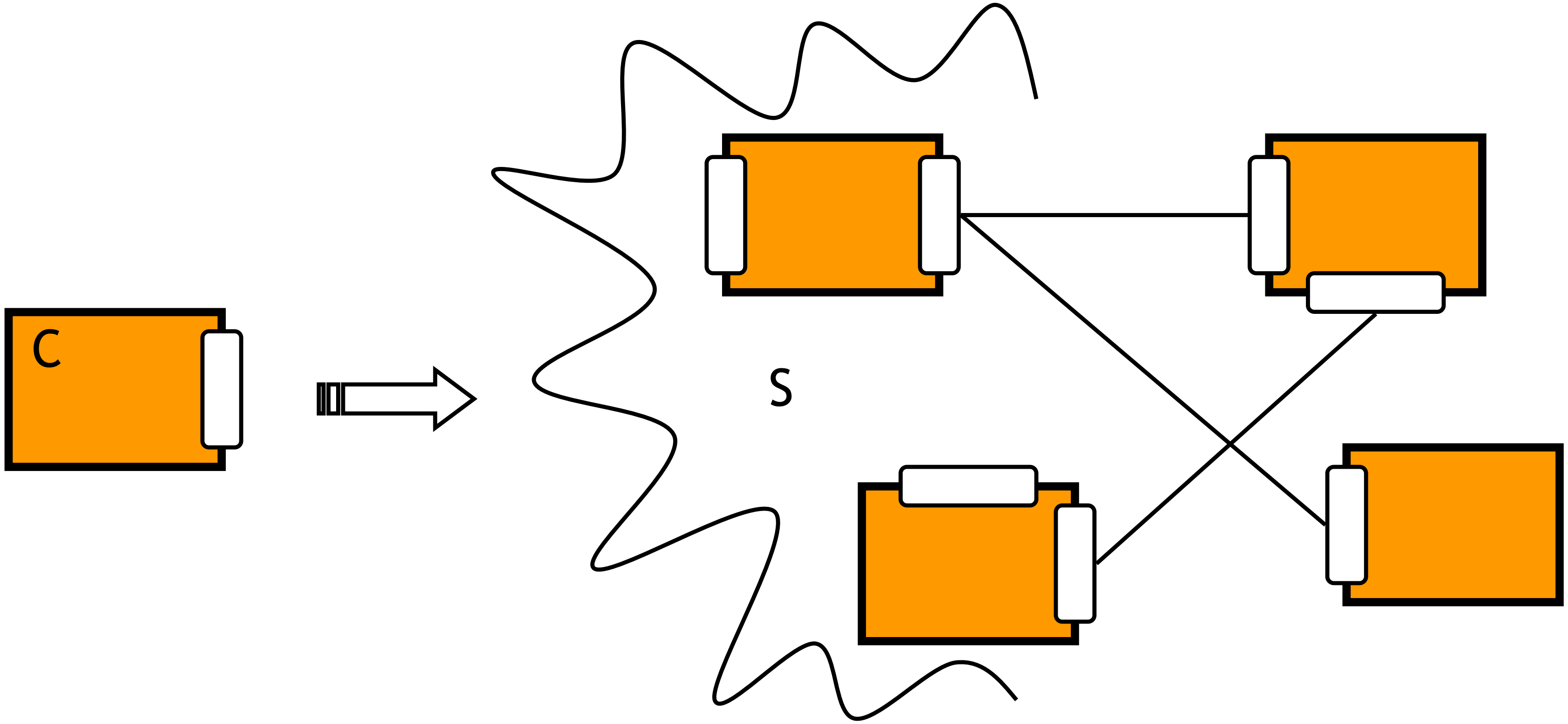
Synchronizing Behavioural Mismatch in Software Composition

Carlos Canal

Pascal Poizat

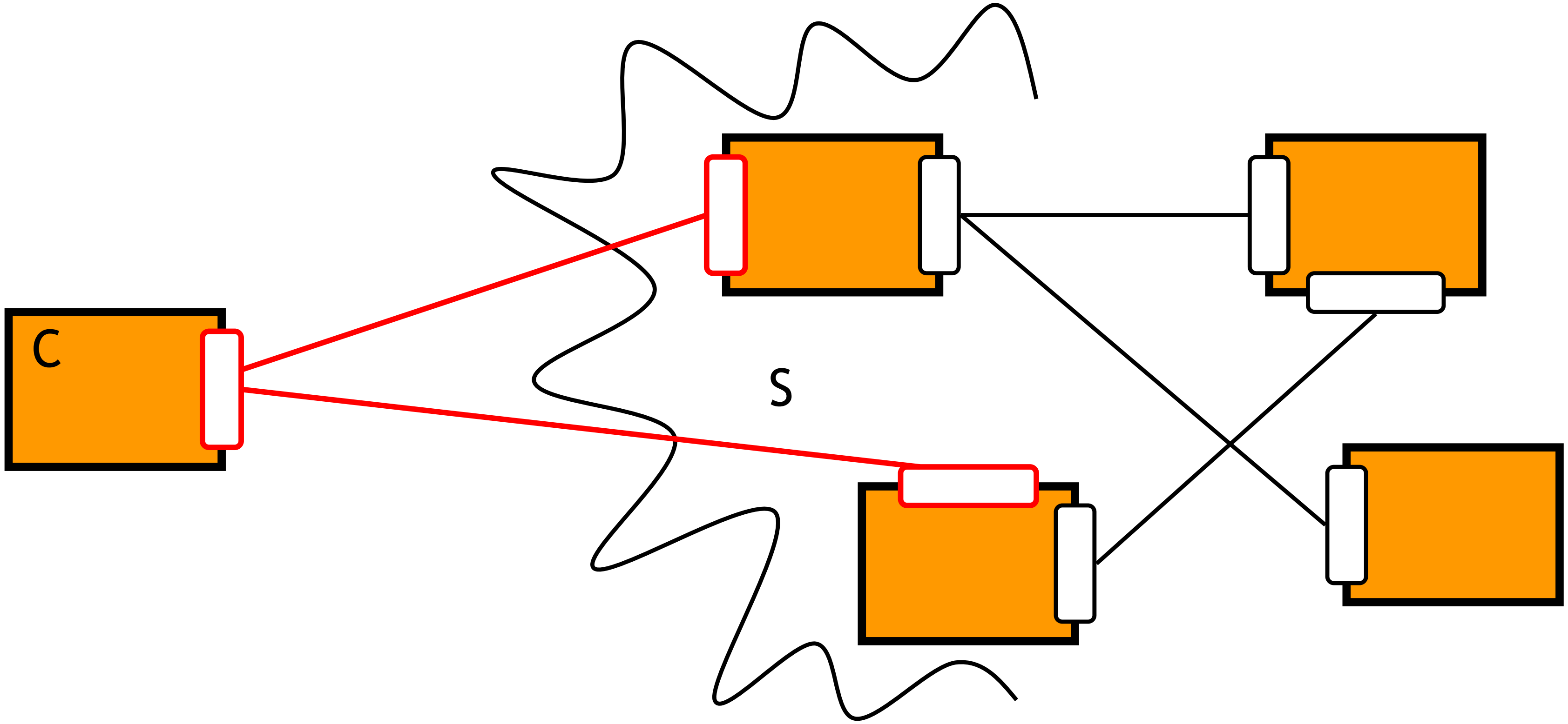
Gwen Salaün

Composition Scenario



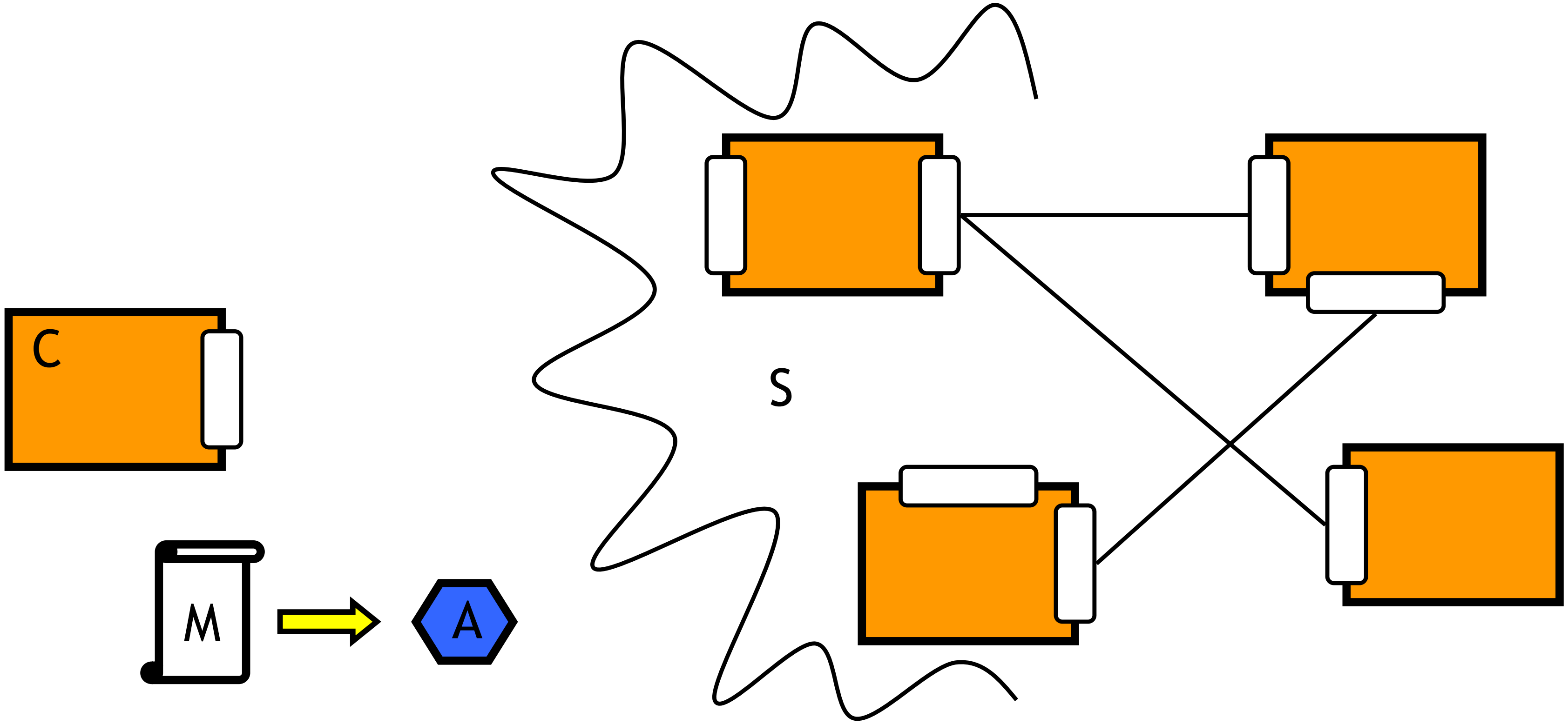
- Addition of a component C to the system S

Composition Scenario



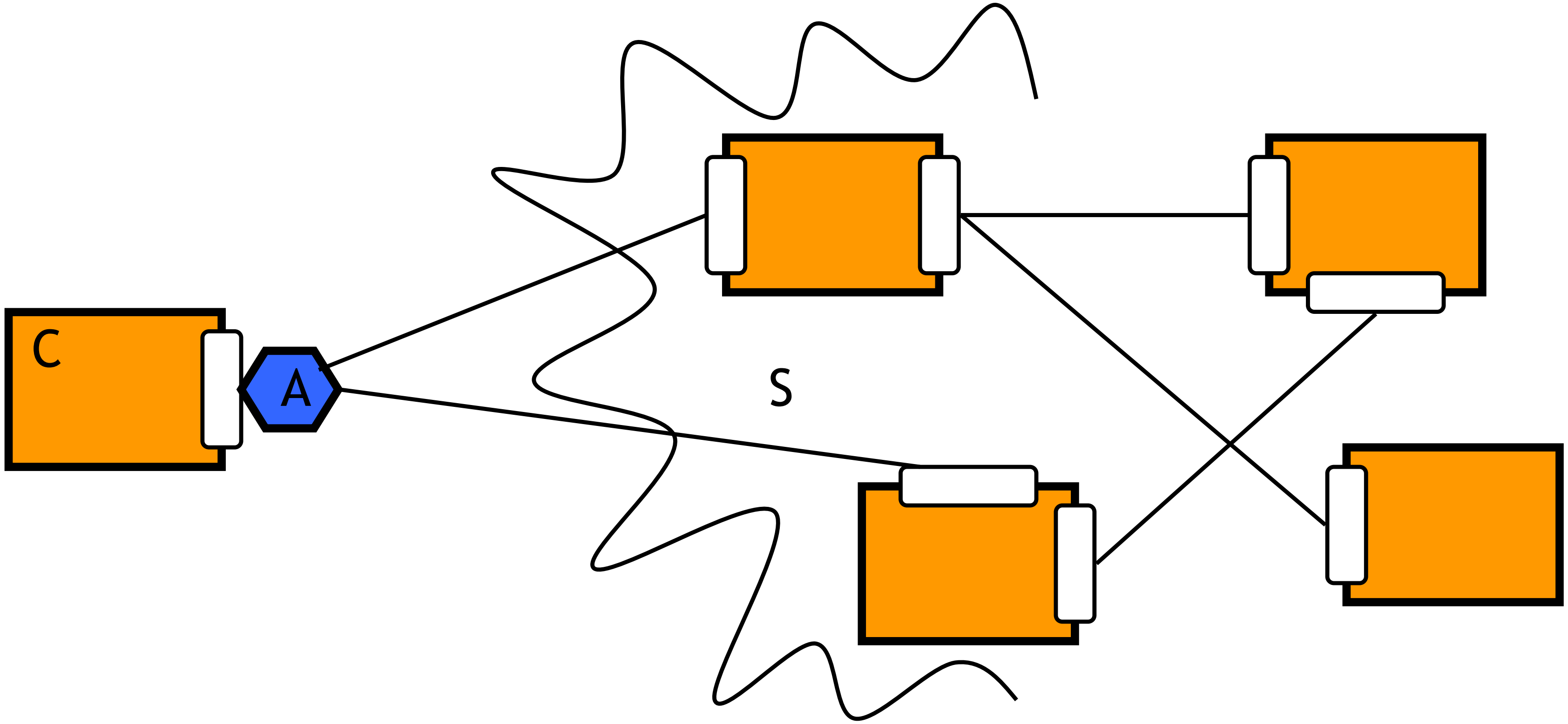
- Addition of a component C to the system S
- Connection is not possible: C|S is blocking!

Composition Scenario



- Addition of a component C to the system S
- Connection is not possible: C|S is blocking!
- Generation of an adaptor A using a mapping M

Composition Scenario



- Addition of a component C to the system S
- Connection is not possible: C|S is blocking!
- Generation of an adaptor A using a mapping
- The system C|A|S is not blocking

Motivation

- Component-based systems are built by composition and reuse of existing components
- Several levels of interoperability which may raise incompatibilities:
  - + Signatures, behaviours
  - – Semantic aspects, quality of services
- A component is seldom used directly as it is and needs some adaptations
- Reusing components as automatically as possible

Outline of the Presentation

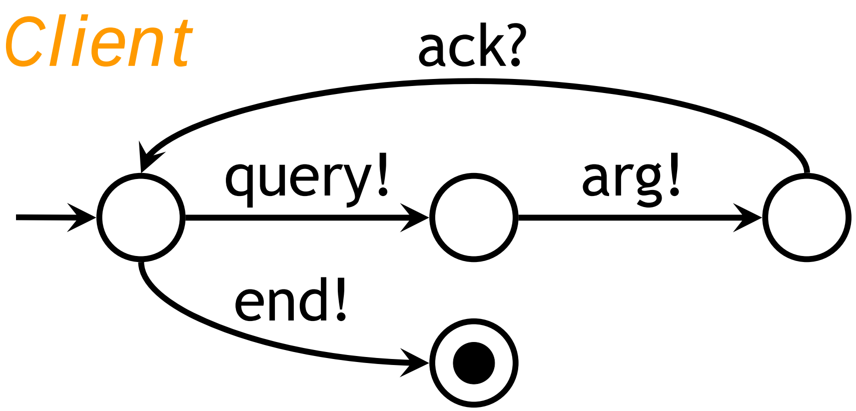
- Overview of our Approach
- Adaptation using Vectors
- Adaptation using Regular Expressions
- Conclusion and Future Work
- Demo

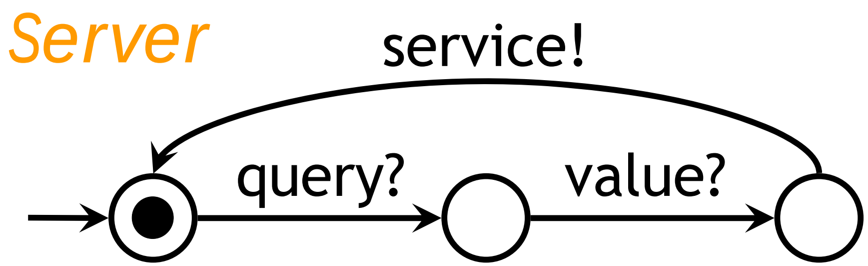
Behavioural Interfaces (BIDL)

- Signatures of operations
- LTS : (A, S, I, F, T)
- Simple process algebra (sequential CCS):

P ::= 0 | m!.P | m?.P | P1+P2 | A

with [i] and [f] for initial and final states

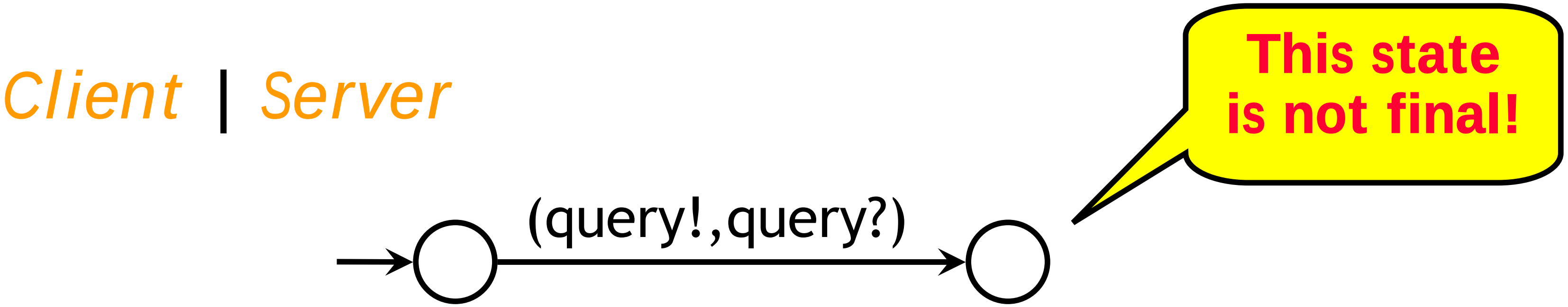




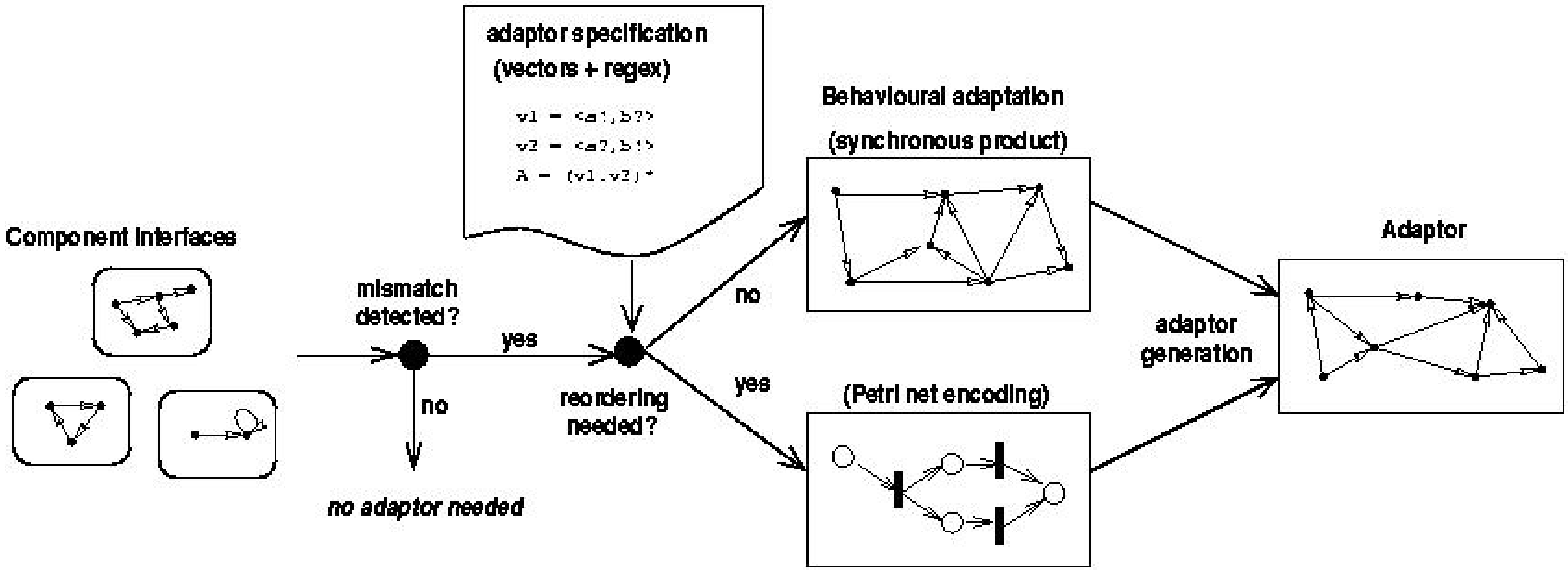
Server[i,f] = query?.value?.service!.Server

Compatibility Check

- Synchronized product [Arnold94] to build a unique LTS from several LTS components
- A deadlock in a LTS if one state is reachable yet not final
- Several components are compatible if the product of their interfaces is deadlock-free



Overview of our Approach

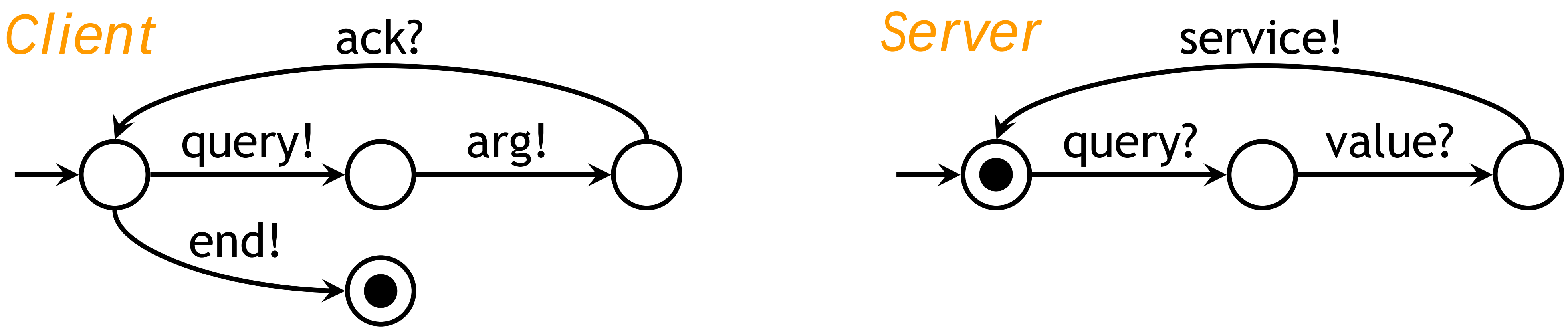


Outline of the Talk

- Overview of our Approach
- Adaptation using Vectors
- Adaptation using Regular Expressions
- Conclusion and Future Work
- Demo

Synchronization Vectors

- Being given a set of LTS $L_i = (A_i, S_i, I_i, F_i, T_i)$, a vector is a tuple $<e_i>$ with $e_i \in A_i \cup \{\varepsilon\}$
- Example:



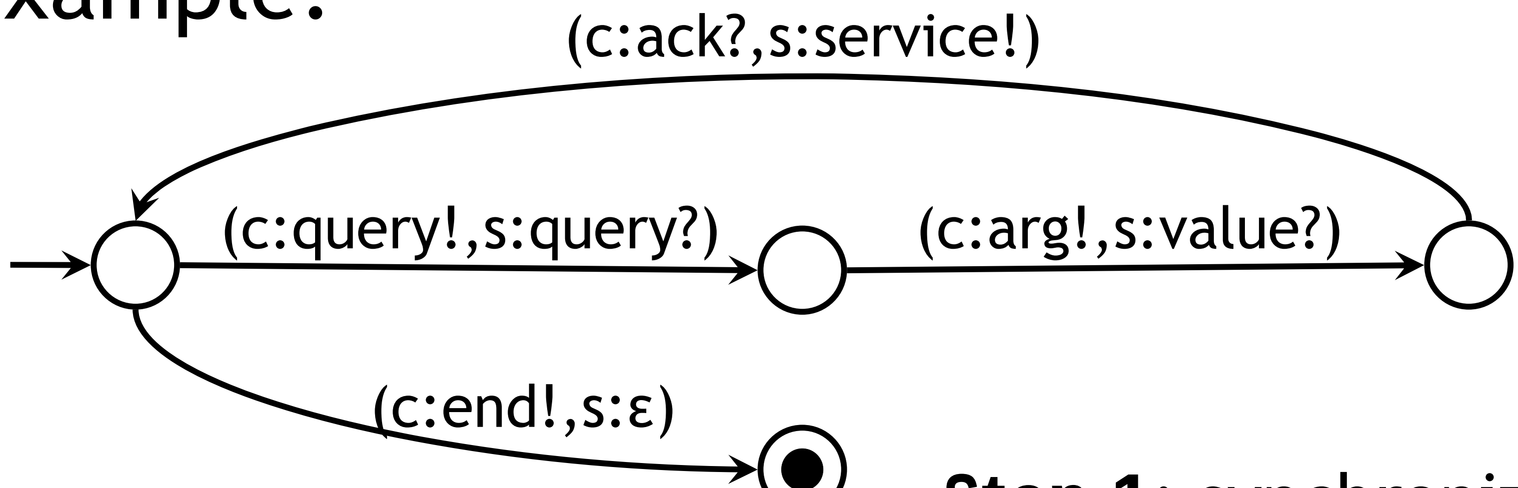
- Vectors:

<c:query!, s:query?> <c:arg!, s:value?>

<c:ack?, s:service!> <c:end!, s:ε>

Adaptation without Reordering

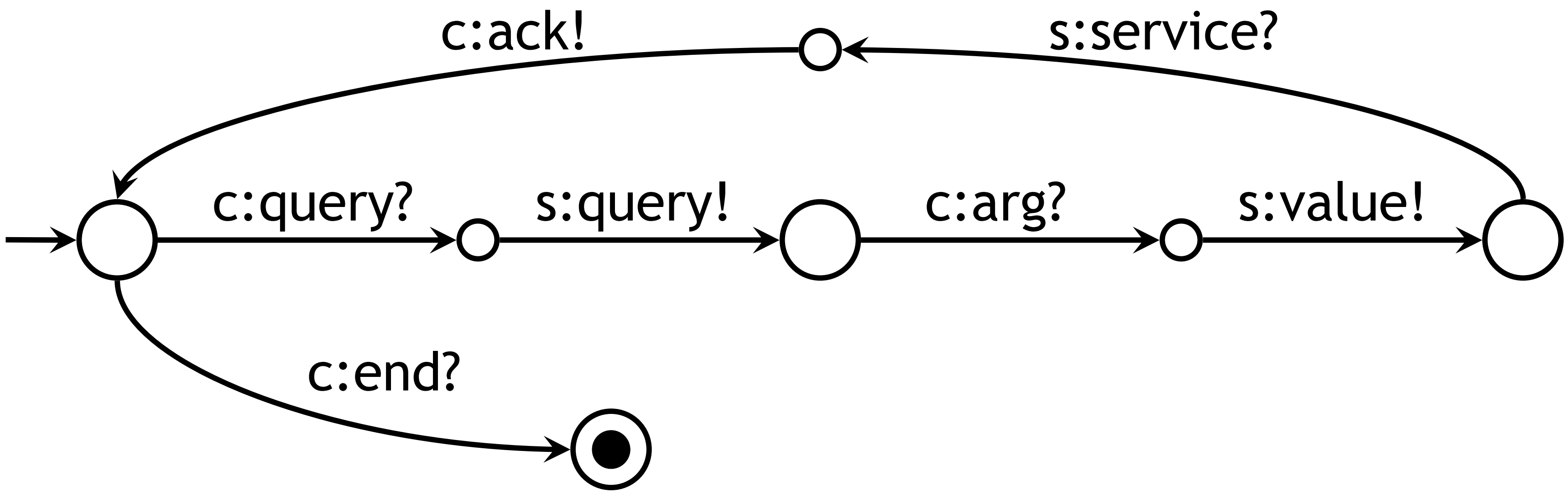
- Algorithm:
  - Compute the synchronized product from LTS L_i and vectors V
  - Remove paths to deadlock
  - Compute permutations and reverse actions
- Example:



Step 1: synchronized product

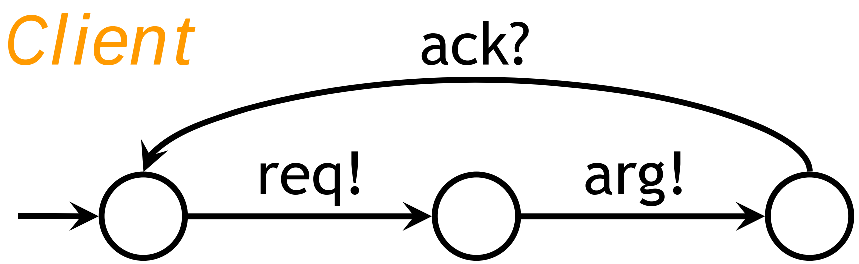
Adaptation without Reordering

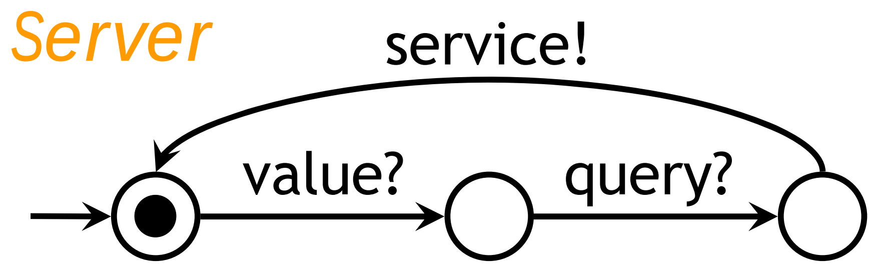
- **Step 2**: no deadlock to remove
- **Step 3**: reverse actions



Adaptation with Reordering

- Algorithm:
  - Encode all LTSs and vectors V into Petri nets
  - Compute marking / cover graph (TINA)
  - Remove paths to deadlock
  - Apply reduction on adaptor to remove τ (CADP)
- Example:





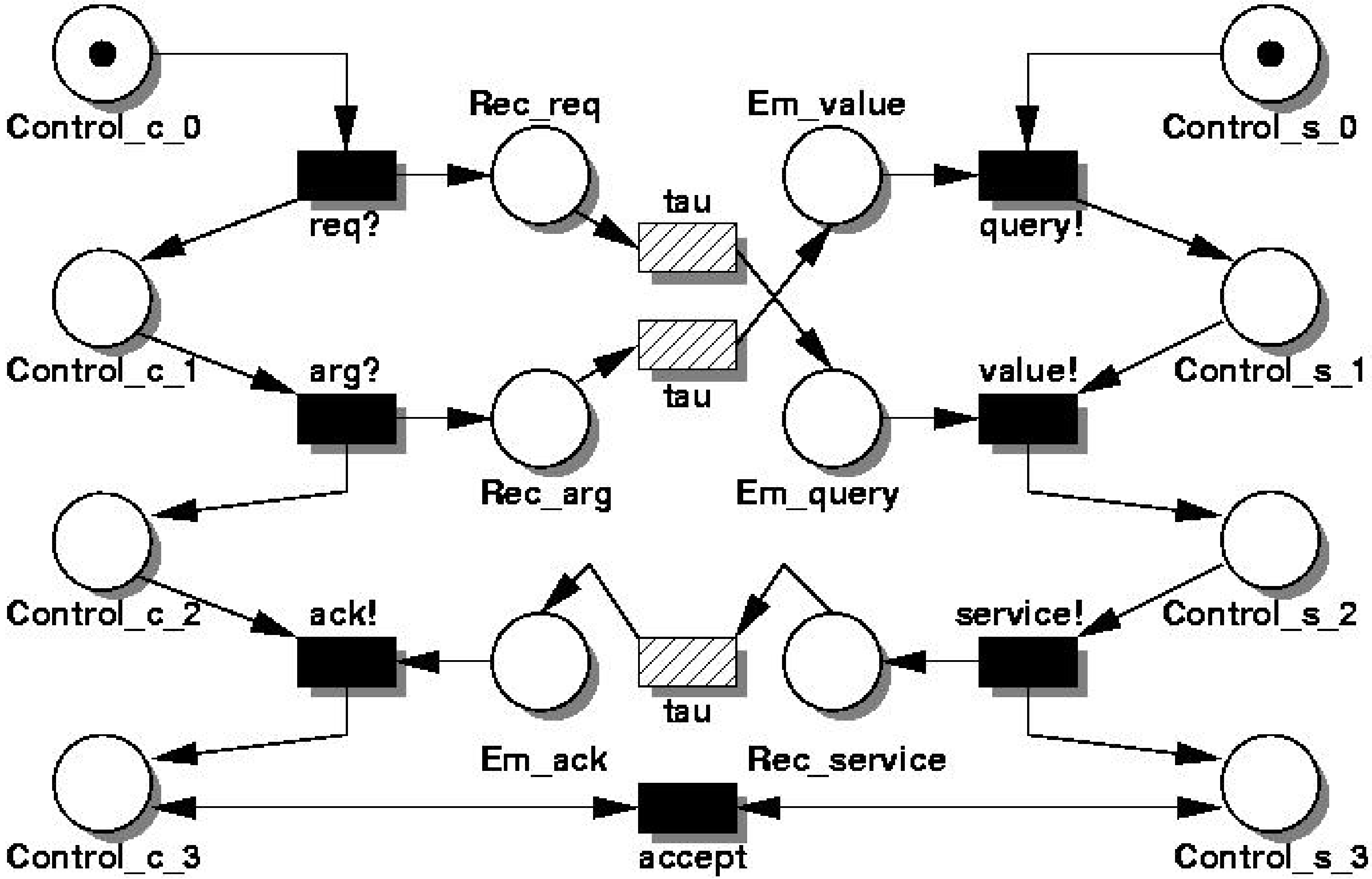
<c:req!, s:query?>

<c:ack?, s:service!>

<c:arg!, s:value?>

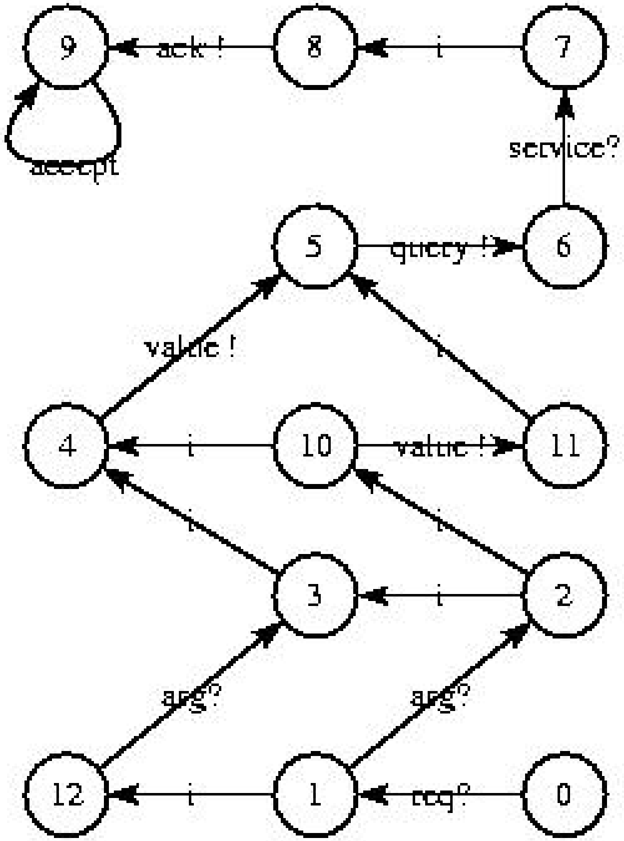
Adaptation with Reordering

- **Step 1**: encoding into Petri nets

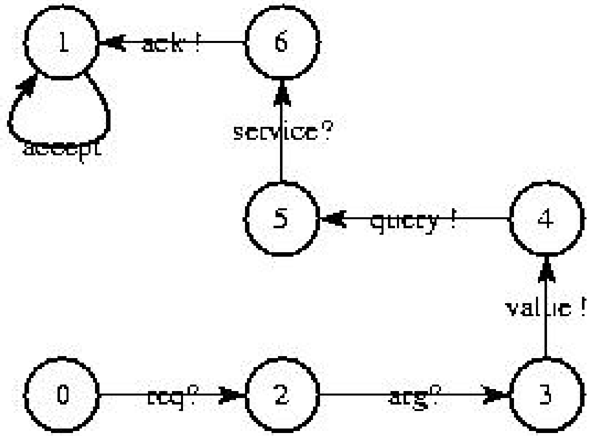


Adaptation with Reordering

- **Step 2**: marking graph computation



- **Step 3**: no deadlock to remove
- **Step 4**: reduction to remove τ actions



Outline of the Talk

- Overview of our Approach
- Adaptation using Vectors
- Adaptation using Regular Expressions
- Conclusion and Future Work
- Demo

Regular Expressions

- Ordering needed: regular expressions of vectors

  R ::= v | R1.R2 *(SEQ)* | R1+R2 *(CH)* | R* *(ITER)*

- Example:

- Vectors:

  v_0=<c:log!, s:log?> v_1=<c:ε, s:log?>

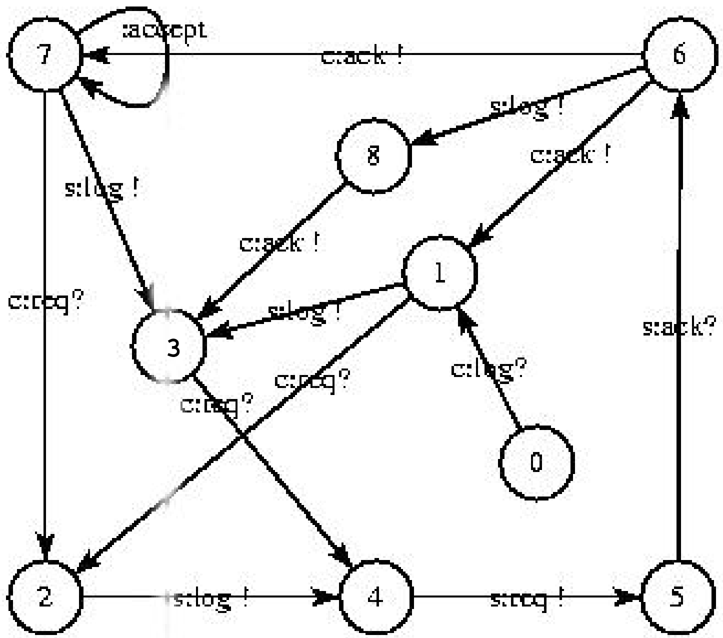
  v_2=<c:req!, s:req?> v_3=<c:ack?, s:ack!>

- Regexp: $v_0.v_2.v_3.(v_1.v_2.v_3)^*$

Adaptation without Reordering

- Algorithm: replace Step 1 of Vectors algo by
  - Compute the LTS L_R for the regexp R
  - Compute the synchronized product from LTS L_i and LTS L_R
  - Discard elements of R in the resulting LTS
- Example:

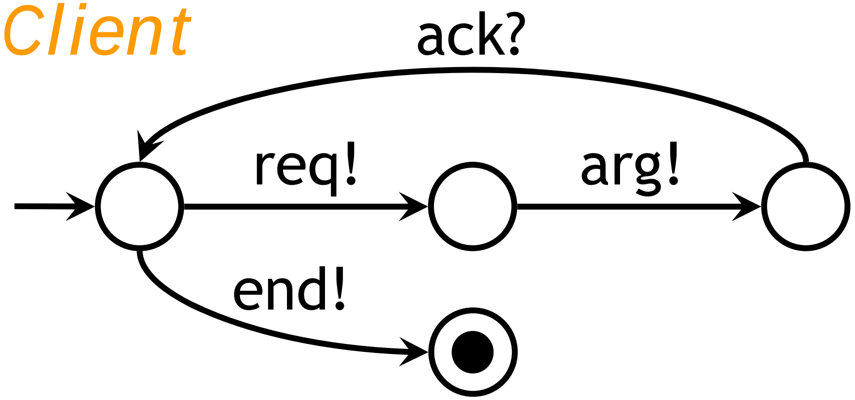
  Abstract adaptor



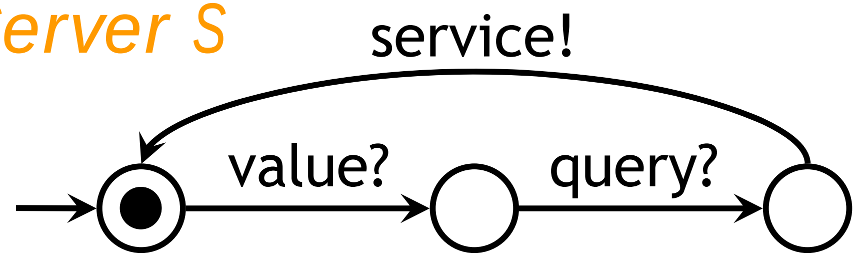
Adaptation with Reordering

- Algorithm: replace Step 1 of Vectors algo by
  - Encode all LTSs and the LTS L_R of the regexp R into Petri nets
- Example:

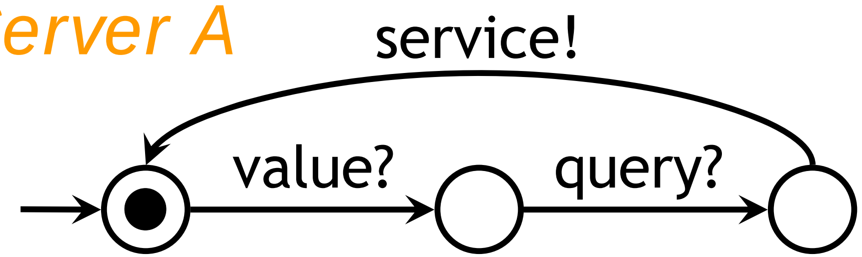
Client



Server S



Server A



v_{s1}=<c:req!, s:query?, a:ε> v_{a1} = <…>

v_{s2}=<c:arg!, s:value?, a:ε> v_{a2} = <…>

v_{s3}=<c:ack?, s:service!, a:ε> v_{a3} = <…>

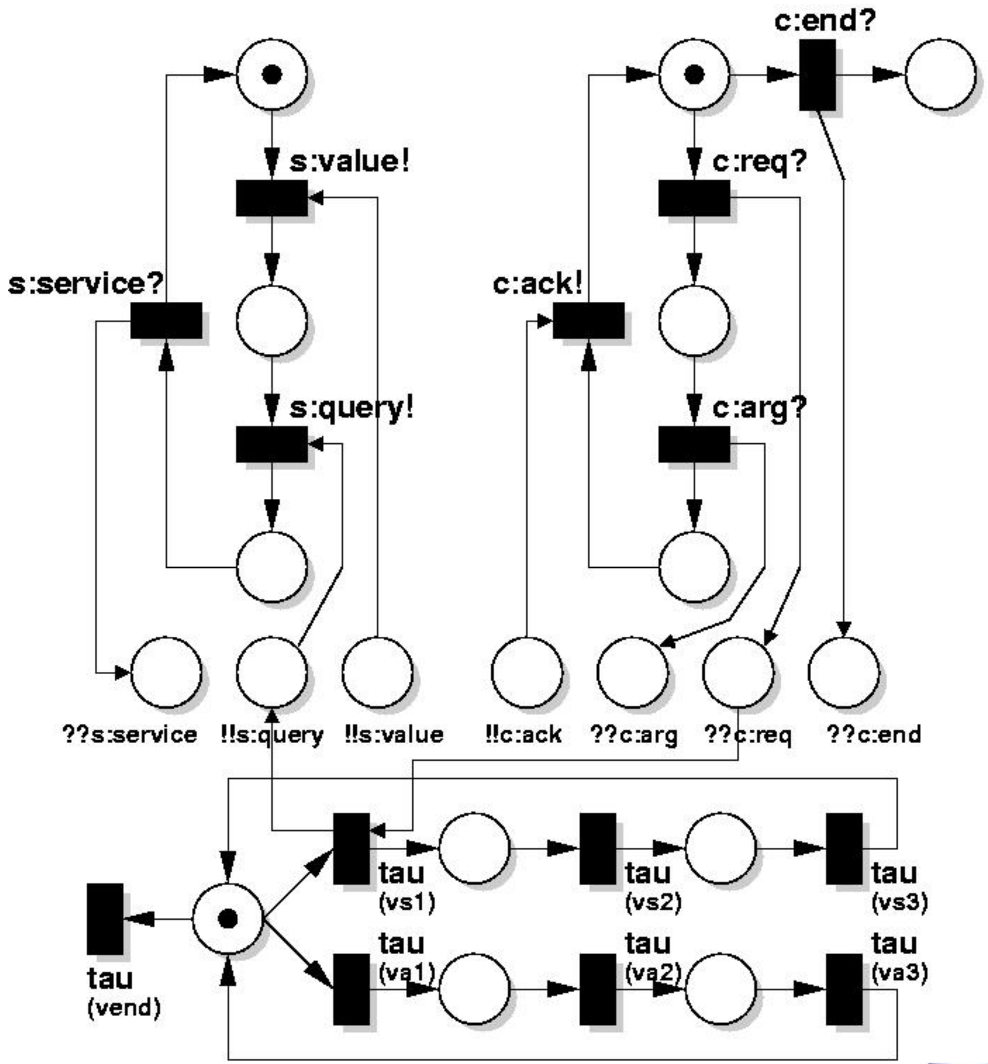
v_{end}= <c:end!, s:ε, a:ε>

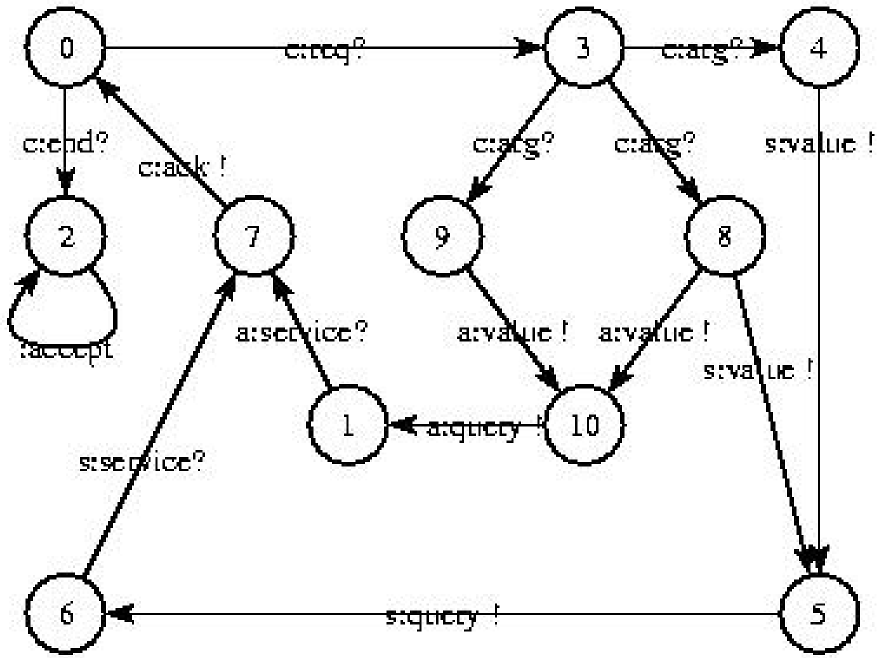
Regexp:

$(v_{s1}.v_{s2}.v_{s3} + v_{a1}.v_{a2}.v_{a3})^*.v_{end}$

Adaptation with Reordering

- Encoding into Petri nets and abstract adaptor





Outline of the Talk

- Overview of our Approach
- Adaptation using Vectors
- Adaptation using Regular Expressions
- Conclusion and Future Work
- Demo

Conclusion

- An approach for component adaptation based on vectors and regular expressions
- Supported by algorithms and tools:
  - CADP: compatibility check, τ-reductions
  - TINA: Petri nets marking and cover graphs
- Fully automated by a prototype (ADAPTOR)

  ⇒ a demo is coming soon!
- Significant insight compared to existing related works…

Comparison with Related Work

Criteria	Inverardi & Tivoli	Brogi et al.	Ours
BIDL	Automata	Proc. algebra	LTS Proc. Algebra
Properties	No deadlock LTL	No deadlock	No deadlock Regexp
Abstraction	Yes	Yes	Yes
Incomp. names	No	Yes	Yes
Data	No	Yes	No
Reordering	No	Yes	Yes
System	Yes	No	Yes
Tools	Yes	No	Yes

Future Work

- Adaptations of messages with data
- Extending our approach with semantic aspects or quality of services (resources, time)
- Automating the generation of mappings
- Formal proof of the algorithms correction
- Connection with implementation component models

Demo

(by Pascal Poizat)