Choreographic Programming of Adaptive Applications

Ivan Lanese
Computer Science Department
University of Bologna/INRIA
Italy

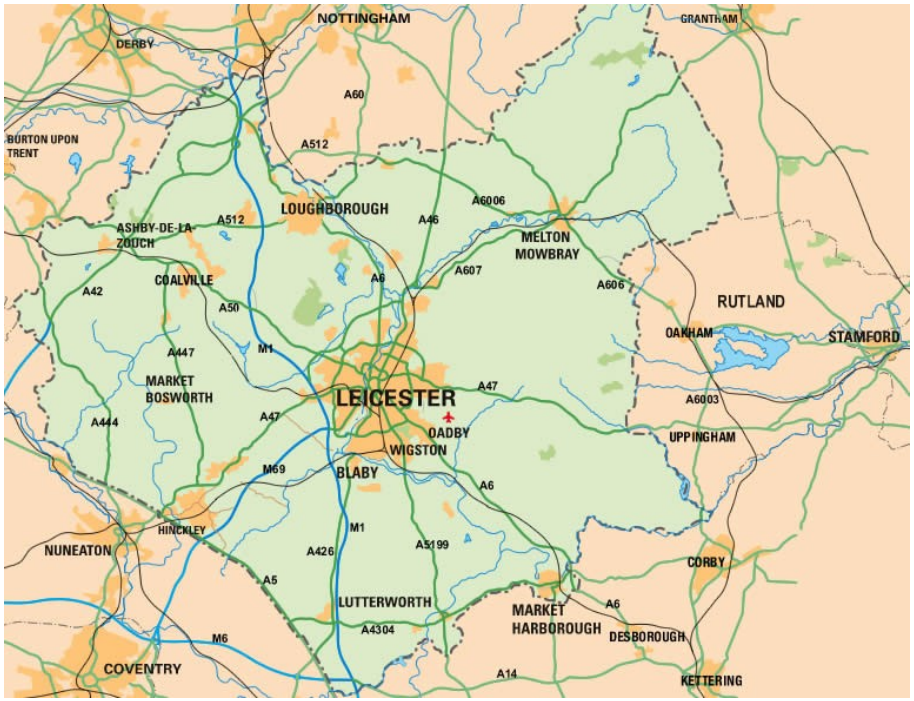
Joint work with Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo and Jacopo Mauro

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

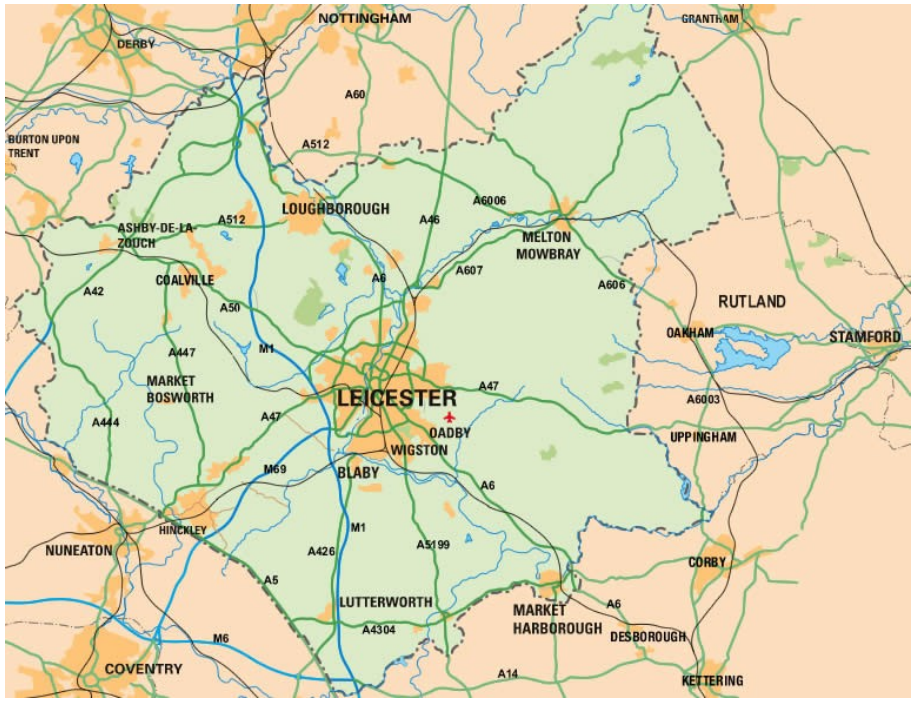
Map of the course

- Choreographic programming
- Choreographic programming in the real world
- Dynamic updates
- Dynamic updates in the real world
- Conclusion



Map of the course

- Choreographic programming
- Choreographic programming in the real world
- Dynamic updates
- Dynamic updates in the real world
- Conclusion



Programming distributed applications

- Distributed applications are composed by many nodes
- Each node executes a program, possibly different from the ones executing on the other nodes
- The nodes interact by exchanging messages
  - Main primitives: send and receive of a message
- The same in message-based concurrent applications
- Distributed applications are everywhere
  - Facebook, google, expedia, …
  - Banking applications, online shops, ...

Distributed programming: pitfalls

- Programming distributed applications is difficult and error-prone
- One has to reason about multiple threads of computation, and on their possible interleavings and interactions
- There can be specific errors due to concurrency
- Deadlocks: a process waits for a message that will never arrive
- Orphan messages: a message is sent to a target not expecting it
- Races: two messages are expected in a given order, but may arrive in a different one

Choreographic programming: aim

- Choreographic programming aims at avoiding these pitfalls
- Pitfalls due to a mismatch between send and receive
- Solution: send and receive cannot be written in isolation, but only composed inside an interaction
- An interaction is a communication between two participants:

$$\text{op: alice}(z+5) \rightarrow \text{bob}(x)$$

Choreographic programming: what it is?

- A programming paradigm for message-based concurrent/distributed applications
- Describing the whole distributed application as a unique program
- The term choreographic programming appeared first in: Fabrizio Montesi: Kickstarting Choreographic Programming. WS-FM 2015: 3-10
- A few approaches existed also before, e.g.: Ivan Lanese, Claudio Guidi, Fabrizio Montesi, Gianluigi Zavattaro: Bridging the Gap between Interaction- and Process-Oriented Choreographies. SEFM 2008: 323-332

Choreographic programming: basics

- A single interaction needs to be executed by two participants
  - Not just one as for standard programming language constructs
- This forces a single choreographic program to describe multiple participants
- A choreographic program is built by composing interactions using standard constructs: sequences, conditionals, loops,...
- Each construct may describe the behavior of one, two, or more participants

Relation with multiparty session types

- Choreographic programming and multiparty session types have commonalities and differences
  - Similar aim: avoid communication errors in message-passing programs
  - Similar idea: use interactions as building block
  - Different level: programming language vs type system
  - Similar technicalities
    - choreographic programs ≈ global types + data + conditions

Relation with multiparty session types (2)

- Choreographic programming and multiparty session types have different advantages and disadvantages

Choreographic programming	Multiparty session types
Used to write programs	Used to check programs
Requires a dedicated language	Can be applied to existing languages and programs
You only need to think about programs	You need to think about both programs and types

- Choreographic programming also allows for some more flexibility

Choreographic syntax

- C ::= o: s(e) → r(x) — interaction
  - | x@r = e — assignment
  - | skip — do nothing
  - | C ; C’ — sequence
  - | C | C’ — parallel
  - | if b@r {C} else {C’} — conditional
  - | while b@r {C} — loop

- Notation: o for operations, s,r for roles, e for expressions, b for Boolean expressions, x for variables
- The @ symbol specifies which role takes some action
- Key feature: a single construct may involve multiple roles

Execution model

- Each participant corresponds to an executable program
- Participants can be deployed in the same or in different machines
  - In our examples they are deployed in the same machine for simplicity
- Each participant has its own state (no global state)
- Interactions occur only by exchanging messages
- Communication is synchronous
  - Asynchronous communication is doable, but requires some more work

A sample choreographic program

product_name@client = getInput("Insert product name");

quote: client(product_name) → seller(sel_product);

price@seller = getInput("Quote " + sel_product)

Choreographic programming advantages

- Unique description of the whole distributed system ⇒
  Clear view of the global behavior
- The global description is a program ⇒
  Automatic compilation into a distributed application
- Sends and receives are paired into interactions ⇒
  No deadlocks, orphan messages nor races by construction

How to execute choreographic programs?

- A choreographic program describes the behavior of many participants
- We want to compile the choreographic program generating multiple executable programs, one for each participant
- When executed, the derived local programs should interact as specified in the choreographic program
  - Correctness of the compilation
  - No deadlocks, orphan messages nor races
- Compilation is quite tricky
- We give an high-level description of compilation as a projection function
  - cfr. endpoint projection in multiparty session types

The target language

- P ::= o: send e to r — message send
  - | o: receive x from r — message receive
  - | x = e — assignment
  - | skip — do nothing
  - | P ; P’ — sequence
  - | P | P’ — parallel
  - | if b {P} else {P’} — conditional
  - | while b {P} — loop

- A distributed program is composed by multiple named participants r_i each executing its own program P_i

Projection: basic idea

- Projection takes a choreographic program and a role r and produces the local program of r
- An interaction o: s(e) → r(x) is projected as
  - o: send e to r on s
  - o: receive x from s on r
  - skip on all the other participants
- Assignments x@r = e are executed by the role r
- Other constructs are projected homomorphically

- Very simple…
- …but it does not work

Projection problems: sequentiality

- Actions of different participants are independent

  $o_1: s_1(5) \rightarrow r_1(x)$; $o_2: s_2(7) \rightarrow r_2(y)$

- Interaction on o_2 should happen after interaction on o_1
  - No participant can force this
  - (Some approaches treat this as hidden parallel)
- No such problem in

  $o_1: s_1(5) \rightarrow r_1(x)$; $o_2: r_1(7) \rightarrow r_2(y)$

  $o_1: s_1(5) \rightarrow r_1(x)$; $o_2: s_1(7) \rightarrow r_2(y)$

  $o_1: s_1(5) \rightarrow r_1(x)$; $o_3: r_1(\text{“go”}) \rightarrow s_2(\text{aux})$; $o_2: s_2(7) \rightarrow r_2(y)$

Projection problems: choices (and loops)

- The behavior of a participant may depend on the decision of another participant
  if $b@s_1$ $\{o: s_2(5) \rightarrow r(x)\}$ else $\{o: s_2(7) \rightarrow r(x)\}$
- Participant s_2 should send 5 or 7 according to a local decision of s_1
- Similar problem in
  while $b@s_1$ $\{o: s_2(5) \rightarrow r(x)\}$
- s_2 and r do not know whether to loop or to exit, s_1 does not know when to re-evaluate the condition

Projection: 2 possible solutions

- Restriction approach: restrict to those cases where the projection works
  - In a sequence C;C’ the last participant in C should be the first participant in C’
  - In a conditional if b@r {C} else {C’} participant r should be the first participant in C and C’
- Smart projection approach: introduce auxiliary communications to force the desired behaviour
  - In a sequence C;C’, the participants in C notify the participants in C’ when C is finished and C’ can start
  - In if b@r {C} else {C’} participant r notifies participants in C and C’ of the outcome of the choice

Smart projection example

- The conditional
  if b@s_1 {o: s_2(5) → r(x)} else {o: s_2(7) → r(x)}
- Will be projected as follows
- On s_1: if b {cnd*: send true to s_2 | cnd*: send true to r}
  else {cnd*: send false to s_2 | cnd*: send false to r}
  On s_2: cnd*: receive z from s_1; if z {o: send 5 to r}
  else {o: send 7 to r}
  On r: cnd*: receive z from s_1; if z {o: receive x from s_2}
  else {o: receive x from s_2}
- Operations with * (like cnd*) are auxiliary operations

Restriction vs smart projection

	Restriction	Smart projection
Usability of the language	-	+
Simplicity of compilation (and proofs)	+	-
Efficiency	+	- (could be optimized)

- Smart projection could deal with other problems too (e.g., adding security guarantees)

Semantics

- In order to prove properties (e.g., correctness of compilation) we need to give semantics to choreographies and projected systems
- Both semantics are defined as labelled transition systems (LTS)
- Generated from operational semantics

Semantics of choreographies

- Quite simple semantics (the one in the paper has a few more technical additions)
- Sample rules:

$$\frac{[\![e]\!]_{\Sigma(s)} = v}{\langle \Sigma, o : s(e) \to r(x) \rangle \overset{o : s(v) \to r(x)}{\to} \langle \Sigma, x@r = v \rangle}$$

$$\frac{[\![e]\!]_{\Sigma(r)} = v}{\langle \Sigma, x@r = e \rangle \overset{\tau}{\to} \langle \Sigma[{}^{v}\!/_{x}, r], 1 \rangle}$$

$$\langle \Sigma, 1 \rangle \overset{\surd}{\to} \langle \Sigma, 0 \rangle$$

$$\frac{\langle \Sigma, I \rangle \overset{\mu}{\to} \langle \Sigma', I' \rangle \qquad \mu \neq \surd}{\langle \Sigma, I ; J \rangle \overset{\mu}{\to} \langle \Sigma', I' ; J \rangle}$$

$$\frac{\langle \Sigma, I \rangle \overset{\surd}{\to} \langle \Sigma', I' \rangle \qquad \langle \Sigma', J \rangle \overset{\mu}{\to} \langle \Sigma'', J' \rangle}{\langle \Sigma, I ; J \rangle \overset{\mu}{\to} \langle \Sigma'', J' \rangle}$$

Semantics of projected systems

- Slightly more complex (synchronization close to pi calculus)
- Sample simplified rules:

$$\frac{[\![e]\!]_\Gamma = v}{(\Gamma, o : \text{send } e \text{ to } r)_s \xrightarrow{o\langle v\rangle \text{ from } s \text{ to } r} (\Gamma, 1)_s}$$

$$(\Gamma, x = v)_r \xrightarrow{\tau} (\Gamma[{}^{v}\!/_{x}], 1)_r$$

$$(\Gamma, o : \text{receive } x \text{ from } s)_r \xrightarrow{o : x \leftarrow v \text{ from } s \text{ to } r} (\Gamma, x = v)_r$$

$$(\Gamma, 1)_s \xrightarrow{\surd} (\Gamma, 0)_s$$

$$\frac{N \xrightarrow{o\langle v\rangle \text{ from } s \text{ to } r} N' \qquad N'' \xrightarrow{o : x \leftarrow v \text{ from } s \text{ to } r} N'''}{N \| N'' \xrightarrow{o : s(v) \to r(x)} N' \| N'''}$$

$$\frac{N \xrightarrow{\surd} N' \qquad N'' \xrightarrow{\surd} N'''}{N \| N'' \xrightarrow{\surd} N' \| N'''}$$

Results: correctness of compilation

- When the projected programs are executed in parallel, they implement the desired behaviour
- The behaviour is represented by the LTS
- The LTSs of the choreographic program and of the projected system are weak equivalent
  - Equivalent: can be trace equivalence or bisimilarity
  - Weak: we only consider non-auxiliary interactions and ✓
  - (We have some pruning on the LTS of the projected system, but this can be avoided with a smarter LTS)
- Non-auxiliary interactions and successful termination are preserved and reflected

Results: freedom from concurrency errors

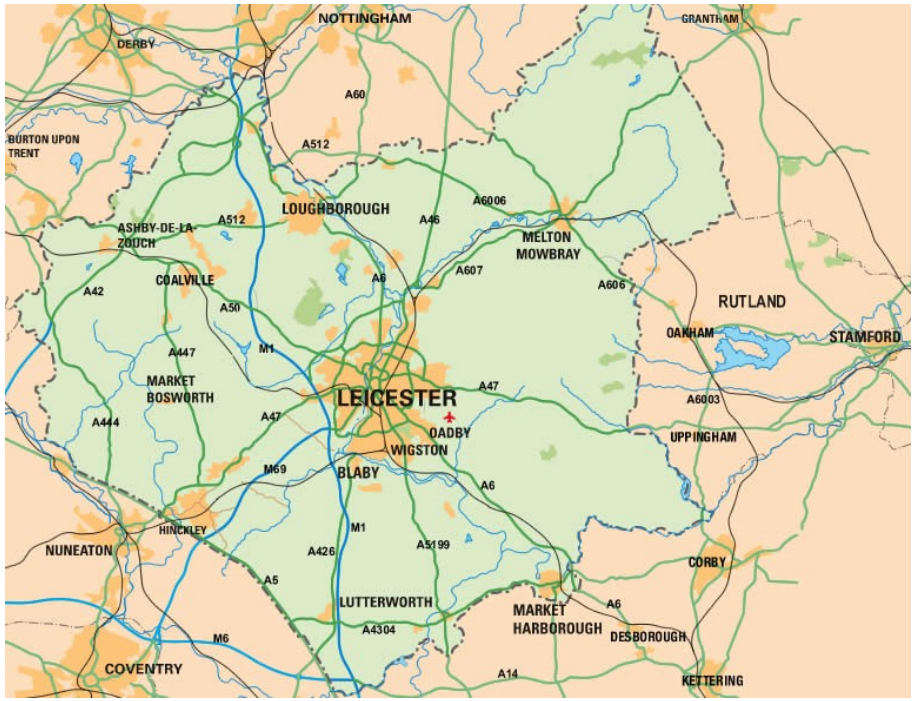
- Deadlocks, orphan messages and message races are avoided by construction
  - These behaviours cannot be written in a choreographic language
  - Projection reflects behaviours, hence they cannot occur in the projected system either

An example: deadlock

- A system has a deadlock when some roles have not finished their computation, yet no transition is possible
- If all roles have successfully terminated, the operational semantics produces a ✓
- There are no other ways of producing a ✓
- Hence, we have a deadlock if there is a finite trace which does not end with ✓
- All finite traces of choreographic programs end with ✓
- Since weak traces are reflected by projection, the projected system never deadlocks

Map of the course

- Choreographic programming
- Choreographic programming in the real world
- Dynamic updates
- Dynamic updates in the real world
- Conclusion



From abstract theory to the real world

- We implemented AIOCJ as an instance of this theory
- AIOCJ is composed by
  - An Eclipse plugin allowing one to write and compile choreographic programs
  - A middleware to enable adaptation (we will see it in the next days)
- Projection generates microservices in the Jolie language
- Choreographic programs could be projected to any language but…
- ...the choice of the Jolie language has some advantages
- The microservices can be executed, obtaining the desired behaviour

A note on microservices

- Services are independent entities communicating via message passing over well-defined APIs
  - Enables communication among heterogeneous entities
  - Thanks to standards such as XML, HTTP and SOAP
- Microservices is an architectural style for service-oriented systems that takes to an extreme the emphasis on modularity and scalability
- Suited to be deployed in Docker containers, frequently on the Cloud
  - We do not yet support this possibility in AIOCJ
- The choreographic approach combines well with modularity

A note on the Jolie language

- The Jolie programming language
  - is a language to program services, and microservices in particular
  - provides suitable abstractions (e.g., communication ports with well-defined APIs)
  - natively supports multiple standards from the area (SOAP, HTTP, RMI, JSON, …)
  - developped at University of Bologna
  - currently supported and exploited by company ItalianaSoftware
  - see https://www.jolie-lang.org/
- AIOCJ inherits some of its features

A note on projection

- AIOCJ projection uses a mixed approach
- Restriction for sequence
  - The Eclipse plugin checks this
- Smart projection for conditionals and loops
  - Auxiliary communications are introduced during compilation

Integration with Legacy Software

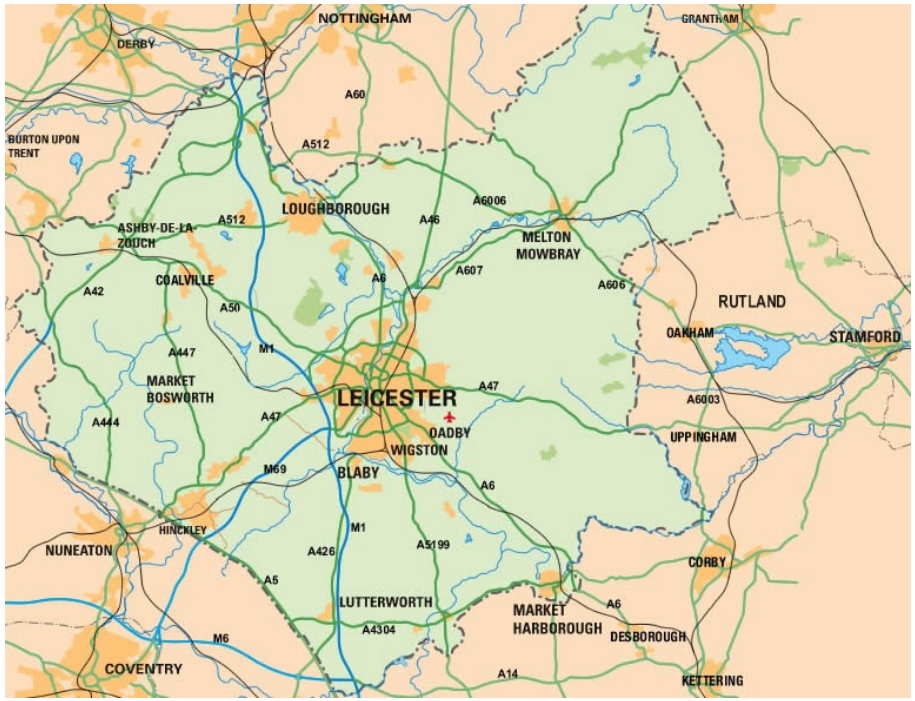
- In the real world applications are not built from scratch
- One uses external libraries, services (e.g., google maps), existing applications…
- We want to integrate them into our framework
- We want AIOCJ distributed applications to be able to interact with existing software
  - While preserving the correctness guarantees
- External software invocations are seen as function invocations from a theoretical point of view
  - They need to have functional behaviour
  - That is, they are invoked and they give back a value
  - No synchronization (e.g., wait on a monitor)

Declaring external dependences

- External function should be declared including
  - where they are: location (URL, ...)
  - which technologies (sockets, bluetooth, ...) and protocols (SOAP, HTTP, …) need to be used
- include makePayment from "socket://localhost:8001/IBAN" with HTTP
- Leverages Jolie ability to interact using different technologies and protocols
- No assumption on the language/technology used to implement them
  - In particular, not necessarily Jolie services

Map of the course

- Choreographic programming
- Choreographic programming in the real world
- Dynamic updates
- Dynamic updates in the real world
- Conclusion



Adaptation, update, evolution

- Applications need to change during their lifetime
- This can be referred to as adaptation, update, evolution
- Slightly different flavours, but the same basic idea, and terminology is not always used in a consistent way
- We will use them as synonyms

Why to update?

- Error fixing
- Environment that evolves
  - Old services no more available
  - Partner applications that evolve
  - New services may become available, exploiting them may improve the results/non functional properties
- Completely new environments
  - Mobile or configurable applications
- New requirements
  - New functionalities
  - Improving non functional properties

Static vs dynamic updates

- Update can be performed statically or dynamically
- Static update: we update the code and then (re)start the application
- Dynamic update: we update a running application, without shutdown and with limited service disruption
- Dynamic update is more challenging
  - Need to interact with running code
  - New code should use pre-existing data
  - Need to keep attention to transient behaviour
  - In a distributed system, need to synchronize updates to different participants

Why dynamic updates?

- Many systems (should) never stop
  - Internet
  - Online services (bank, shopping, ...)
  - Industrial plants
- Mainly for economic reasons
  - Customers do not want services to be down
  - No production on shutdown times
- Yet these systems need to be updated

Our approach to (very) dynamic updates

- We want dynamic updates: we want to change the code of running applications
- An update replaces an existing piece of choreographic program with a new piece of choreographic program coming from outside
  - The external piece of code is called an update fragment
  - The update fragment is not known when the application is started
- A single update may concern more than one participant

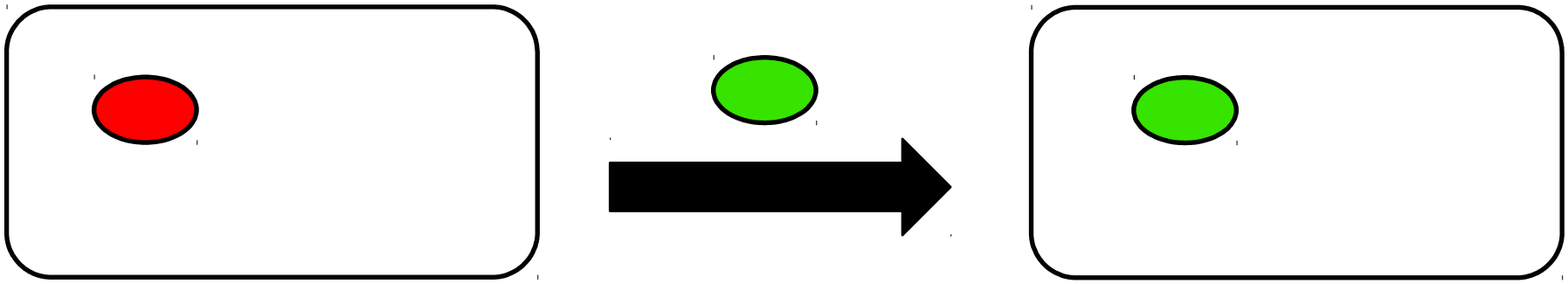
Timing of updates

- At any time there is a set of available update fragments
- New update fragments may become available at any time and without notice
- Update fragments may stop to be available at any time and without notice
- The running application performs a check for update for a given piece of choreographic program just before executing it
  - but the same piece of choreographic program could have been already executed before

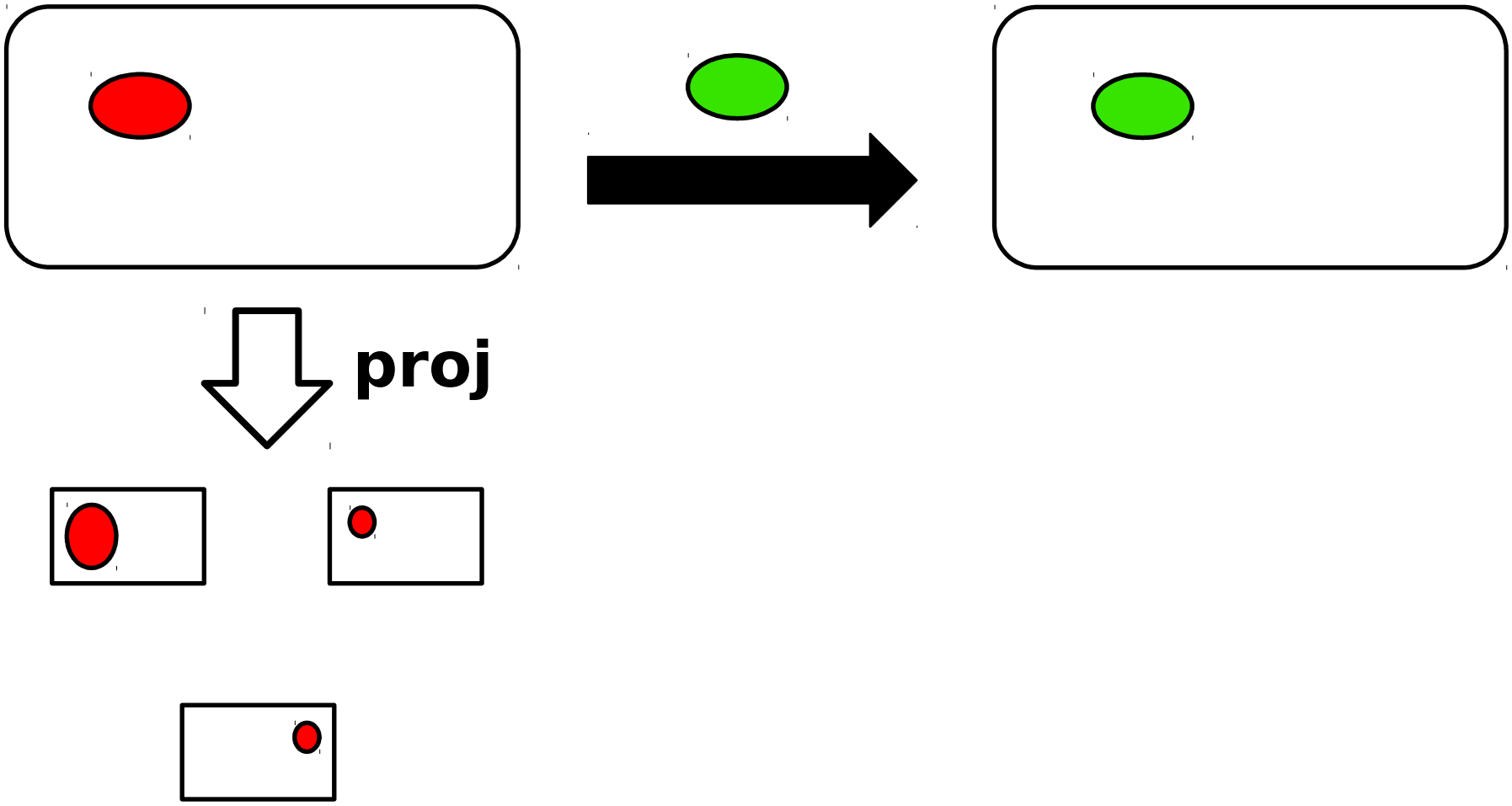
Our approach, syntactically

- Pair a running application with a set of update fragments
  - Each update fragment is a choreographic program
  - The set of update fragments may change at any time
- At the choreographic level, the update fragment may replace a part of the choreographic program
  - Which part?
- Extend choreographic programs with scopes
  - scope @r {C}
  - Normally executes C, but executes an update fragment instead if a suitable one is available
  - That is, a check for update is performed before executing C
  - Participant r is in charge of managing the update procedure

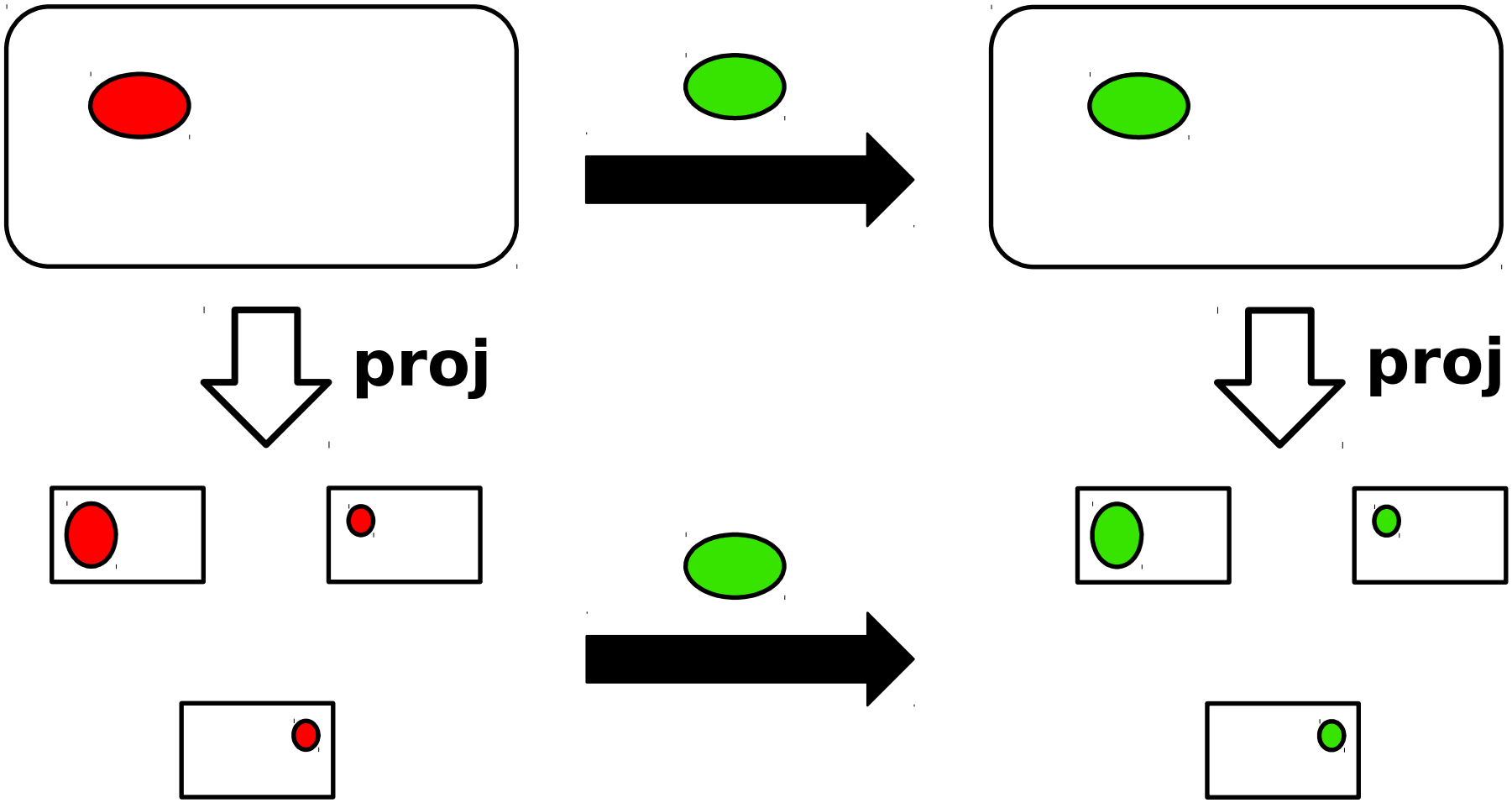
Our approach, graphically



Our approach, graphically



Our approach, graphically



Dynamic updates: challenges

- All the participants should agree on
  - whether to update a scope or not
  - in case, which update fragment to use
- All the participants need to retrieve (their part of) the update fragment
  - Not easy, since update fragments may disappear
- No participant should start executing the code inside a scope that needs to be updated

Dynamic updates: our approach

- For each scope a single participant coordinates its execution
  - Decides whether to update it or not, and in case which update fragment to apply
  - Gets the update fragment, and sends to the other participants their part
- The other participants wait for the decision before executing the scope

How to program with scopes: difficulties

- Deciding where to put scopes in your code is tricky
- If a piece of code is not inside a scope it cannot be updated
- One need to foresee which piece of code may need adaptation
- If a scope is too small, you may want to change more than one scope in a coordinated way, what is not possible
- If a scope is too large
  - You can only update it at the very beginning
  - You need to provide the whole new code, even if you want to update a single line
- Intersecting scopes must be nested
- (And, of course, scopes cause some overhead)

How to program with scopes: guidelines

- Scopes must enclose pieces of code with a precise aim
  - Roughly, what you would put into a procedure
- You should select pieces of code that may need to be updated
  - Pieces of code depending on some regulation that may change
  - Pieces of code depending on the environment
  - Pieces of code critical for performance or security reasons
  - ...
- You do not neet to foresee when they will be updated, why or how

Dynamic updates: formalization (1)

- Updates can be formalized by extending the choreographic language and the target language with scopes
- At the choreographic level scope @r {C} has two kinds of transitions:
  - Either it reduces to its body C
  - Or it reduces to an available update fragment C’
- At the projected system level there are two kinds of scopes, coordinator scopes and non-coordinator scopes
- The construct scope @r {C} is projected to
  - A coordinator scope on r
  - A non-coordinator scope on the other roles in C
  - skip on the other roles

Dynamic updates: formalization (2)

- A coordinator scope:
  - Decides whether to update or not
  - Sends the decision and the projections of the selected update fragment, if any, to the other roles
  - Executes its own code (original or updated)
  - Waits for an end-of-scope message from the other roles
- A non-coordinator scope:
  - Waits for a message telling him whether to update or not, and containing in the first case its new code
  - Executes its own code (original or updated)
  - Sends an end-of-scope message to the coordinator

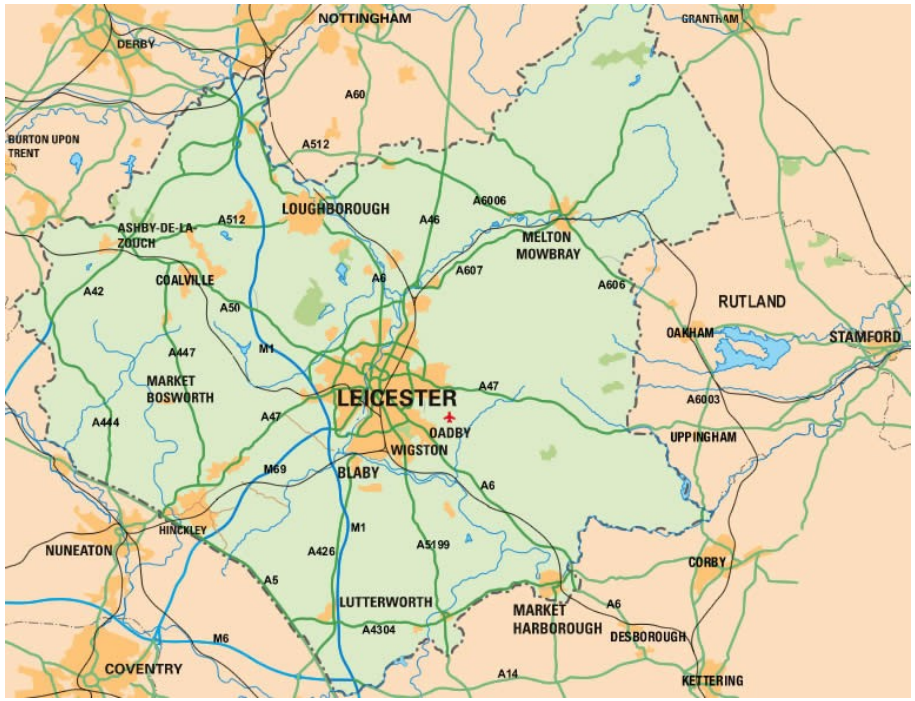
Results

- A choreographic program and its projection behave the same
  - Their LTSs are weak equivalent
  - Under all possible, dynamically changing, sets of updates
- The projected application has no deadlocks, orphan messages nor races by construction

- These results are strong given that we are considering an application which is
  - distributed
  - dynamically updatable

Map of the course

- Choreographic programming
- Choreographic programming in the real world
- Dynamic updates
- Dynamic updates in the real world
- Conclusion



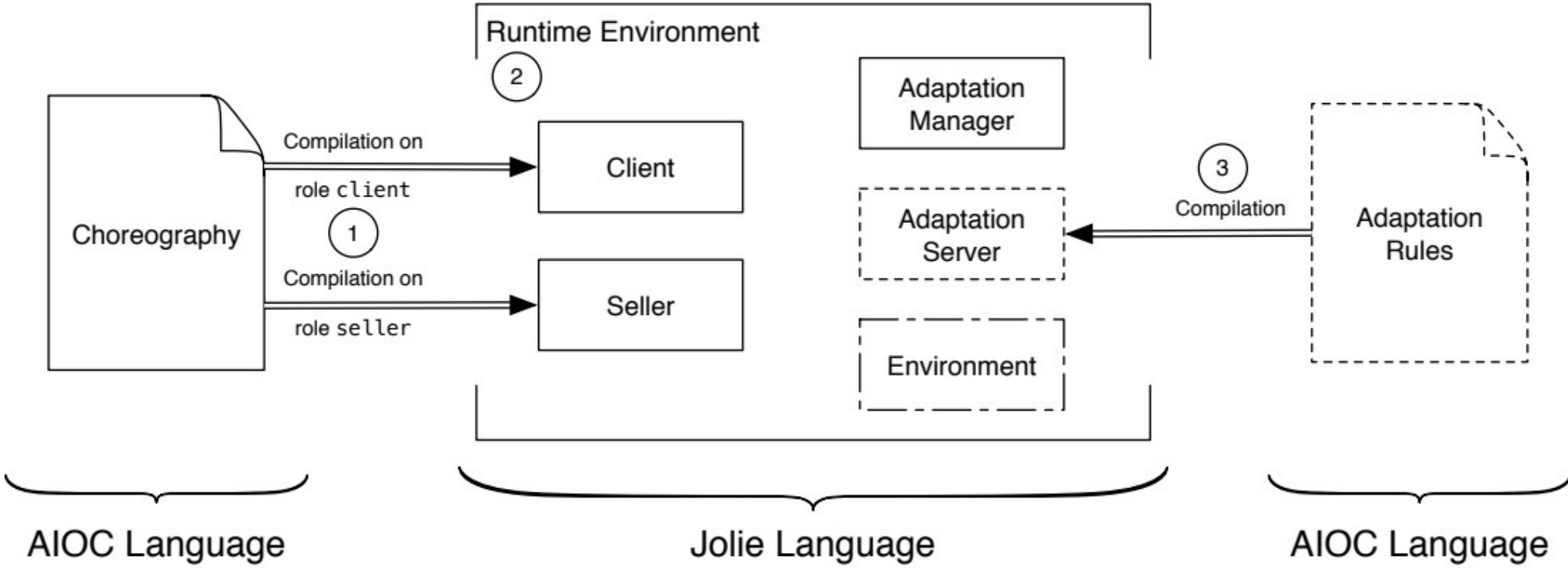
From update fragments to adaptation rules

- Non-determinstic application of updates is not practical
- We use adaptation rules composed by an update fragment and its applicability condition
  - May refer to environment variables, scope properties, and local variables of the coordinator of the update
- If at least one adaptation rule is applicable, update is performed
- This does not avoid nondeterminism, but gives some control
  - Other mechanisms could be added (e.g., priorities)

AIOCJ middleware

- Adaptation managed by a dedicated middleware composed of microservices
- Rules are managed by dedicated microservices called adaptation servers
- A microservice called adaptation manager coordinates the adaptation procedure
- A microservice stores environment information

Our approach, architecturally



Adding, removing and applying rules

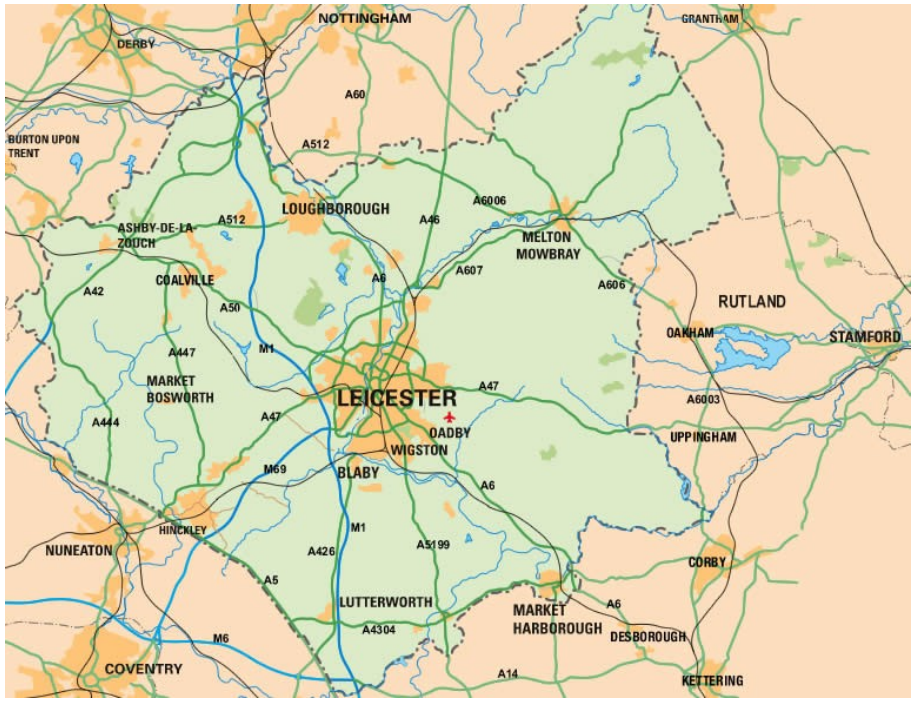
- Adaptation servers can be added and removed at any time
- When added, an adaptation server registers to the adaptation manager
- In order to check for update, the role managing adaptation interacts with the adaptation manager and, as a result, with all available adaptation servers
- Each rule is checked for applicability
- The first applicable rule, if any, is applied
  - Of course, different policies would be meaningful

Implementing adaptation

- We need a mechanism for runtime code update
- Not directly supported by Jolie
  - (Erlang would support it)
- Scopes' body externalized as services
- When a scope needs to be updated, the updated body is deployed as a service, and the invocation switched to it
- All the involved services need to have access to variables
- Variables externalized in a state service

Map of the course

- Choreographic programming
- Choreographic programming in the real world
- Dynamic updates
- Dynamic updates in the real world
- Conclusion



Conclusion

- A choreographic approach to program distributed applications
- The derived distributed application follows the behaviour defined by the choreographic program
- Applications have no deadlocks, orphan messages nor races
- We allow integration of external functionalities
- We allow dynamic update, preserving the correctness guarantees
- We instantiated the theoretical framework into a real programming language

Future work

- Is choreographic programming practical?
  - We need to allow asynchronous communication
  - We need to improve performance
    - Drop redundant auxiliary communications
  - We need error handling
- Only a serious case study can tell
  - Not so easy to find a suitable case study for this framework

A question for our industrial partners

- Could you imagine a use case for AIOCJ
  - You could assume that future work discussed on previous slide has been done
- If yes, which one?
- If no, why? Would something else be needed? What?
- And for the part without adaptation?

(Partial) Bibliography 1

- For a general introduction to choreographies and related topics:
  Hans Hüttel et al.: Foundations of Session Types and Behavioural Contracts. ACM Comput. Surv. 49(1): 3:1-3:36 (2016)
- AIOCJ:
  Mila Dalla Preda et al.: AIOCJ: A Choreographic Framework for Safe Adaptive Distributed Applications. SLE 2014: 161-170
- Theory:
  Mila Dalla Preda et al.: Dynamic Choreographies - Safe Runtime Updates of Distributed Applications. COORDINATION 2015: 67-82
- Comprehensive journal paper:
  Mila Dalla Preda et al.: Dynamic Choreographies: Theory and Implementation. LMCS 13(2) (2017)

(Partial) Bibliography 2

- Using AIOCJ in the software development life cycle:
  Saverio Giallorenzo et al.: ChIP: A Choreographic Integration Process. OTM Conferences (2) 2018: 22-40
- AIOCJ tutorial:
  Saverio Giallorenzo et al.: Programming Adaptive Microservice Applications: An AIOCJ Tutorial, chapter 7 of book Behavioural Types: from Theory to Tools, pages 147-167, River Publisher, 2017.
- AIOCJ website: http://www.cs.unibo.it/projects/jolie/aiocj.html
- Jolie website: http://www.jolie-lang.org/

End of talk