

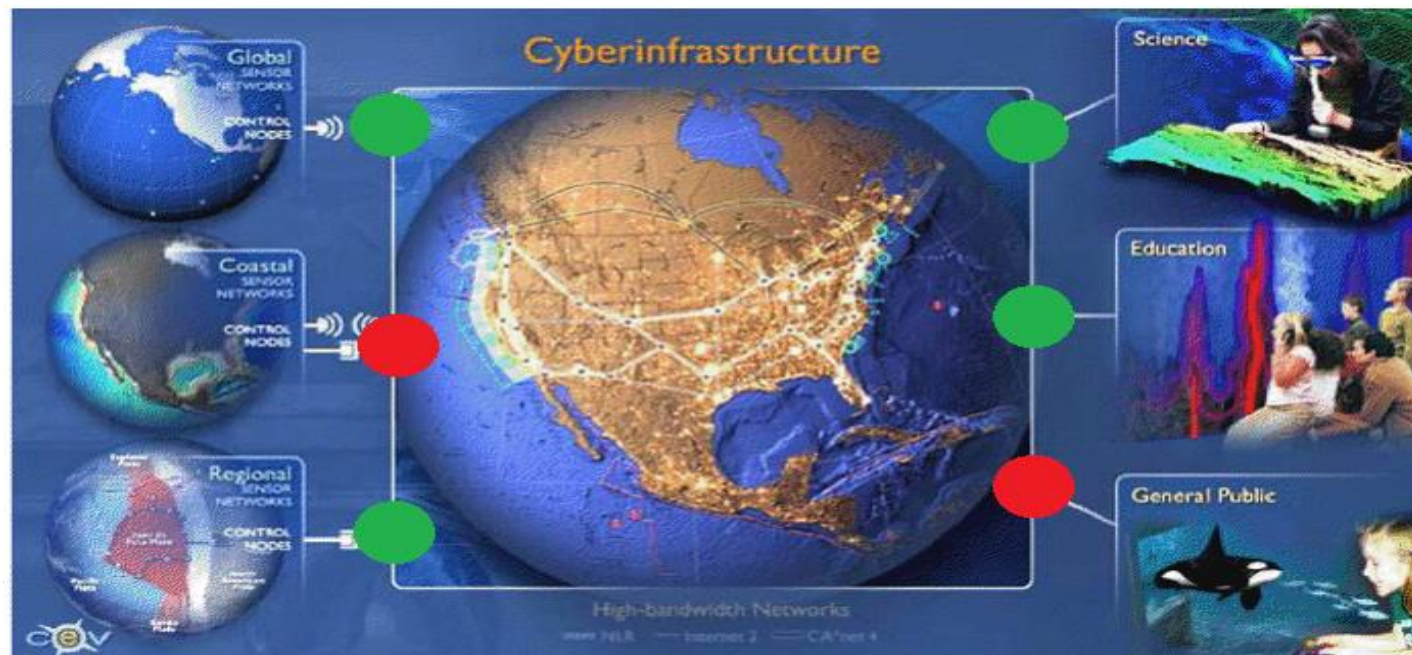


# Multiparty Session Types for Runtime Verification

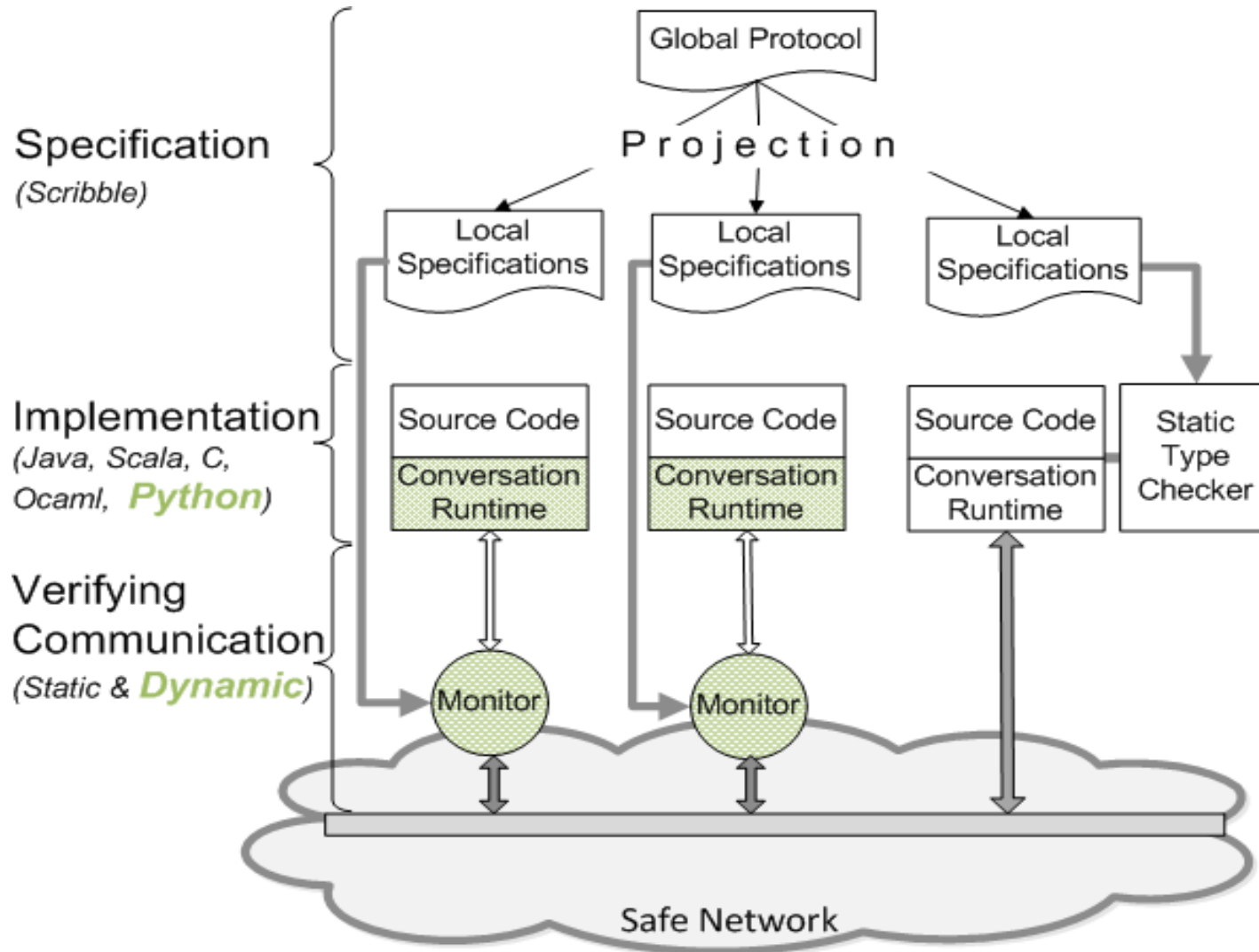
Rumyana Neykova

# OOI: verification challenges

- ▶ applications written in **different** languages, running on **heterogeneous** hardware in an **asynchronous** network.
- ▶ different authentication domains, external **untrusted** applications based on various application protocols
- ▶ requires correct, safe interactions



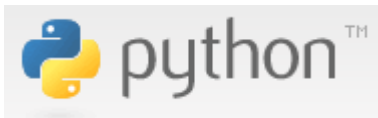
# MPST for Runtime Verification



# The players

---

- ▶ Specify a global protocol (**in Scribble**) that describes how the communication (use case) will proceed
- ▶ Implement processes according to the protocol (using **conversation API in Python**)
- ▶ The communication is intercepted by a monitor
- ▶ The processes are verified dynamically (**monitoring**) against the protocol



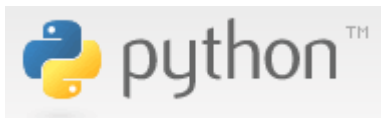
# 3-level Verification

---

1. Writing correct global protocols with **Scribble Compiler**

2. Verify programs via *local monitors*

3. Build additional verification modules via *annotations*



---

1. Writing correct global protocols with **Scribble Compiler**

2. Verify programs via *local monitors*

3. Build additional verification modules via *annotations*



# Scribble Toolchain

# Scribble

## Protocol Language



*"Scribbling is necessary for architects, either physical or computing, since all great ideas of architectural construction come from that unconscious moment, when you do not realise what it is, when there is no concrete shape, only a whisper which is not a whisper, an image which is not an image, somehow it starts to urge you in your mind, in so small a voice but how persistent it is, at that point you start scribbling." Kohei Honda 2007.*

## What is Scribble?

Scribble is a language to describe application-level protocols among communicating systems. A protocol represents an agreement on how participating systems interact with each other. Without a protocol, it is hard to do a meaningful interaction: participants simply cannot communicate effectively, since they do not know when to expect the other parties to send their data, or whether the other party is ready to receive a datum it is sending. In fact it is not clear what kinds of data is to be used for each interaction. It is too costly to carry out communications based on guess works and with inevitable communication mismatch (synchronisation bugs). Simply, it is not feasible as an engineering practice.

## Documents

> [Protocol Language Guide](#)

## Downloads

> [Java Tools](#)

## Community

> [Discussion Forum](#)

> [Java Tools](#)

[Issues](#)

[Wiki](#)

> [Python Tools](#)

[Issues](#)

[Wiki](#)

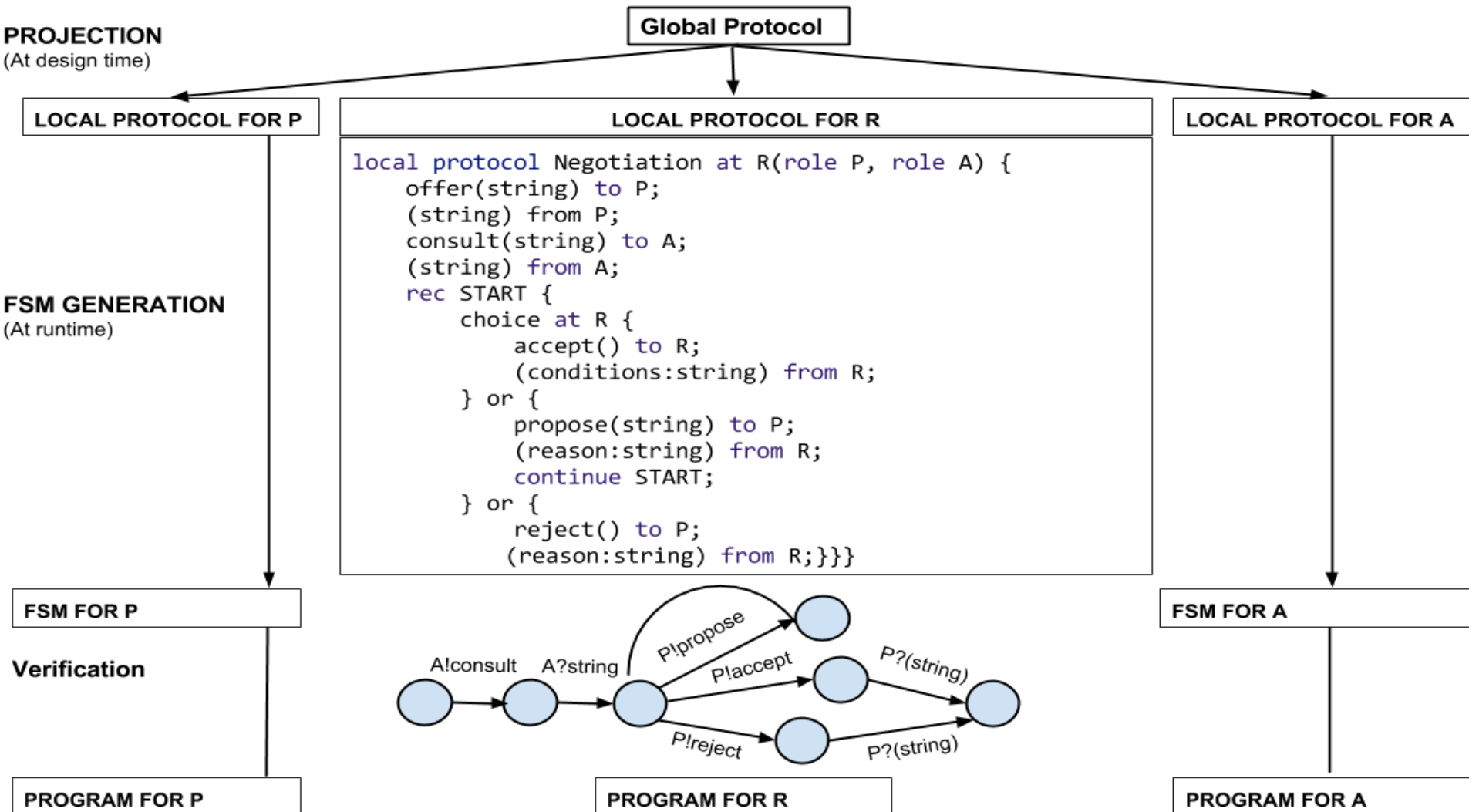
1. Writing correct global protocols with **Scribble Compiler**

2. Verify programs via *local monitors*

3. Build additional verification modules via *annotations*



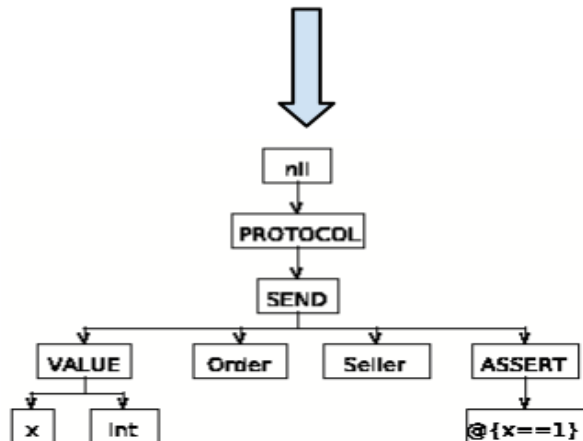
# Local Protocol Conformance



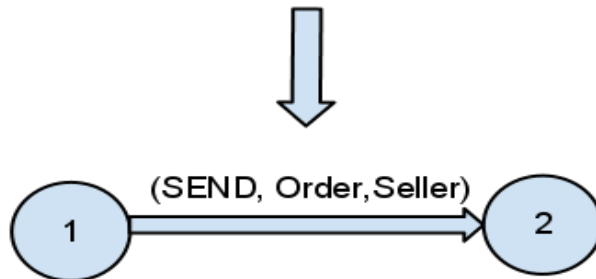
# FSM Generator

**Scribble:** Order(x:int) to Seller @ {x==1}

**AST:**

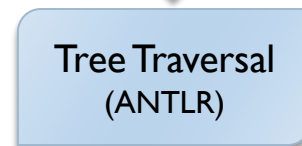
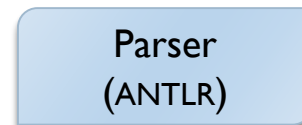


**FSM:**




**FSM transition\_table:**

(1, (send, order, seller) ->  
(2, assertion\_object, {'x': 'int'})



# Runtime Verification: Implications

1. Global specification and its
2. local projections are given in  Scribble

```
protocol Hello(me, you)
{
  Hello from me to you;
  Hello from you to me;
}
```

3. Applications for the communication are conversation-aware ...

## Application

Application Layer

Conversation Layer

Transport Layer

...and can read/write conversation messages

## Header:

...

Sender: you

Receiver: me

Label: Hello

...

# Conversation Layer

---

## ▶ Participant

- ▶ addressable entity on the network, has a public address and can be invited to take part in a conversation,
- ▶ It can also start a conversation with another addressable entities (if he has the right capabilities)

## ▶ Conversation

- ▶ Encapsulate conversation related information like: the protocol that will run; the roles that take part; memory (mapping between addresses and conversation endpoints)

## ▶ ConversationEndpoint – these are the roles in the conversation and also the processes that we want to run,

- ▶ send(), receive()

# Conversation Runtime

Conversation API	
<b>create</b> (configuration)	$\bar{a} < s[r]: T >$
<b>join</b> (role, principal_name)	$a < s[r]: T >$
<b>send</b> (to_role, op, args)	$k[r_1, r_2]! l < e >$
<b>receive</b> (from_role)	$k[r_1, r_2]? \{l < e >\}. P_i$
<b>receive_async</b> (from_role, callback)	

1. Sending to roles in a conversation, not to addresses (only the initiator knows all addresses)
2. *Operation is unspecified (can be used for method name or class name or as annotation)*
3. *Update the conversation header in a message and the local routing table*

# API Example 1/2

```
global protocol DataAquisition(role U,
role A, role I) {
  Request(string:info) from U to A;
  Request(string:info) from A to I;
  choice at I {
    Support from I to A;
    rec Poll{
      Poll from A to I;
      choice at I {
        Raw(data) from I to A
        Formatted(data) from I to U;
        Poll;
      } or {
        Stop from I to A;
        Stop from A to U;}}
  } or {
    NotSupported from I to A;
    Stop from A to I;
    Stop from A to U;}}
```

```
class ClientApp(BaseApp):
  def start(self, p):
    c = p.create(
      'DataAquisition', 'config.yml')
    c.join('U', 'alice')

    resource_request = c.receive('U')
    c.send('I', resource_request)
    req_result = c.receive('I')

    if (req_result == SUPPORTED):
      c.send('I', 'Poll')
      op, data = c.receive('I')

      while (op != 'Stop'):
        formatted_data = format(data)
        c.send('U', fomratted_data)
        c.send('U', stop)
      else:
        c.send('U, I', stop)
        c.stop()
```





# API Example – event-driven

```
class ClientApp(BaseApp)
def start(self, p):
    c = p.create(
        'DataAquisition', 'config.yml')
    c.join('U', 'alice')

    resource_request = c.receive('U')
    c.send('I', resource_request)
    req_result = c.receive('I')

    if (req_result == SUPPORTED):
        c.send('I', 'Poll')
        op, data = c.receive('I')

        while (op != 'Stop'):
            formatted_data = format(data)
            c.send('U', fomratted_data)
            c.send('U', stop)
    else:
        c.send('U, I', stop)
        c.stop()
```

```
class ClientApp(BaseApp):
def start(self, p):
    c = p.create(...)
    c.join('U', 'alice')
    c.receive_async('U', on_req_rcv)

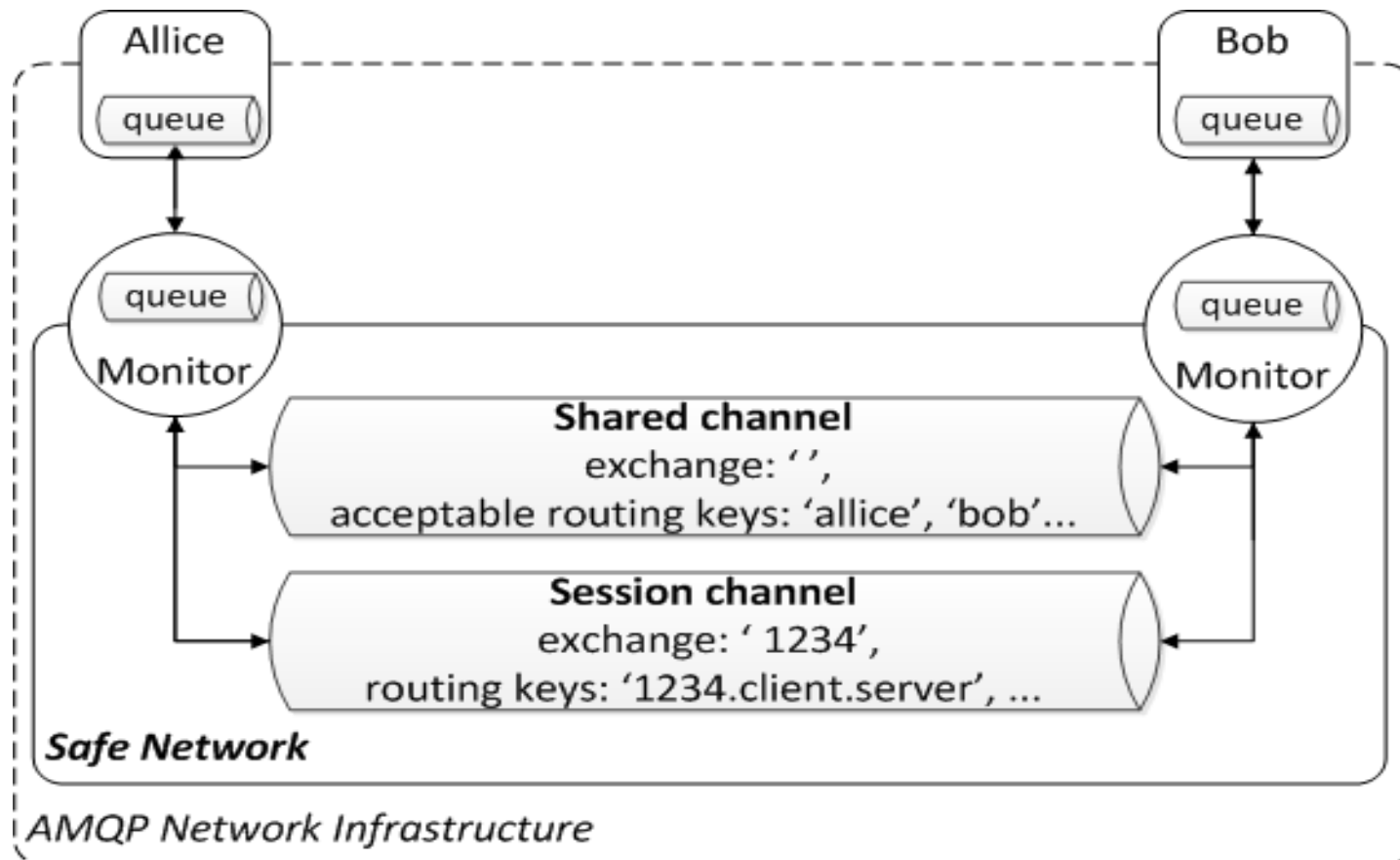
def on_req_rcv(self, conv, op, msg):
    if (op == SUPPORTED):
        conv.send('I', 'Poll')
        conv.receive_async('I',
            on_data_rcv)
    else: conv.send('U', 'Stop')

def on_data_rcv(self, conv, op, payload):
    if (operation != 'Stop'):
        formatted_data = format(payload)
        c.send('U', formatted_data)
    else:
        conv.send('U', 'Stop')
        conv.stop()
```

Same Monitor!!!

# Conversations in AMQP

Use AMPQ abstractions to mimic public and session channels.



---

1. Writing correct global protocols with **Scribble Compiler**

2. Verify programs via *local monitors*

3. Build additional verification modules via *annotations*

# Validation via Annotations

## Annotations = @{Logic} Scribble Construct

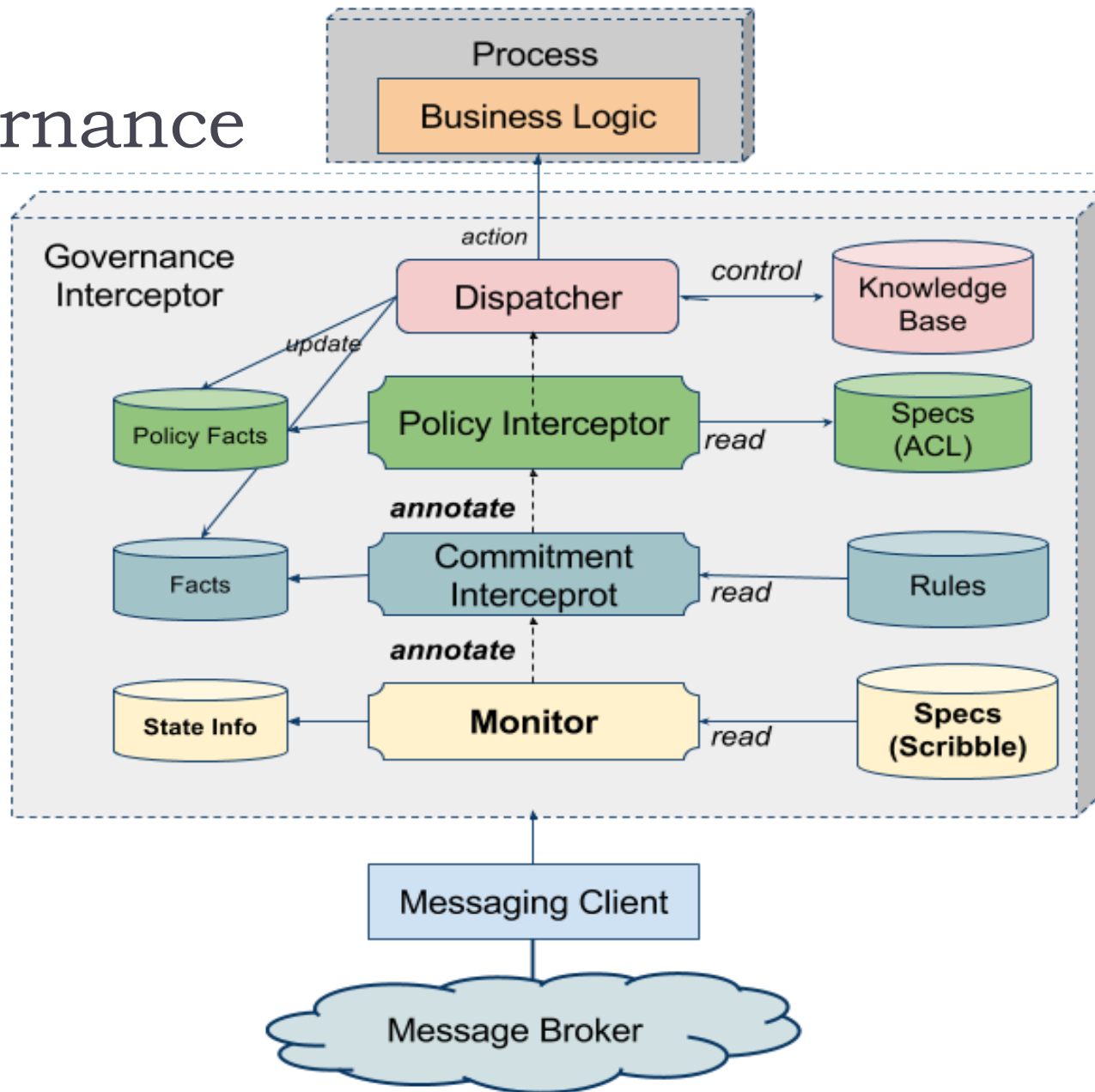
```
...  
@{assert: payment + overdraft >= 1000}  
offer(payment: int) from C to I;  
...
```

```
...  
@{deadline: 5s}  
offer(conditions string) from C to I;  
...
```

```
...  
rec Loop {  
  @{guard: repeat < 10}  
  propose(string) from C to I;  
  ...  
}
```

- ▶ The monitor passes {‘type’:param, ...} to the upper layers
- ▶ Upper layers recognize and process the annotation type or discard it
- ▶ Statefull assertion

# Governance

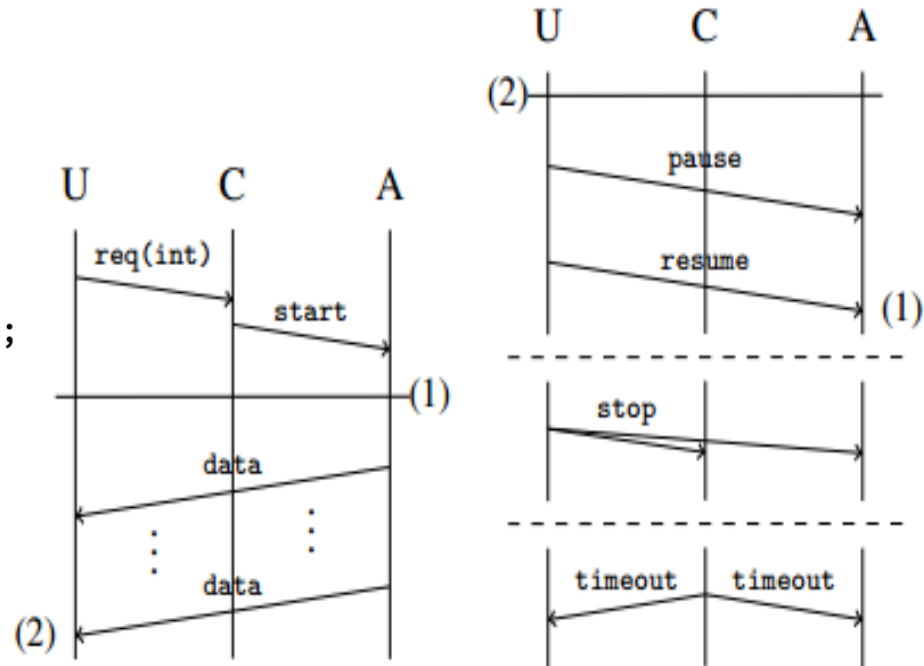


# Extensions: Flexible Interrupt

```

global protocol RAC(role User as U,
    role Controller as C, role Agent as A) {
    req(duration:int) from U to C;
    start() from C to A;
    interruptible {
        rec X {
            interruptible {
                rec Y {
                    data() from A to U;
                    continue Y;
                }
            } with {
                pause by U;
            }
            resume() from U to A;
            continue X;
        }
    } with {
        stop() by U;
        timeout() by C;
    }
}

```





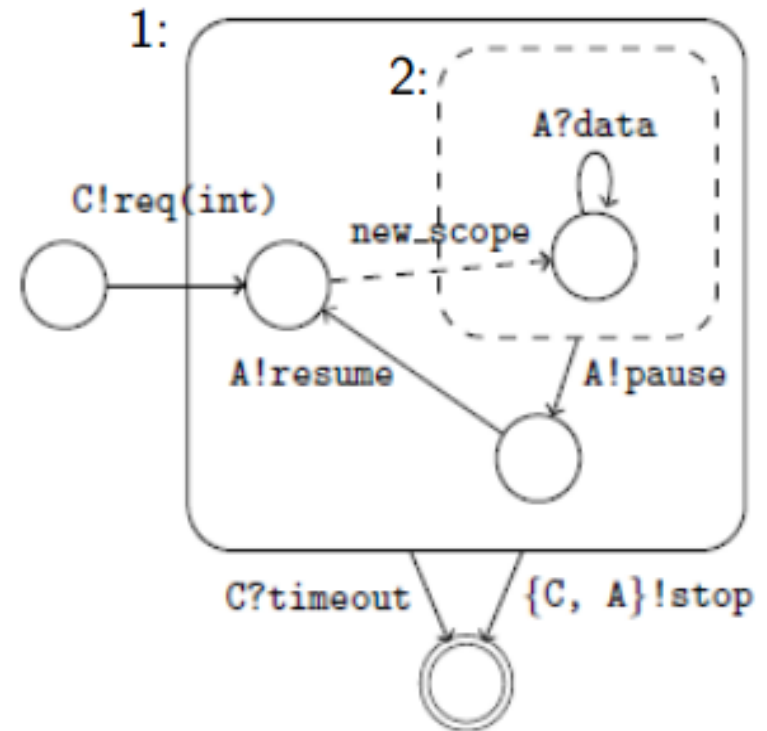
# Extensions: Flexible Interrupt

## Interruptible scopes modelled by nested FSMs

```

req(duration:int) to C;
1: interruptible {
  rec X {
    2: interruptible {
      rec Y {
        data() from A;
        continue Y;
      }
      throws pause() to A;
      resume() to A;
      continue X;
    }
  }
  throws stop() to A, C;
  catches timeout() from C;
}

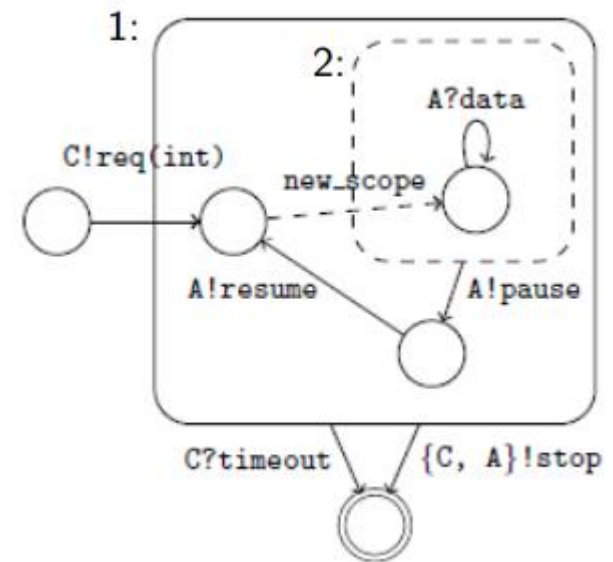
```



# Scope based control flow

Dynamic tracking of MPST by endpoint monitoring to drive scope-based conversation operations

```
with conv.join('user') as c:
    c.send(controller, 'req', 100)
    with c.scope('timeout' 'stop') as c1:
        while not self.enough_data():
            with c1.scope('timeout', 'stop') as c2:
                while not batch.full():
                    next = c2.recv(agent, 'data')
                    batch.append(next)
                    c2.interrupt('pause')
                    process_data(batch)
            c1.send(agent, 'resume')
        c1.interrupt('stop')
```



# Conclusion replacement

---

Type are used for early error detection

- Session types **prevent** cluttering the network with wrong messages.

Types are used to optimise the memory layout

- Session types can be used to **optimise the communication flow**, grouping messages together

Types are used as a guidance when we design programs

- Session types can be used for **Testable architectures** - checking the specification match the implementation

# It is your turn ...

---

**global protocol** Q&A(**role you**, **role me**)

{

**rec Loop**

{

Questions **from you to me**;

Answers **from me to you**;

Loop;

}

}