Dynamic Choreographies

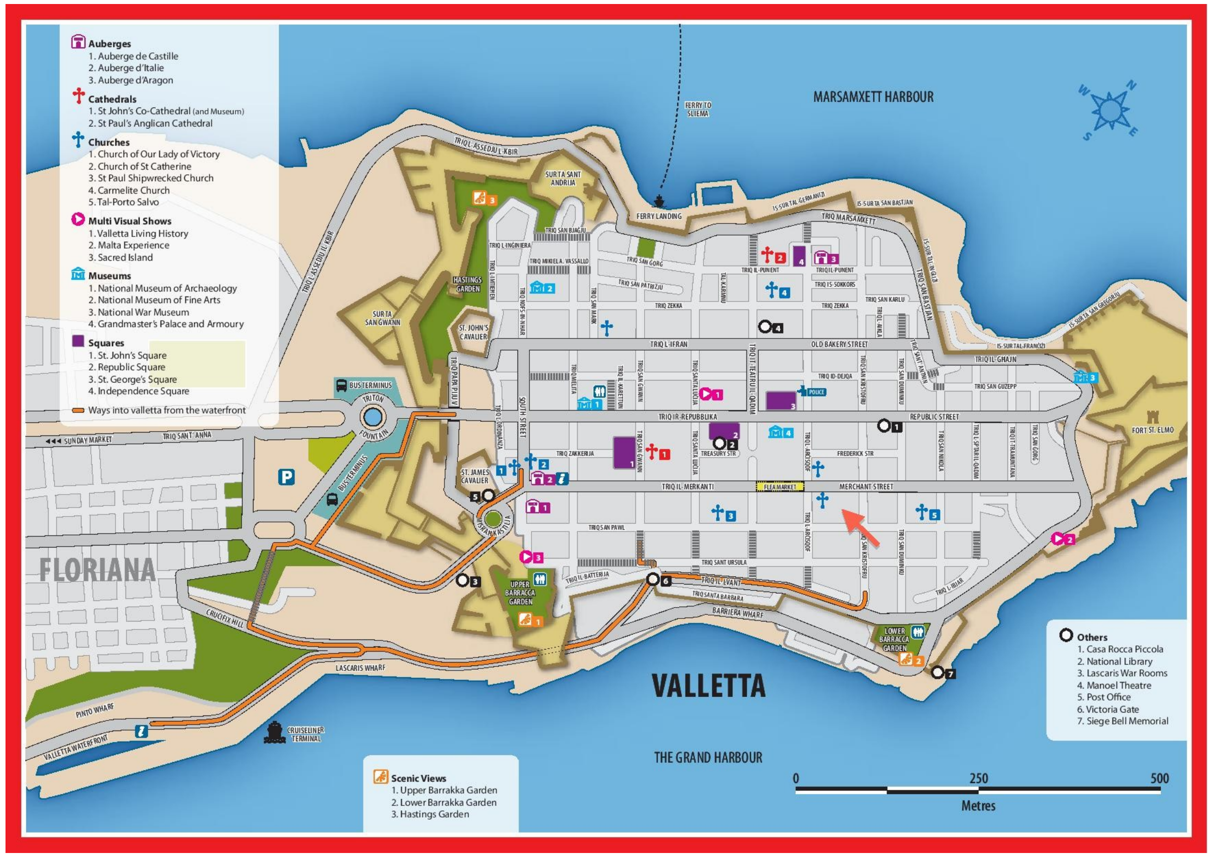
Safe Runtime Updates of Distributed Applications

Ivan Lanese
Computer Science Department
University of Bologna/INRIA
Italy

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

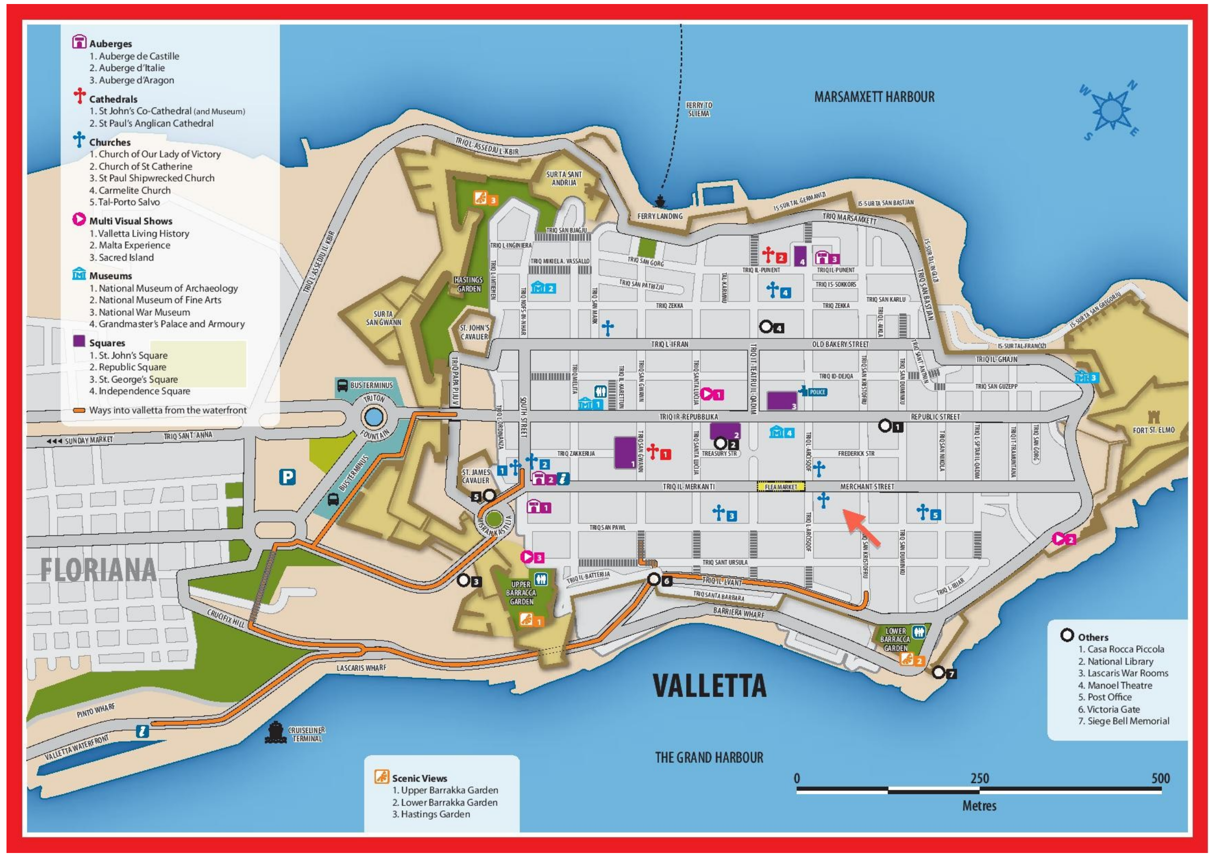
Map of the talk

- Choreographic programming
- Dynamic updates
- Results and applications
- Conclusion



Map of the talk

- Choreographic programming
- Dynamic updates
- Results and applications
- Conclusion



Choreographic programming

- Choreographic programming applies the ideas of global types and endpoint projection
  - not at the level of types
  - but at the level of programming language
- Choreographic programs ≈ global types + data + conditions
- One choreographic program describes a whole distributed application
- The basic building block is an interaction, i.e. a communication between two participants
- Interactions can be composed using standard constructs: sequences, conditionals, cycles,...

Choreographic programming: syntax

- $I ::=$

$o : r(e) \rightarrow s(x)$	interaction
$x@r = e$	assignment
1	skip
$I ; I'$	sequence
$I \mid I'$	parallel
if $b@r$ { I } else { I' }	conditional
while $b@r$ { I }	loop

- o are operations, r,s are roles, e expressions, b boolean expressions, x variables

A sample choreographic program

- *prodName@buyer* = getInput*();*
  priceReq : buyer (prodName) $\rightarrow$ *seller (pName);*
  price@seller = getPrice*(pName);*
  offer : seller (price) $\rightarrow$ *buyer (pr);*
  …

Advantages of choreographic programming

- Same as for global types
- Clear view of the global behavior
- No deadlocks and no races by construction
- … and you are in an untyped setting!

How to execute choreographic programs?

- Most constructs involve many participants
- What each participant should do?
- We want to compile one choreographic program generating a local code for each participant
- We define a projection function to this end
  - Similar to endpoint projection for multiparty session types
- When executed, the derived participants should interact as specified in the choreographic program
  - Correctness of the compilation (close to session fidelity)
  - No deadlocks and no races

The target language

- P ::=

$o : e$ to r	send
$o : x$ from r	receive
$x = e$	assignment
$\mathit{1}$	skip
$P \,;\, P'$	sequence
$P \mid P'$	parallel
if b $\{ P \}$ else $\{ P' \}$	conditional
while b $\{ P \}$	loop

- A distributed application is composed by named participants, each executing a program P

Projection: basic idea

- An interaction $o : r(e) \to s(x)$ becomes
  - A send $o : e$ to s on r
  - A receive $o : x$ from r on s
  - A skip 1 on all the other participants
- Assignments and guard evaluations are executed by the declared role
- Other constructs are projected homomorphically
- Very simple…

Projection: basic idea

- An interaction $o : r(e) \rightarrow s(x)$ becomes
  - A send $o : e$ to s on r
  - A receive $o : x$ from r on s
  - A skip 1 on all the other participants
- Assignments and guard evaluations are executed by the declared role
- Other constructs are projected homomorphically

- Very simple…
- …but it does not work

Projection: problems

- Participants are independent

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

- Interaction on o_2 should happen after interaction on o_1
  - No participant can force this
- Participants' execution may depend on other participants

  $$\text{if } x@r_1 \ \{o : r_2(5) \rightarrow s(x)\} \text{ else } \{o : r_2(7) \rightarrow s(x)\}$$

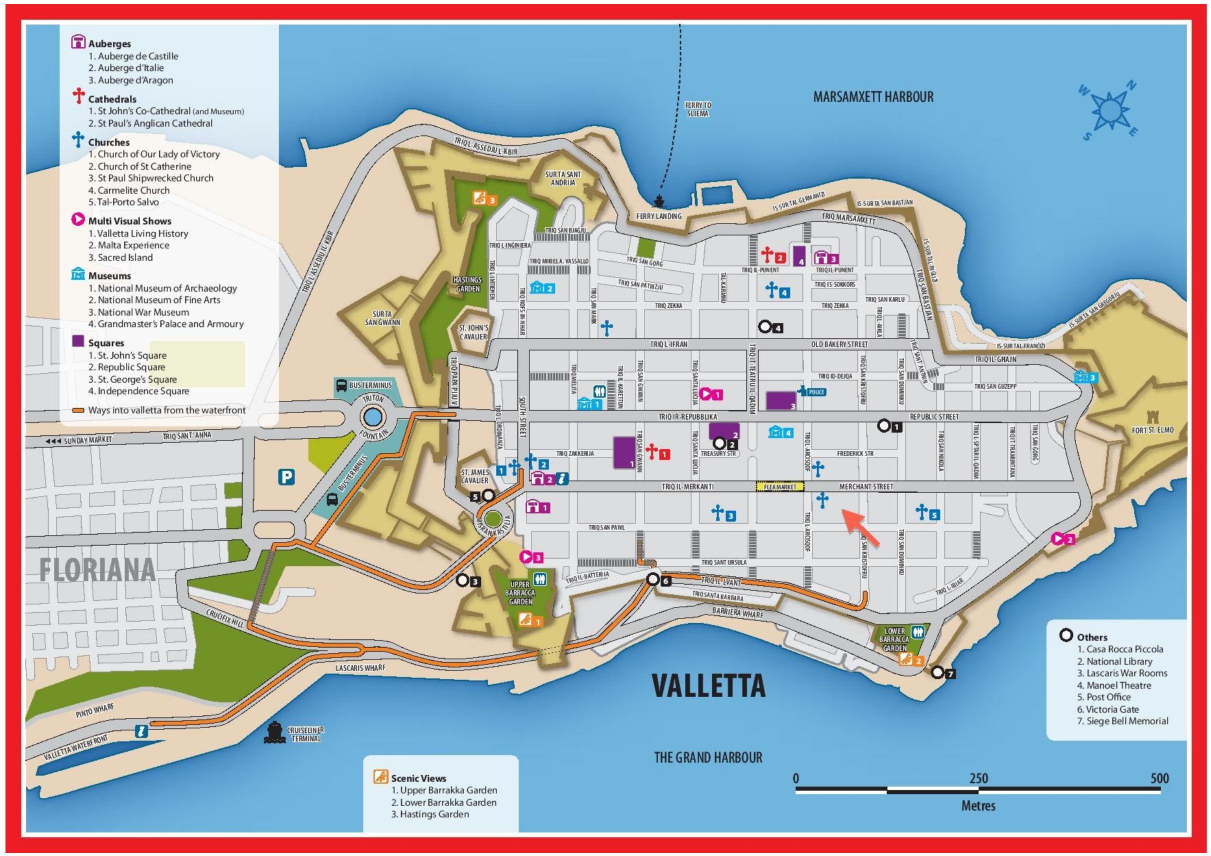
- Participant r_2 should send 5 or 7 according to a local decision of r_1

Projection: solutions

- Two kinds of solutions
- Restricting the allowed compositions (connectedness)
  - More difficult for the programmer to write code satisfying the requirements
  - Easier compilation
- Adding auxiliary communications beyond the ones specified
  - Easier for the programmer
  - More difficult compilation, and additional communications cause overhead
- We use both the approaches, depending on the construct

Map of the talk

- Choreographic programming
- Dynamic updates
- Results and applications
- Conclusion



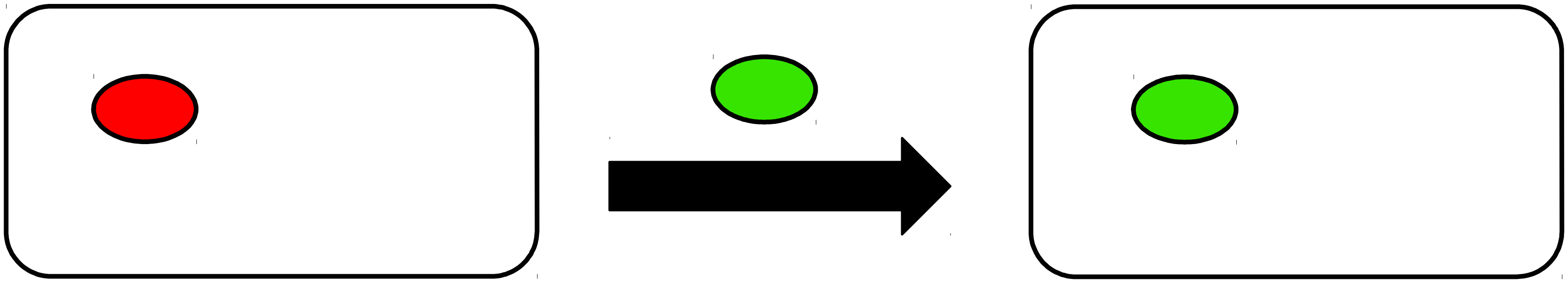
Dynamic updates

- We want to change the code of running applications, by integrating new pieces of code coming from outside
- Those pieces of code are called updates
- The set of updates
  - is not known when the application is designed, programmed or even started
  - may change at any moment and without notice
- Many possible uses
  - Deal with emergency behavior
  - Deal with changing business rules or environment conditions
  - Specialise the application to user preferences

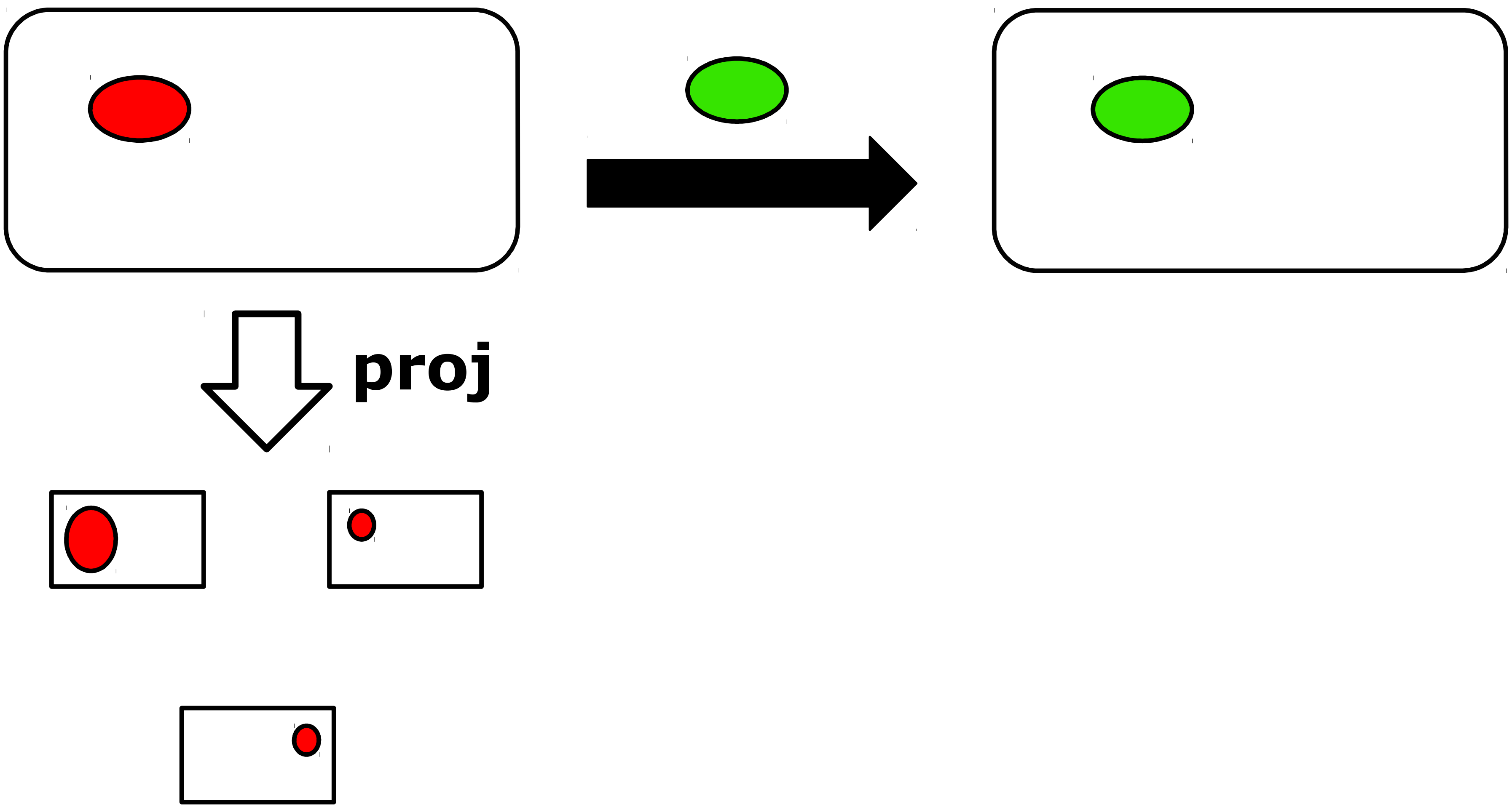
Our approach, syntactically

- Pair a running application with a set of updates
  - Each update is a choreographic program
  - The set of updates may change at any time
- At the choreographic level, the update may replace a part of the application
  - Which part?
- Extend choreographic programs with scopes
  - *scope @r { I }*
  - Before starting, the scope may be replaced by an update

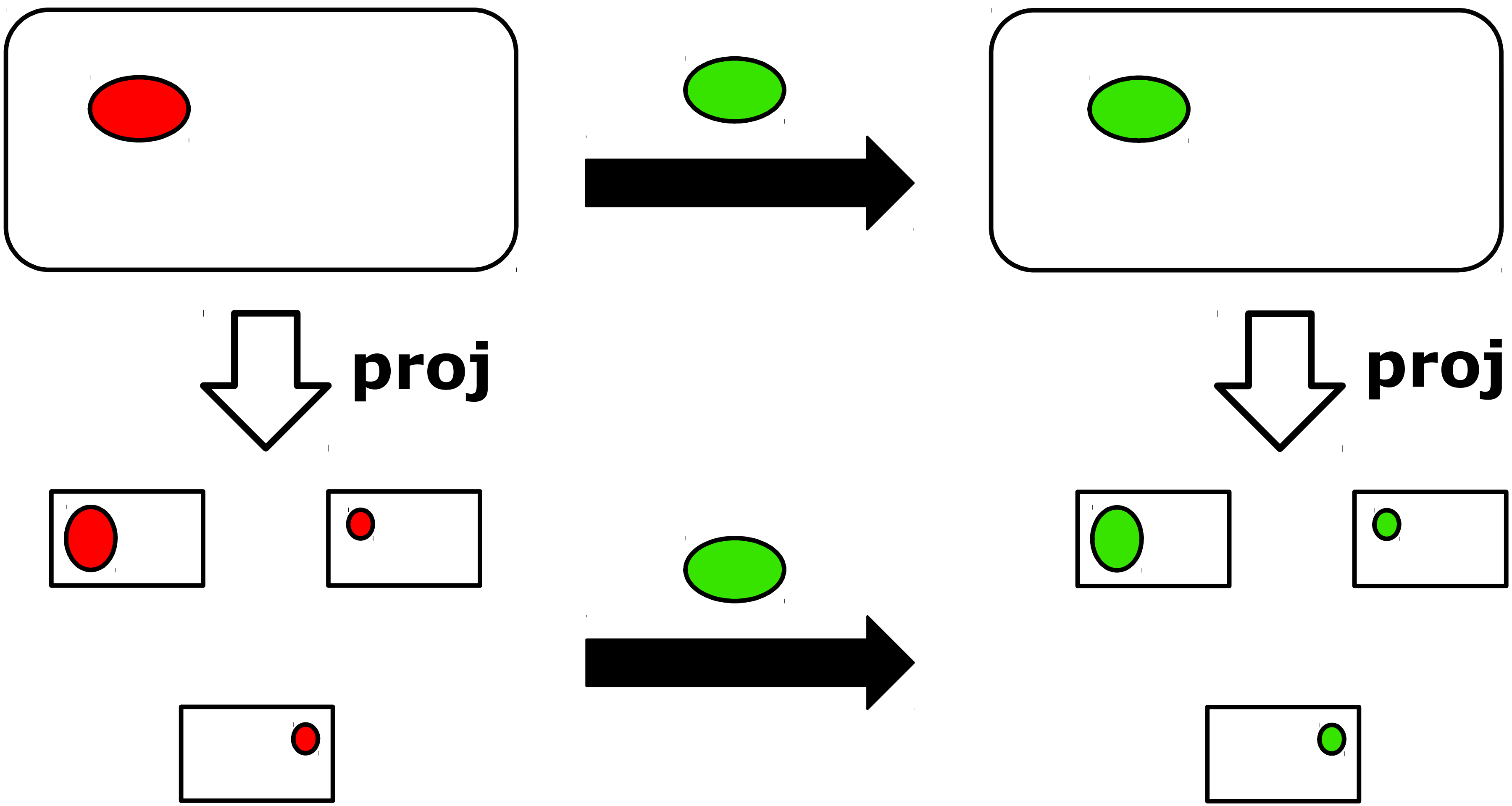
Our approach, graphically



Our approach, graphically



Our approach, graphically



Our approach, semantically

- A scope can either execute, or be replaced by an update

$$< \Sigma , \boldsymbol{I} , scope\ @r\ \{ I \} > \xrightarrow{\text{no-up}} < \Sigma , \boldsymbol{I} , I >$$

$$\frac{roles(I') \subseteq roles(I) \qquad I' \in \boldsymbol{I} \qquad I' \text{ connected}}{< \Sigma , \boldsymbol{I} , scope\ @r\ \{ I \} > \xrightarrow{I'} < \Sigma , \boldsymbol{I} , I' >}$$

- The set of available updates can change at any time

$$< \Sigma , \boldsymbol{I} , I > \xrightarrow{\boldsymbol{I'}} < \Sigma , \boldsymbol{I'} , I >$$

A sample update

- *cardReq : seller () → buyer ();*
  cardSend : buyer (cardId) → seller (buyerId);
  if (isValid*(buyerId)) @ seller*
  { price@seller = getPrice*(pName) * 0.8; }*
  else
  { price@seller = getPrice*(pName);}*
  offer : seller (price) → buyer (pr)

Making the choreographic program updatable

- *prodName@buyer* = getInput*();*
  priceReq : buyer (prodName) → seller (pName);
  price@seller = getPrice*(pName);*
  offer : seller (price) → buyer (pr);
  …

Making the choreographic program updatable

- *prodName@buyer* = getInput*();*
  priceReq : buyer (prodName) → seller (pName);
  scope @seller {
      price@seller = getPrice*(pName);*
      offer : seller (price) → buyer (pr)
  }
  …

Dynamic updates: challenges

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

Dynamic updates: our approach

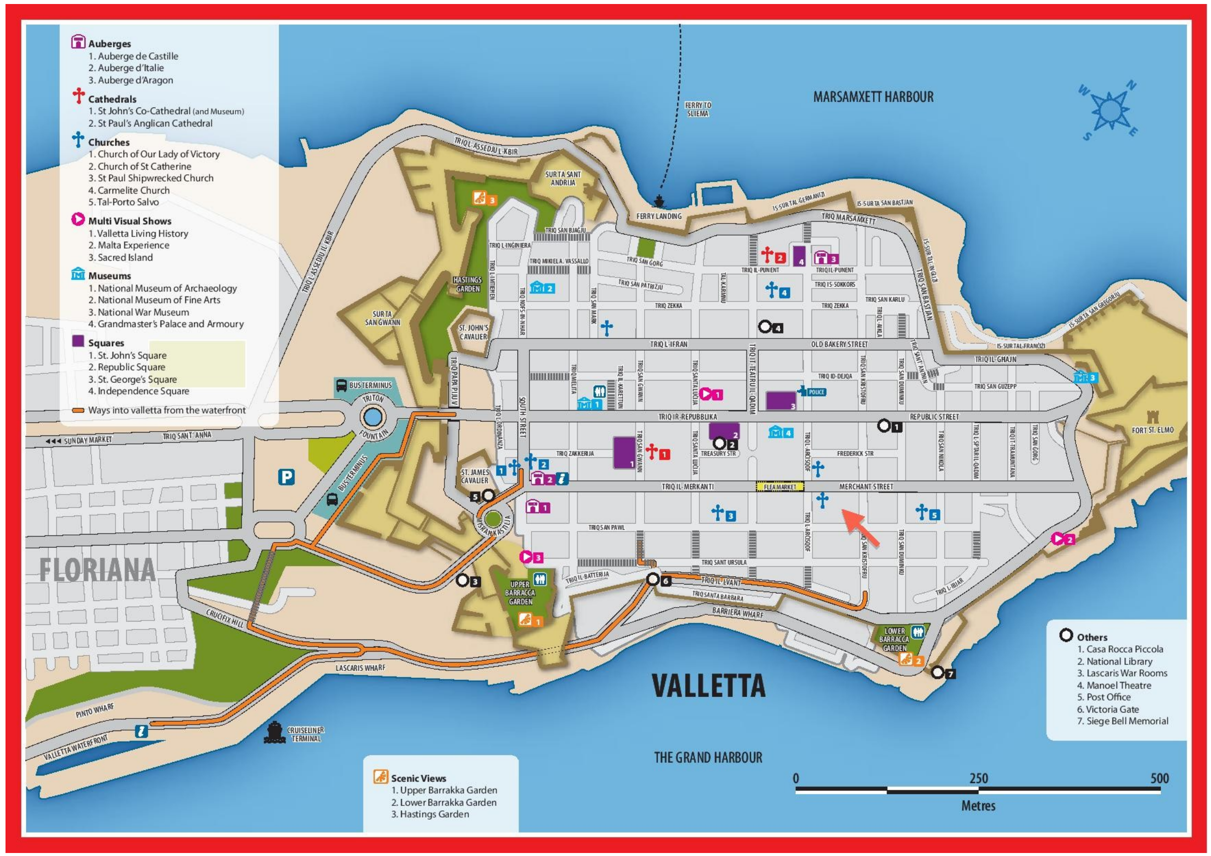
- For each scope a single participant coordinates its execution
  - Decides whether to update it or not, and which update to apply
  - Gets the update, and sends to the other participants their part
- The other participants wait for the decision before executing the scope
- We add scopes (and higher-order communications) to the target language, with the informal semantics above

Compositionality issue

- Applying an update at the choreographic level results in a new choreographic program, composed by
  - The unchanged part of the old choreographic program
  - The update
- Even if the two parts are connected, the result may not be connected
- Auxiliary communications are added to ensure connectedness

Map of the talk

- Choreographic programming
- Dynamic updates
- Results and applications
- Conclusion



Results

- A choreographic program and its projection behave the same
  - They have the same set of weak traces (abstracting away auxiliary actions)
  - Under all possible, dynamically changing, sets of updates
- The projected application is deadlock free and race free by construction

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

An instance for rule-based adaptation

- Our result is quite abstract
  - Whether to update or not, and which update to apply is nondeterminstic
- Different instances are possible, reducing nondeterminism
- AIOCJ [SLE 2014] explores one such possibility
- A framework for safe rule-based adaptation of distributed applications
- Available as an eclipse plugin
- http://www.cs.unibo.it/projects/jolie/aiocj.html
- Projection produces service-oriented code

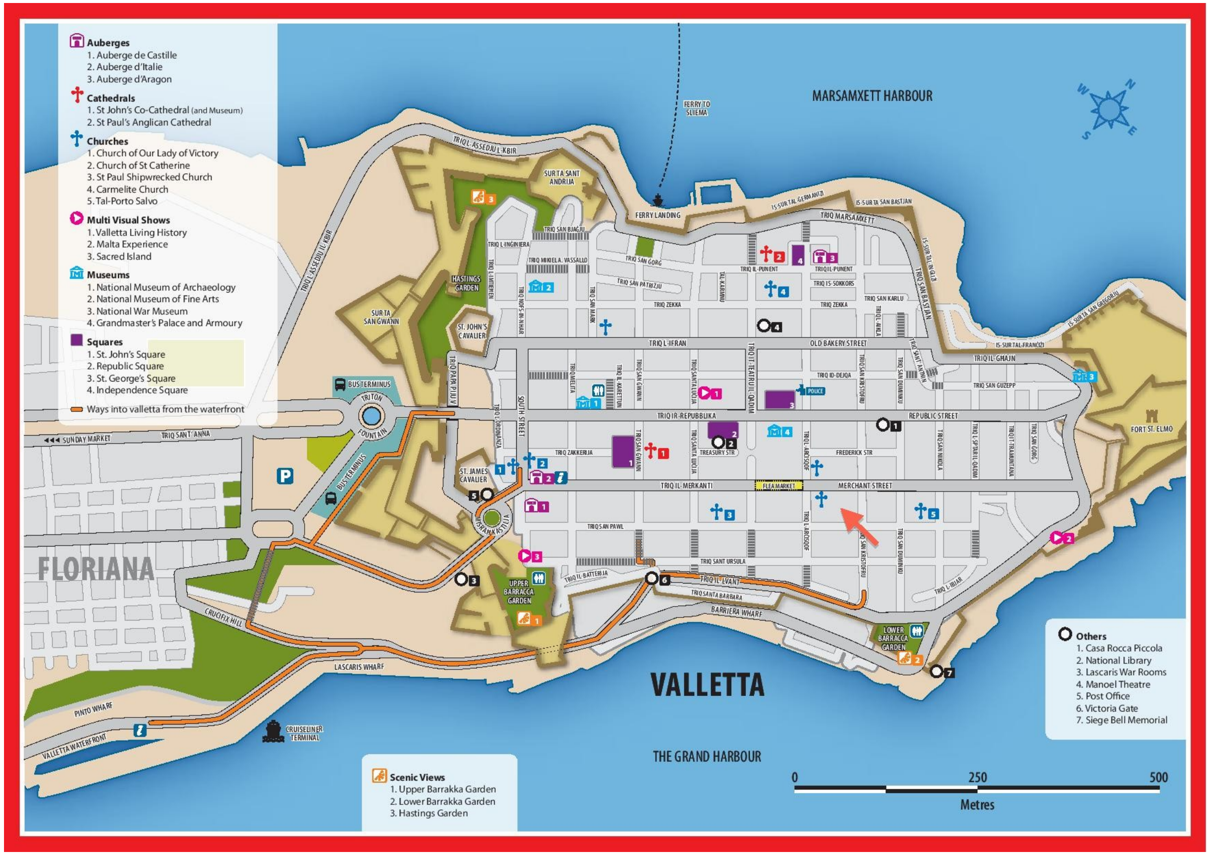
What AIOCJ adds?

- Scopes include some information describing the current implementation
- The framework includes an environment providing contextual information
- A rule is an update plus an applicability condition
  - A Boolean formula taking into account scope information, environmental information and state information
- An adaptation manager allows one to load sets of rules dynamically

Demo time

Map of the talk

- Choreographic programming
- Dynamic updates
- Results and applications
- Conclusion



Conclusion

- A choreographic approach to dynamic updates
- The derived distributed application follows the behavior defined by the choreographic program
- We ensure deadlock freedom and race freedom in a challenging setting
- We instantiated the theoretical framework to adaptable service-oriented applications

Future work

- Extend the approach to asynchronous communication
- How to cope with multiple interleaved sessions?
- How to improve the performance?
  - Drop redundant auxiliary communications
- Can we instantiate our approach on existing frameworks for adaptation?
  - E.g., dynamic aspect-oriented programming
  - To inject correctness guarentees

End of talk

Thanks!

Questions?