Gamma; \langle p, \mathsf{send}(\ell)+\omega, h,\theta,e\rangle \mid \Pi$$

Controlled semantics

- Rollback and replay are sequences of uncontrolled steps
- We use a recursive algorithm to select the steps
- We use the uncontrolled semantics to actually undo or redo them
- To rollback action A in process p
  - Start undoing actions in p
  - If A is undone then stop
  - If A cannot be undone due to a dependency on action A1 in process p1 then rollback A1 in p1, then continue undoing A
- Replay is analogous

Roadmap



- Motivation
- Causal-consistent rollback and replay
- Suitability for debugging
- Demo

Why do we want all this?

- Debugging concurrent/distributed systems is hard
- Misbehaviors may appear/disappear depending on the interleaving
- Re-executing a program may not reproduce the same misbehaviors
  - But replaying it using our approach does
- From a misbehavior it may be difficult to find the bug
  - May be in a different process
  - But it is a cause of the misbehavior
  - We can replay the misbehavior …
  - … and use rollback to reach the bug

Why not using breakpoints as usual?

- In standard debugging, one puts a breakpoint and executes forward from there
- If the breakpoint is too late, one should re-execute with an earlier breakpoint
  - Or better replay, to reproduce the misbehavior
  - In our approach, if we go too forward we can go backward again, and vice versa
- Going backward helps the programmer:
  - (S)he can follow causality links from the visible misbehavior towards the bug
  - Going forward there is no clue on what to execute
    - Which process? Up to where?

Future directions

- I could have put this slide at the beginning ...
- Support Erlang instead of Core Erlang
  - Not technically difficult, but time consuming
- Support a larger subset of the language
  - Distribution, constructs for fault tolerance, ...
- Improve efficiency
  - In particular, reducing the time overhead due to logging
  - Critical since logging needs to be done in production environment

Finally

Thanks!

Questions?