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
- Conclusion
Map of the talk

- Choreographic programming
- Dynamic updates
- Results
- Conclusion
Choreographic programming: aim

- Distributed applications are normally programmed using send and receive primitives
  - Difficult and error-prone
  - Deadlocks, races, ...
- Choreographic programming aims at solving these problems by raising the level of abstraction
Choreographic programming: basics

- The basic building block is an interaction, i.e. a communication between two participants
  - Not a send or a receive
- Interactions can be composed using standard constructs: sequences, conditionals, cycles,...
- One choreographic program describes a whole distributed application
  - Not a single participant
Choreographic syntax

- $I ::= o : r(e) \to s(x)$
  - $x@r = e$
  - 1
  - $I ; I'$
  - $I | I'$
  - if $b@r \{ I \}$ else $\{ I' \}$
  - while $b@r \{ I \}$

For multiparty session types addicts
choreographic programs $\approx$ global types + data + conditions
A sample choreographic program

- \( \text{prodName@buyer} = \text{getInput}(); \)
  \( \text{priceReq: buyer(prodName)} \rightarrow \text{seller(pName)}; \)
  \( \text{price@seller} = \text{getPrice(pName)}; \)
  \( \text{offer: seller(price)} \rightarrow \text{buyer(pr)}; \)
  ...

Advantages of choreographic programming

- Clear view of the global behavior
- No deadlocks and races since send and receive are paired into interactions
How to execute choreographic programs?

- Most constructs involve many participants
- What each participant should do?
- We want to compile the choreographic program into a local code for each participant
- We define a projection function to this end
- When executed, the derived participants should interact as specified in the choreographic program
  - Correctness of the compilation
  - No deadlocks and no races
The target language

- $P ::= o : e \text{ to } r$
  - $o : x \text{ from } r$
  - $x = e$
  - 1
  - $P ; P'$
  - $P | P'$
  - if $b$ {P} else {P'}
  - while $b$ {P}

- A distributed application is composed by named participants executing Ps
Projection: basic idea

- An interaction $o_1 : r_1(5) \rightarrow s_1(x)$ becomes
  - A send $o_1 : 5$ to $s_1$ on $r_1$
  - A receive $o_1 : x$ from $r_1$ on $s_1$
  - A skip 1 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 and solutions

- 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
  if \( x \cdot r_1 \) \( \{ o: r_2(5) \rightarrow s(x) \} \) else \( \{ o: r_2(7) \rightarrow s(x) \} \)
- Participant \( r_2 \) should send 5 or 7 according to a local decision of \( r_1 \)
- These problems are solved by
  - adding auxiliary communications beyond the ones specified
  - restricting the allowed compositions (connectedness)
Map of the talk

- Choreographic programming
- Dynamic updates
- Results
- 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, semantically

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

\[
\langle \Sigma, I, \text{scope @}r \{I\} \rangle \xrightarrow{\text{no-up}} \langle \Sigma, I, I' \rangle
\]

\[
\text{roles}(I') \subseteq \text{roles}(I) \quad I' \in I \quad I' \text{ connected}
\]

\[
\langle \Sigma, I, \text{scope @}r \{I\} \rangle \xrightarrow{I'} \langle \Sigma, I, I' \rangle
\]

- Updates can change at any time

\[
\langle \Sigma, I, I \rangle \xrightarrow{I'} \langle \Sigma, I', I \rangle
\]
Our approach, graphically
Our approach, graphically
Our approach, graphically

proj

proj
A sample update

- cardReq: seller() → buyer();
- cardSend: buyer(cardId) → seller(buyerId);
- if(isValid(buyerId))@seller
  
  "price@seller = getPrice(pName) * 0.9;"

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

- \(prodName@buyer = \text{getInput}()\);
- \(priceReq: buyer(prodName) \rightarrow seller(pName)\);
- \(\text{scope } @seller \{\)
- \hspace{1em} \(price@seller = \text{getPrice}(pName)\);
- \hspace{1em} \(offer: seller(price) \rightarrow buyer(pr)\);
- \}
- ...

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 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
- Conclusion
Results

● A choreographic program and its projection behave the same
  – They have the same set of traces (up to 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
Implementation

- Our result is quite abstract, and can be instantiated in different ways
- AIOCJ is one such way [SLE 2014]
- A framework for safe rule-based adaptation of distributed applications
- Updates are embedded into adaptation rules specifying when and where to apply them
- Scopes include some more information driving the application of adaptation rules
- Projection produces service-oriented code
Map of the talk

- Choreographic programming
- Dynamic updates
- Results
- 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 guarantees
End of talk

Thanks!

Questions?