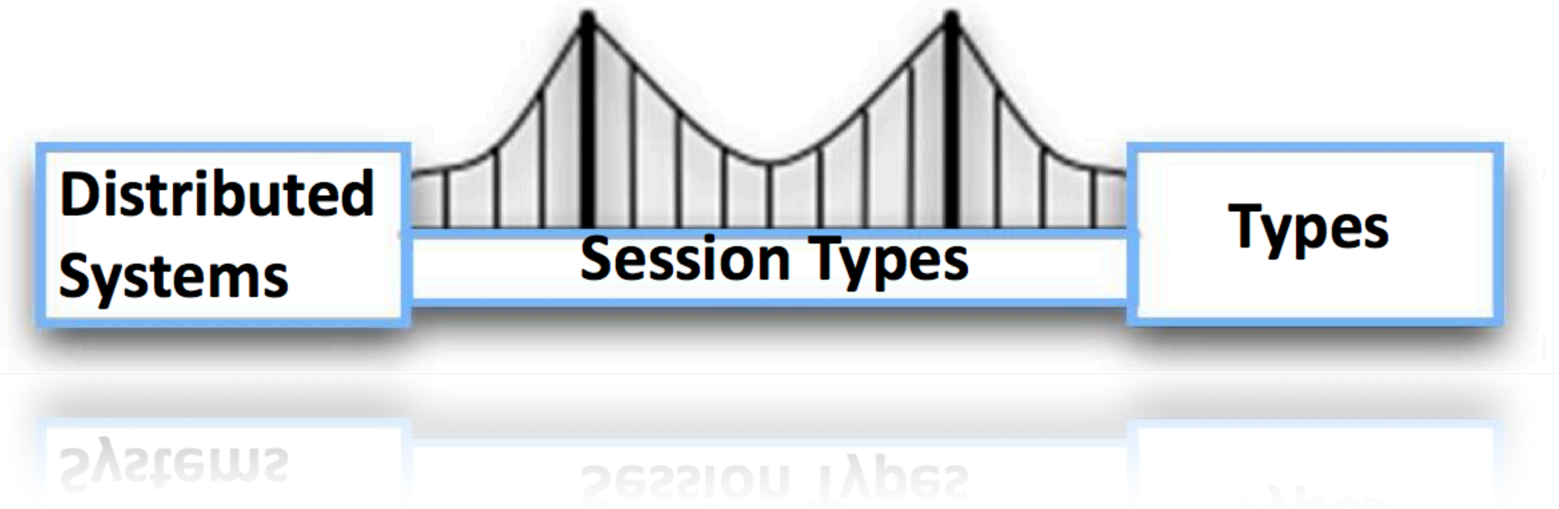


MULTIPARTY SESSION TYPES

Scribble and applications

Multiparty Session Types

Motivation

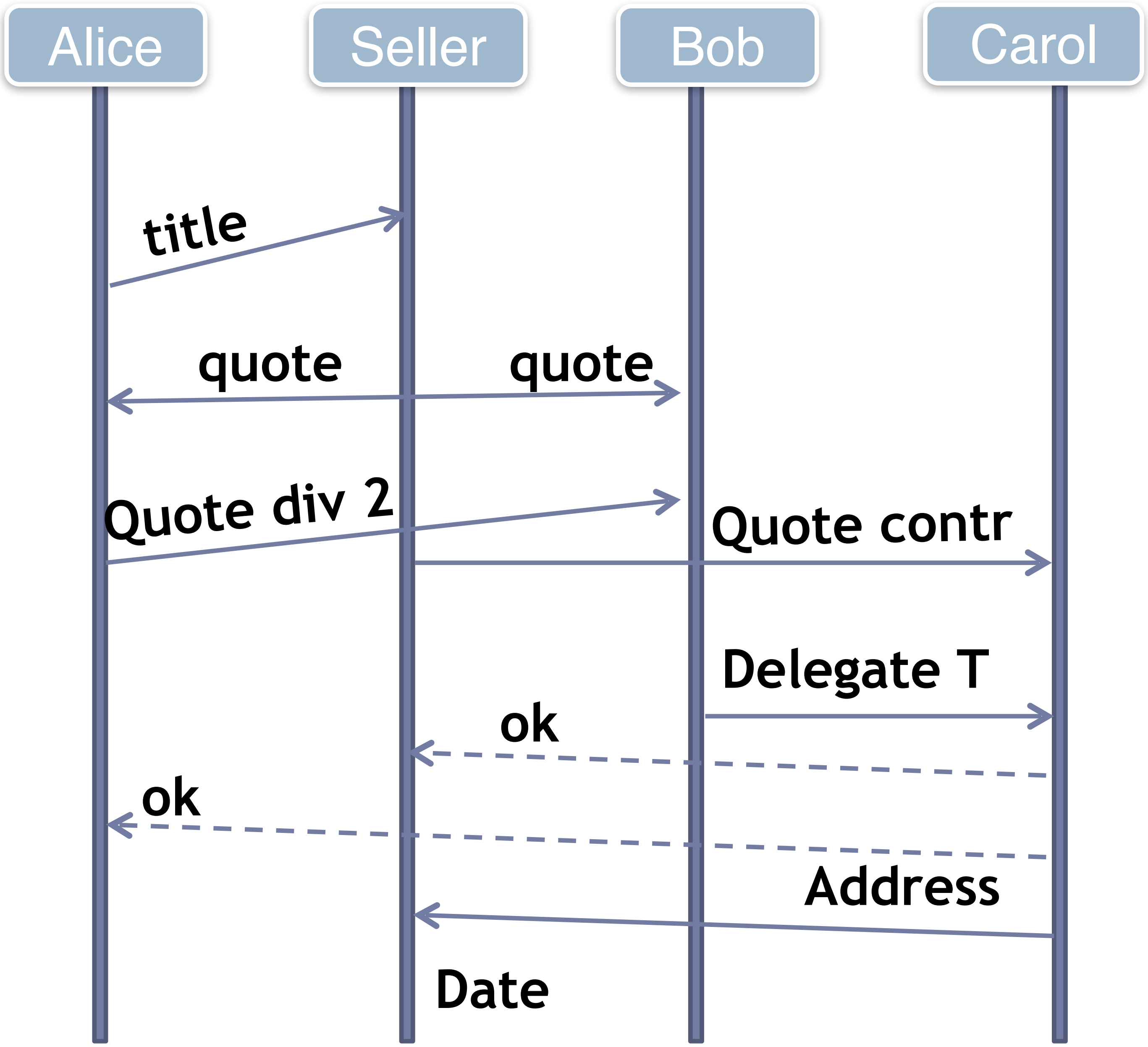
Binary Session Types and Duality

P has T

Q has $\overline{T}$

$P \mid Q$ typable

Wait a minute! What if it is more than 2?

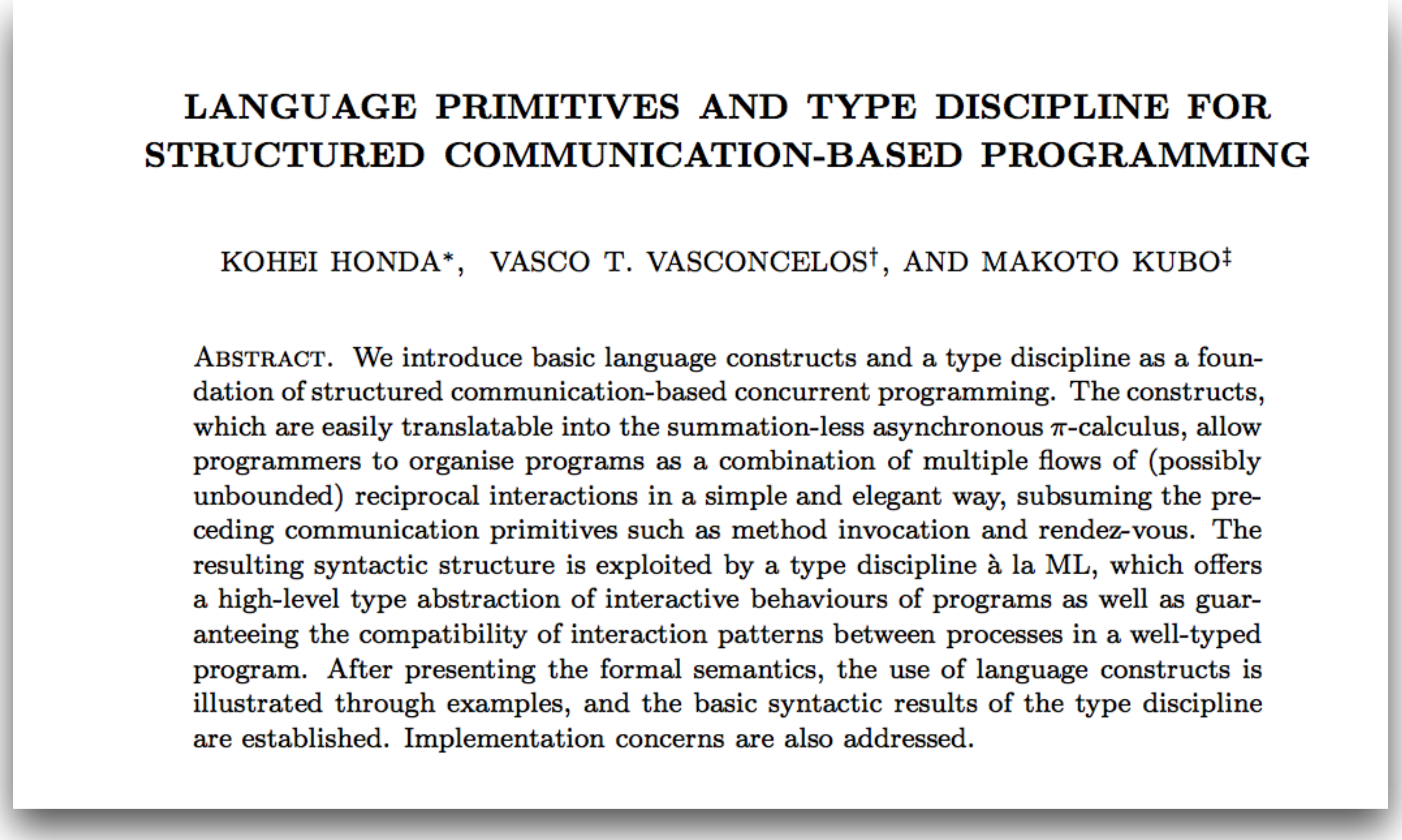


The only problem is…

communication is more like this …

Or… even like this

Original (Binary) Session Types Paper ESOP'98

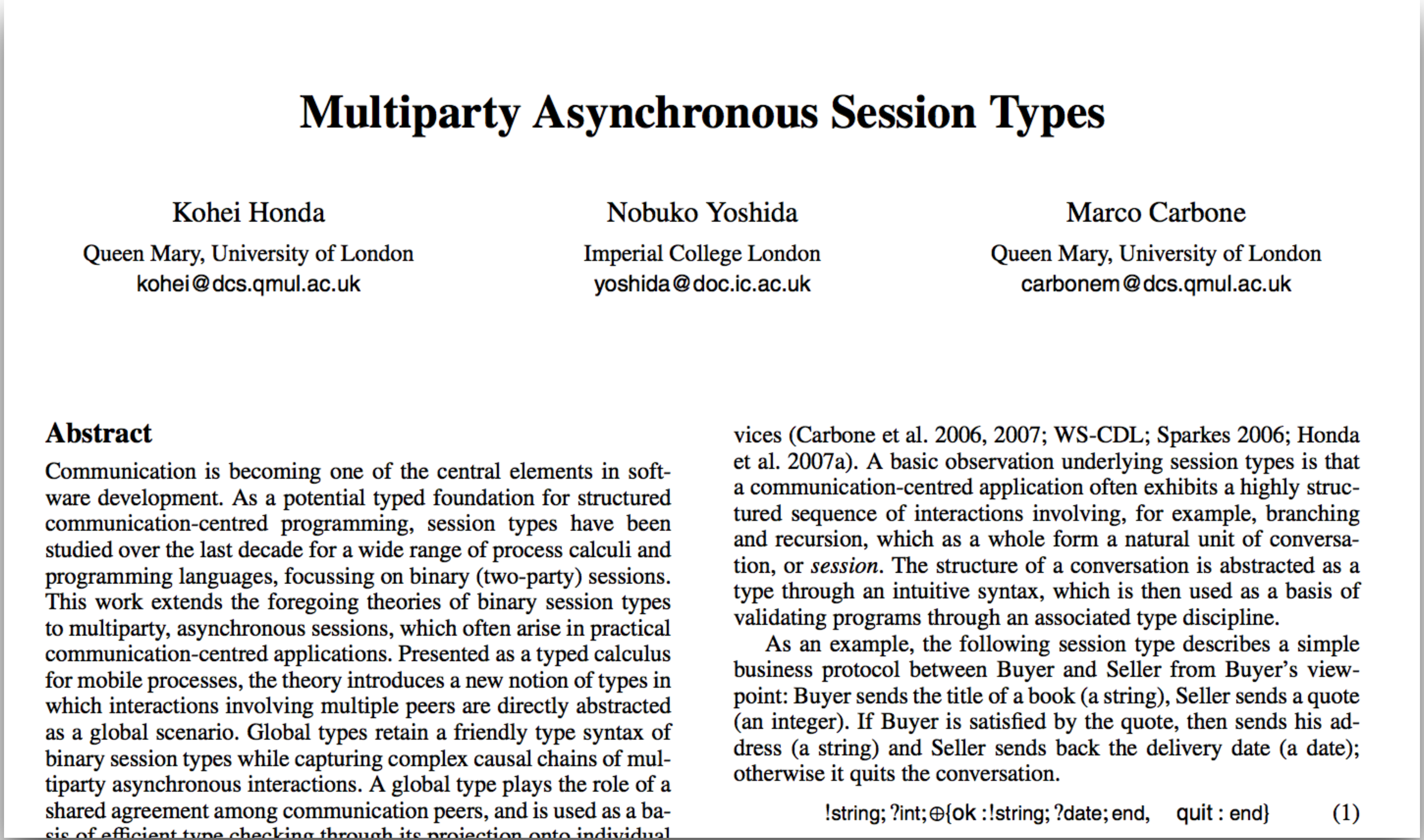


LANGUAGE PRIMITIVES AND TYPE DISCIPLINE FOR STRUCTURED COMMUNICATION-BASED PROGRAMMING

KOHEI HONDA*, VASCO T. VASCONCELOS†, AND MAKOTO KUBO‡

Abstract. We introduce basic language constructs and a type discipline as a foundation of structured communication-based concurrent programming. The constructs, which are easily translatable into the summation-less asynchronous π-calculus, allow programmers to organise programs as a combination of multiple flows of (possibly unbounded) reciprocal interactions in a simple and elegant way, subsuming the preceding communication primitives such as method invocation and rendez-vous. The resulting syntactic structure is exploited by a type discipline à la ML, which offers a high-level type abstraction of interactive behaviours of programs as well as guaranteeing the compatibility of interaction patterns between processes in a well-typed program. After presenting the formal semantics, the use of language constructs is illustrated through examples, and the basic syntactic results of the type discipline are established. Implementation concerns are also addressed.

Multiparty Session Types Paper POPL'08



Multiparty Asynchronous Session Types

Kohei Honda
Queen Mary, University of London
kohei@dcs.qmul.ac.uk

Nobuko Yoshida
Imperial College London
yoshida@doc.ic.ac.uk

Marco Carbone
Queen Mary, University of London
carbonem@dcs.qmul.ac.uk

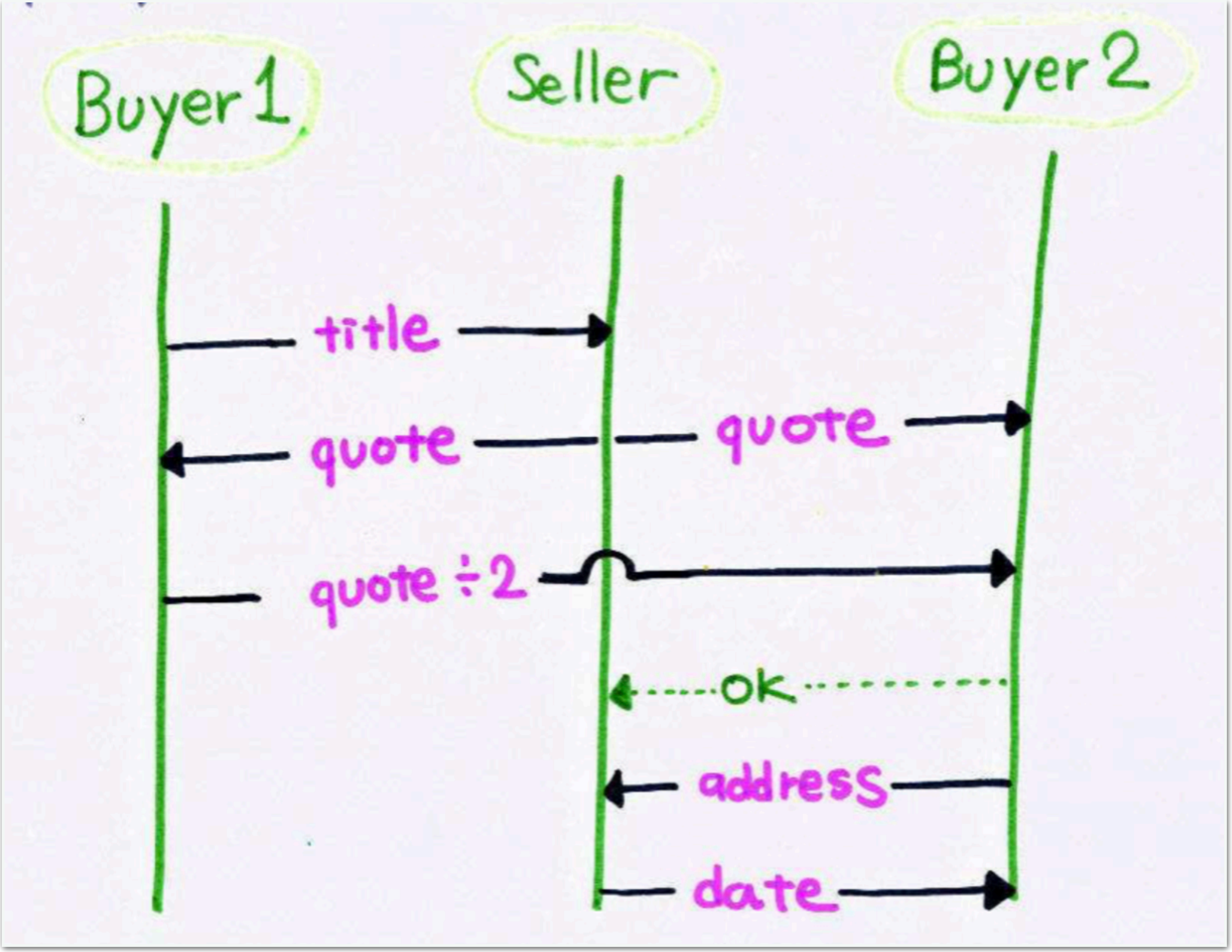
Abstract

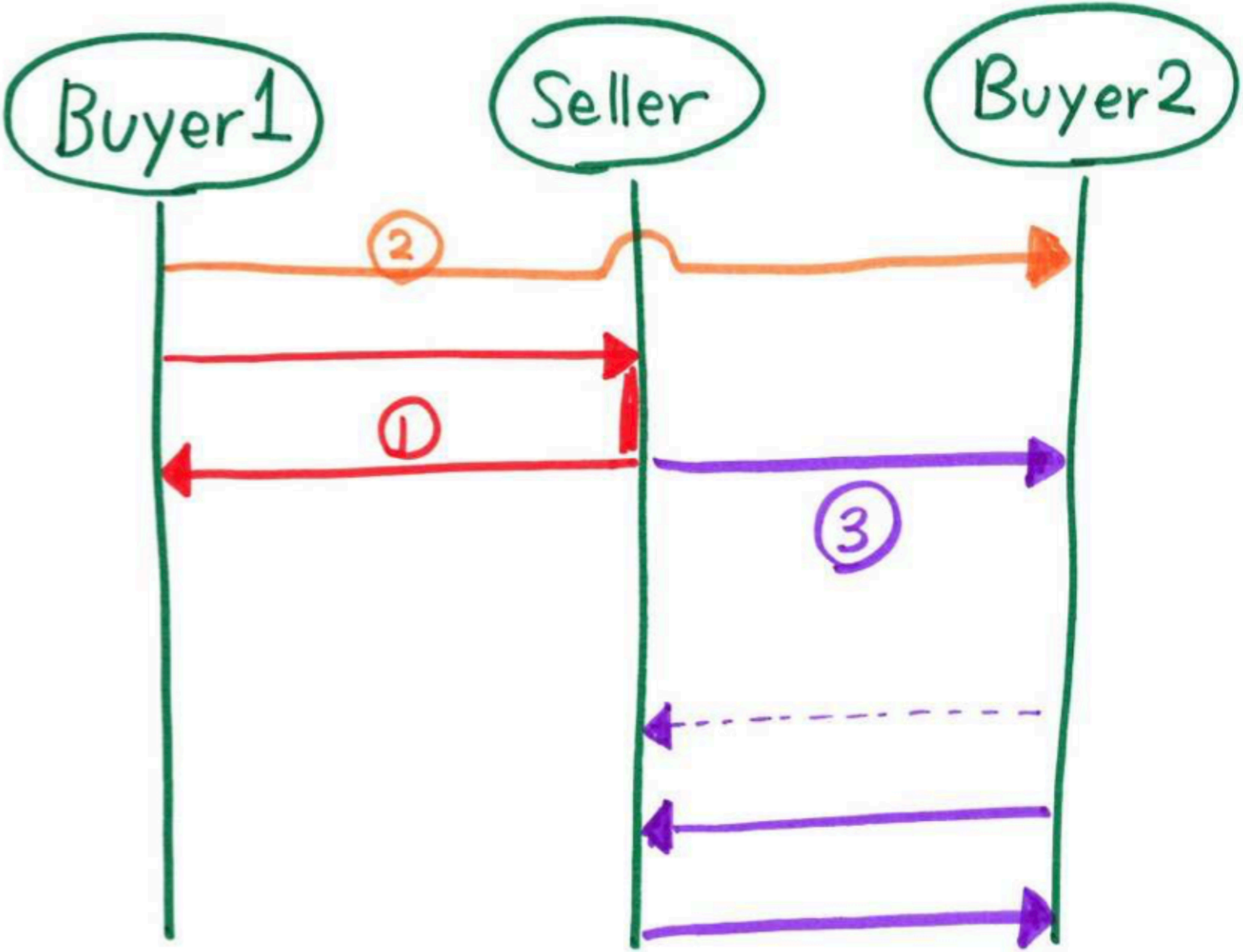
Communication is becoming one of the central elements in software development. As a potential typed foundation for structured communication-centred programming, session types have been studied over the last decade for a wide range of process calculi and programming languages, focussing on binary (two-party) sessions. This work extends the foregoing theories of binary session types to multiparty, asynchronous sessions, which often arise in practical communication-centred applications. Presented as a typed calculus for mobile processes, the theory introduces a new notion of types in which interactions involving multiple peers are directly abstracted as a global scenario. Global types retain a friendly type syntax of binary session types while capturing complex causal chains of multiparty asynchronous interactions. A global type plays the role of a shared agreement among communication peers, and is used as a basis of efficient type checking through its projection onto individual

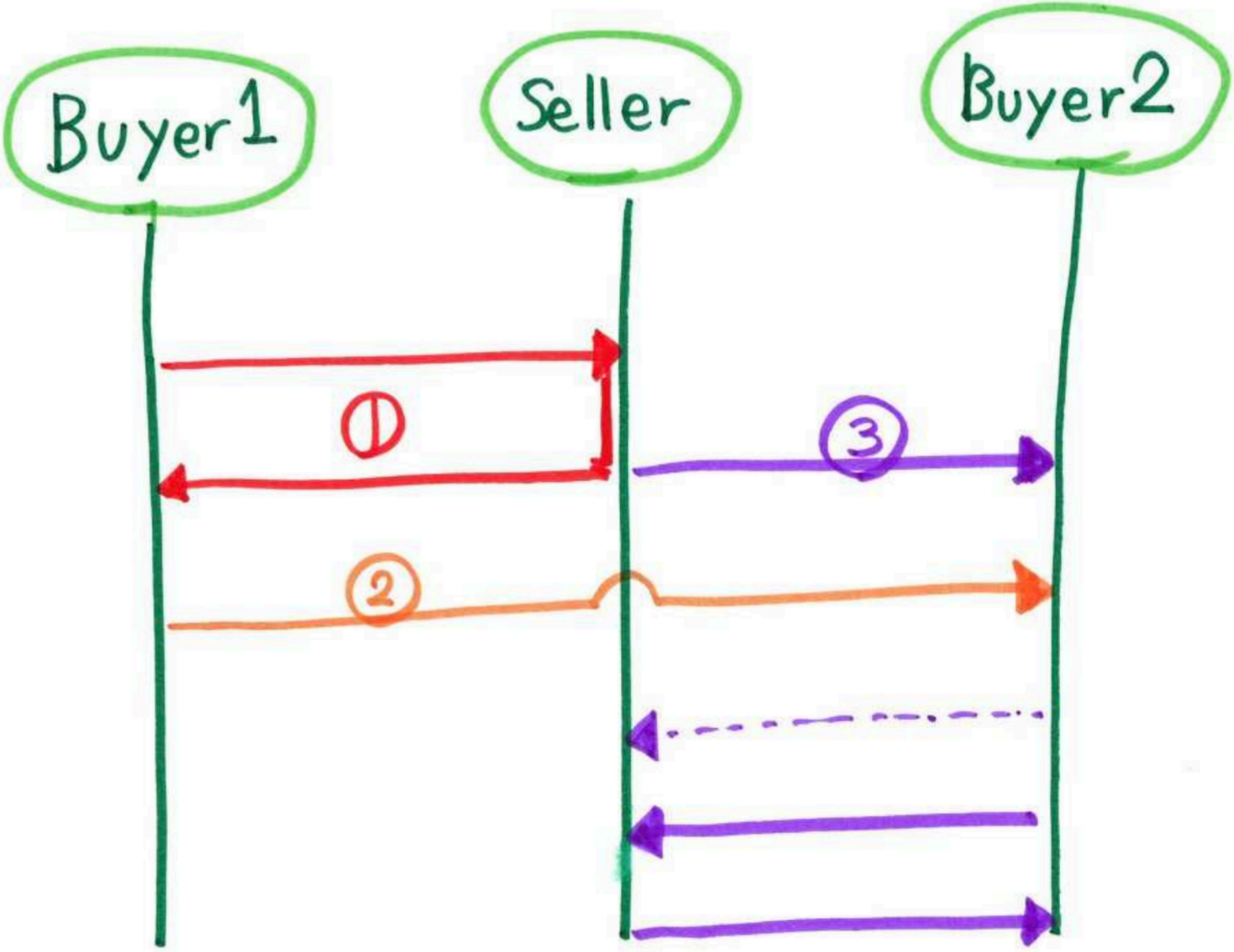
vices (Carbone et al. 2006, 2007; WS-CDL; Sparkes 2006; Honda et al. 2007a). A basic observation underlying session types is that a communication-centred application often exhibits a highly structured sequence of interactions involving, for example, branching and recursion, which as a whole form a natural unit of conversation, or *session*. The structure of a conversation is abstracted as a type through an intuitive syntax, which is then used as a basis of validating programs through an associated type discipline.

As an example, the following session type describes a simple business protocol between Buyer and Seller from Buyer's viewpoint: Buyer sends the title of a book (a string), Seller sends a quote (an integer). If Buyer is satisfied by the quote, then sends his address (a string) and Seller sends back the delivery date (a date); otherwise it quits the conversation.

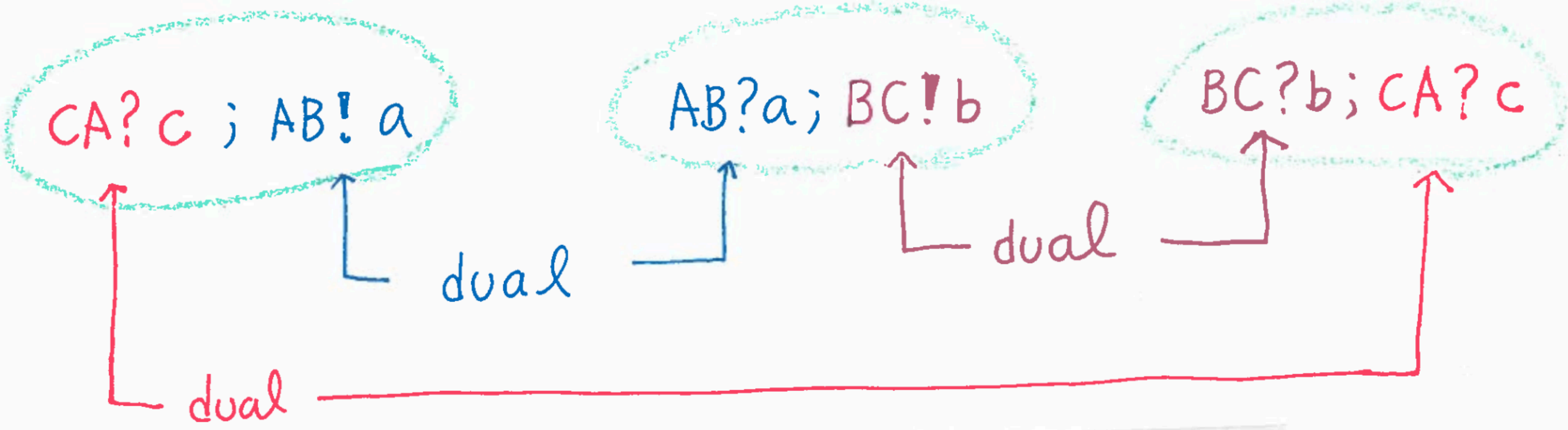
$$!\mathsf{string}; ?\mathsf{int}; \oplus\{\mathsf{ok} : !\mathsf{string}; ?\mathsf{date}; \mathsf{end}, \quad \mathsf{quit} : \mathsf{end}\} \qquad (1)$$


Buyer 1
Seller
Buyer 2
title
quote
quote
quote ÷ 2
ok
address
date


Buyer1
Seller
Buyer2
2
1
3


Buyer1
Seller
Buyer2
1
3
2

Alice Bob Carol

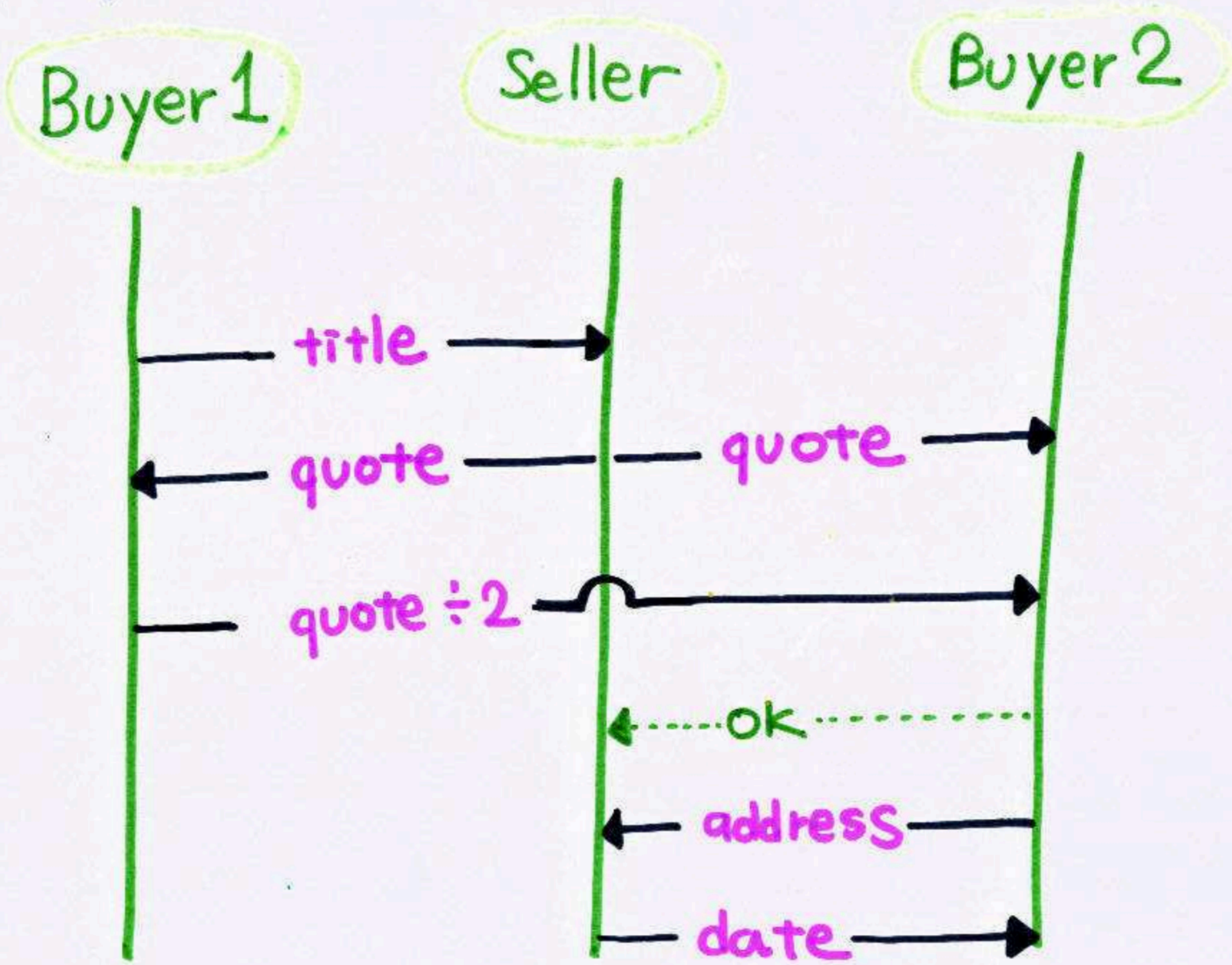


3 dual pairs

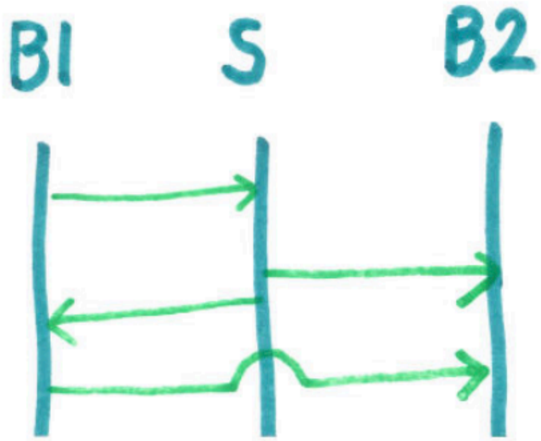
If you use **binary** Session Types ....

Deadlock!

Multiparty Session Types



Multiparty Session Types [Honda, Yoshida, Carbone 2008]



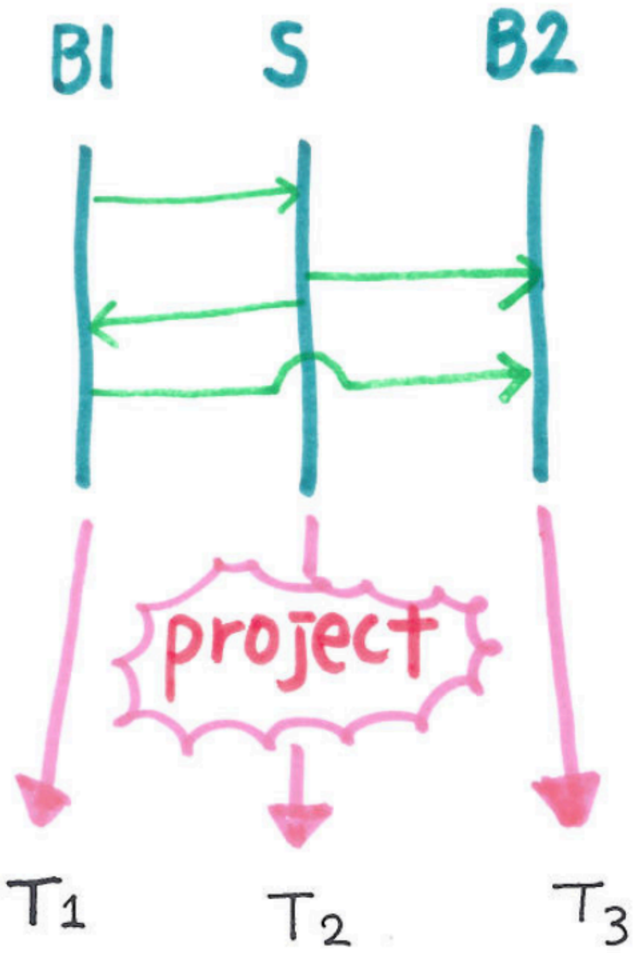
Ⓖ $B1 \to S$ Int.

$S \to B2$ Char

STEP 1

Write Global Type

Multiparty Session Types [Honda, Yoshida, Carbone 2008]



Ⓖ $B1 \to S$ Int.

$S \to B2$ Char

Ⓣ $B_1?\mathrm{Int}.B_2!\mathrm{Char}$

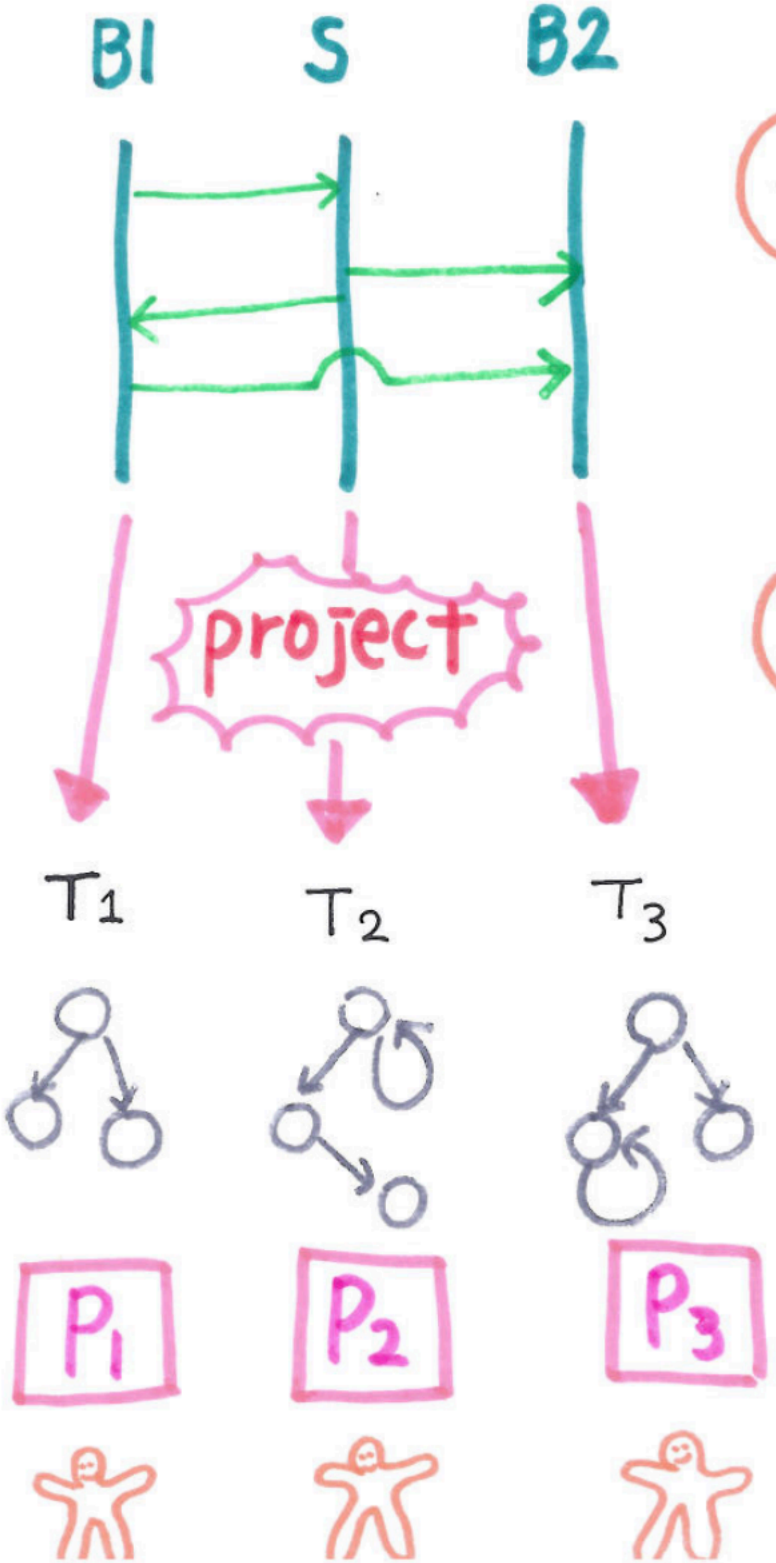
STEP 1

Write Global Type

STEP 2

Project to Local Types

Multiparty Session Types [Honda, Yoshida, Carbone 2008]



G $B1 \rightarrow S$ Int.

$S \rightarrow B2$ Char

T $B_1?\mathrm{Int}.B_2!\mathrm{Char}$

P $B_1?(x).B_2!\langle\text{"apple"}\rangle$

STEP 1

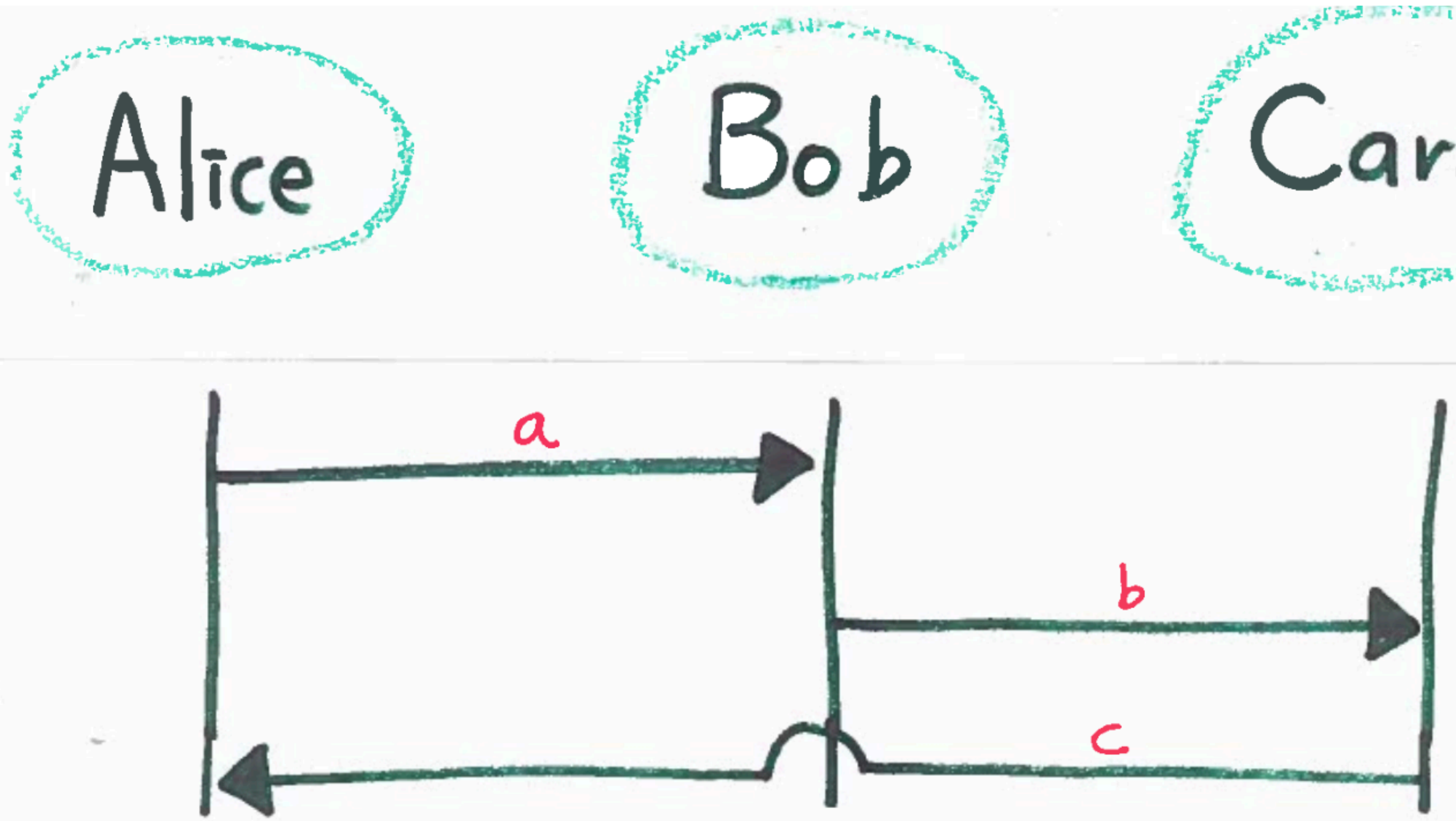
Write Global Type

STEP 2

Project to Local Type

STEP 3

- Static Check
- Generate Code
- Run-time Check



Global Type

Alice AB!a ; CA?c

Bob AB?a; BC!b

BC?b;
Carol CA!c;

LOCAL TYPES

No Deadlock

Binary Session Types and Duality

P has T

Q has $\overline{T}$

P | Q typable

Multiparty Session Types and Projection

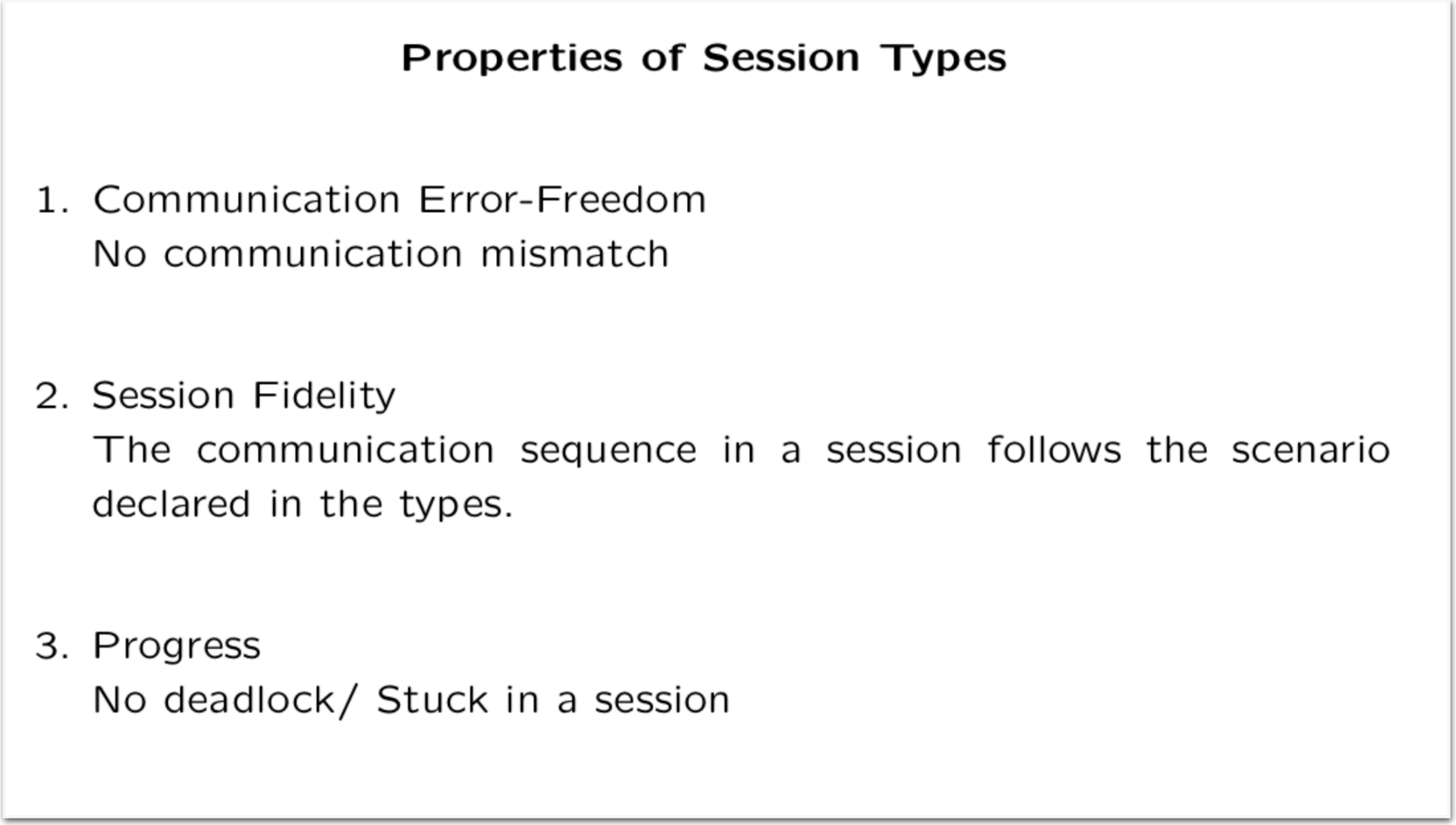
P_1 has type $G \upharpoonright P_1$

P_2 has type $G \upharpoonright P_2$

P_3 has type $G \upharpoonright P_3$

P_4 has type $G \upharpoonright P_4$

$P_1 \mid P_2 \mid P_3 \mid P_4$ is typable



Properties of Session Types

1. Communication Error-Freedom
   No communication mismatch

2. Session Fidelity
   The communication sequence in a session follows the scenario declared in the types.

3. Progress
   No deadlock/ Stuck in a session

“well-typed **channels** are free from communication errors”

Errors (by example)

- ☑ Communication mismatch

```
              A                        B

send(B, Div, int) | recv(A, Add, int)        ✖ Wrong label
send(B, Div, int) | recv(A, Add, string)     ✖ Wrong payload
send(C, Div, int) | recv(A, Div, int)        ✖ Wrong role
```

- ☑ Orphan messages

```
    A         B
send(B)|send(A)
```

- ☑ Deadlock

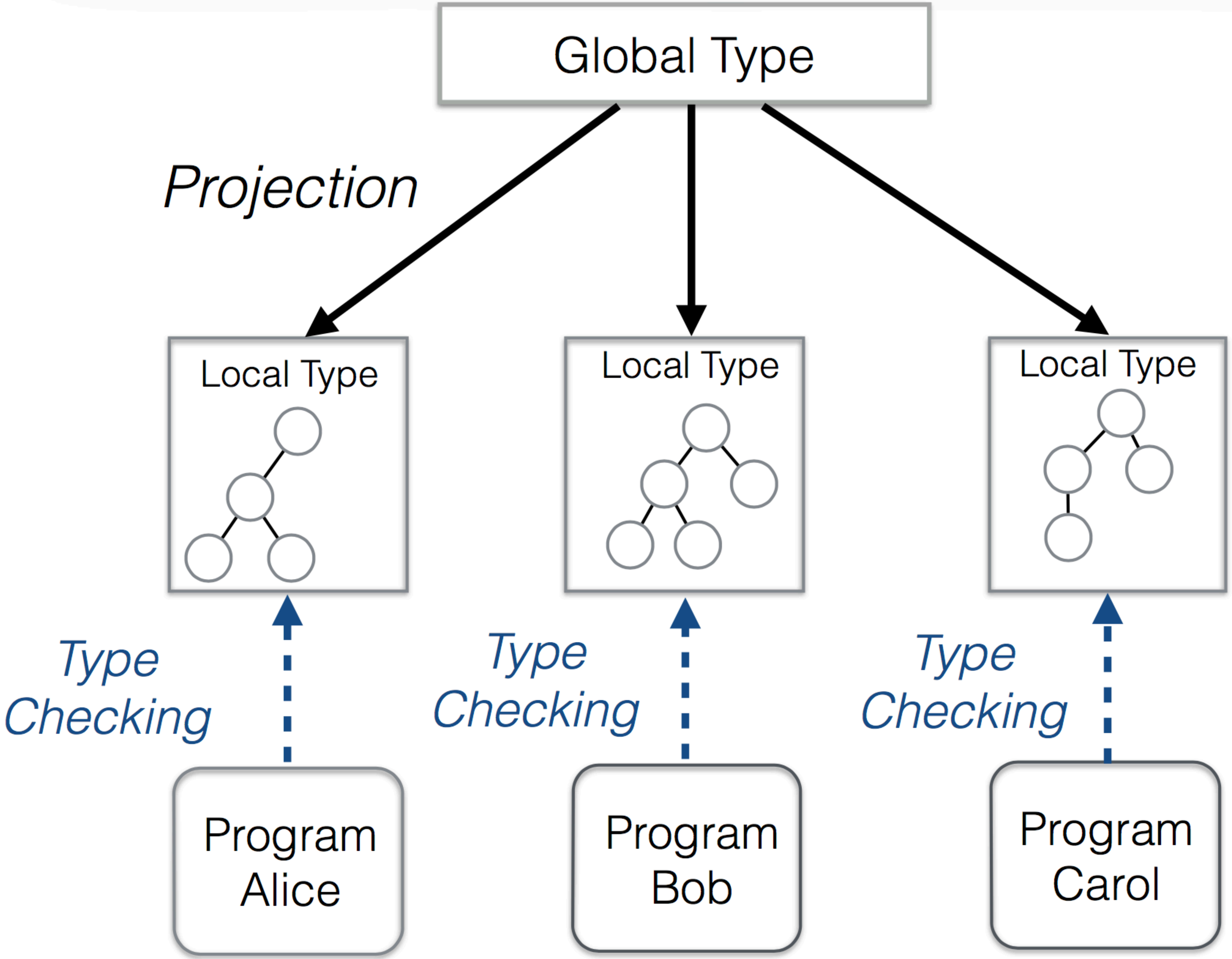
```
   A       B           C

recv(B)|recv(A)
recv(C)|recv(C)|if (n=0) then send(A) else send(B)
```

Session Types

Applications

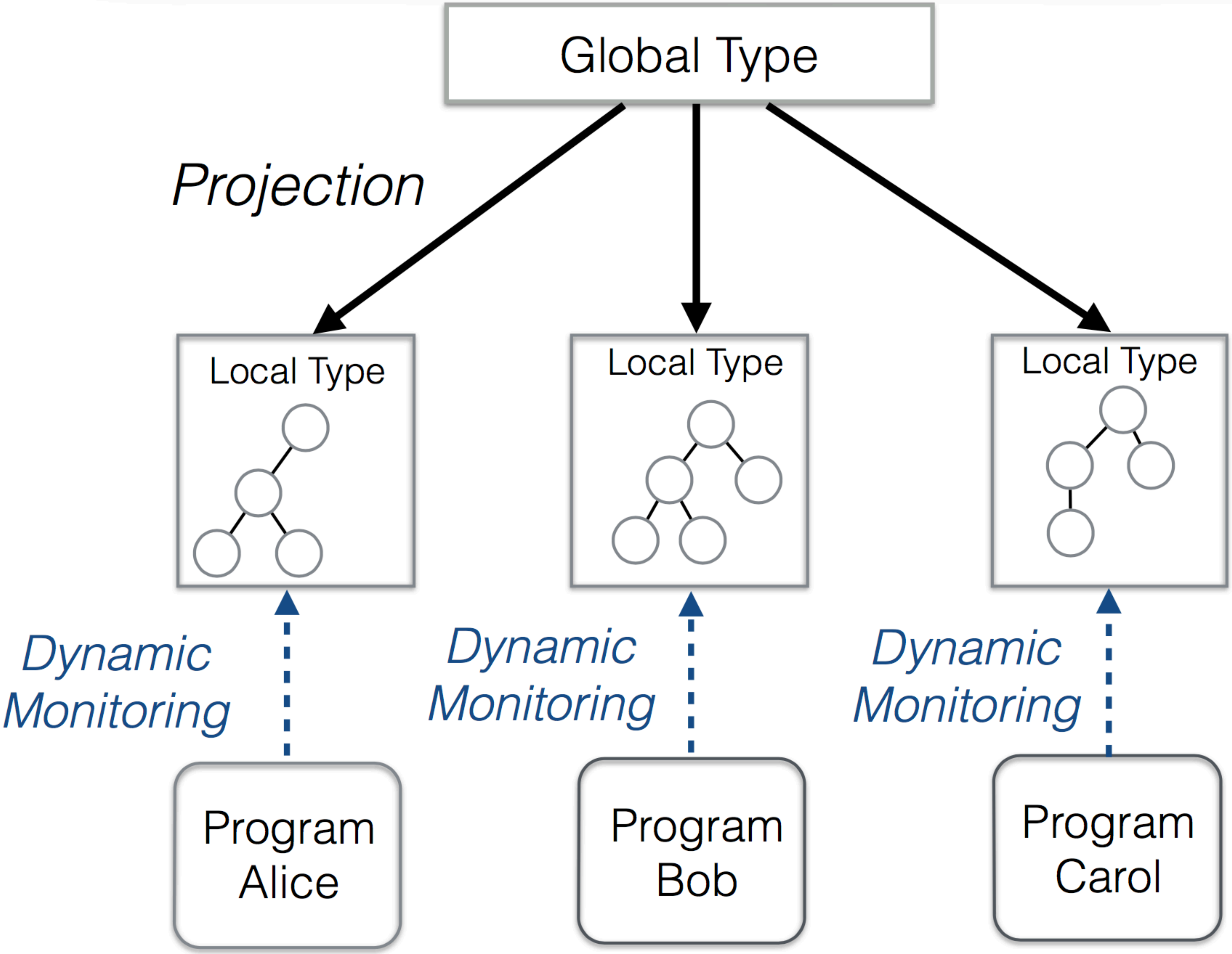
Type Checking

[OOPSLA’15, ECOOP’16, ECOOP’17, COORDINATION’17]



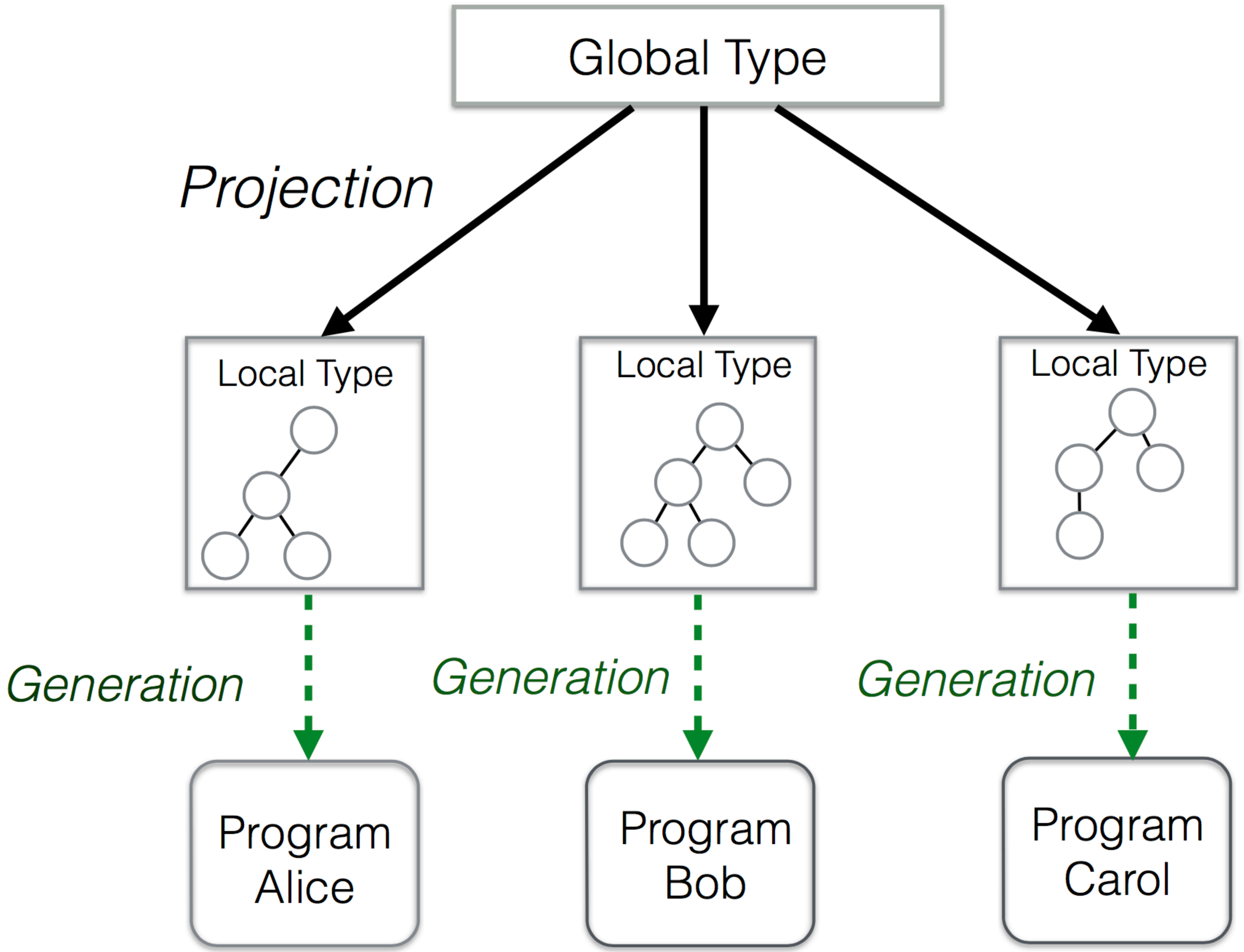
Dynamic Monitoring

[RV’13, COORDINATION’14, FMSD’15, LMCS’17, CC’17]



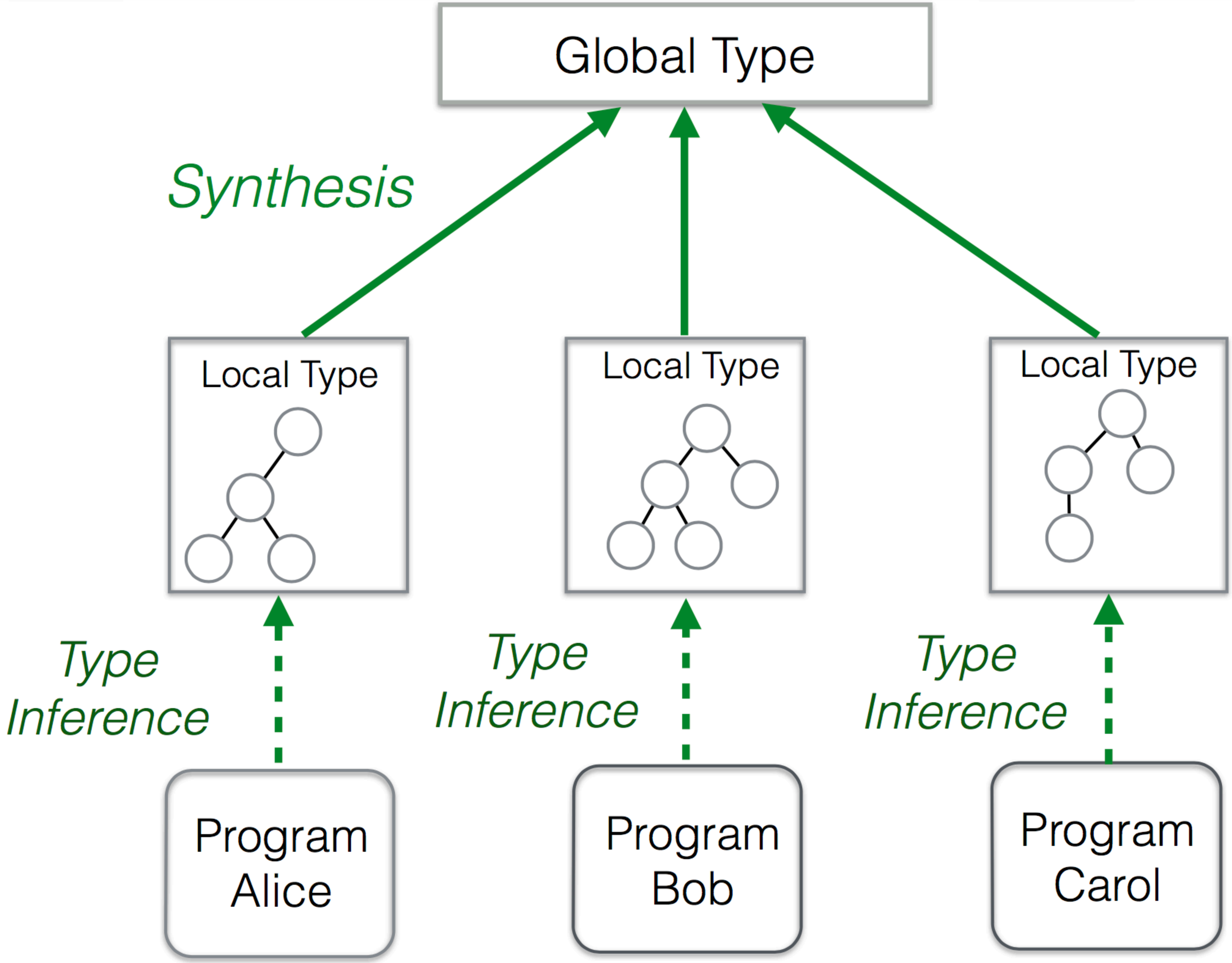
Code Generation

[CC'15, FASE'16, CC'18]



Synthesis

[POPL’15, CONCUR’15, TACAS’16, CC’16, POPL’18, ICSE’18]



MPST

Scribble

- Applications
  - Deadlock Detection (Go)
  - Recovery strategies(Erlang)
  - Type-driven programming (Java, Scala, F#)
  - Static Verification (C, OCaml, Rust)
  - Runtime monitoring (Python)

F#
GO
Scala
Session C
Java
ERLANG
RabbitMQ
Messaging that just works
python
AMQP
Advanced Message Queuing Protocol
OOI
OCEAN OBSERVATORIES INITIATIVE

Applications

Scribble

F#

Go

Scala

Session C

Java

ERLANG

RabbitMQ

Messaging that just works

python

AMQP

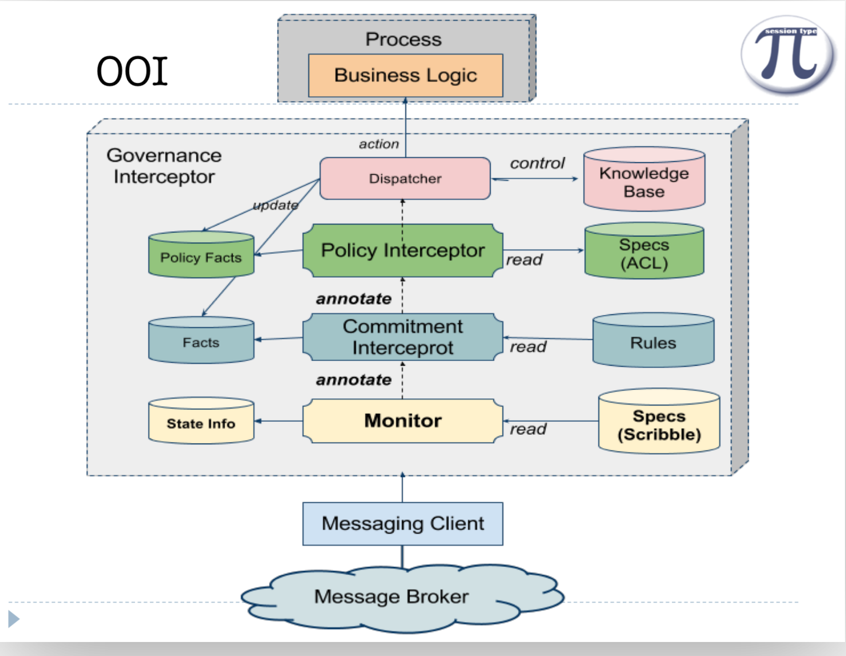
Advanced Message Queuing Protocol

OOI

OCEAN OBSERVATORIES INITIATIVE

Session Type Based Tools

OOI Governance



OOI

Process

Business Logic

action

Governance Interceptor

Dispatcher

control

Knowledge Base

update

Policy Facts

Policy Interceptor

read

Specs (ACL)

annotate

Facts

Commitment Interceprot

read

Rules

annotate

State Info

Monitor

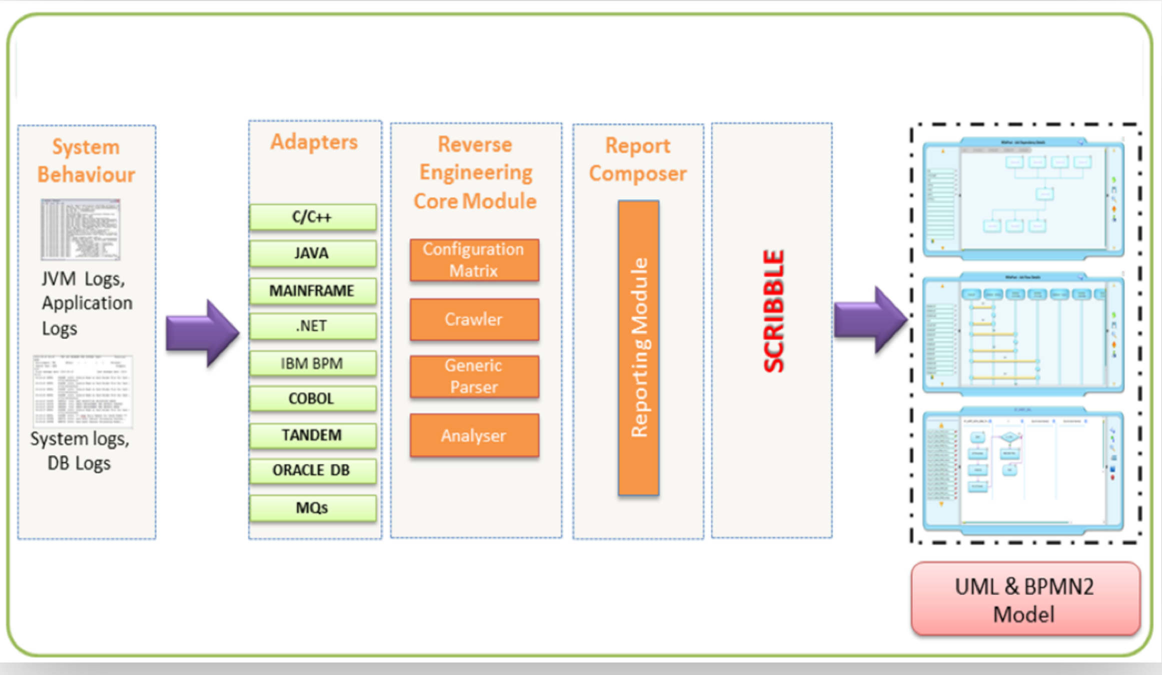
read

Specs (Scribble)

Messaging Client

Message Broker

ZDLC: Process Modeling



System Behaviour

JVM Logs, Application Logs

System logs, DB Logs

Adapters

C/C++

JAVA

MAINFRAME

.NET

IBM BPM

COBOL

TANDEM

ORACLE DB

MQs

Reverse Engineering Core Module

Configuration Matrix

Crawler

Generic Parser

Analyser

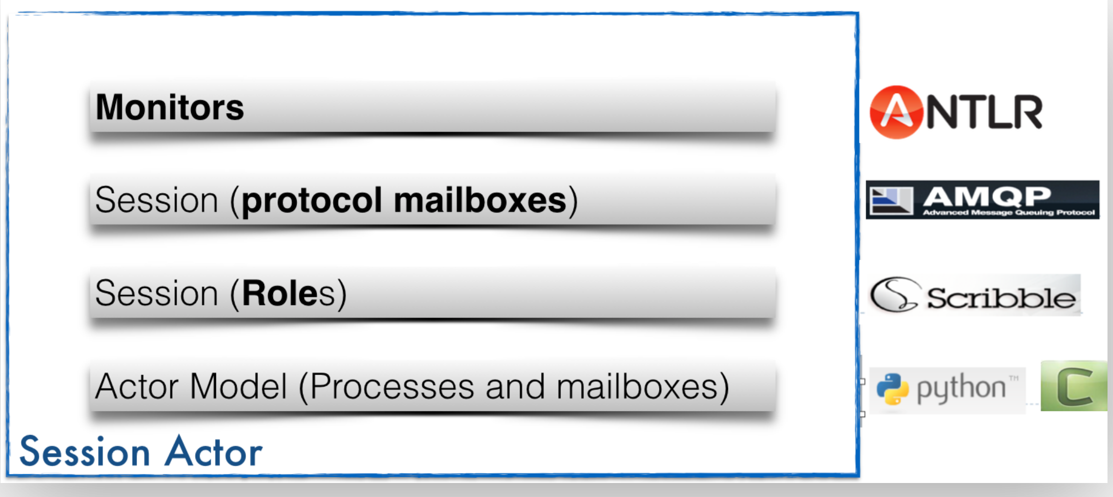
Report Composer

Reporting Module

SCRIBBLE

UML & BPMN2 Model

Actor Verification



Monitors

Session (**protocol mailboxes**)

Session (**Roles**)

Actor Model (Processes and mailboxes)

Session Actor

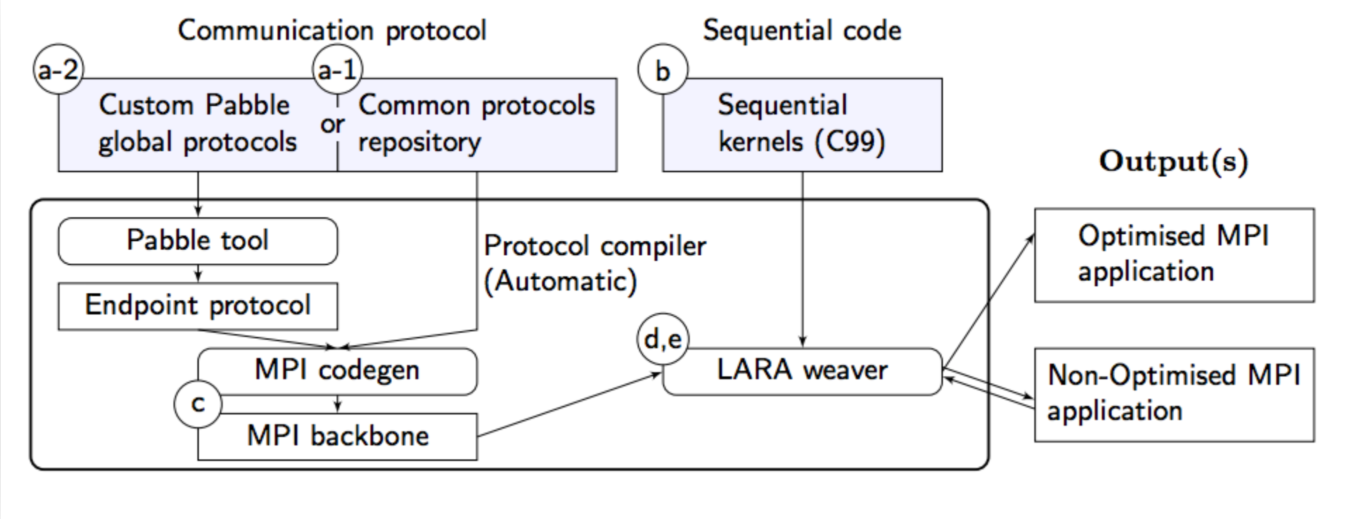
ANTLR

AMQP Advanced Message Queuing Protocol

Scribble

python

MPI code generations



Communication protocol

Sequential code

a-2

Custom Pabble global protocols

or

a-1

Common protocols repository

b

Sequential kernels (C99)

Output(s)

Pabble tool

Endpoint protocol

Protocol compiler (Automatic)

d,e

LARA weaver

MPI codegen

c

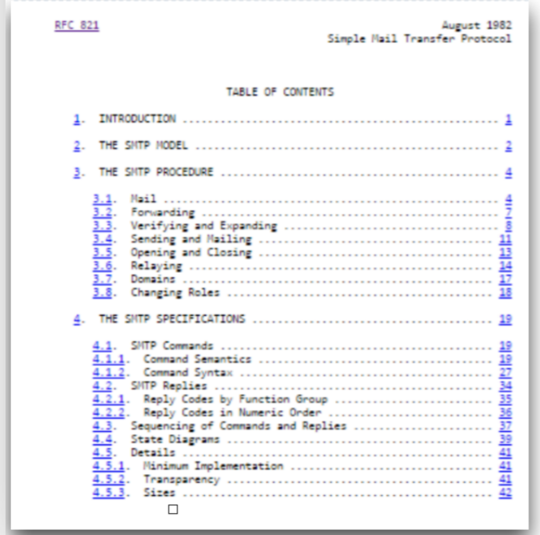
MPI backbone

Optimised MPI application

Non-Optimised MPI application

Session Type based Tools

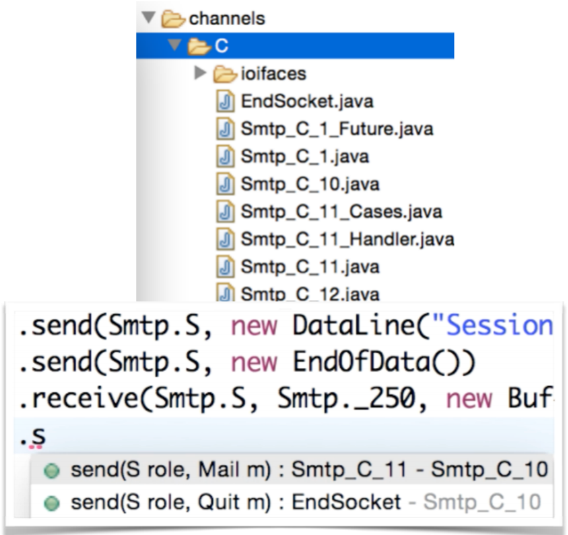
Java API Generation [FASE'16]



RFC 821 August 1982
Simple Mail Transfer Protocol

TABLE OF CONTENTS

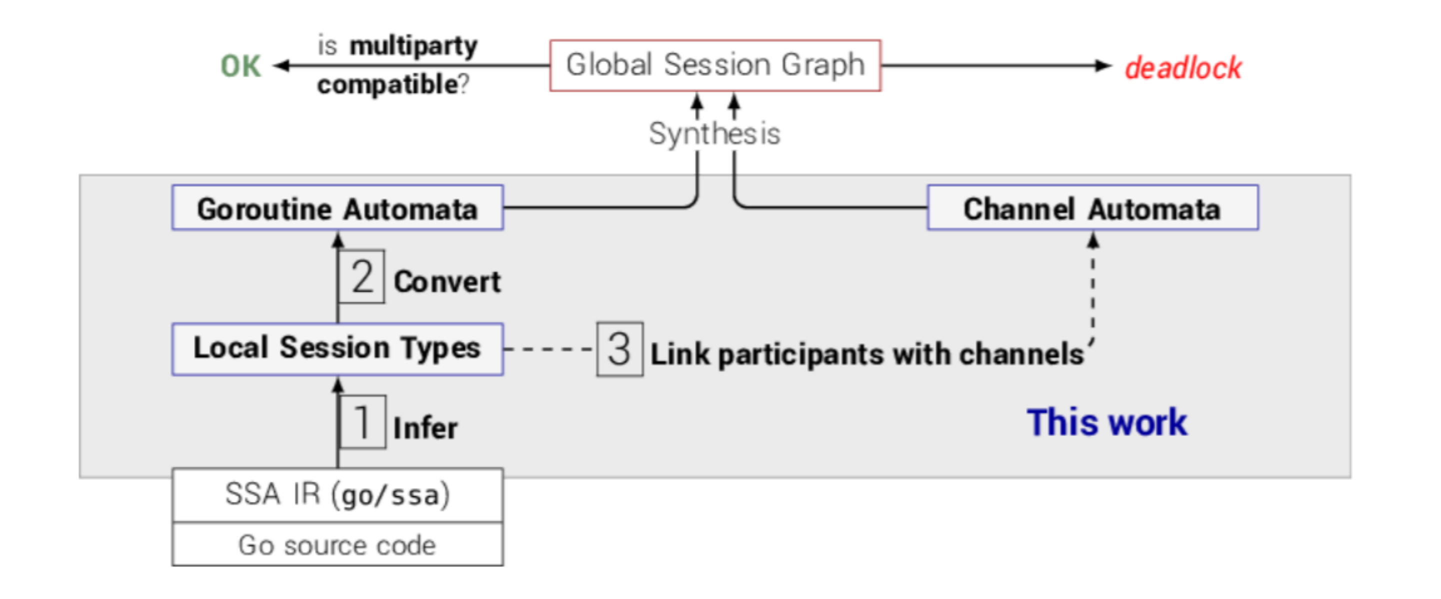
```
1.  INTRODUCTION .............................................. 1

2.  THE SMTP MODEL ............................................ 2

3.  THE SMTP PROCEDURE ........................................ 4

   3.1.  Mail ................................................. 4
   3.2.  Forwarding ........................................... 7
   3.3.  Verifying and Expanding .............................. 8
   3.4.  Sending and Mailing ................................. 11
   3.5.  Opening and Closing ................................. 13
   3.6.  Relaying ............................................ 14
   3.7.  Domains ............................................. 17
   3.8.  Changing Roles ...................................... 18

4.  THE SMTP SPECIFICATIONS .................................. 19

   4.1.  SMTP Commands ....................................... 19
   4.1.1.  Command Semantics ................................. 19
   4.1.2.  Command Syntax .................................... 27
   4.2.  SMTP Replies ........................................ 34
   4.2.1.  Reply Codes by Function Group ..................... 35
   4.2.2.  Reply Codes in Numeric Order ...................... 36
   4.3.  Sequencing of Commands and Replies .................. 37
   4.4.  State Diagrams ...................................... 39
   4.5.  Details ............................................. 41
   4.5.1.  Minimum Implementation ............................ 41
   4.5.2.  Transparency ...................................... 41
   4.5.3.  Sizes ............................................. 42
```



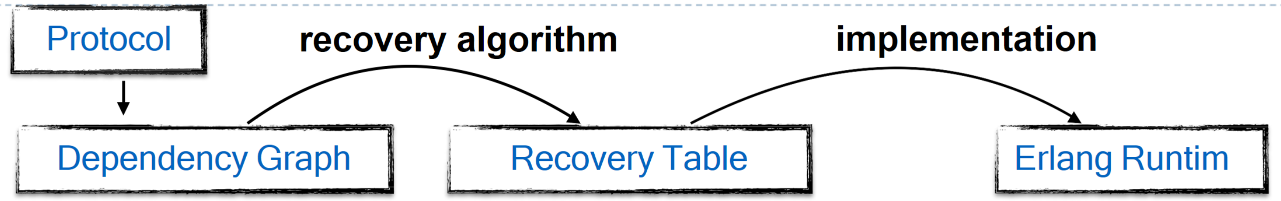
```
.send(Smtp.S, new DataLine("Session
.send(Smtp.S, new EndOfData())
.receive(Smtp.S, Smtp._250, new Buf
.s
```

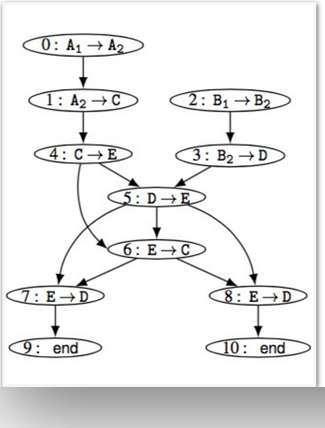
- send(S role, Mail m) : Smtp_C_11 - Smtp_C_10
- send(S role, Quit m) : EndSocket - Smtp_C_10

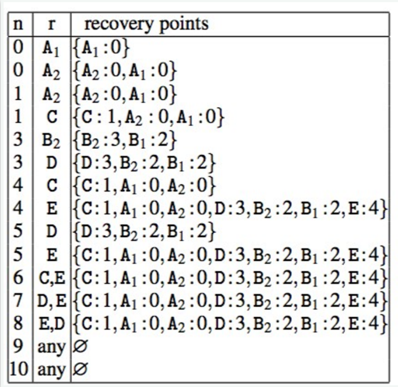
Deadlock Detection for Go [CC'16, POPL'17, ICSE'18]



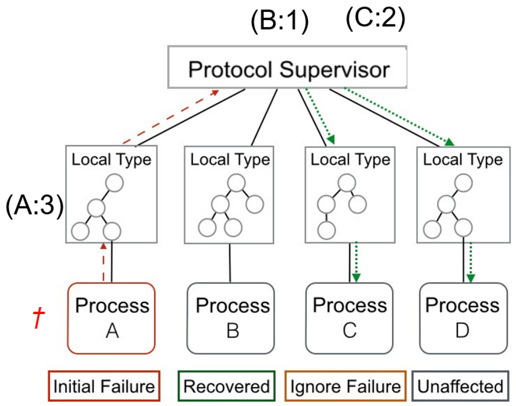
Safe Recovery for Erlang [CC'15]



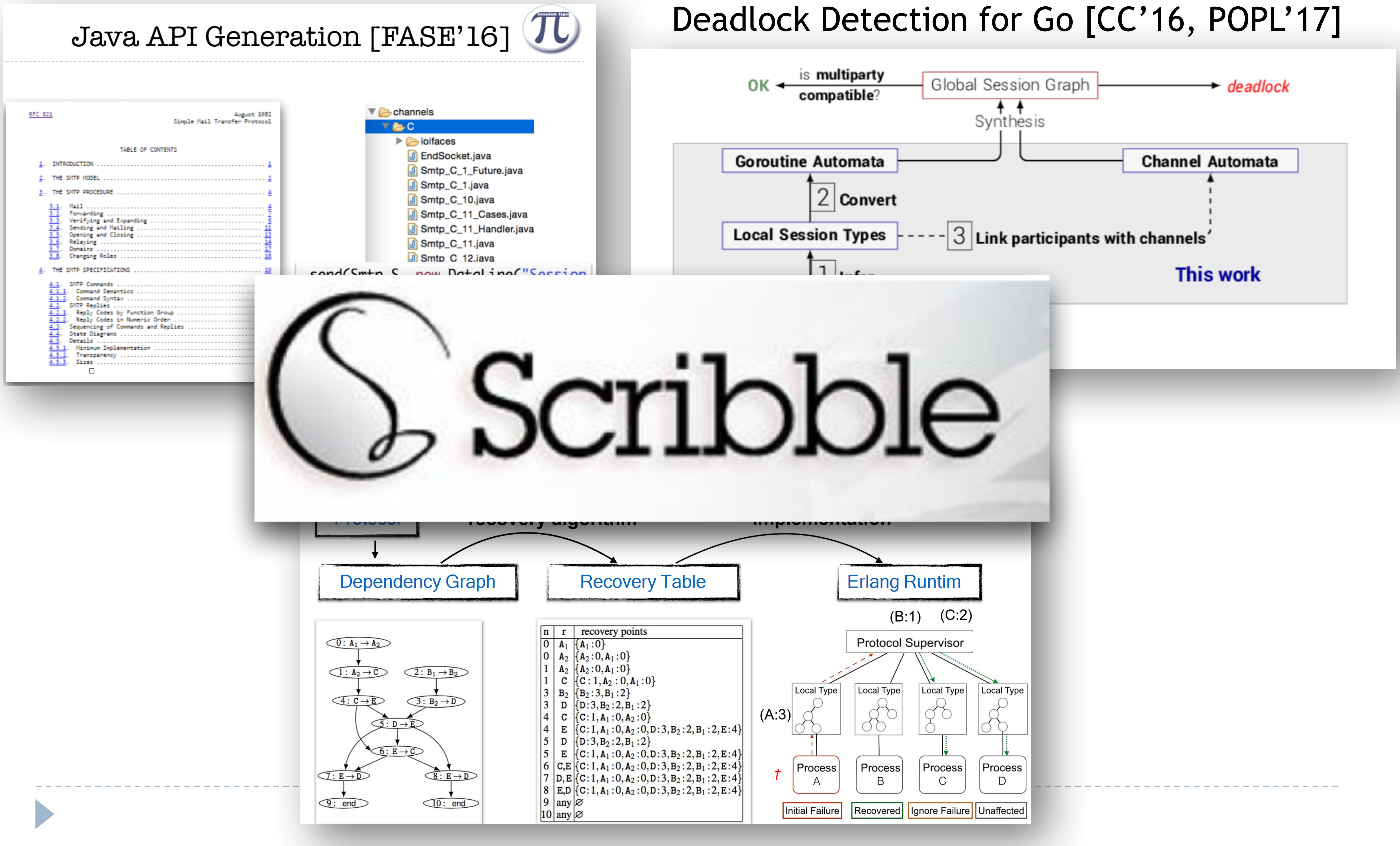




n	r	recovery points
0	A_1	$\{A_1:0\}$
0	A_2	$\{A_2:0, A_1:0\}$
1	A_2	$\{A_2:0, A_1:0\}$
1	C	$\{C:1, A_2:0, A_1:0\}$
3	B_2	$\{B_2:3, B_1:2\}$
3	D	$\{D:3, B_2:2, B_1:2\}$
4	C	$\{C:1, A_1:0, A_2:0\}$
4	E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
5	D	$\{D:3, B_2:2, B_1:2\}$
5	E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
6	C,E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
7	D,E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
8	E,D	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
9	any	∅
10	any	∅



Applications



Java API Generation [FASE'16]

RFC 821 — August 1982
Simple Mail Transfer Protocol

TABLE OF CONTENTS

1. INTRODUCTION 1
2. THE SMTP MODEL 2
3. THE SMTP PROCEDURE 4
   3.1. Mail 4
   3.2. Forwarding 7
   3.3. Verifying and Expanding 8
   3.4. Sending and Mailing 11
   3.5. Opening and Closing 13
   3.6. Relaying 14
   3.7. Domains 17
   3.8. Changing Roles 18
4. THE SMTP SPECIFICATIONS 19
   4.1. SMTP Commands ....
   4.1.1. Command Semantics ....
   4.1.2. Command Syntax ....
   4.2. SMTP Replies ....
   4.2.1. Reply Codes by Function Group ....
   4.2.2. Reply Codes in Numeric Order ....
   4.3. Sequencing of Commands and Replies ....
   4.4. State Diagrams ....
   4.5. Details ....
   4.5.1. Minimum Implementation ....
   4.5.2. Transparency ....
   4.5.3. Sizes ....

- channels
  - C
    - ioifaces
    - EndSocket.java
    - Smtp_C_1_Future.java
    - Smtp_C_1.java
    - Smtp_C_10.java
    - Smtp_C_11_Cases.java
    - Smtp_C_11_Handler.java
    - Smtp_C_11.java
    - Smtp_C_12.java

```
send(Smtp.S, new DataLine("Session
```

Deadlock Detection for Go [CC'16, POPL'17]

Scribble

n	r	recovery points
0	A_1	$\{A_1:0\}$
0	A_2	$\{A_2:0, A_1:0\}$
1	A_2	$\{A_2:0, A_1:0\}$
1	C	$\{C:1, A_2:0, A_1:0\}$
3	B_2	$\{B_2:3, B_1:2\}$
3	D	$\{D:3, B_2:2, B_1:2\}$
4	C	$\{C:1, A_1:0, A_2:0\}$
4	E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
5	D	$\{D:3, B_2:2, B_1:2\}$
5	E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
6	C,E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
7	D,E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
8	E,D	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
9	any	∅
10	any	∅

Session Types

Scribble

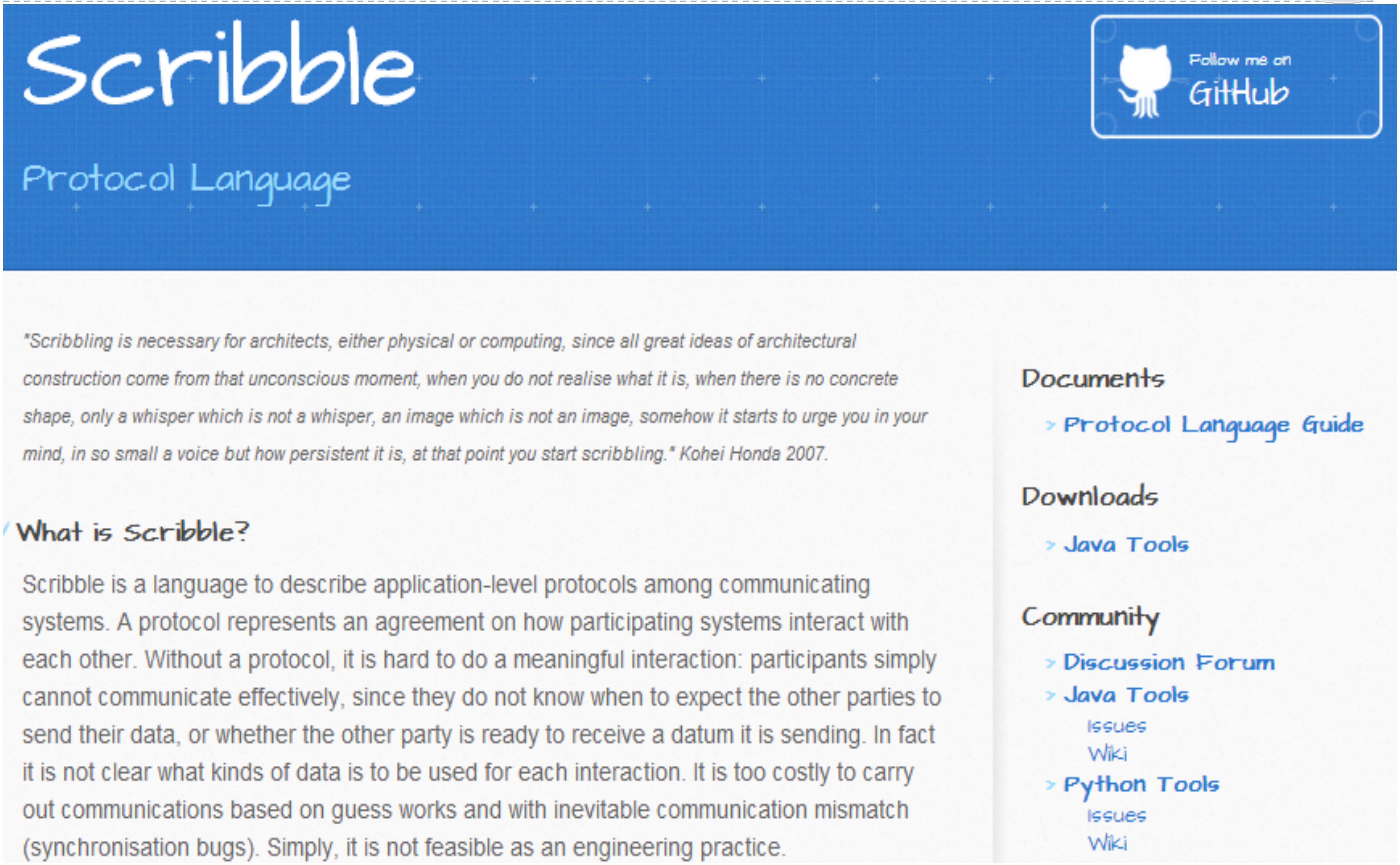
Scribble Protocol

- *"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*

- Basic example:

```
protocol HelloWorld {
    role You, World;
    Hello from You to World;
}
```

www.scribble.org



Scribble

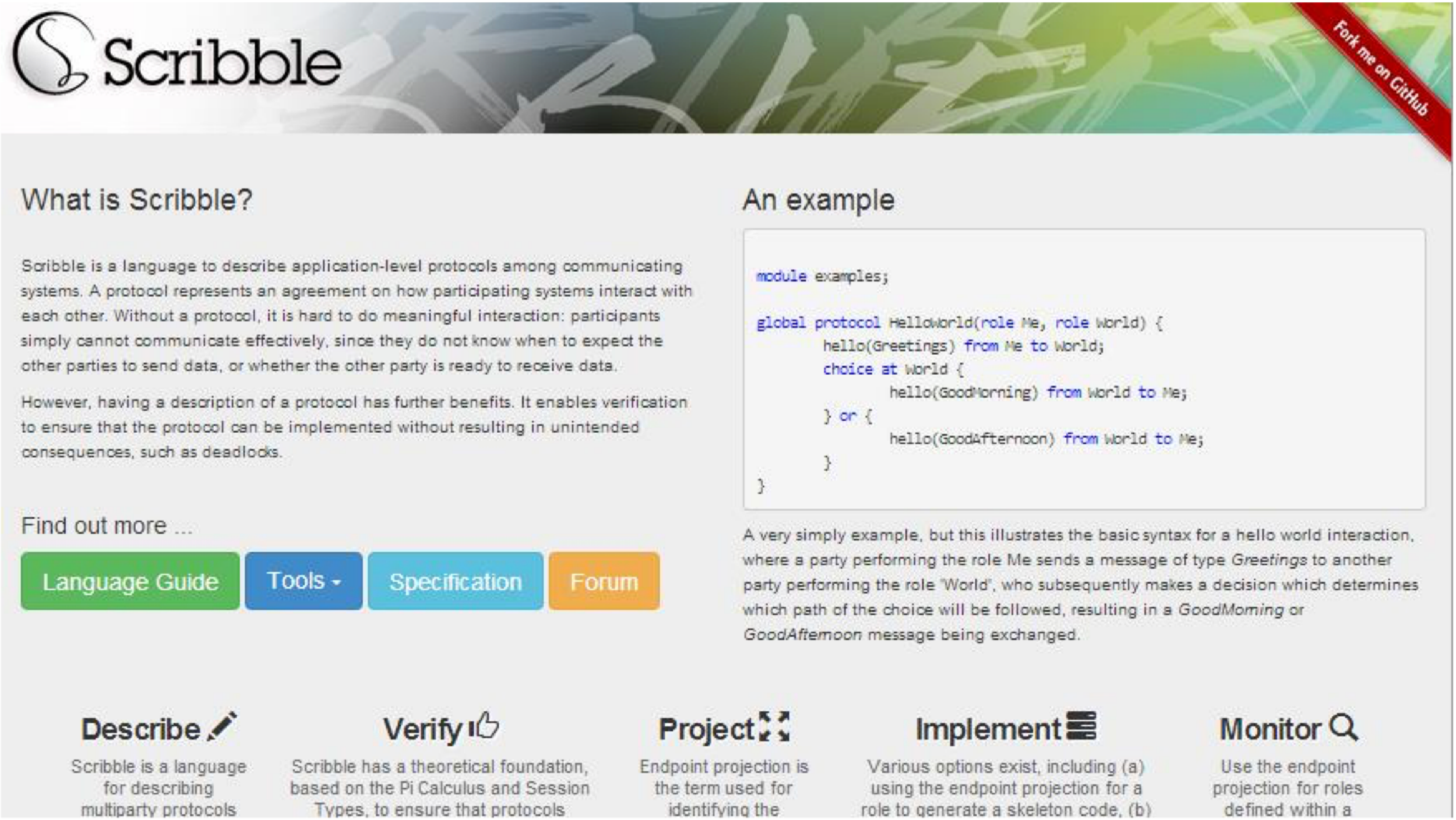
Protocol Language

"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.

Meet Scribble www.scribble.org



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 meaningful interaction: participants simply cannot communicate effectively, since they do not know when to expect the other parties to send data, or whether the other party is ready to receive data.

However, having a description of a protocol has further benefits. It enables verification to ensure that the protocol can be implemented without resulting in unintended consequences, such as deadlocks.

Find out more ...

An example

```
module examples;

global protocol HelloWorld(role Me, role World) {
        hello(Greetings) from Me to World;
        choice at World {
                hello(GoodMorning) from World to Me;
        } or {
                hello(GoodAfternoon) from World to Me;
        }
}
```

A very simply example, but this illustrates the basic syntax for a hello world interaction, where a party performing the role Me sends a message of type *Greetings* to another party performing the role 'World', who subsequently makes a decision which determines which path of the choice will be followed, resulting in a *GoodMorning* or *GoodAfternoon* message being exchanged.

Describe

Scribble is a language for describing multiparty protocols

Verify

Scribble has a theoretical foundation, based on the Pi Calculus and Session Types, to ensure that protocols

Project

Endpoint projection is the term used for identifying the

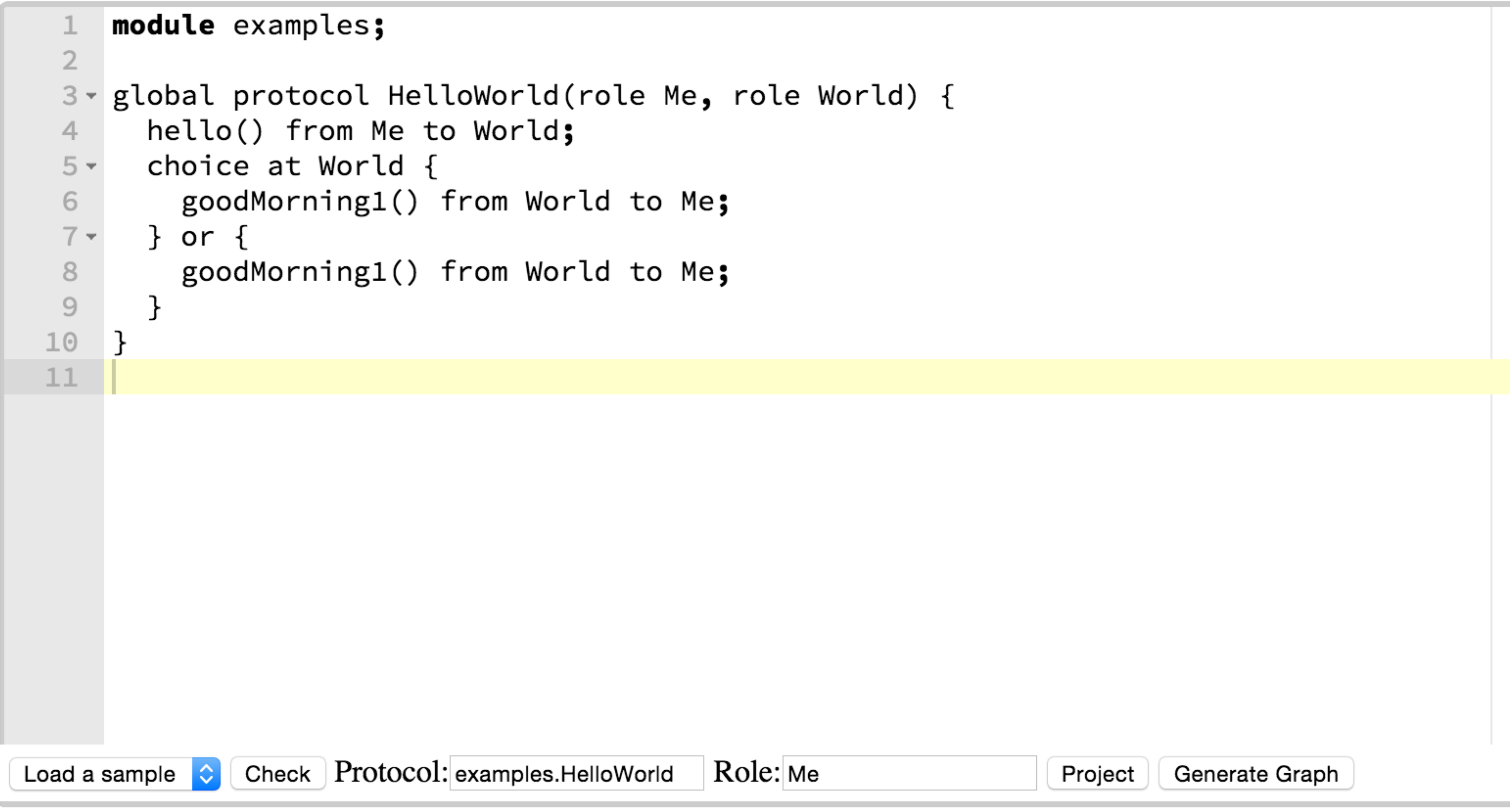
Implement

Various options exist, including (a) using the endpoint projection for a role to generate a skeleton code, (b)

Monitor

Use the endpoint projection for roles defined within a

Let's try some protocols: http://scribble.doc.ic.ac.uk/



```
module examples;

global protocol HelloWorld(role Me, role World) {
  hello() from Me to World;
  choice at World {
    goodMorning1() from World to Me;
  } or {
    goodMorning1() from World to Me;
  }
}
```

Protocol: examples.HelloWorld

Role: Me

Example

protocol def

recursion

send-receive

choice

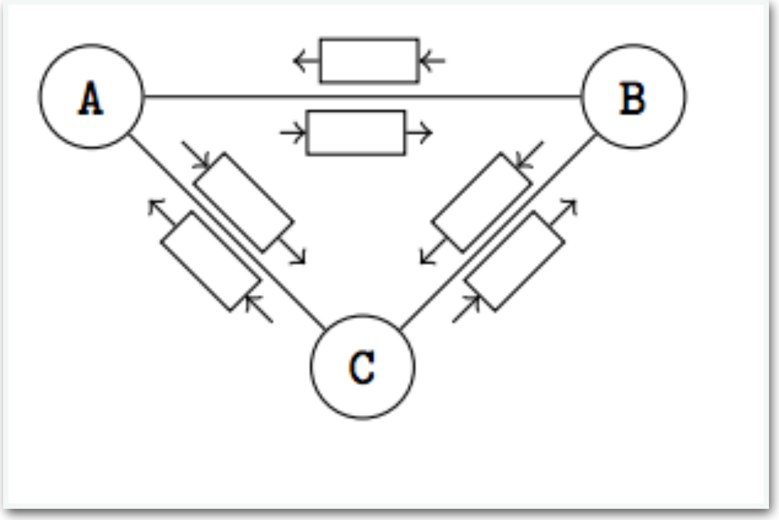
```
global protocol Q&A(role me, role you){
rec loop {
      ask(string) from you to me;
      choice at me
            { response (string) from me to you;
               continue loop; }
       or { enough() from me to you; }}
```

Scribble

Protocol Validation

Good/Bad MPST by example

- Communication model:
  - asynchronous, reliable, role-to-role ordering
  - MPST applies to transports that fit this model
    - TCP, HTTP, …, AMQP, …shared memory
- MPST protocols should be fully specified
  - no implicit messages needed to conduct a session



Next….

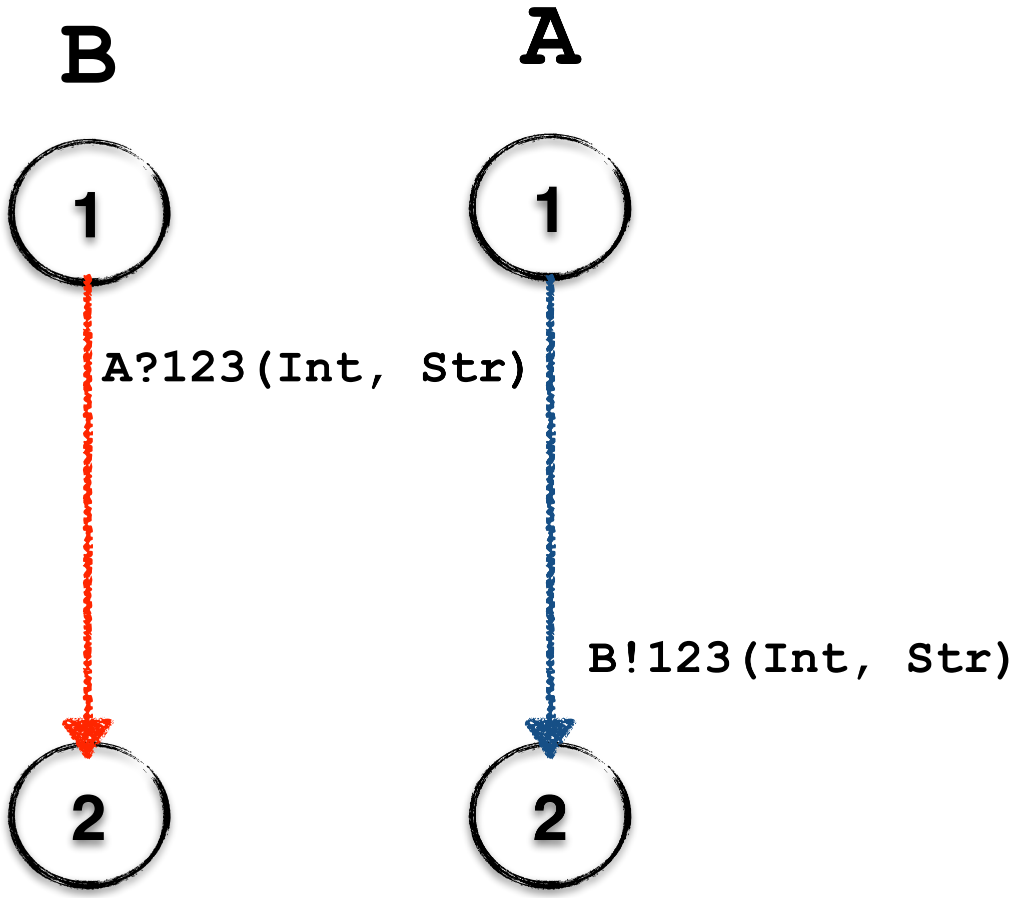
- Core Scribble constructs
- What can go wrong ?
- MPST safety and liveness errors (informally)
- How are they ruled out (syntactically)

Scribble constructs:
Role-to-role Message passing

```
123(Int, String) from A to B;
```

Payload types

Operator (label, header, …)



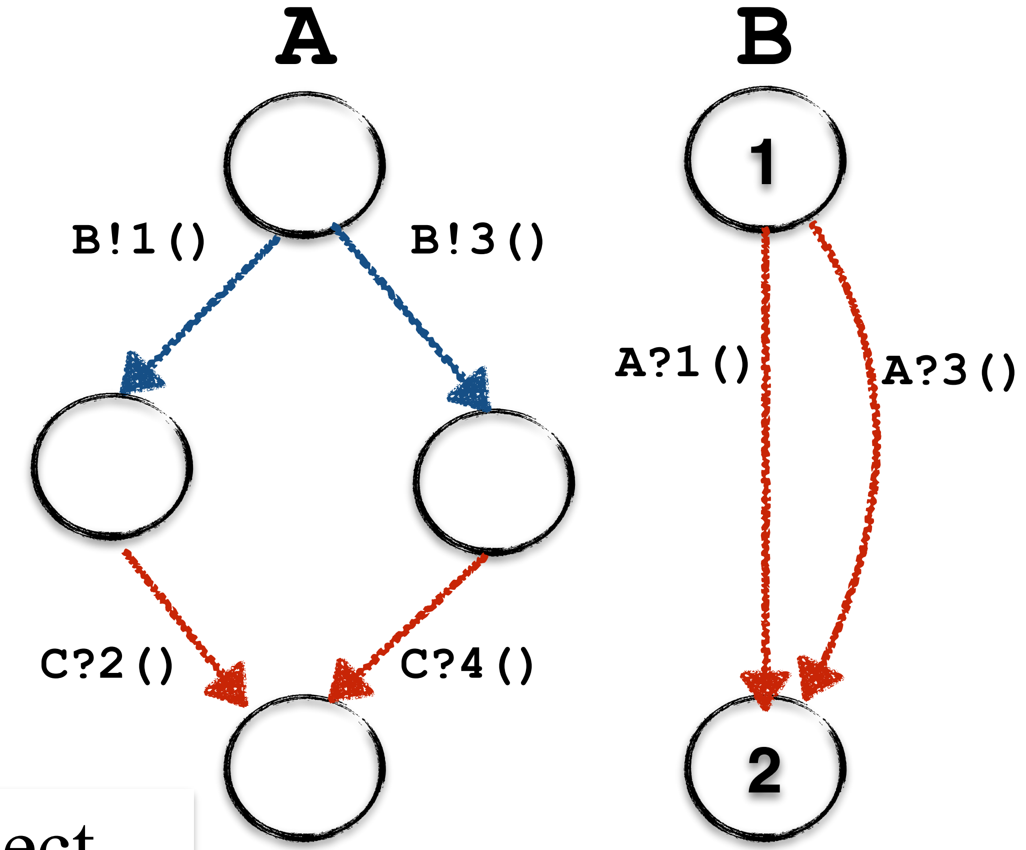
- Empty operator and/or payload is allowed

```
() from A to B;
```

✅

Scribble constructs:
"Located" choice

```
choice at A {
  1() from A to B;
  2() from A to C;
  } or {
  3() from A to B;
  4() from A to C;
  }
}
```



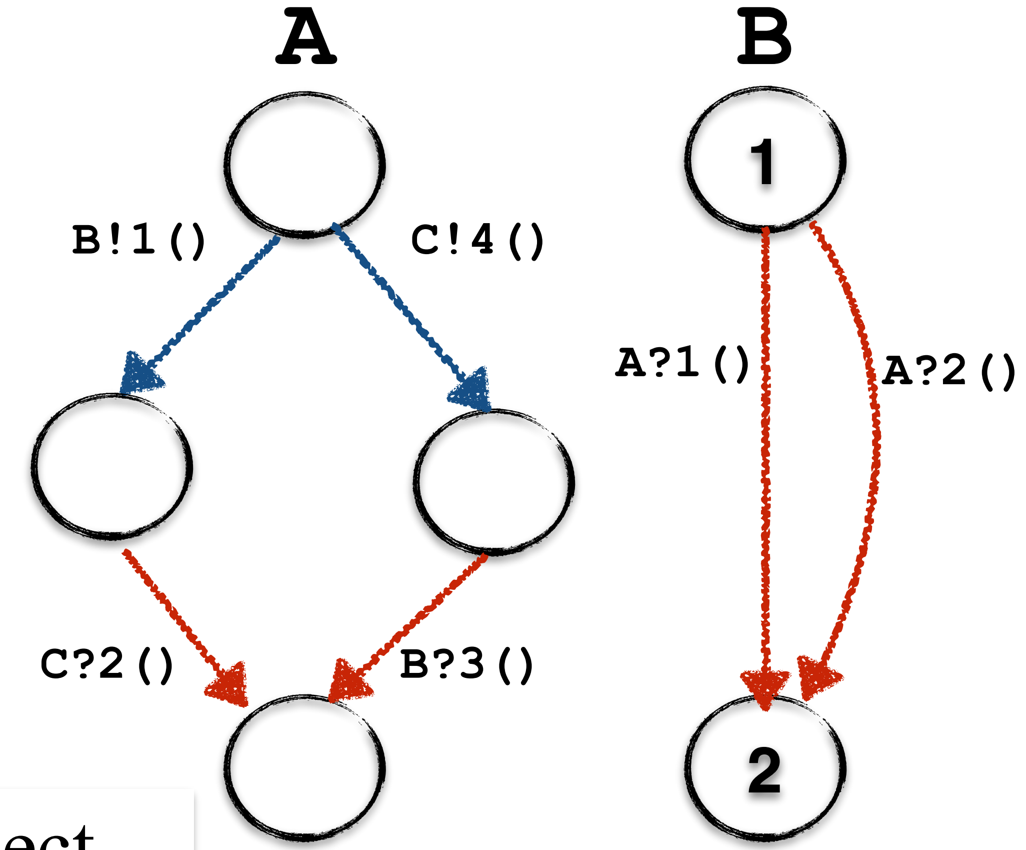
- Internal choice by global choice subject
- External choice for all other roles

Condition

- Only enabled roles can send messages in choice paths
  - Start role enabled, other disabled
  - a role is enabled by receiving a message from an enabled role

Scribble constructs:
“Located” choice

```
choice at A {
  1() from A to B;
  2() from A to C;
  } or {
  4() from A to C;
  3() from A to B;
  }
}
```



- Internal choice by global choice subject
- External choice for all other roles

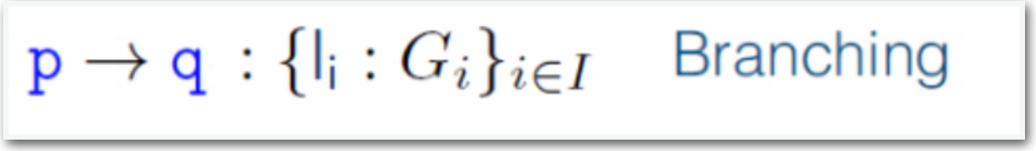
Condition

- Only enabled roles can send messages in choice paths
  - Start role enabled, other disabled
  - a role is enabled by receiving a message from an enabled role

Scribble constructs:
“Located” choice

```
 choice at A {
   buyer1(int) from A to B; // Total to pay
   (int) from B to A;// B will pay that much
   buyer1(int) from A to C; // C pays the remainder
  } or {
   buyer2(x:int, y:int) from A to C; // Total to pay
   (Int) from C to A; // C pays that much
   buyer2(x:int, y:int) from A to  B;// B pays the remainder
  }
}
```

- More flexible than directed choice



$\mathsf{p} \to \mathsf{q} : \{\mathsf{l_i} : G_i\}_{i \in I}$ Branching

- Branching via different payloads not allowed

```
choice at A {1() from A to B;} or {1(int) from A to B;}
```

Exercise:
“Located” choice

Condition

- Only enabled roles can send messages in choice paths
  - Start role enabled, other disabled
  - a role is enabled by receiving a message from an enabled role

```
 choice at A {
   1() from A to B;
   1() from B to C;
   1() from C to A;
 } or {
   2() from B to A;    ✖ Role B not enabled
   choice at B {
     2() from B to C;
   } or {
     3() from B to C;
   }
   4() from C to A;
}
```

What actually goes wrong ?

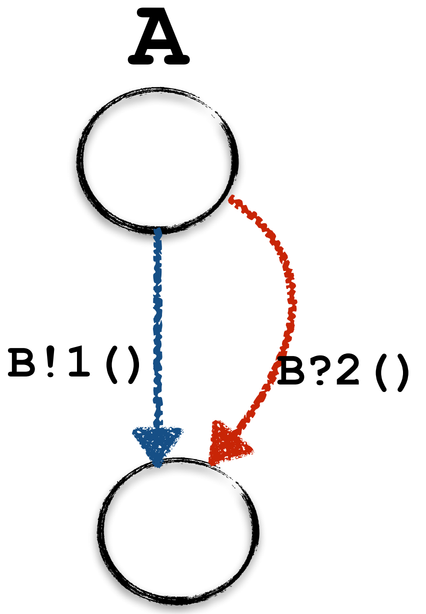
- MPST Safety errors:
  - reception error, orphan message, deadlock

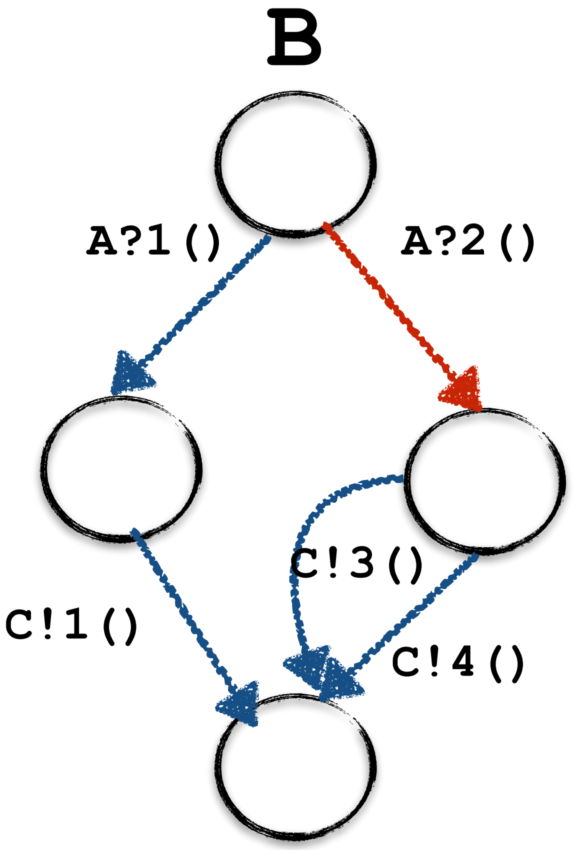
Exercise:
“Located” choice

What actually goes wrong ?

- MPST Safety errors:
  - reception error, orphan message, deadlock

```
choice at A {
  1() from A to B;
  1() from B to C;
  1() from C to A;
} or {
  2() from B to A;  ✖ Role B not enabled
  choice at B {
    2() from B to C;
  } or {
    3() from B to C;
  }
  4() from C to A;
}
```





Is this protocol OK? 1/4

```
choice at A {
   1() from A to B;
   3() from B to C; ❌
   4() from C to A;
} or {
   2() from A to B;
   3() from B to C; ❌
   5() from C to A;
}
```

Errors explained ?

- Ambitious choice for C
  - Should C send a 4 or 5 to A?
  - potential reception errors (4, 5) if interpreted non-deterministically
- *Non-deterministic choice at C* inconsistent with the choice by A
  - Not mergeable in syntactic projections
  - has to merge continuations (undefined for distinct outputs)

Is this protocol OK? 1/4

```
choice at A {
  1() from A to B;
  3() from B to C;
  4() from C to A;
} or {
  2() from A to B;
  3() from A to C;
  5() from A to C;
}
```

How to fix t?

Is this protocol OK? 1/4

```
choice at A {
  1() from A to B;
  3a() from B to C;
  4() from C to A;
} or {
  2() from A to B;
  3b() from A to C;
  5() from A to C;
}
```

Distinguish label 3!

Is this protocol OK? 2/4

✅

```
choice at A {
   1() from A to B;
   3() from B to C;
   do Merge(A, C);
} or {
   2() from A to B;
   3() from B to C;
   do Merge(A, C);
}

global protocol Merge(role A, role C){
   4() from A to C;
}
```

- Duplicate cases inherently mergeable, e.g [POPL’11]

Is this protocol OK? 2/4

✅

```
choice at A {
   1() from A to B;
   3() from B to C;
   do Merge(A, C);
} or {
   2() from A to B;
   3() from B to C;
   do Merge(A, C);
}

global protocol Merge(role A, role C){
  choice at A {
     4() from A to C;
  } or {
     5() from A to C;
  }
}
```

- Duplicate cases inherently mergeable, e.g [POPL’11]

Is this protocol OK? 3/4

```
choice at A {
  1a() from A to B;
  2() from A to C;
  3() from B to C;  ❌
  4() from C to A;
} or {
  1b() from A to B;
  3() from B to C;  ❌
  4() from C to A;
}
```

Errors explained ?

- "Race condition" on choice on C due to asynchrony
  - What should C do after receiving a 3?
  - Potential orphan message (2) if interpreted as multi-queue FIFO
- Inconsistent external choice subject
  - (trivially non-mergeable in standard MPST)
  - A role must be enabled by the same role in choice paths

Is this protocol OK? 4/4

```
choice at A {
   1() from A to B;
   2() from A to C;  ✖
 } or {
   3() from A to B;
}
```

Errors explained ?

- Unrealisable choice at C
  - No implicit message can be assumed, e.g end of session
  - How can C determine if a message is coming?
  - Potential deadlock (C waiting for A), or potential orphan (2), depending on the interpretation
- Empty action option to terminal state
  - can’t merge end type with anything else

Quiz: Mergeability

```
choice at A {
   1() from A to B;
   2() from C to B;
} or {
   3() from A to D;
   4() from D to B;
}
```

✗

```
choice at A {
   1() from A to B;
   2() from C to D;
} or {
   3() from A to B;
   4() from C to D;
}
```

✓

```
choice at A {
   1() from A to C;
   2() from C to D;
} or {
   3() from A to B;
   2() from C to D;
}
```

✗

```
choice at A {
   1() from A to C;
   2() from B to C;
} or {
   3() from A to B;
   4() from B to C;
}
```

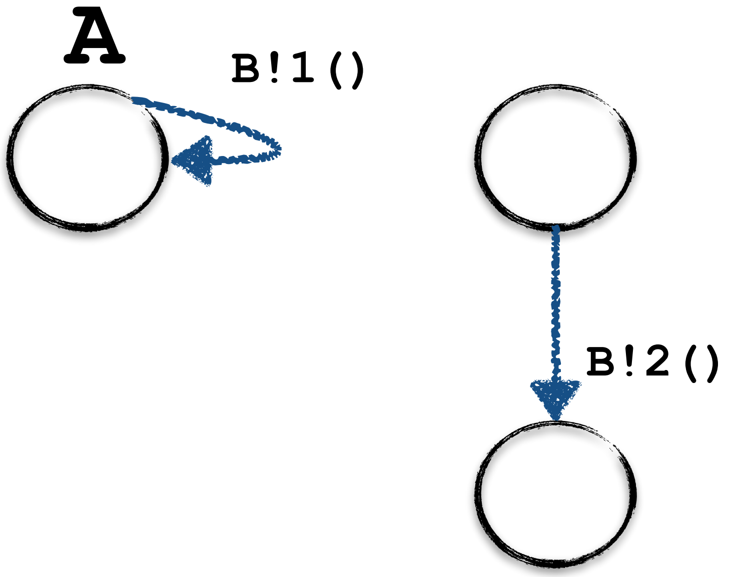
✓

Scribble construct: **Recursion**

- Tail recursion with recursive scopes

```
rec X {
  1() from A to B;
  continue X;
}
2() from A to B;
```

✖ **Dead code**



Condition

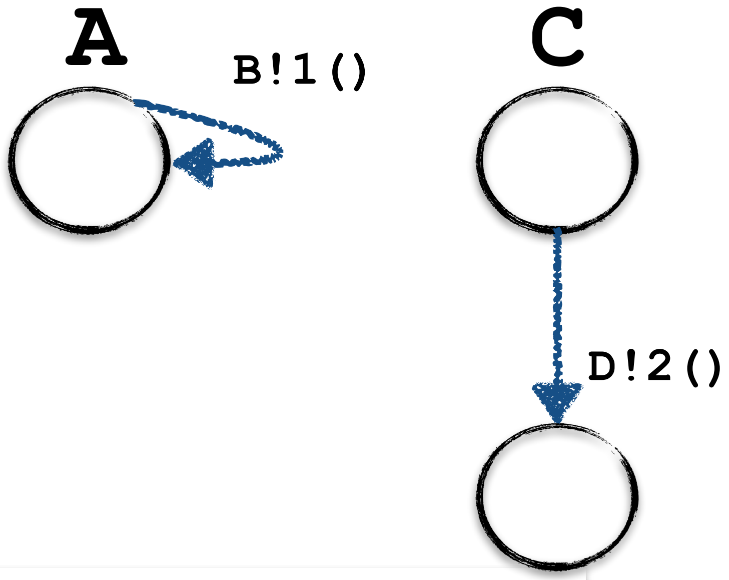
- Reachability of protocol states (no “dead code”)
  - Checked via projection (reachability w.r.t per-role protocol flow)
- Regular interaction structure at endpoints (CFSM)

Scribble construct: **Recursion**

- Tail recursion with recursive scopes

```
rec X {
  1() from A to B;
  continue X;
}
2() from A to B; ❌ Dead code
```

```
rec X {
  1() from A to B;
  continue X;
}
2() from C to D; ✅
```



Condition

- Reachability of protocol states (no “dead code”)
  - Checked via projection (reachability w.r.t per-role protocol flow)
- Regular interaction structure at endpoints (CFSM)

Is this protocol ok? 1/4

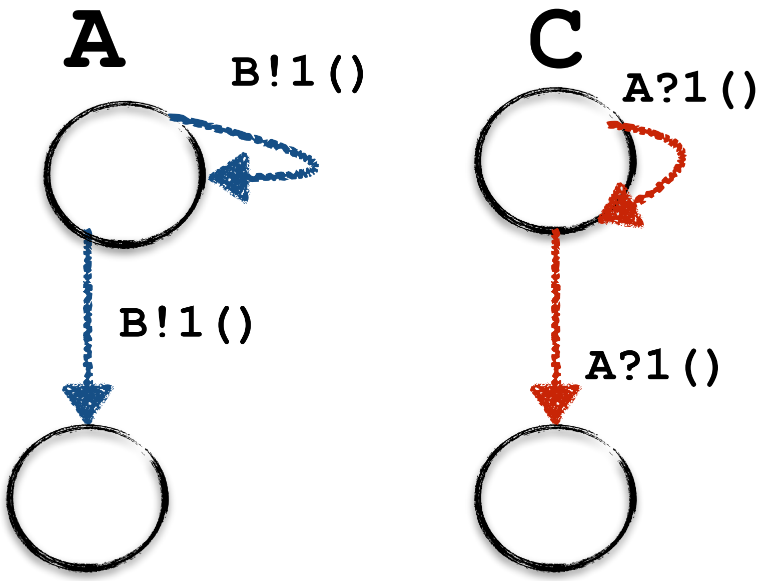
```
rec X {
  choice at A {
    1() from A to B;
    continue X;
    //2() from A to B;
  } or {
    3() from A to B;
}
4() from A to B; ✖
}
5() from A to B;
```

Condition

- Reachability of protocol states (no “dead code”)
  - Checked via projection (reachability w.r.t per-role protocol flow)
- Regular interaction structure at endpoints (CFSM)

Is this protocol OK? 2/4

```
rec X {
   choice at A {
      1() from A to B;
      continue X;
   } or {
      1() from A to B;
   }
}
```



Potential ***deadlocks*** or ***orphans***

Is this protocol ok? 3/4

```
  choice at A {
    rec X {
      1() from A to B;
      1() from B to C;
      continue X;
    }
  } or {
    2() from A to B;
    2() from B to C;
}
```

- Safety errors? (reception errors, orphan messages, deadlock)
  - Consider the FSM at A?

Is this protocol ok? 3/4

```
choice at A {
   rec X {
     1() from A to B;
     //1() from B to C;
     continue X;
   }
} or {
  2() from A to B;
  2() from B to C;
}
```

- Safety errors?
  - hint: Consider the FSM at A?
  - How about now?

Is this protocol ok? 3/4

```
choice at A {
  rec X {
    1() from A to B;
    //1() from B to C;
    continue X;
} or {
  2() from A to B;
  2() from B to C;    ✖
}
```

- Safety errors?
  - hint: Consider the FSM at A?
  - How about now?

- Liveness errors?
  - Role progress

Is this protocol ok? 4/4

```
choice at A {
  rec X {
    1() from A to B;
    //1() from B to C;
    continue X;
  }
} or {
    2() from A to B;

} 2() from C to B; ✖
```

- Safety errors?
  - hint: Consider the FSM at A?
  - How about now?

- Liveness errors?
  - Role progress
  - Message liveness (Eventual reception)

Is this protocol ok? 4/4

```
rec X {
  choice at A {
    1() from A to B;
    continue X;
  } or {
    2() from A to B;
    2() from B to C;
  }
}
```

- But is this a good protocol
  - depends …*fairness* of output choices

Homework

```
rec X {
  choice at A {
    1() from A to B;
    2() from B to C;
    3() from C to B;
} or {
    4() from A to C;
    5() from C to B;
}
continue X;}
```

Why does Scribble not allow this protocol?

Scribble

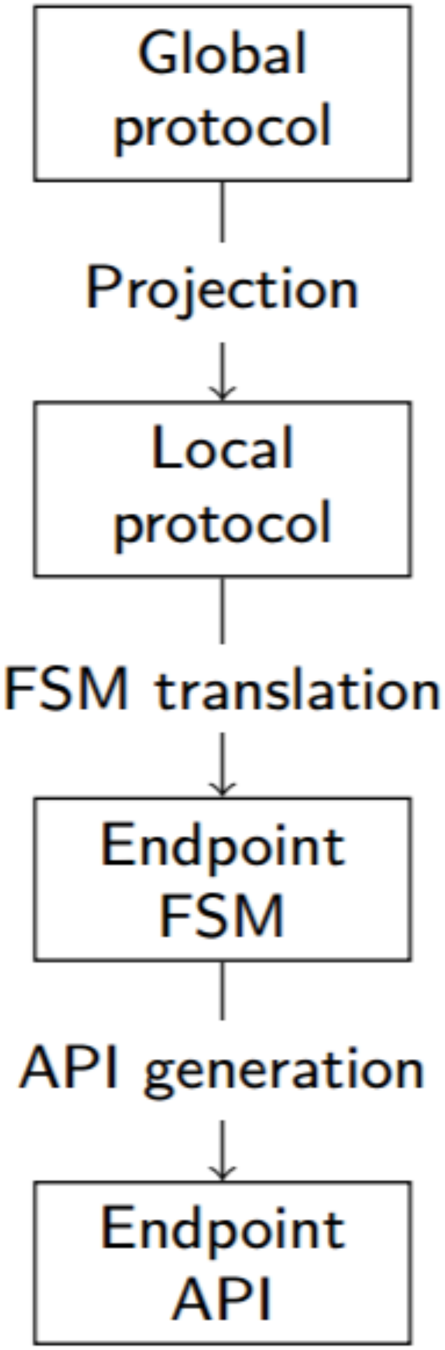
Program Verification

or….

How to program SMTP in 5 min

Scribble Endpoint API generation toolchain

- Protocol spec. as Scribble protocol (asynchronous MPST)
  - Global protocol validation
    (safely distributable asynchronous protocol)
  - Syntactic projection to local protocols
    (static session typing if supported)
  - Endpoint FSM (EFSM) translation
    (dynamic session typing by monitors)
    - Protocol states as state-specific channel *types*
    - Call chaining API to link successor states
- Java APIs for implementing the endpoints



Global protocol

Projection

Local protocol

FSM translation

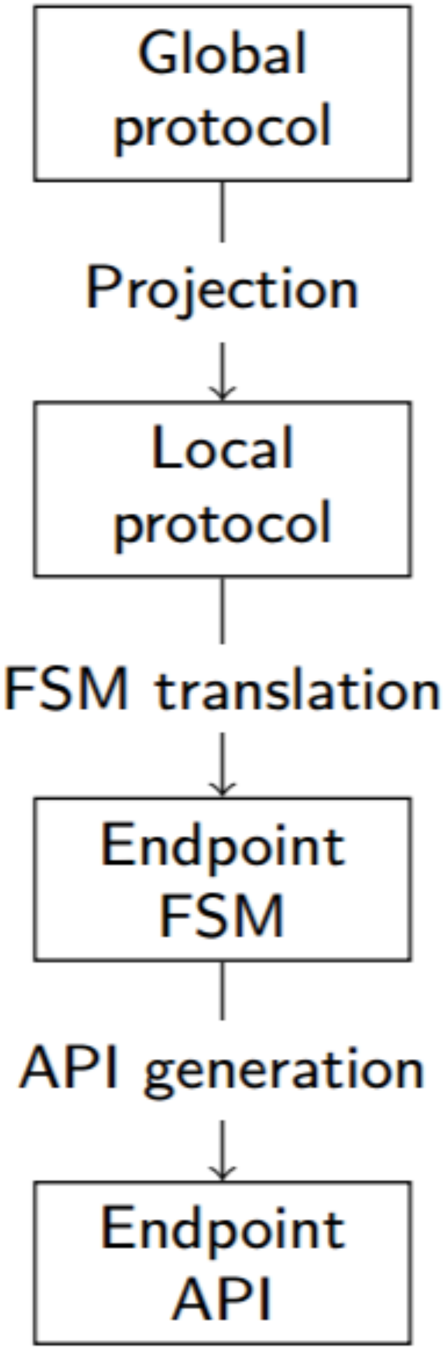
Endpoint FSM

API generation

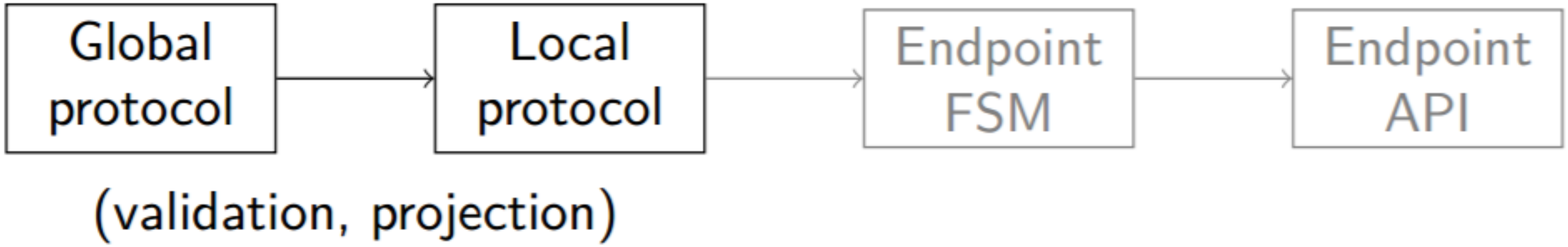
Endpoint API

Scribble Endpoint API generation toolchain

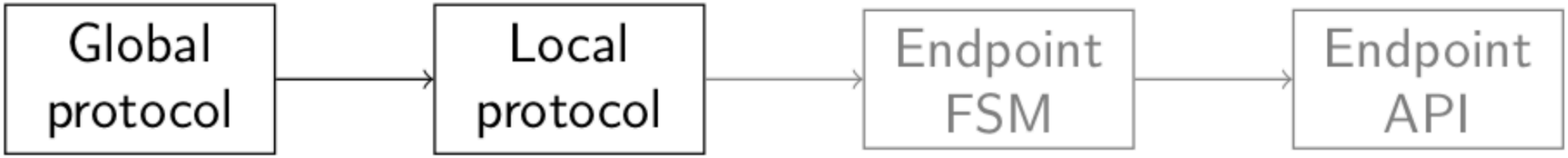
- Protocol spec. as Scribble protocol (asynchronous MPST)
  - Global protocol validation
    (safely distributable asynchronous protocol)
  - Syntactic projection to local protocols
    (static session typing if supported)
  - Endpoint FSM (EFSM) translation
    (dynamic session typing by monitors)
    - Protocol states as state-specific channel *types*
    - Call chaining API to link successor states
- Java APIs for implementing the endpoints



Example: Adder



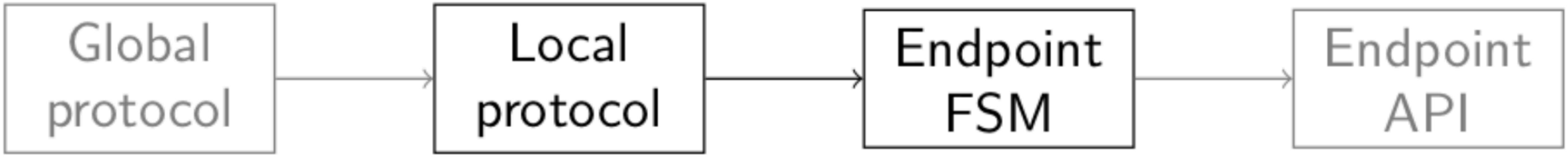
```
global protocol Adder(role C, role S) {
  choice at C {
    Add(Integer, Integer) from C to S;
    Res(Integer) from S to C;
    do Adder(C, S);
  } or {
    Bye() from C to S;
    Bye() from S to C;
  }
}
```



(validation, projection)

```
global protocol Adder(role C, role S) {
  choice at C {
    Add(Integer, Integer) ~~from C~~ to S;
    Res(Integer) from S ~~to C;~~
    do Adder(C, S);
  } or {
    Bye() ~~from C~~ to S;
    Bye() from S ~~to C;~~
  }
}
```

```
local protocol Adder_C(role C, role S) {
  choice at C {
      Add(Integer, Integer) to S;
      Res(Integer) from S;
      do Adder(C, S);
   } or {
      Bye() to S;
      Bye() from S;
  }}
```



Global protocol

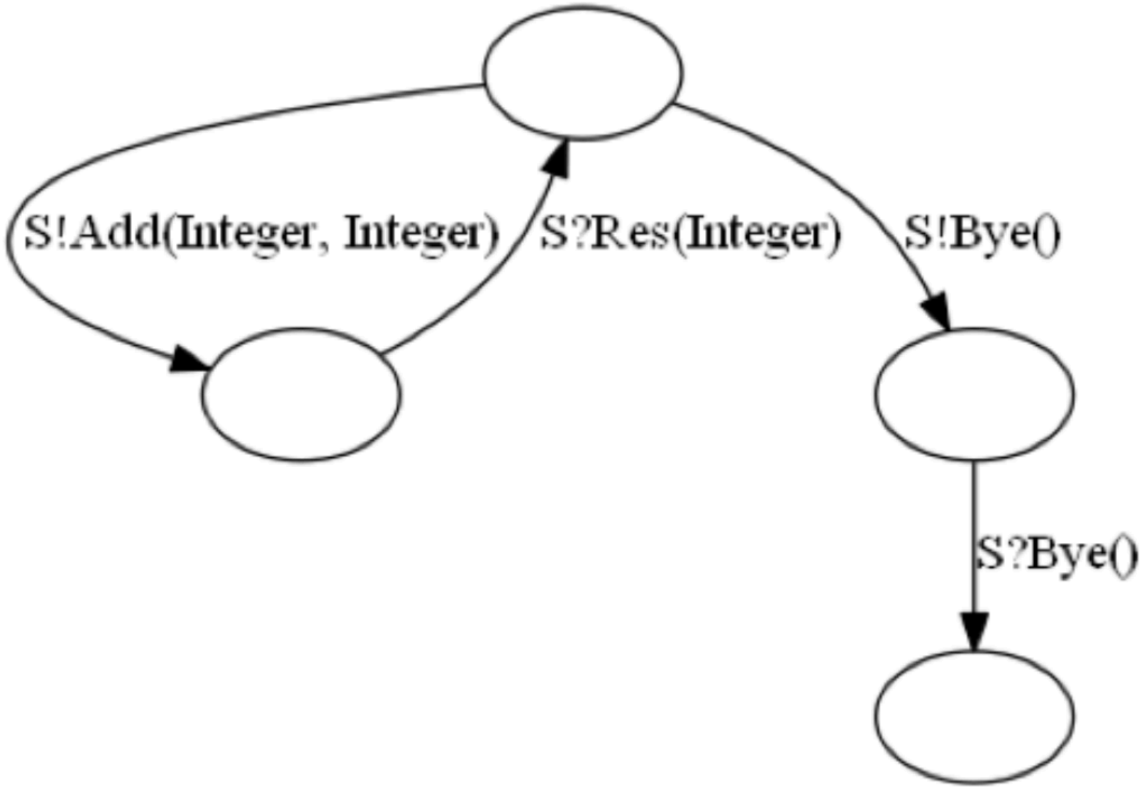
Local protocol

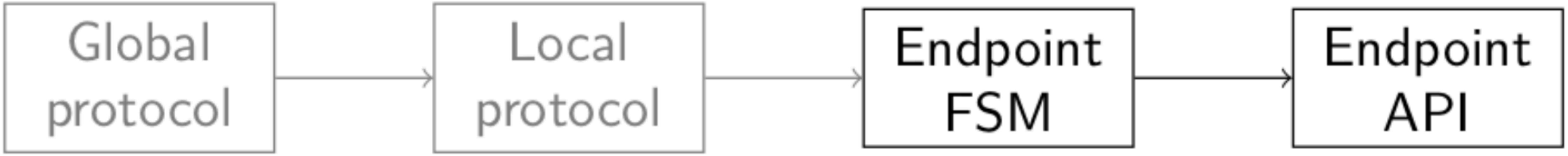
Endpoint FSM

Endpoint API

(FSM translation)

```
local protocol Adder_C(role C, role S) {
  choice at C {
      Add(Integer, Integer) to S;
      Res(Integer) from S;
      do Adder(C, S);
  } or {
     Bye() to S;
    Bye() from S;
  }}
```





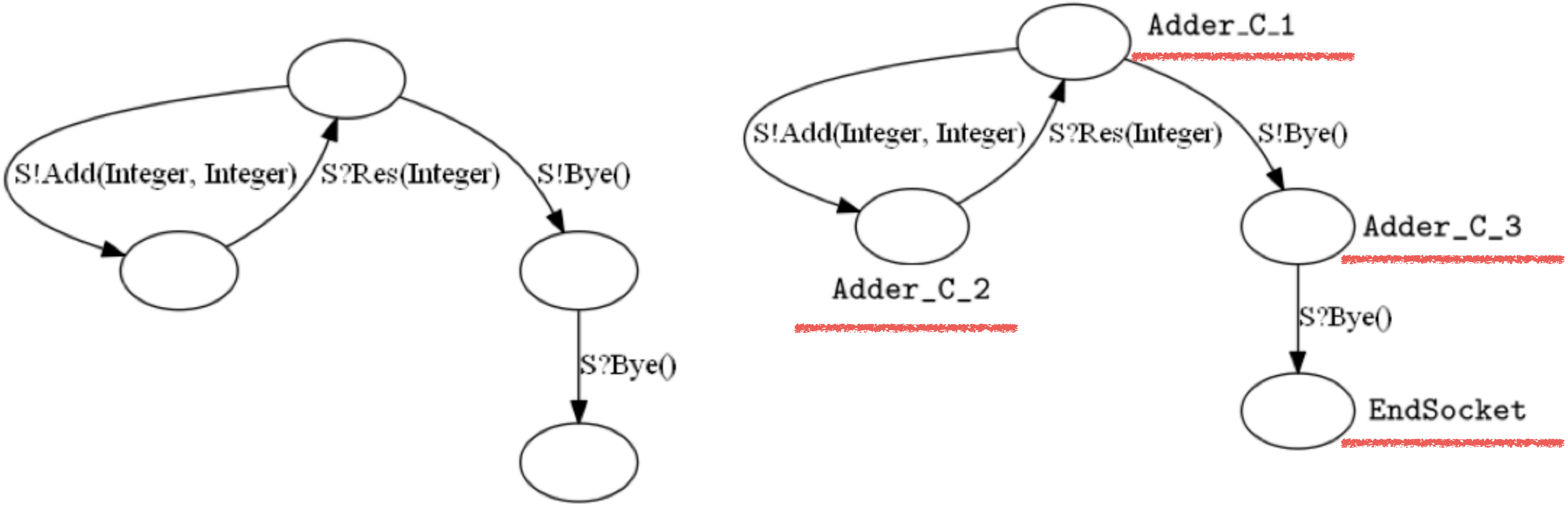
Global protocol

Local protocol

Endpoint FSM

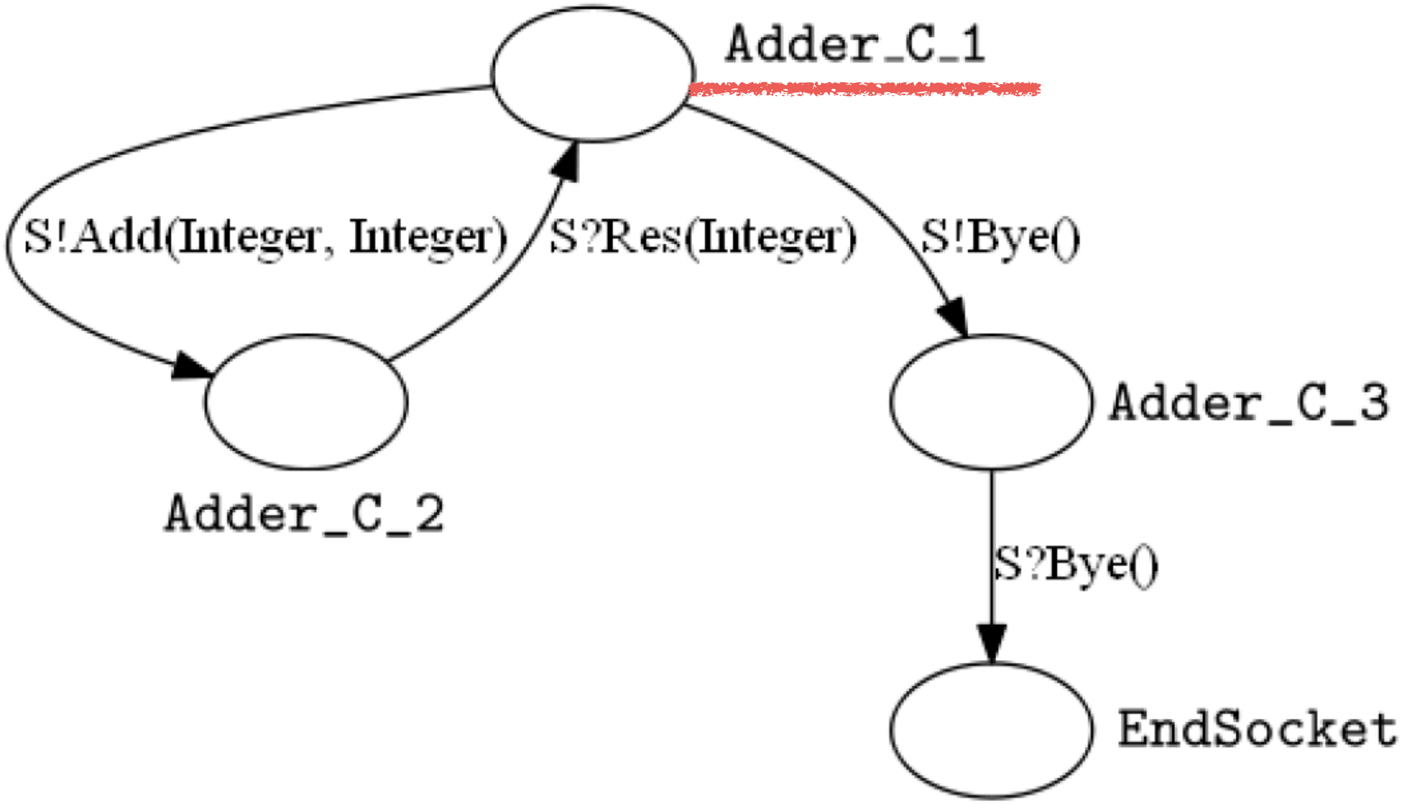
Endpoint API

API generation



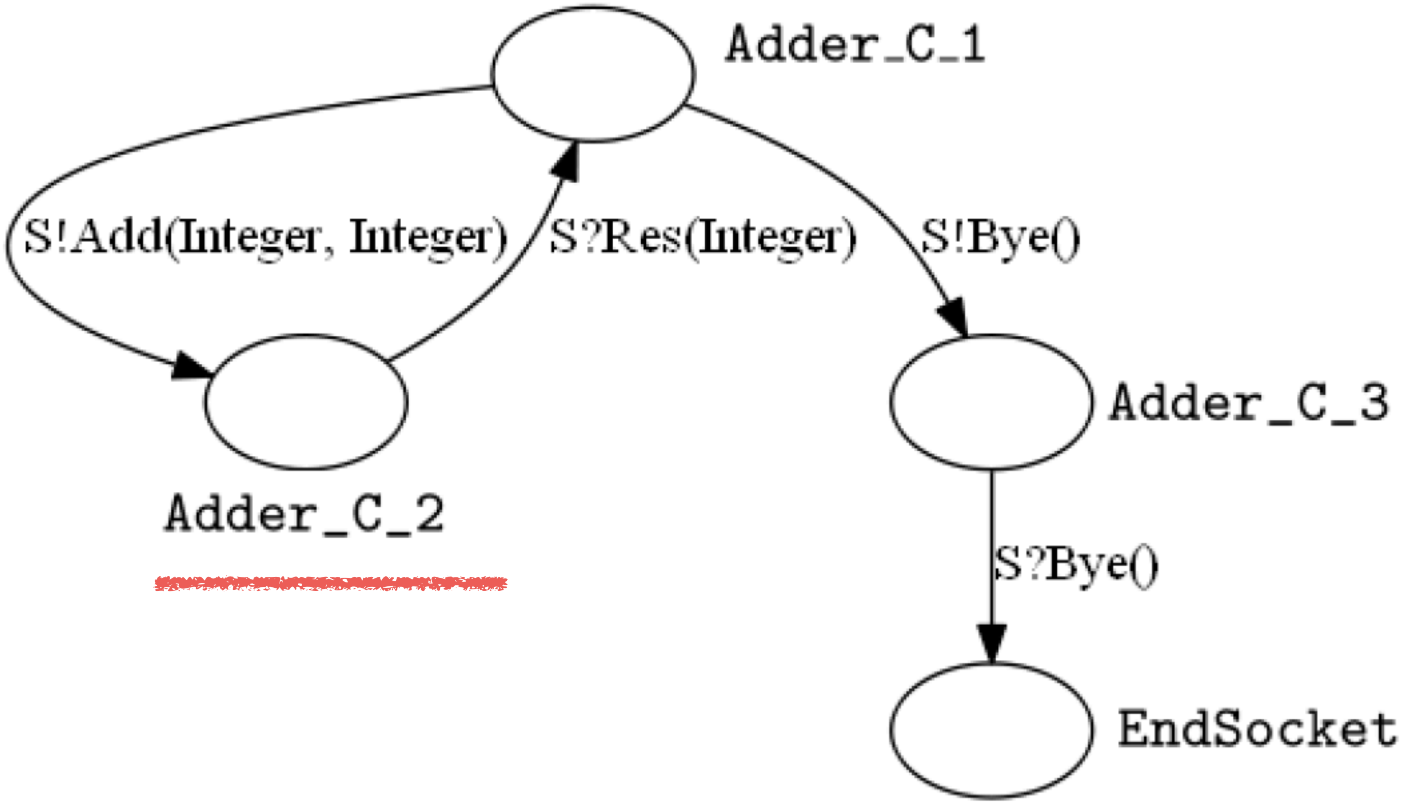
Turn each state into a class

Generated State/Channel Classes offer exactly the valid operation



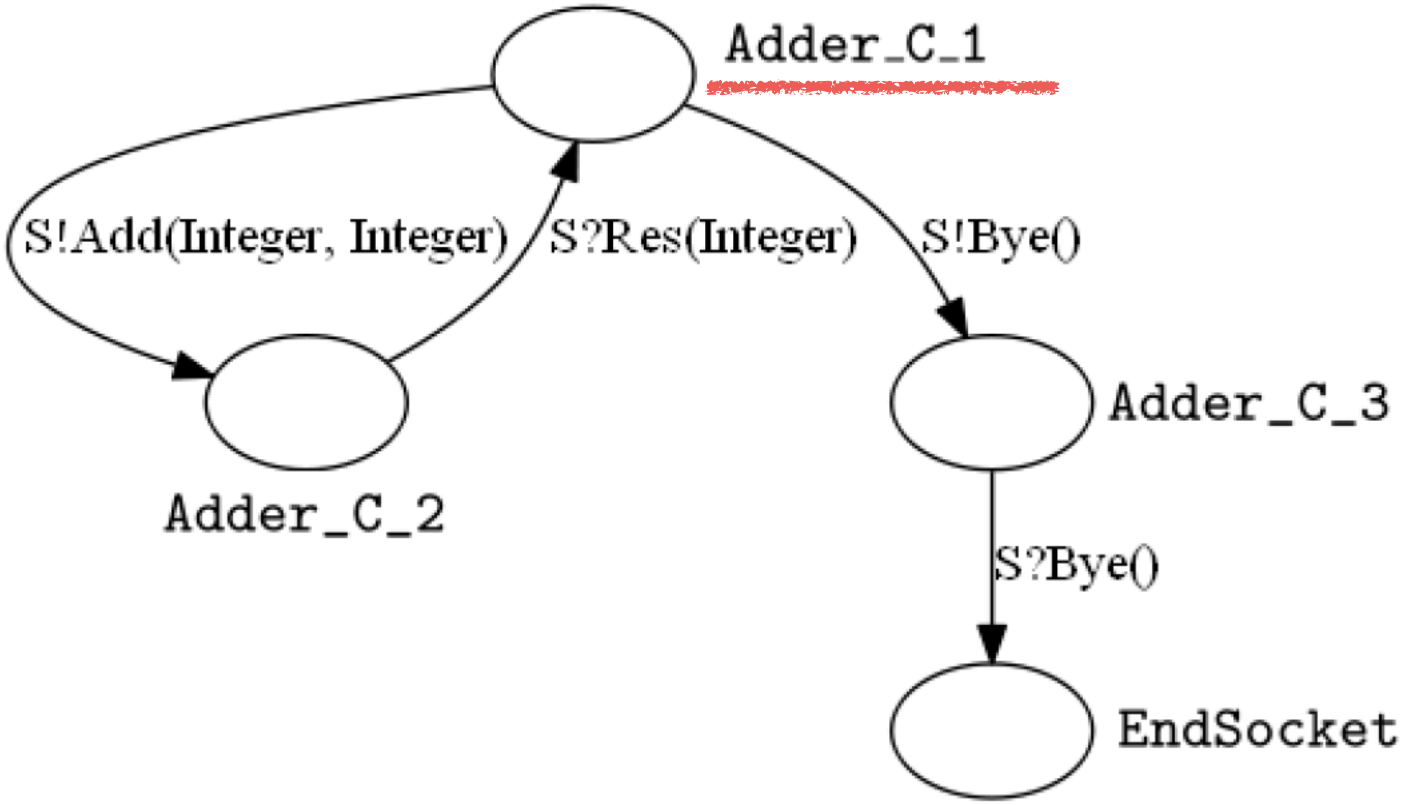
Adder_C_1

```
    Adder_C_2 send(S role, Add op, Integer arg0, Integer arg1) t
    Adder_C_3 send(S role, Bye op) throws ...
```

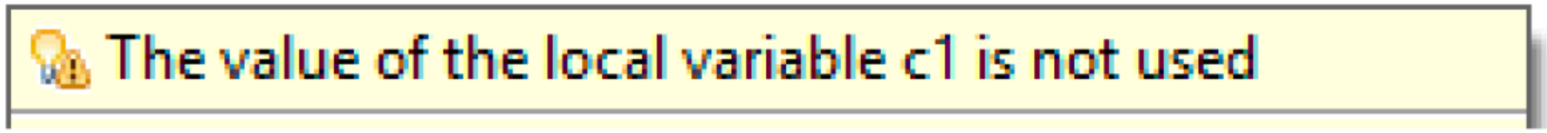


Adder_C_2

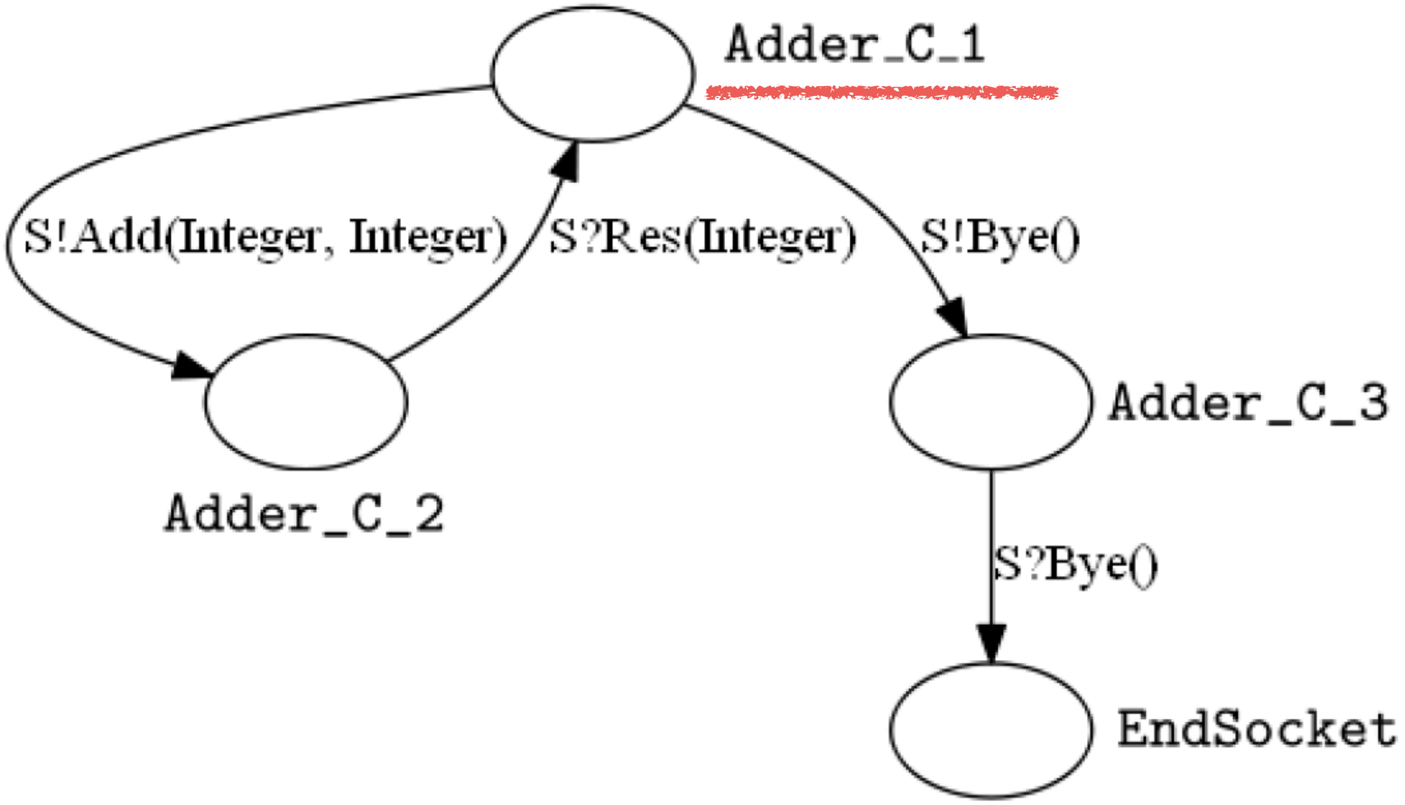
```
Adder_C_1 receive(S role, Res op, Buf<? super Integer> arg1)
```



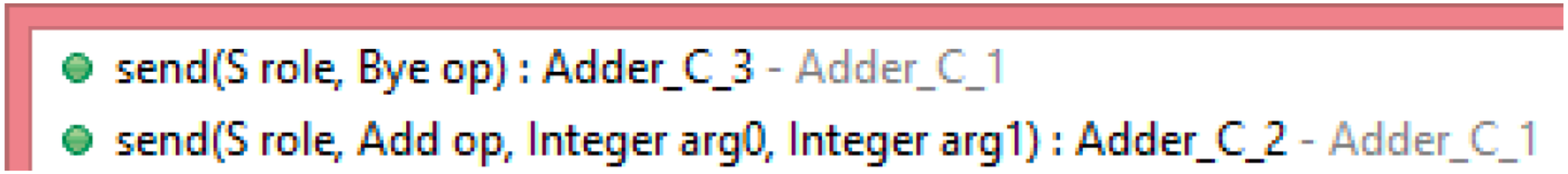
```
Adder_C_1 c1 = new Adder_C_1(...);
```



The value of the local variable c1 is not used

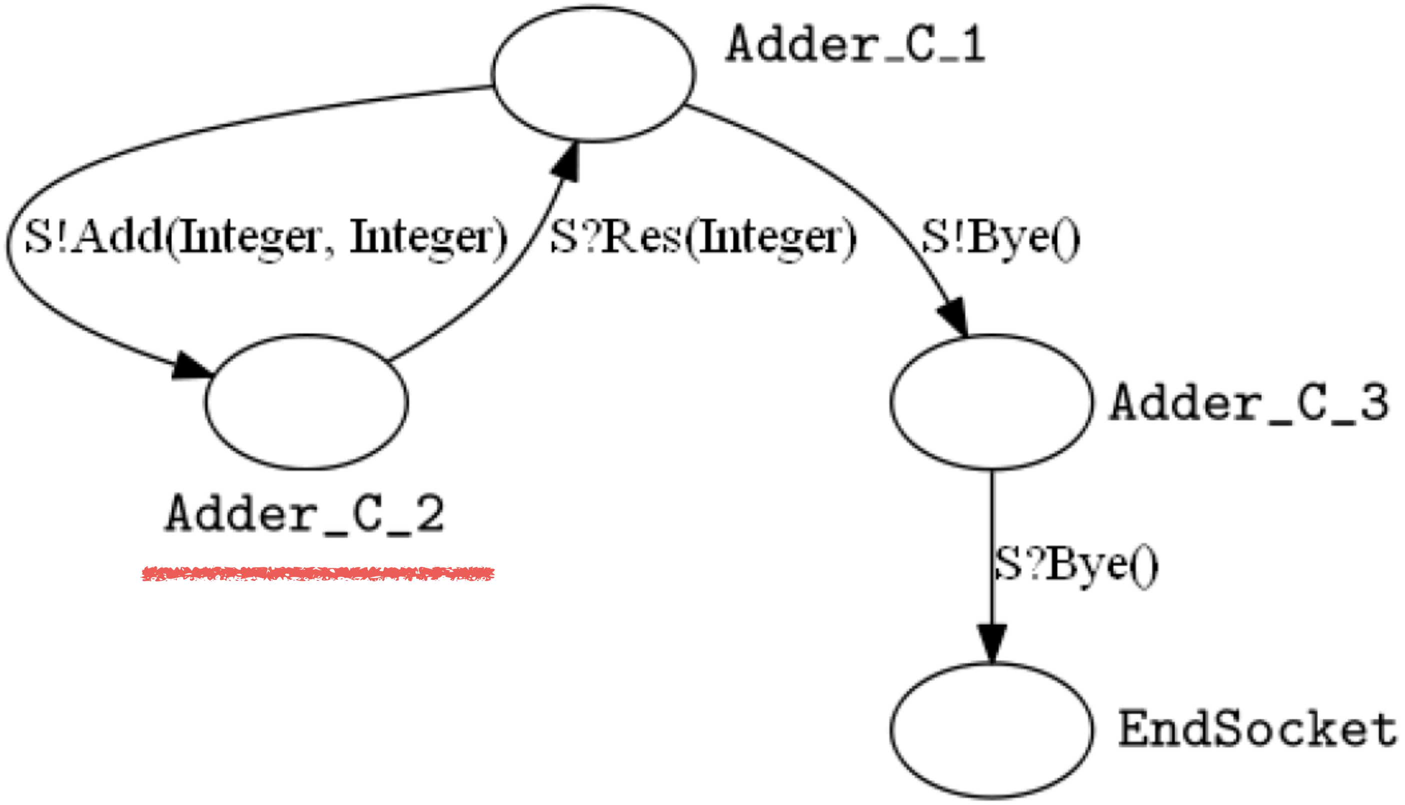


```
Adder_C_1 c1 = new Adder_C_1(...);

c1.
```

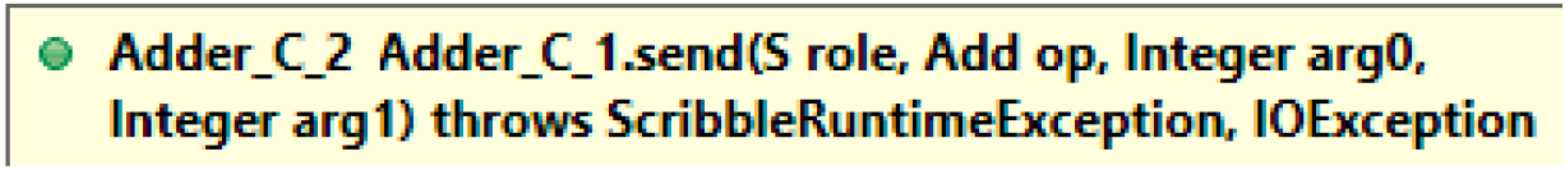


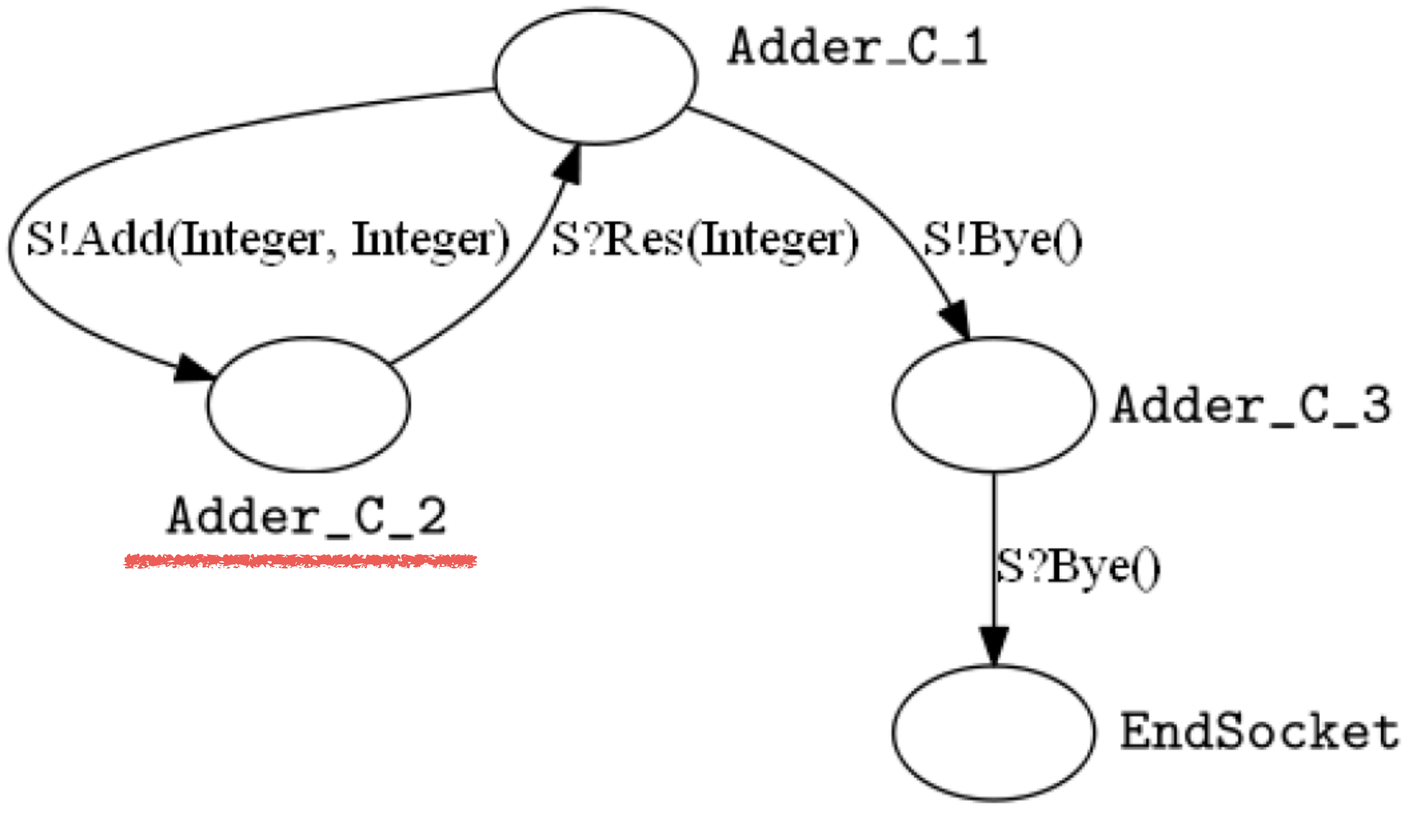
● send(S role, Bye op) : Adder_C_3 - Adder_C_1

● send(S role, Add op, Integer arg0, Integer arg1) : Adder_C_2 - Adder_C_1

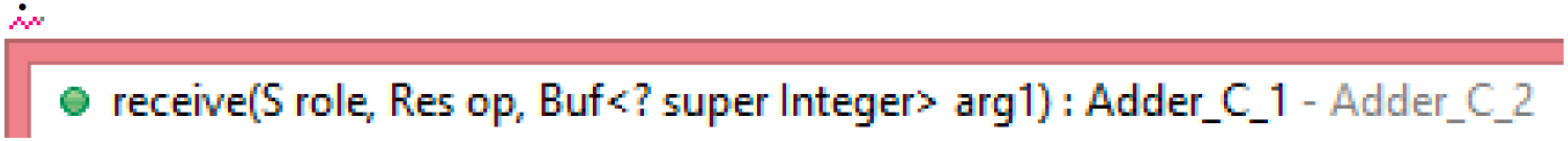


```
Adder_C_1 c1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
c1.send(S, Add, i.val, i.val);
```

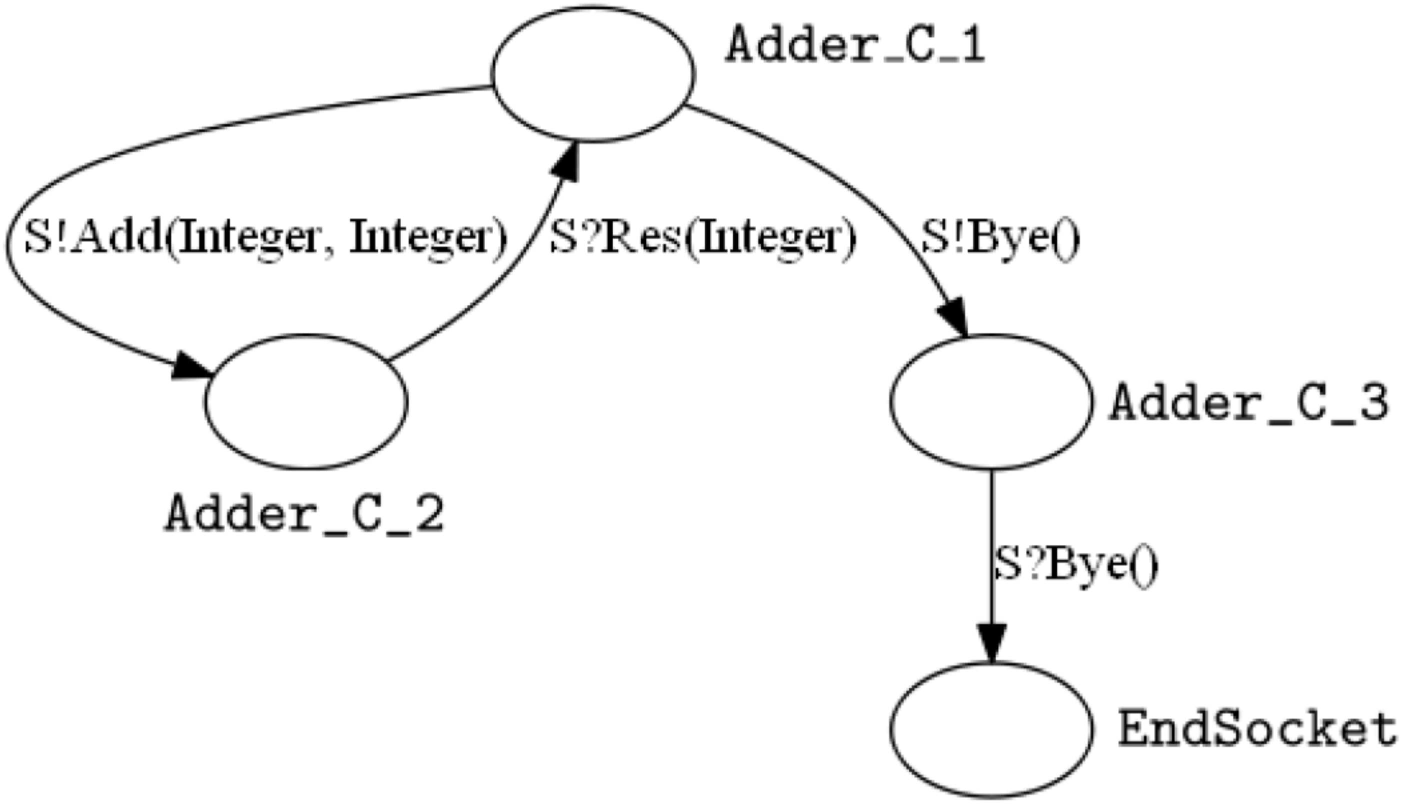

Adder_C_2 Adder_C_1.send(S role, Add op, Integer arg0, Integer arg1) throws ScribbleRuntimeException, IOException



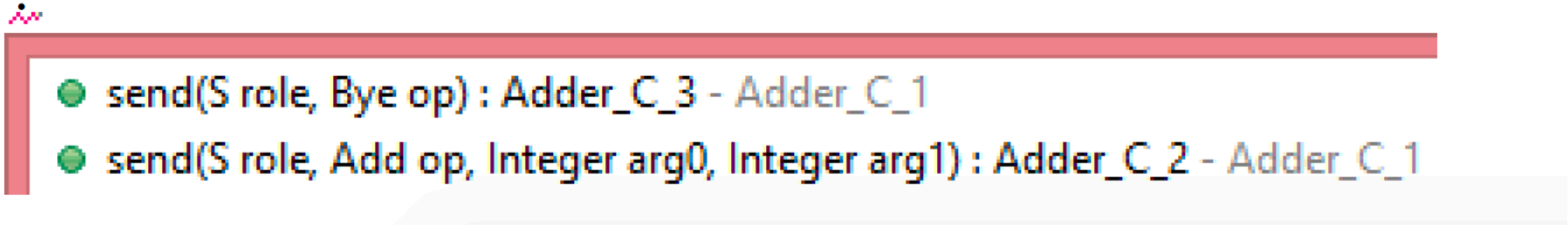
```
Adder_C_1 c1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
c1.send(S, Add, i.val, i.val)
```



● receive(S role, Res op, Buf<? super Integer> arg1) : Adder_C_1 - Adder_C_2

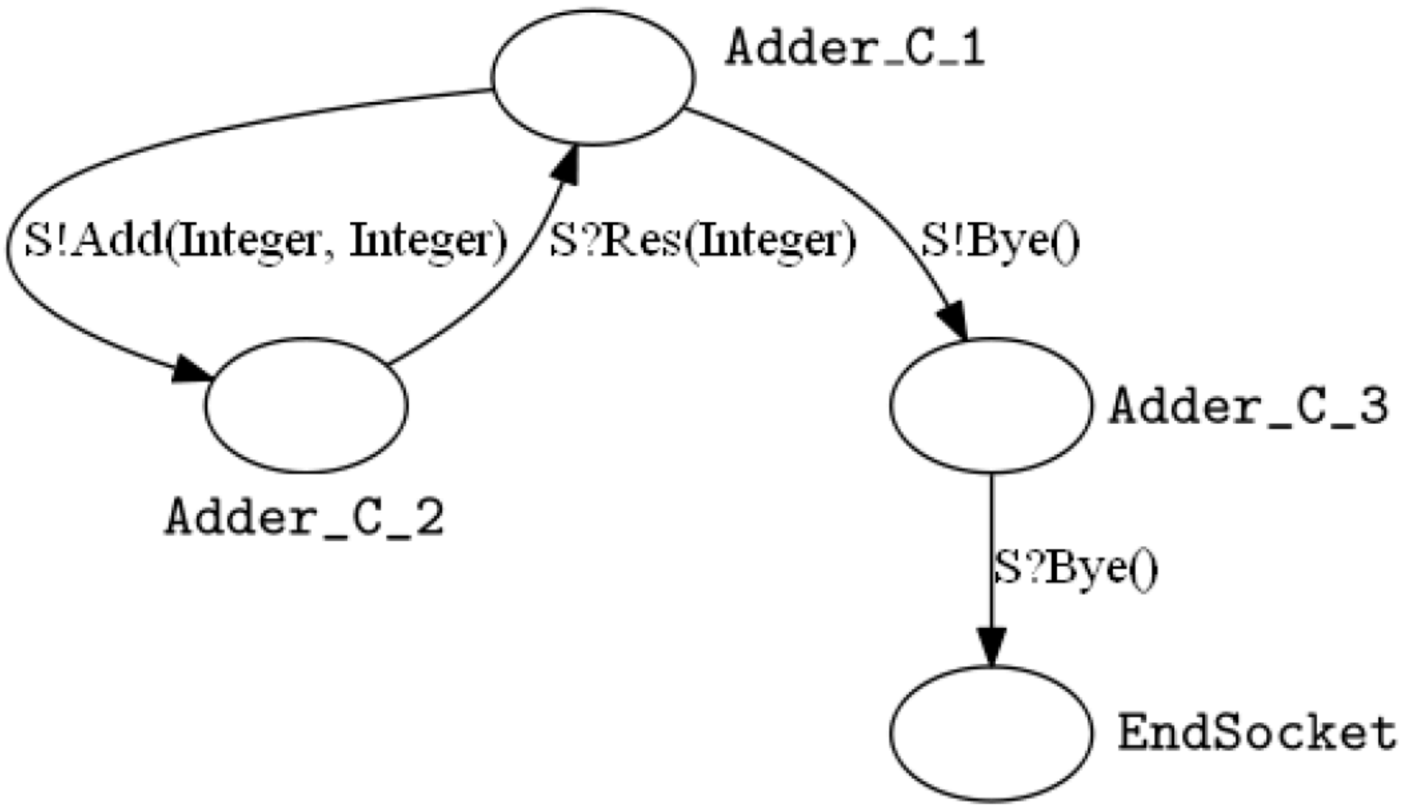


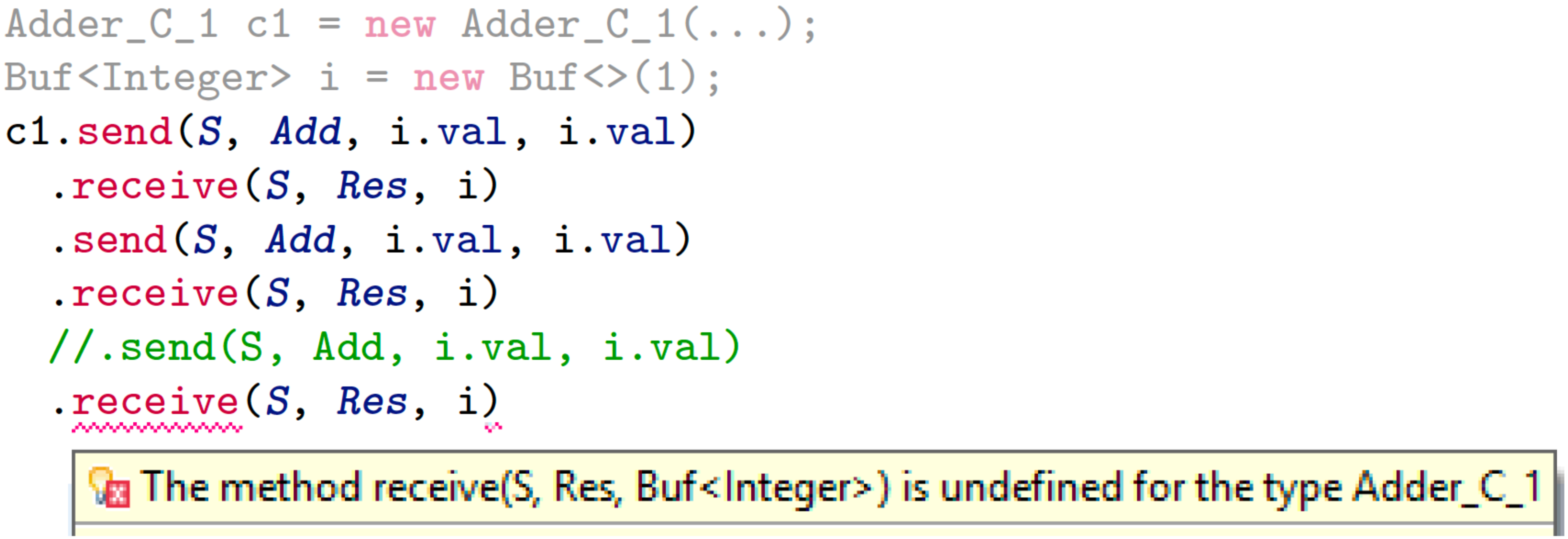
```
Adder_C_1 c1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
c1.send(S, Add, i.val, i.val)
  .receive(S, Res, i)
  .send(S, Add, i.val, i.val)
  .receive(S, Res, i)
  .send(S, Add, i.val, i.val)
  .receive(S, Res, i)
```



● send(S role, Bye op) : Adder_C_3 - Adder_C_1

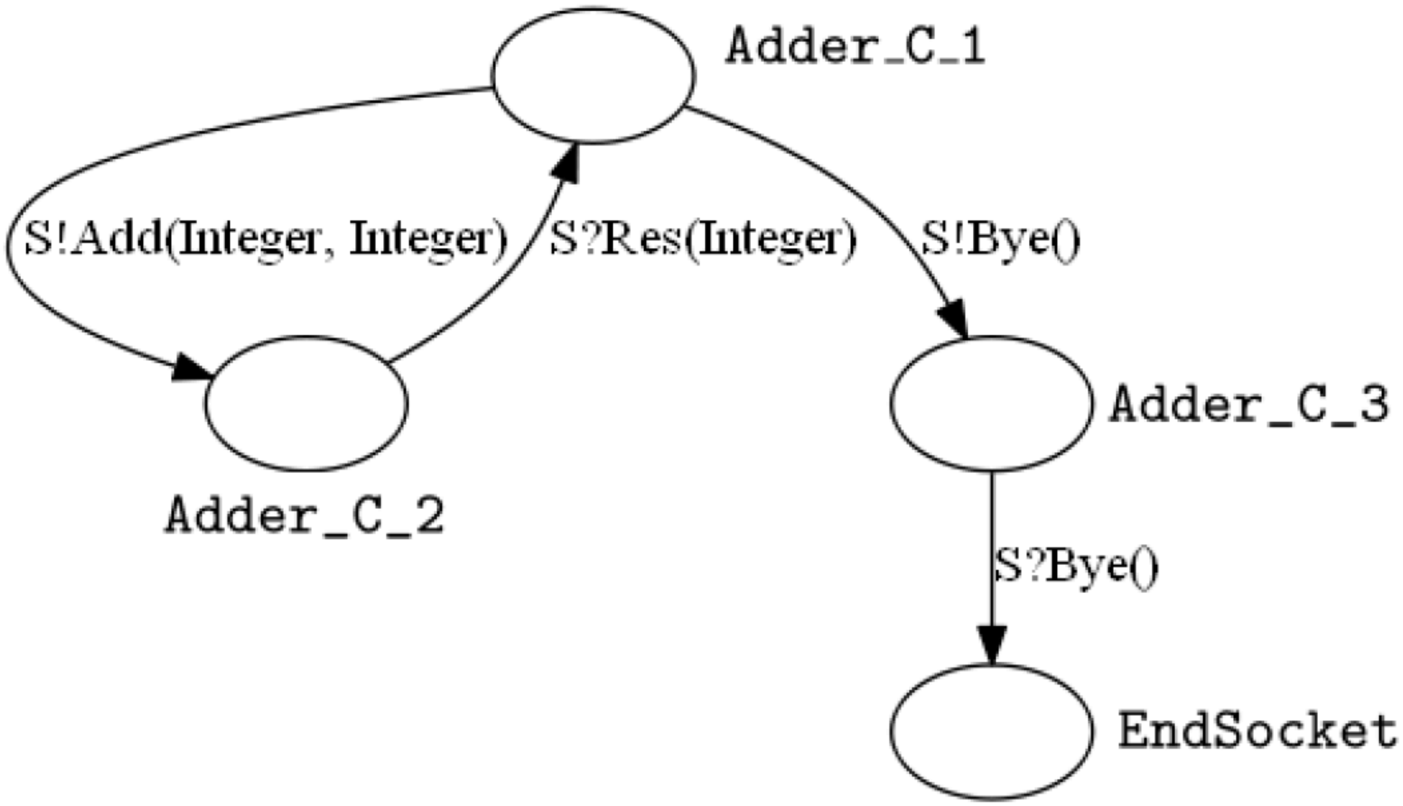
● send(S role, Add op, Integer arg0, Integer arg1) : Adder_C_2 - Adder_C_1



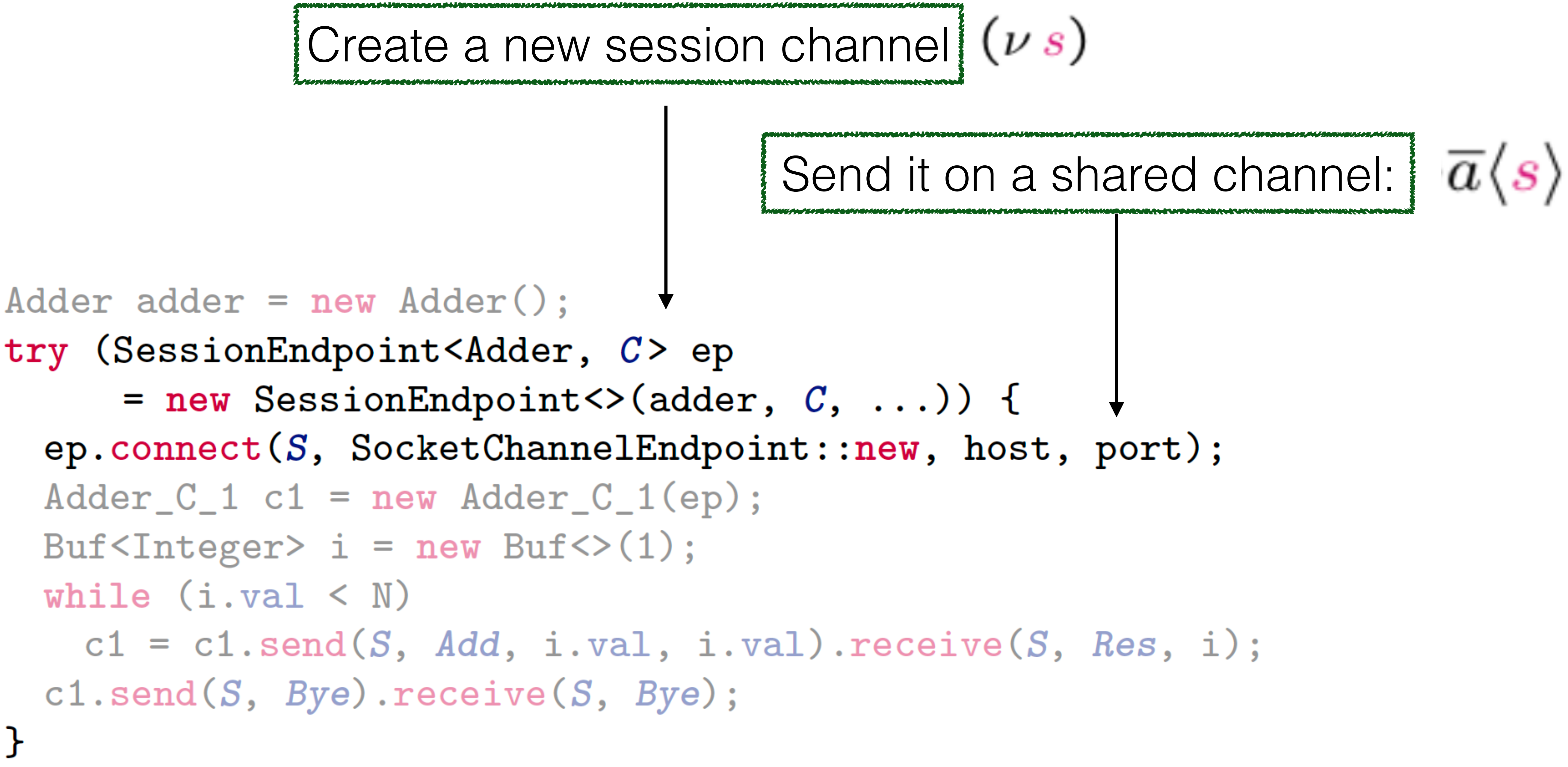


```
Adder_C_1 c1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
c1.send(S, Add, i.val, i.val)
  .receive(S, Res, i)
  .send(S, Add, i.val, i.val)
  .receive(S, Res, i)
  //.send(S, Add, i.val, i.val)
  .receive(S, Res, i)
```

The method receive(S, Res, Buf<Integer>) is undefined for the type Adder_C_1



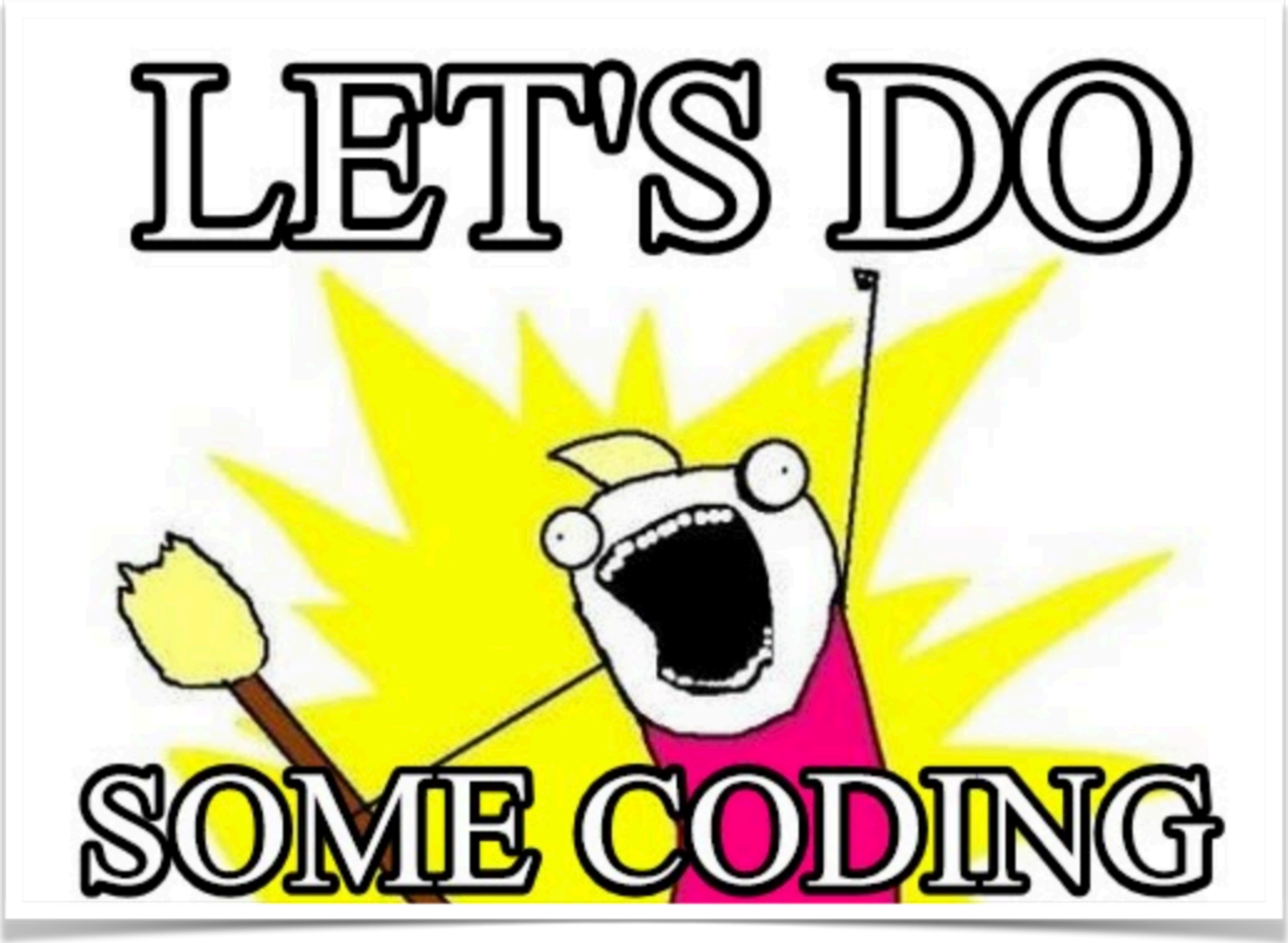
```
Adder_C_1 c1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
while (i.val < N)
  c1 = c1.send(S, Add, i.val, i.val).receive(S, Res, i);
c1.send(S, Bye).receive(S, Bye);
```



Create a new session channel $(\nu\, s)$

Send it on a shared channel: $\overline{a}\langle s\rangle$

```
Adder adder = new Adder();
try (SessionEndpoint<Adder, C> ep
      = new SessionEndpoint<>(adder, C, ...)) {
  ep.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 c1 = new Adder_C_1(ep);
  Buf<Integer> i = new Buf<>(1);
  while (i.val < N)
    c1 = c1.send(S, Add, i.val, i.val).receive(S, Res, i);
  c1.send(S, Bye).receive(S, Bye);
}
```


LET'S DO
SOME CODING