

AIOCJ: a Choreographic Framework for Safe Adaptive Distributed Applications

Mila dalla Preda¹, **Saverio Giallorenzo**², Ivan Lanese², Jacopo Mauro², and Maurizio Gabbrielli²

¹Department of Computer Science - Univ. of Verona

²Department of Computer Science and Engineering - Univ. of Bologna / INRIA

Conference on Software Language Engineering, 2014

Why Choreographic?

Bob

```
msg = "Want to dance?";  
sendMessage: msg to Alice;  
ok: response from Alice
```

Alice

```
sendMessage: msg from Bob;  
response = show( msg );  
ok: response to Bob
```





Why **Choreographic**?

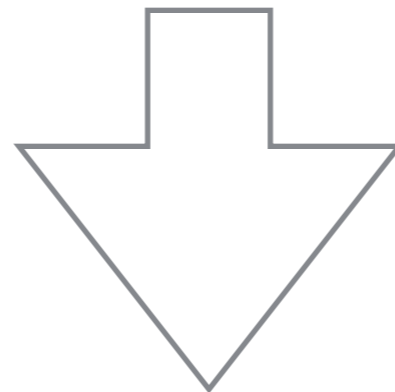
```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );  
response@Alice = show( msg );  
ok: Alice( response ) -> Bob( response )
```

Why **Choreographic**?

```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );
```

```
response@Alice = show( msg );
```

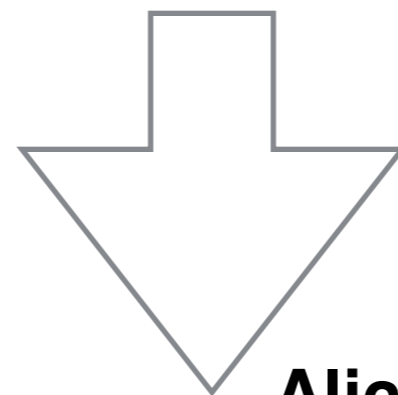
```
ok: Alice( response ) -> Bob( response )
```



Projects to

Why Choreographic?

```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );
response@Alice = show( msg );
ok: Alice( response ) -> Bob( response )
```



Projects to

Bob

```
msg = "Want to dance?";
sendMessage: msg to Alice;
ok: response from Alice
```

Alice

```
sendMessage: msg from Bob;
response = show( msg );
ok: response to Bob
```

Why Choreographic?

```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );  
response@Alice = show( msg );  
ok: Alice( response ) -> Bob( response )
```



Why Choreographic?

```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );  
response@Alice = show( msg );  
ok: Alice( response ) -> Bob( response )
```

**What if we
want to change
(parts of it) at
runtime?**



Safe Adaptive Choreographies

Choreographies are suitable for programming safe distributed systems.

Can we make them suitable for programming **safe** and **adaptable** distributed systems?

AIOCJ is our attempt at giving a positive answer to this question.

Safe Adaptive Choreographies

We deem **AIOCJ** suitable because:

1. It gives a general and neat overview of the (interaction in the) whole system;
2. It injects “good” (desirable) properties on distributed systems;
3. It has proven to be a feasible implementation of formal results. (We ensure “good” properties to hold on the distributed system at runtime and after any step of adaptation).



1. Neat overview

Neat overview | The **AIOC** Language

```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );
```

```
response@Alice = show( msg );
```

```
ok: Alice( response ) -> Bob( response )
```

Neat overview | The **AIOC** Language

```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );
```

```
response@Alice = show( msg );
```

```
ok: Alice( response ) -> Bob( response )
```

- **Interactions** (synchronous);

Neat overview | The **AIOC** Language

```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );
```

```
response@Alice = show( msg );
```

```
ok: Alice( response ) -> Bob( response )
```

- Interactions (synchronous);
- **Local Computation;**

Neat overview | The **AIOC** Language

```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );  
response@Alice = show( msg );  
ok: Alice( response ) -> Bob( response )
```

- Interactions (synchronous);
- Local Computation;
- **Participants;**

Neat overview | The **AIOC** Language

```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );
```

```
response@Alice = show( msg );
```

```
ok: Alice( response ) -> Bob( response )
```

- Interactions (synchronous);
- Local Computation;
- Participants;
- **Operations;**

Neat overview | The **AIOC** Language

```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );
```

```
response@Alice = show( msg );
```

```
ok: Alice( response ) -> Bob( response )
```

- Interactions (synchronous);
- Local Computation;
- Participants;
- Operations;
- **Functions;**

Neat overview | The **AIOc** Language

```
sendMessage: Bob( "Want to dance?" ) -> Alice( msg );
```

```
response@Alice = show( msg );
```

```
ok: Alice( response ) -> Bob( response )
```

- Interactions (synchronous);
- Local Computation;
- Participants;
- Operations;
- Functions;
- **Data.**



1. Neat overview,



1. Neat overview, also when programming adaptation

Neat overview | The **AIOC** Language

Scopes

Neat overview | The **AIOC** Language

A **scope** defines a part of the interaction that can be replaced (adapted).

Neat overview | The **AIOC** Language

A **scope** defines a part of the interaction that can be replaced (adapted).

```
scope @Bob {  
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )  
} prop { N.scopename = "hangout" }
```

- **Scope Declaration;**

Neat overview | The **AIOC** Language

A **scope** defines a part of the interaction that can be replaced (adapted).

```
scope @Bob {
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
} prop { N.scopename = "hangout" }
```

- Scope Declaration;
- **Scope Leader;**

Neat overview | The **AIOC** Language

A **scope** defines a part of the interaction that can be replaced (adapted).

```
scope @Bob {
```

```
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
```

```
} prop { N.scopename = "hangout" }
```

- Scope Declaration;
- Scope Leader;
- **Sub-choreography;**

Neat overview | The **AIOC** Language

A **scope** defines a part of the interaction that can be replaced (adapted).

```
scope @Bob {
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
} prop { N.scopename = "hangout" }
```

- Scope Declaration;
- Leader;
- Sub-choreography;
- **Scope properties;**

Neat overview | The **AIOCI** Language

Neat overview | The **AIOCI** Language

Rules

Neat overview | The **AIOC** Language

A **rule** defines a choreography that can replace a scope.

Neat overview | The **AIOC** Language

A **rule** defines a choreography that can replace a scope.

```
rule {  
  on { N.scopename == "hangout" }  
  do {  
    sendMessage: Bob( "What about Movies?" ) ->  
      Alice( msg )  
  }  
}
```

- **Rule Declaration;**

Neat overview | The **AIOC** Language

A **rule** defines a choreography that can replace a scope.

```
rule {
```

```
on { N.scopename == "hangout" }
```

```
do {
```

```
  sendMessage: Bob( "What about Movies?" ) ->
    Alice( msg )
```

```
}
```

```
}
```

prefixes

N. - properties of the scope;

E. - environmental variables;

non prefixed variables are local to the leader.

- Rule Declaration;
- **Applicability Condition;**

Neat overview | The **AIOC** Language

A **rule** defines a choreography that can replace a scope.

```
rule {
  on { N.scopename == "hangout" }
  do {
    sendMessage: Bob( "What about Movies?" ) ->
      Alice( msg )
  }
}
```

- Rule Declaration;
- Applicability Condition;
- **New Choreography.**

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```
scope @Bob {  
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )  
} prop { N.scopename = "hangout" };  
response@Alice = show( msg );  
ok: Alice( response ) -> Bob( response )
```

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```
scope @Bob {
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
} prop { N.scopename = "hangout" };
response@Alice = show( msg );
ok: Alice( response ) -> Bob( response )
```

```
rule {
  on { N.scopename == "hangout" }
  do {
    sendMessage: Bob( "What about Movies?" ) ->
      Alice( msg )
  }
}
```

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```
scope @Bob {
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
} prop { N.scopename = "hangout" };
```

```
response@Alice = show( msg );
ok: Alice( response ) -> Bob( response )
```

```
rule {
  on { N.scopename == "hangout" }
  do {
    sendMessage: Bob( "What about Movies?" ) ->
      Alice( msg )
  }
}
```

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```
scope @Bob {
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
} prop { N.scopename = "hangout" };
response@Alice = show( msg );
ok: Alice( response ) -> Bob( response )
```

```
rule {
  on { N.scopename == "hangout" }
  do {
    sendMessage: Bob( "What about Movies?" ) ->
      Alice( msg )
  }
}
```

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```
scope @Bob {
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
} prop { N.scopename = "hangout" };
response@Alice = show( msg );
ok: Alice( response ) -> Bob( response )
```

```
rule {
  on { N.scopename == "hangout" }
  do {
    sendMessage: Bob( "What about Movies?" ) ->
      Alice( msg )
  }
}
```

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```

scope @Bob {
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
} prop { N.scopename = "hangout" };
response@Alice = show( msg );
ok: Alice( response ) -> Bob( response )

rule {
  on { N.scopename == "hangout" }
  do {
    sendMessage: Bob( "What about Movies?" ) ->
      Alice( msg )
  }
}

```


Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```

scope @Bob {
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
} prop { N.scopename = "hangout" };
response@Alice = show( msg );
ok: Alice( response ) -> Bob( response )

rule {
  on { N.scopename == "hangout" } ✓
  do {
    sendMessage: Bob( "What about Movies?" ) ->
      Alice( msg )
  }
}

```

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```
scope @Bob {
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
} prop { N.scopename = "hangout" };
response@Alice = show( msg );
ok: Alice( response ) -> Bob( response )
```

```
rule {
  on { N.scopename == "hangout" }
  do {
    sendMessage: Bob( "What about Movies?" ) ->
      Alice( msg )
  }
}
```

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```
scope @Bob {
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
} prop { N.scopename = "hangout" };
```

```
response@Alice = show( msg );
ok: Alice( response ) -> Bob( response )
```

```
rule {
  on { N.scopename == "hangout" }
  do {
```

```
  sendMessage: Bob( "What about Movies?" ) ->
    Alice( msg )
```

```
}
}
```

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```
scope @Bob {
  sendMessage: Bob( "Want to dance?" ) -> Alice( msg )
} prop { N.scopename = "hangout" };
```

```
response@Alice = show( msg );
ok: Alice( response ) -> Bob( response )
```

```
rule {
  on { N.scopename == "hangout" }
  do {
```

```
sendMessage: Bob( "What about Movies?" ) ->
  Alice( msg )
```

```
}
}
```

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```
sendMessage: Bob( "What about Movies?" ) -> Alice( msg );
```

```
response@Alice = show( msg );
```

```
ok: Alice( response ) -> Bob( response )
```

```
rule {
```

```
  on { N.scopename == "hangout" }
```

```
  do {
```

```
    sendMessage: Bob( "What about Movies?" ) ->
      Alice( msg )
```

```
  }
```

```
}
```

Neat overview | The **AIOC** Language

What happens at runtime? Easy to figure out.

```
sendMessage: Bob( "What about Movies?" ) -> Alice( msg );
```

```
response@Alice = show( msg );
```

```
ok: Alice( response ) -> Bob( response )
```



2. “Good” properties

“Good” Properties
|
Deadlock- and Race-freedom by
construction.

“Good” properties | **deadlock- and race-freedom**

Choreographies are deadlock- and race-free by construction.

1. **Interactions are atomic**
2. We enforce **well-formed choreographies** both in AIOCJ programs and rules.
3. **Correctness of projection**

“Good” Properties II Consistency of Adaptation

“Good” properties | **consistency of adaptation**

When a scope of an AIOCJ program adapts, the adaptation is **consistent** among the participants.

“Good” properties | **consistency of adaptation**


When a scope of an AIOCJ program adapts, the adaptation is **consistent** among the participants.

```
scope @Bob {  
    sendMessage: Bob( a ) -> Alice( b )  
} prop { N.scopename = “hangout” }
```

“Good” properties | **consistency of adaptation**

When a scope of an AIOCJ program adapts, the adaptation is **consistent** among the participants.

```
scope @Bob {  
  sendMessage: Bob( a ) -> Alice( b )  
} prop { N.scopename = "hangout" }
```

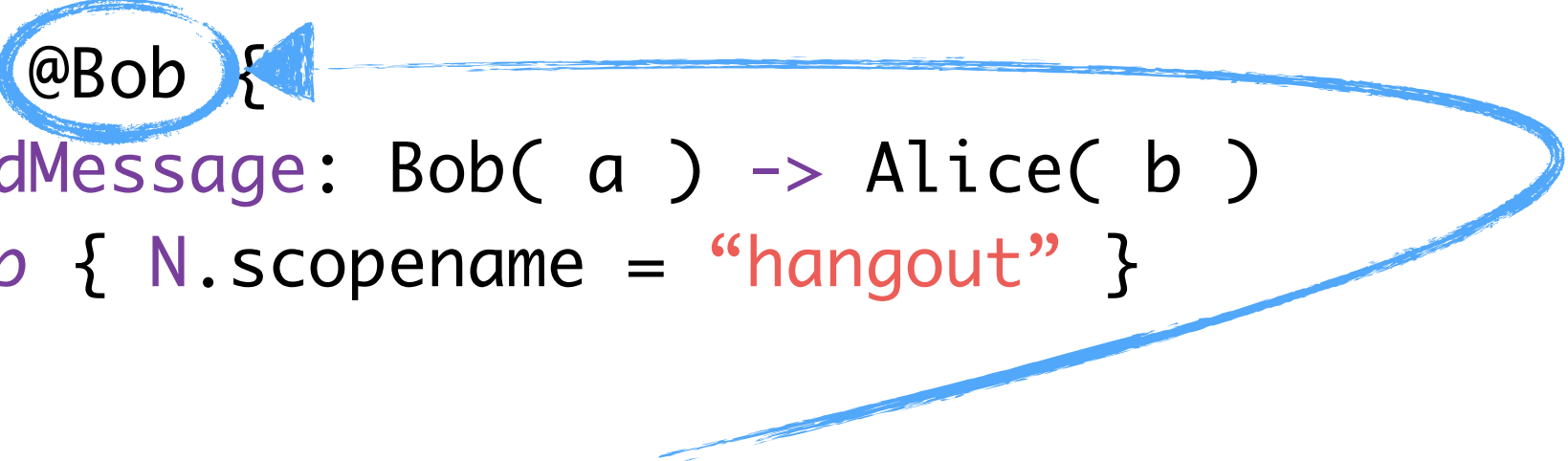


Bob is the **leader** of this scope of adaptation.

“Good” properties | **consistency of adaptation**

When a scope of an AIOCJ program adapts, the adaptation is **consistent** among the participants.

```
scope @Bob {
  sendMessage: Bob( a ) -> Alice( b )
} prop { N.scopename = "hangout" }
```

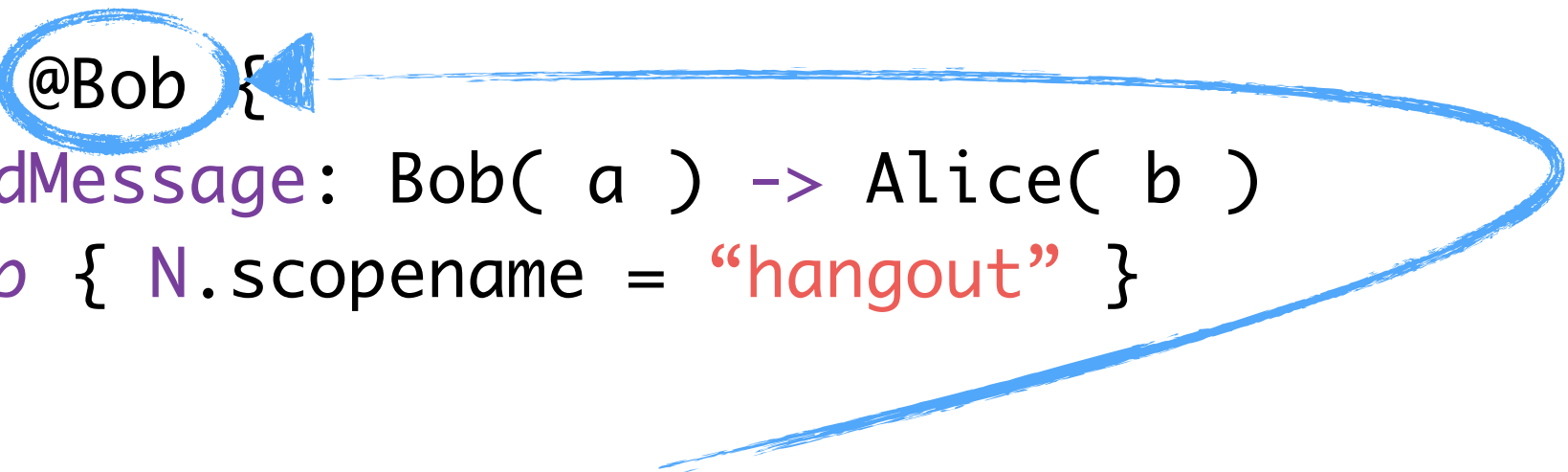


Bob is the **leader** of this scope of adaptation.
Only Bob can query the repositories of rules.

“Good” properties | **consistency of adaptation**

When a scope of an AIOCJ program adapts, the adaptation is **consistent** among the participants.

```
scope @Bob {
  sendMessage: Bob( a ) -> Alice( b )
} prop { N.scopename = "hangout" }
```

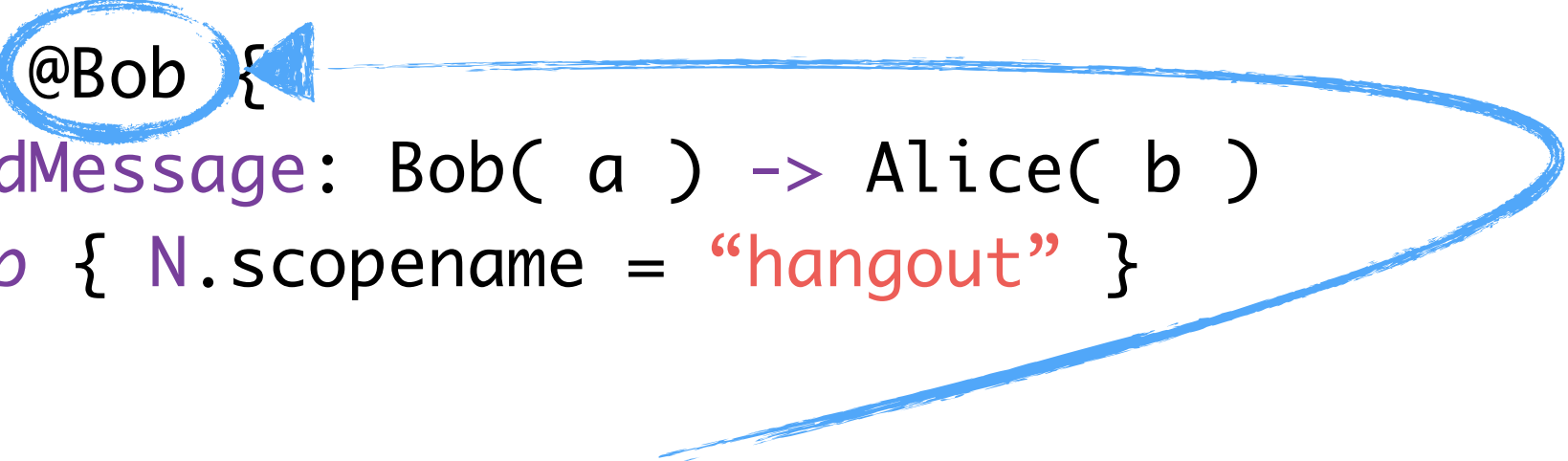


Bob is the **leader** of this scope of adaptation. Only Bob can query the repositories of rules. He decides whether to adapt and which rule applies.

“Good” properties | **consistency of adaptation**

When a scope of an AIOCJ program adapts, the adaptation is **consistent** among the participants.

```
scope @Bob {
  sendMessage: Bob( a ) -> Alice( b )
} prop { N.scopename = "hangout" }
```



Bob is the **leader** of this scope of adaptation. Only Bob can query the repositories of rules. He decides whether to adapt and which rule applies.

This allows rules to change at runtime!

The screenshot shows an IDE window with a Project Explorer on the left and a code editor on the right. The Project Explorer shows a folder named 'AIOCJ_Demo' containing two files: 'AIOCJ_example_rule.ioc' and 'AIOCJ_example.ioc'. The code editor displays the content of 'AIOCJ_example.ioc' with line numbers 1 through 29. The code is as follows:

```

1 include isFreeDay from "socket://localhost:8000"
2 include getTicket from "socket://localhost:8001"
3
4 preamble { starter: bob }
5
6 aioc {
7   end@bob = false;
8
9   while( !end )@bob{
10
11     scope @bob {
12       free_day@bob = getInput( "Insert your free day" );
13       proposal: bob( free_day ) -> alice( bob_free_day );
14       is_free@alice = isFreeDay( bob_free_day )
15     } prop { N.scope_name = "matching day"};
16
17     if( is_free )@alice {
18       scope @bob {
19         proposal: bob( "cinema" ) -> alice( event );
20         agreement@alice = getInput( "Bob proposes " + event +
21           ", do you agree?[y/n]");
22         if( agreement == "y" )@alice{
23           end@bob = true;
24           book: bob( bob_free_day ) -> cinema( book_day );
25           ticket@cinema = getTicket( book_day );
26           { notify: cinema( ticket ) -> bob( ticket )
27             | notify: cinema( ticket ) -> alice( ticket ) }
28         }
29       } prop { N.scope_name = "event selection" }

```

3. Feasible

Website: <http://bit.do/aiocj>

Feasible | The AIOCJ Framework

AIOCJ-ecl.

Plug-in for Eclipse.

Provides:

- syntax highlighting;
- syntax checking;
- online correctness checking;

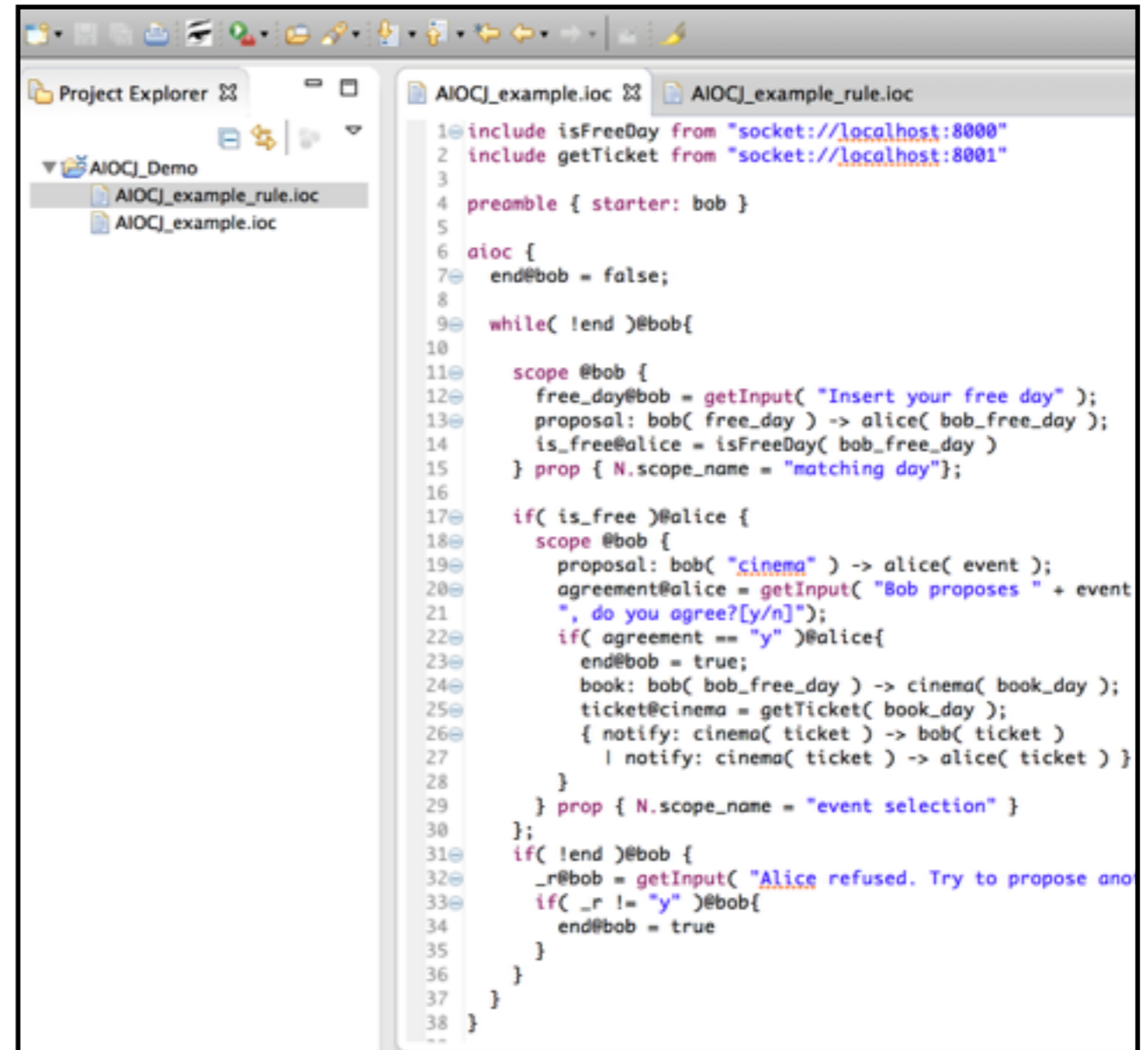
- Projection to



Jolie

www.jolie-lang.org

Website: <http://bit.do/aiocj>



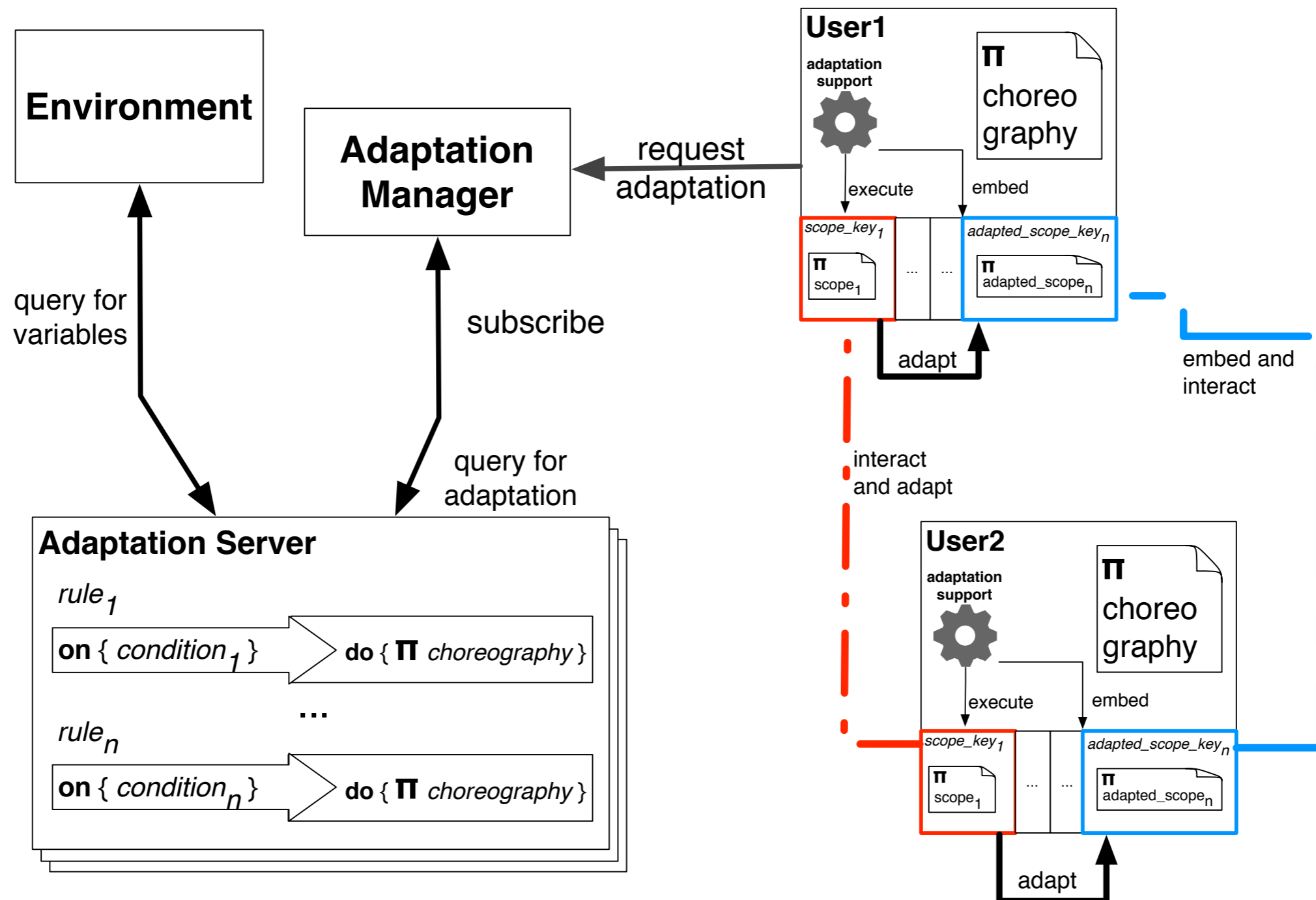
```

1 include isFreeDay from "socket://localhost:8000"
2 include getTicket from "socket://localhost:8001"
3
4 preamble { starter: bob }
5
6 aioc {
7   end@bob = false;
8
9   while( !end )@bob{
10
11     scope @bob {
12       free_day@bob = getInput( "Insert your free day" );
13       proposal: bob( free_day ) -> alice( bob_free_day );
14       is_free@alice = isFreeDay( bob_free_day );
15     } prop { N.scope_name = "matching day";
16
17     if( is_free )@alice {
18       scope @bob {
19         proposal: bob( "cinema" ) -> alice( event );
20         agreement@alice = getInput( "Bob proposes " + event
21           ", do you agree?[y/n]");
22         if( agreement == "y" )@alice{
23           end@bob = true;
24           book: bob( bob_free_day ) -> cinema( book_day );
25           ticket@cinema = getTicket( book_day );
26           { notify: cinema( ticket ) -> bob( ticket )
27             | notify: cinema( ticket ) -> alice( ticket ) }
28         }
29       } prop { N.scope_name = "event selection" }
30     };
31     if( !end )@bob {
32       _r@bob = getInput( "Alice refused. Try to propose ano
33       if( _r != "y" )@bob{
34         end@bob = true
35       }
36     }
37   }
38 }

```

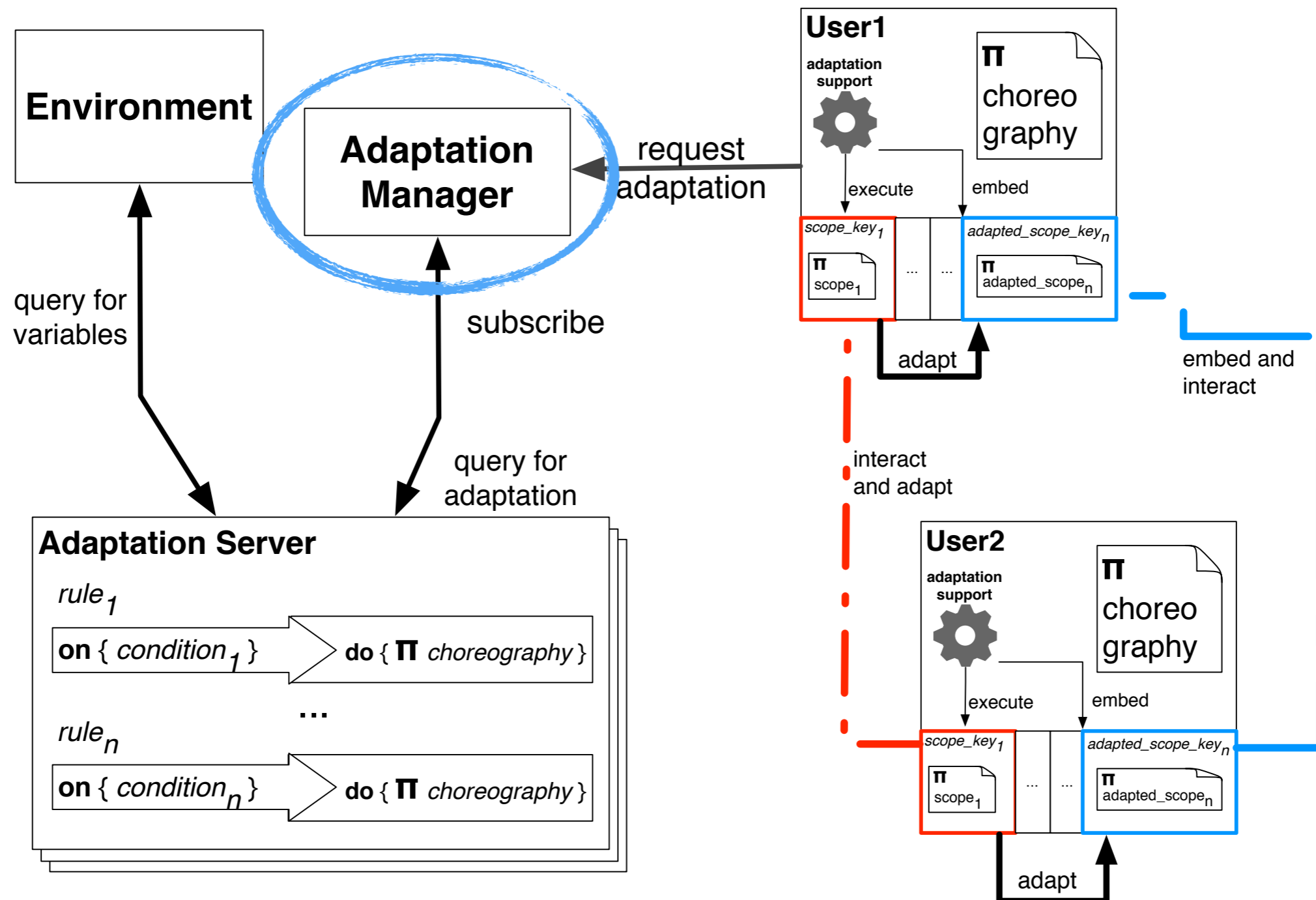
Feasible | The AIOCJ Framework

AIOCJ-mid



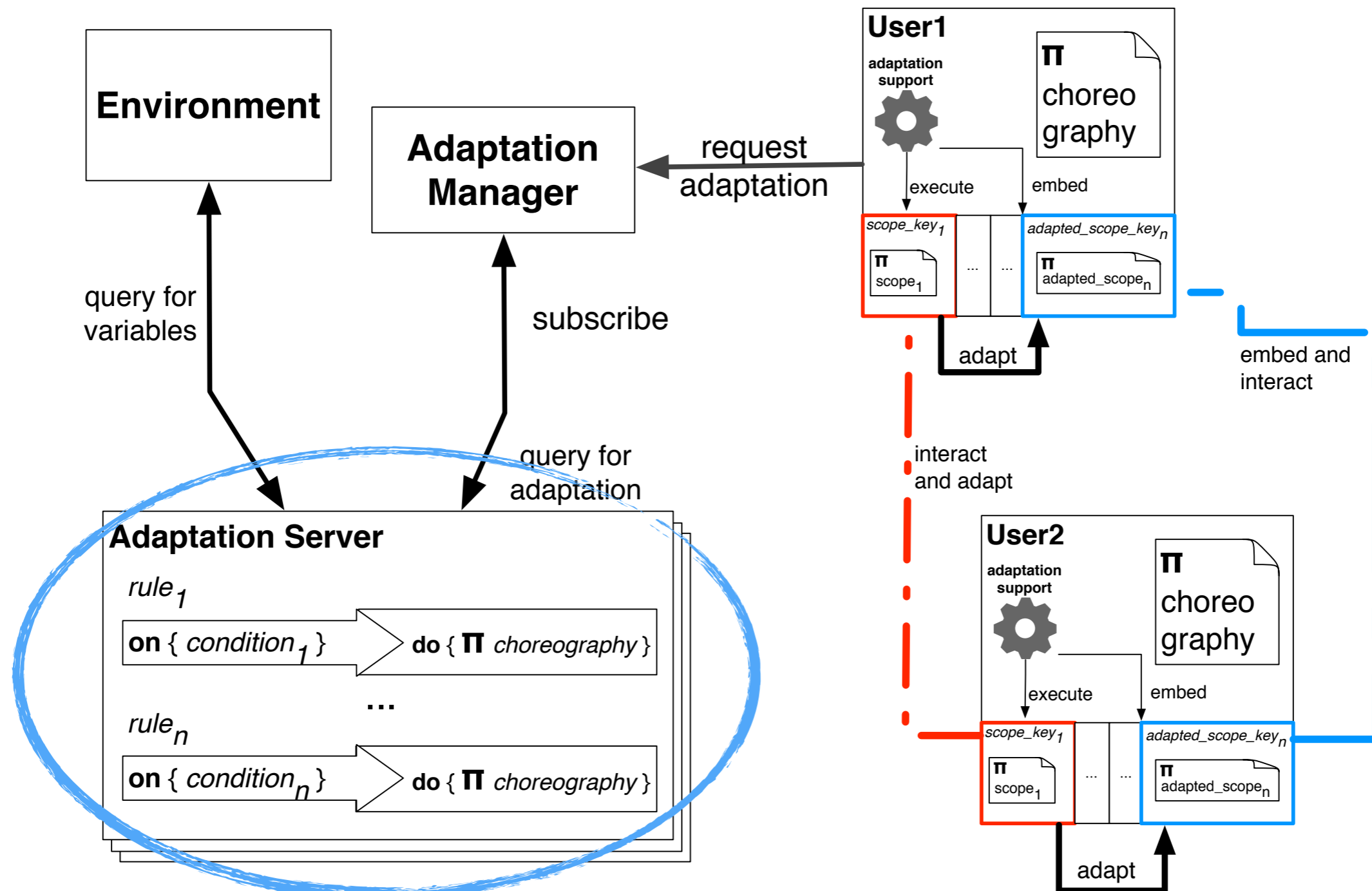
Feasible | The AIOCJ Framework

AIOCJ-mid



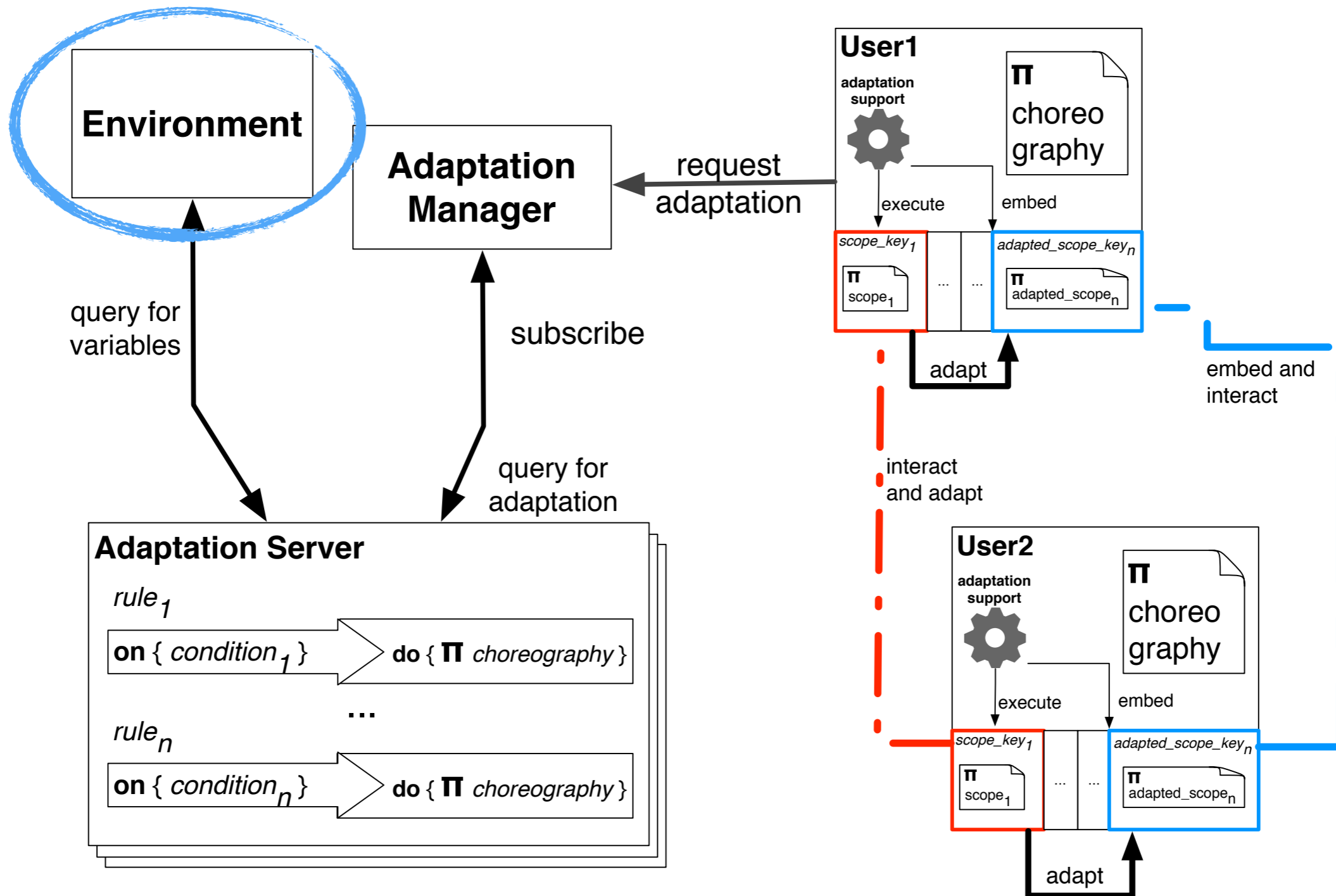
Feasible | The AIOCJ Framework

AIOCJ-mid



Feasible | The AIOCJ Framework

AIOCJ-mid





Demonstration

Safe Adaptive Choreographies | RECAP

Choreographies are suitable for programming safe distributed systems.

With **AIOCJ**, we made a first attempt at making them suitable for programming **safe** and **adaptable** distributed systems.

Website: <http://bit.do/aiocj>

Safe Adaptive Choreographies | RECAP

Main features of **AIOCJ**:

1. It gives a general and neat overview of the (interaction in the) whole system;
2. It injects “good” (desirable) properties on distributed systems;
3. It has proven to be a feasible implementation of formal results. (We ensure “good” properties to hold on the distributed system at runtime and after any step of adaptation).

Future Work

What is still missing?

- Communications in AIOCJ are synchronous. We are planning to include also **asynchronous** communications;
- Sessions;
- Injection of AIOCJ “good” properties in other adaptation mechanisms. E.g., Aspect-Oriented or Context-Oriented Programming, etc...

Thanks for your time

A blackboard with white chalk writing. The text reads "ANY QUESTIONS?" in a casual, handwritten style. The word "ANY" is on the top line, "QUESTIONS" is on the second line, and a question mark is on the third line.

Want to discuss offline?
Please, contact me at:
sgiallor@cs.unibo.it

AIOCJ

Framework for Safe Adaptive Distributed Applications

Mila dalla Preda

¹Department of Computer Science - Univ. of Verona

²Department of Computer Science and Engineering - Univ. of Bologna / INRIA

Conference on Software Language Engineering, 2014

Appendix

Connectedness Properties

“Good” properties | **deadlock- and race-freedom**

Choreographies are deadlock-free by construction.

Connectedness for sequence

`op1: Bob(a) -> Alice(b);`

`op2: Alice(b) -> Bob(c);`

“Good” properties | **deadlock- and race-freedom**

Choreographies are deadlock-free by construction.

Connectedness for sequence

`op1: Bob(a) -> Alice(b);`

`op2: Alice(b) -> Bob(c);`

`op3: Carol(d) -> Dave(e)`

“Good” properties | **deadlock- and race-freedom**

Choreographies are deadlock-free by construction.

Connectedness for sequence

`op1: Bob(a) -> Alice(b);`

`op2: Alice(b) -> Bob(c);`

`op3: Carol(d) -> Dave(e);`



No causality
relation between
either Alice, Bob,
Carol or Dave

“Good” properties | **deadlock- and race-freedom**

Choreographies are deadlock-free by construction.

Connectedness for sequence

op1: Bob(a) -> Alice(b);

op2: Alice(b) -> Bob(c)

op3: Carol(d) -> Dave(e)



A natural enforcement.

Probably the programmer wanted the last two instructions to run in parallel

“Good” properties | **deadlock- and race-freedom**

Choreographies are deadlock-free by construction.

Connectedness for parallel

op1: Bob(a) -> Alice(c)

op1: Bob(b) -> Alice(d)

There might be interference between these interactions.

Interactions with the same signature (operation, sender, receiver) in parallel are forbidden.

AIOC Language Syntax

AIOC Program Syntax

$$\begin{array}{lcl}
 \mathcal{C} & ::= & \text{Include}^* \\
 & & \text{Preamble} \\
 & & \text{aioc } \{ \mathcal{I} \} \\
 \text{Include} & ::= & \text{include } f^+ \text{ from Location} \\
 \text{Preamble} & ::= & \text{preamble } \{ \\
 & & \quad \text{starter} : r \\
 & & \quad \text{Deployment}^* \\
 & & \} \\
 \text{Deployment} & ::= & \text{location}@r : \text{Location}
 \end{array}$$

AIOC Behaviour Syntax

$$\begin{array}{l}
 \mathcal{I} ::= o^? : r_1(e) \rightarrow r_2(x) \quad | \quad \mathcal{I};\mathcal{I}' \quad | \quad \mathcal{I}|\mathcal{I}' \\
 \quad | \quad x@r = local \quad | \quad skip \quad | \quad while \ b@r \ \{\mathcal{I}\} \\
 \quad | \quad if \ b@r \ \{\mathcal{I}\} \ else \ \{\mathcal{I}'\} \\
 \quad | \quad scope \ @r \ \{\mathcal{I}\} \\
 \quad [\ prop \ \{\text{list of } N.x = e\}] \\
 \quad [\ roles \ \{r_i, \dots, r_j\}]
 \end{array}$$

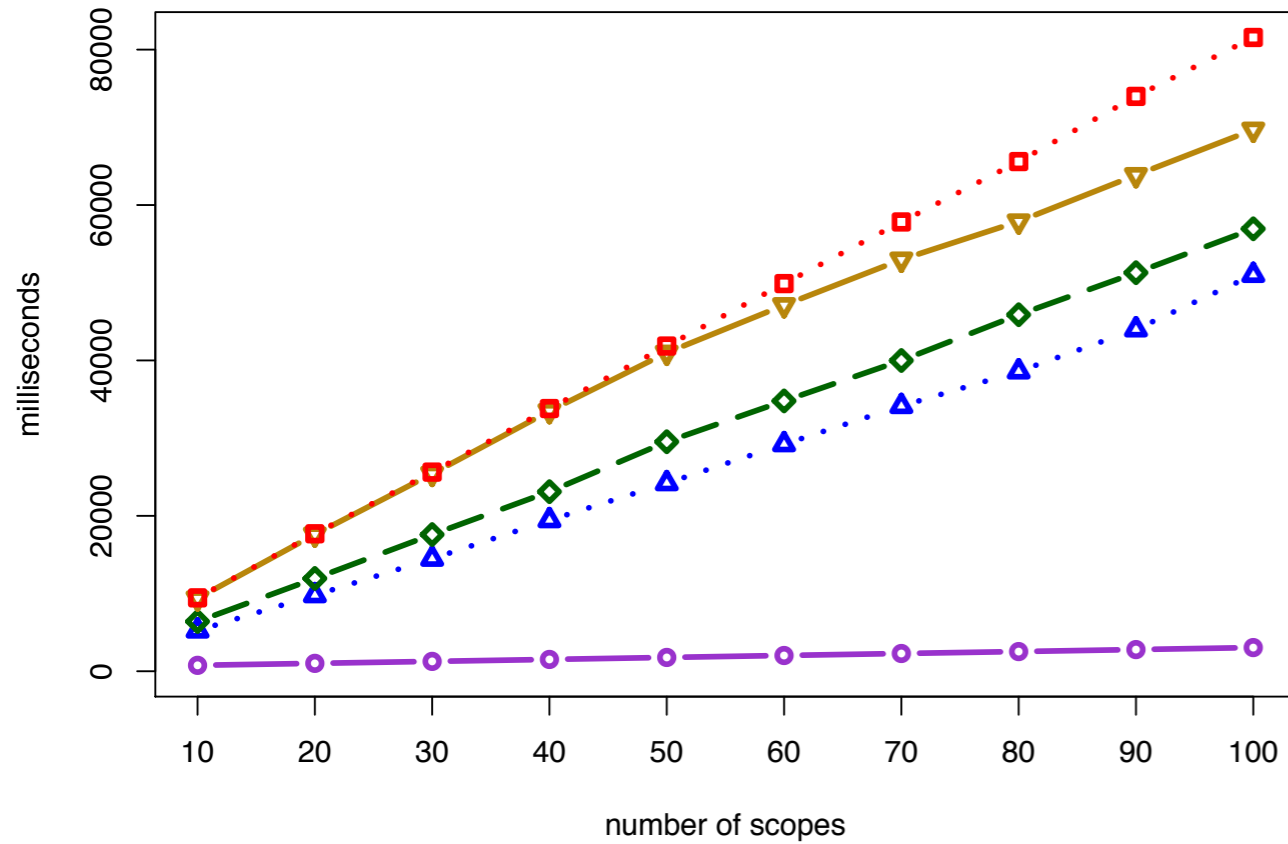
$$local ::= e \quad | \quad f \quad | \quad getInput(x) \quad | \quad show(x)$$

Rules Syntax

$$\mathcal{R} ::= \text{rule } \{$$
$$\quad \text{Include}^*$$
$$\quad \text{on } \{ \mathcal{B} \}$$
$$\quad \text{do } \{ \mathcal{I} \}$$
$$\quad \}$$

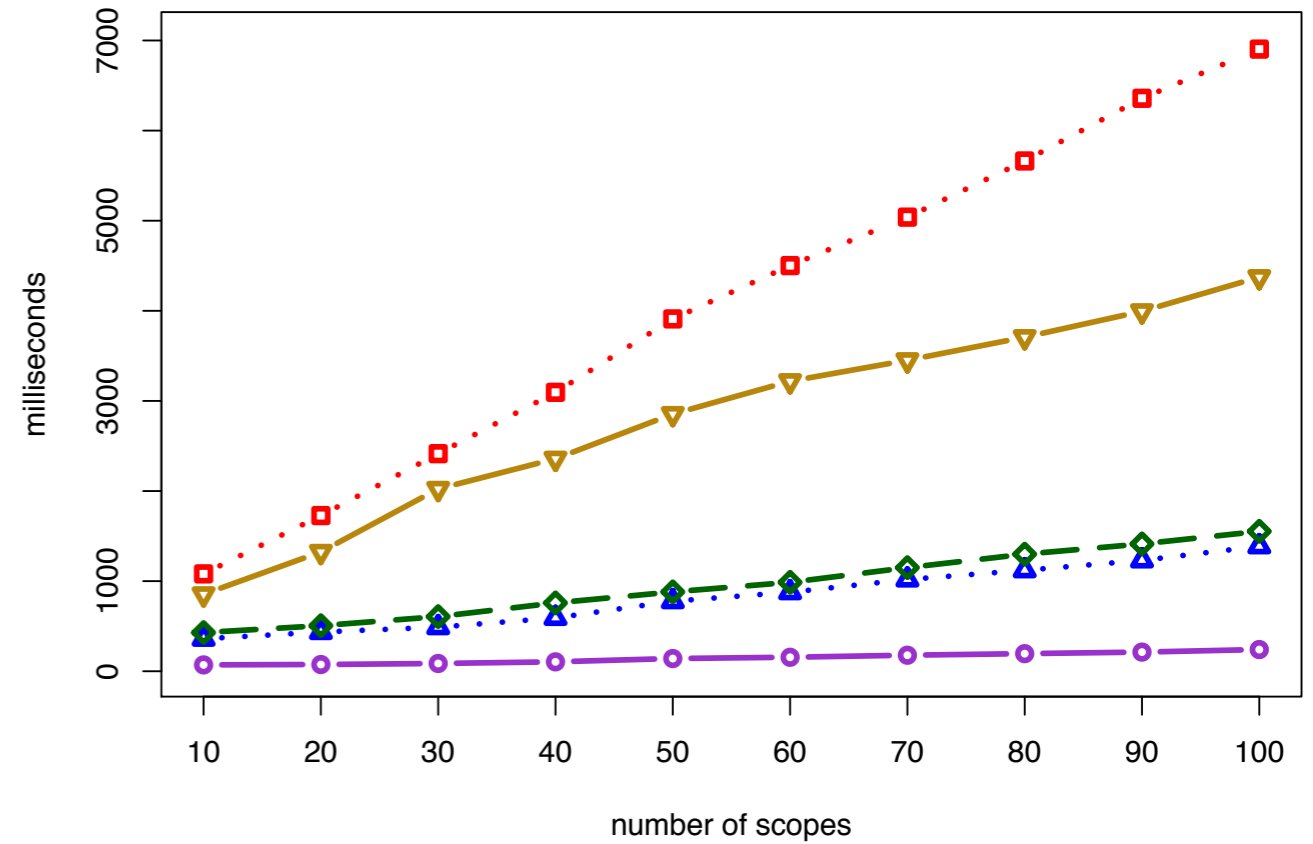
Performances

Pipe



- C1: no scopes
- △· C2: scopes, no adaptation server
- ◇— C3: scopes, 1 adaptation server, no rules

Fork-Join



- ▽— C4: scopes, 1 adaptation server, 50 rules
- C5: scopes, 1 adaptation server, 100 rules