Hybrid session verification through Endpoint API generation

Raymond Hu and Nobuko Yoshida

Imperial College London

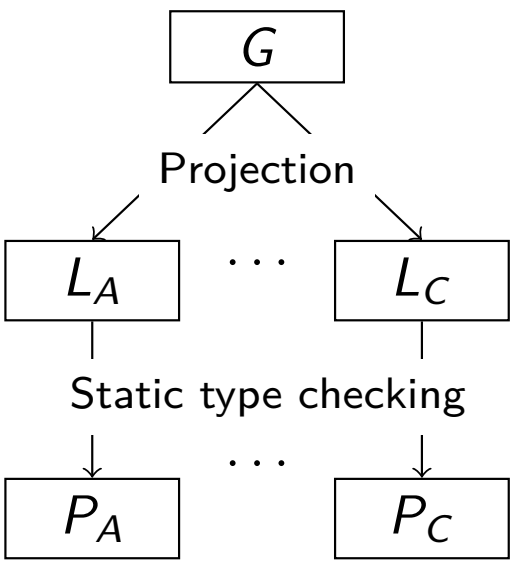
Outline

- Background: multiparty session types (MPST)
  - Implementations and applications of MPST
- Hybrid session verification through Endpoint API generation
  - Practical MPST-based (Scribble) toolchain
  - Simple example: Adder service
  - Real-world example: Simple Mail Transfer Protocol (SMTP)

Multiparty session types (background)

- Programming distributed applications
  - From: protocol spec. (e.g. natural language, sequence diagrams, . . .)
  - To: endpoint programs that faithfully implement their role in the protocol
  - Potential errors:
    - × Communication mismatch: e.g. receiver is sent an unexpected message
    - × Protocol violation: executed interaction does not follow the protocol
    - × Deadlock: e.g. all endpoints blocked on input

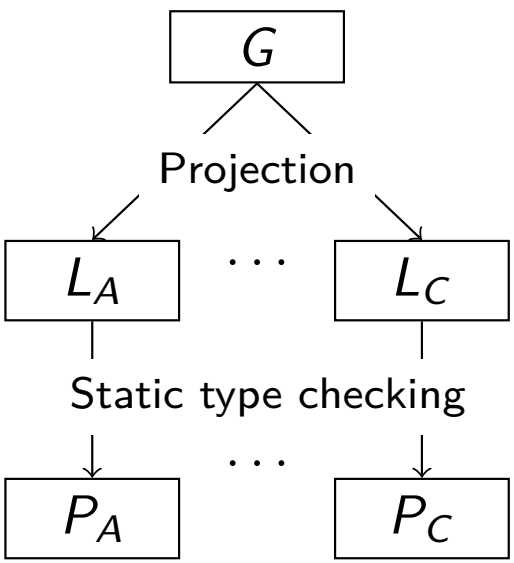
- Types for specification and verification of message passing programs
  - Originally developed as a type theory in the π-calculus [POPL08]

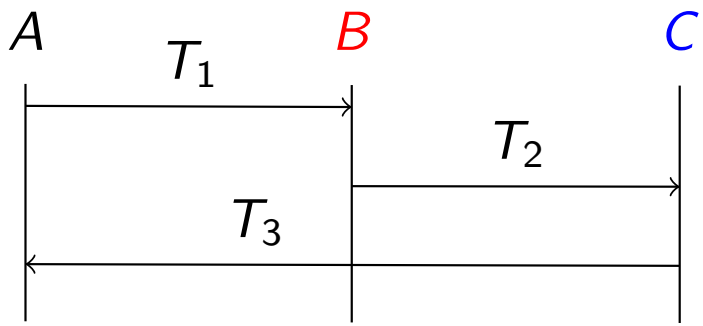


Multiparty session types (background)

- Programming distributed applications
  - From: protocol spec. (e.g. natural language, sequence diagrams, . . .)
  - To: endpoint programs that faithfully implement their role in the protocol
  - Potential errors:
    - × Communication mismatch: e.g. receiver is sent an unexpected message
    - × Protocol violation: executed interaction does not follow the protocol
    - × Deadlock: e.g. all endpoints blocked on input

- Types for specification and verification of message passing programs
  - Originally developed as a type theory in the π-calculus [POPL08]

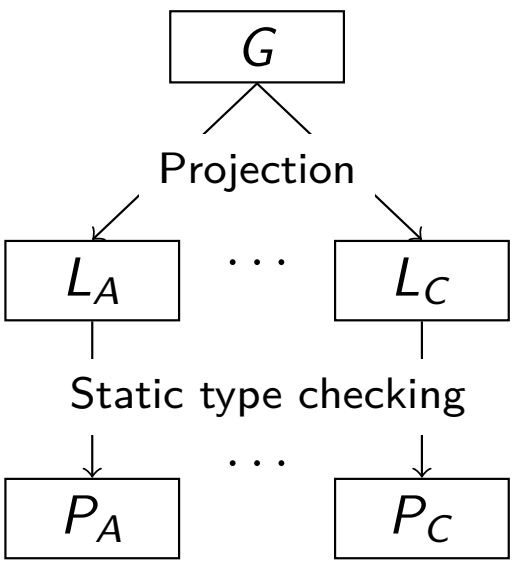


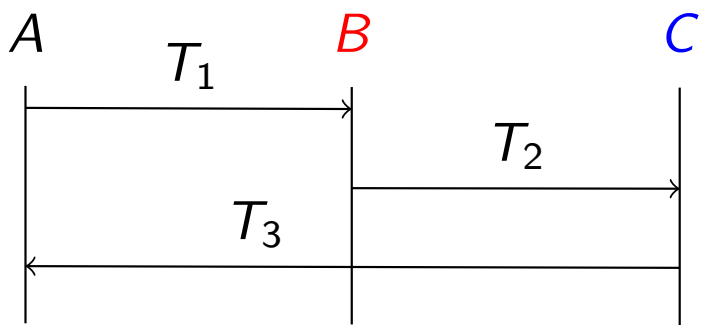


Multiparty session types (background)

- Programming distributed applications
  - From: protocol spec. (e.g. natural language, sequence diagrams, . . .)
  - To: endpoint programs that faithfully implement their role in the protocol
  - Potential errors:
    - × Communication mismatch: e.g. receiver is sent an unexpected message
    - × Protocol violation: executed interaction does not follow the protocol
    - × Deadlock: e.g. all endpoints blocked on input

- Types for specification and verification of message passing programs
  - Originally developed as a type theory in the π-calculus [POPL08]



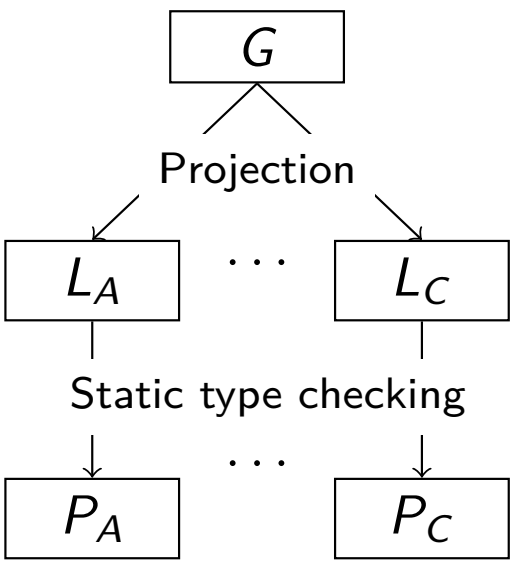


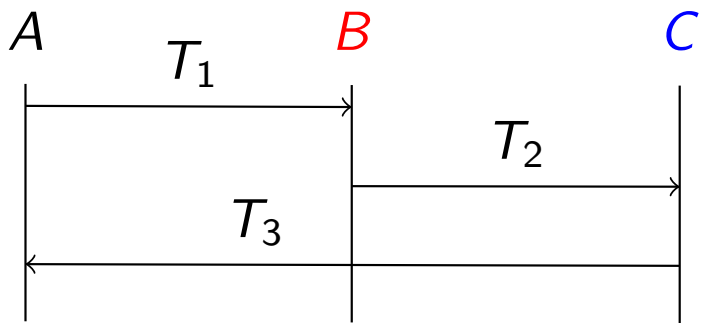
$$G = A \to B : T_1.$$
$$B \to C : T_2.$$
$$C \to A : T_3.\text{end}$$

Multiparty session types (background)

- Programming distributed applications
  - From: protocol spec. (e.g. natural language, sequence diagrams, . . .)
  - To: endpoint programs that faithfully implement their role in the protocol
  - Potential errors:
    - × Communication mismatch: e.g. receiver is sent an unexpected message
    - × Protocol violation: executed interaction does not follow the protocol
    - × Deadlock: e.g. all endpoints blocked on input

- Types for specification and verification of message passing programs
  - Originally developed as a type theory in the π-calculus [POPL08]





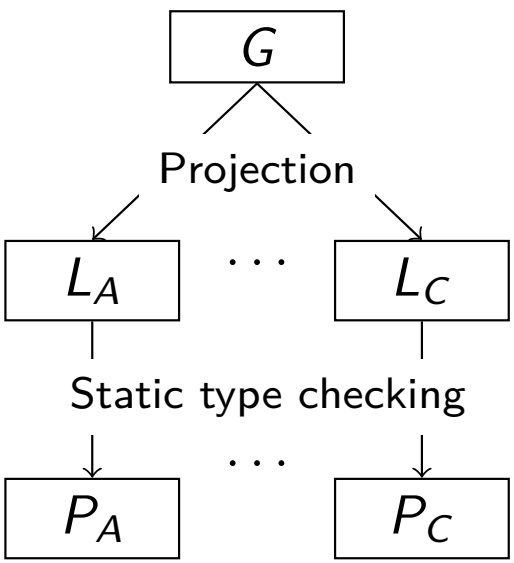
$G = A \to B : T_1.$
$B \to C : T_2.$
$C \to A : T_3.\text{end}$

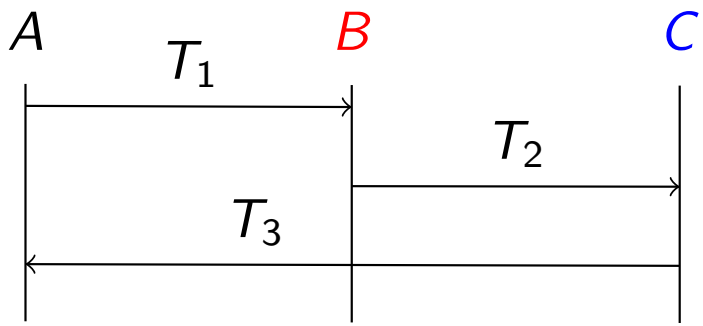
$A :!\langle B, T_1\rangle.?(C, T_3).\text{end}$
$B :?(A, T_1).!\langle C, T_2\rangle.\text{end}$
$C :?(B, T_2).!\langle A, T_3\rangle.\text{end}$

Multiparty session types (background)

- Programming distributed applications
  - From: protocol spec. (e.g. natural language, sequence diagrams, . . .)
  - To: endpoint programs that faithfully implement their role in the protocol
  - Potential errors:
    - × Communication mismatch: e.g. receiver is sent an unexpected message
    - × Protocol violation: executed interaction does not follow the protocol
    - × Deadlock: e.g. all endpoints blocked on input

- Types for specification and verification of message passing programs
  - Originally developed as a type theory in the π-calculus [POPL08]





$$G = A \to B : T_1.\ B \to C : T_2.\ C \to A : T_3.\mathsf{end}$$

$$A :!\langle B, T_1\rangle.?(C, T_3).\mathsf{end}$$
$$B :?(A, T_1).!\langle C, T_2\rangle.\mathsf{end}$$
$$C :?(B, T_2).!\langle A, T_3\rangle.\mathsf{end}$$

$$\bar{a}[A](x).x!\langle B, t_1\rangle.x?(C, u_3).\mathbf{0}$$
$$a[B](y).y?(A, u_1).y!\langle C, t_2\rangle.\mathbf{0}$$
$$a[C](z).z?(B, u_2).z!\langle A, t_3\rangle.\mathbf{0}$$

Multiparty session types (background)

- Programming distributed applications
  - From: protocol spec. (e.g. natural language, sequence diagrams, . . .)
  - To: endpoint programs that faithfully implement their role in the protocol
  - Potential errors:
    - × Communication mismatch: e.g. receiver is sent an unexpected message
    - × Protocol violation: executed interaction does not follow the protocol
    - × Deadlock: e.g. all endpoints blocked on input

- Types for specification and verification of message passing programs
  - Originally developed as a type theory in the π-calculus [POPL08]
  - Static safety properties [MSCS15]
    - ✓ Communication safety
    - ✓ Protocol fidelity
    - ✓ Deadlock-freedom (or progress)

[SFM15MP] *A Gentle Introduction to Multiparty Asynchronous Session Types*. Coppo, Dezani-Ciancaglini, Luca Padovani and Yoshida.

[POPL08] *Multiparty asynchronous session types*. Honda, Yoshida and Carbone.

[MSCS15] *Global Progress for Dynamically Interleaved Multiparty Sessions*. Coppo, Dezani-Ciancaglini, Yoshida and Padovani.

Implementing and applying session types (related work)

- Static session typing
  - Extending existing mainstream languages, e.g.
    - SJ (binary ST in Java) [ECOOP08]
    - STING (MPST in Java) [SCP13]
  - Need language support for tractability
    - First-class channel I/O primitives (e.g. session initiation, choice, etc)
    - Linearity/aliasing control of channel endpoints

[ECOOP08] *Session-Based Distributed Programming in Java*. Hu, Yoshida and Honda.
[SCP13] *Efficient sessions*. Sivaramakrishnan, Ziarek, Nagaraj and Eugster.

Implementing and applying session types (related work)

- Static session typing
  - Embedding into existing languages, e.g. Haskell
    - Neubauer and Thiemann [PADL04] (no session interleaving)
    - `simple-sessions` [HASKELL08] ("manual" typing environment management)
    - `effect-sessions` [POPL16] (synchronous)
  - Varying tradeoffs involving expressiveness and usability

[PADL04] *An Implementation of Session Types*. Neubauer and Thiemann.
[HASKELL08] *Haskell session types with (almost) no class*. Pucella and Tov.
[POPL16] *Effects as sessions, sessions as effects*. Orchard and Yoshida.

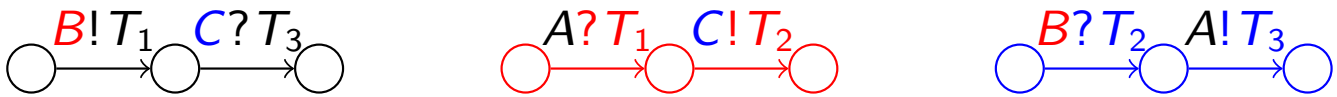
  - New languages, e.g.
    - SILL (sessions in linear logic) [FoSSaCS13]

[FoSSaCS13] *Polarized Substructural Session Types*. Pfenning and Griffith.

Implementing and applying session types (related work)

- Run-time session monitoring
  - Generate protocol-specific endpoint I/O monitors from source protocol

$$A \to B : T_1.B \to C : T_2.C \to A : T_3.\text{end}$$



  - Direct application of ST to existing (and non-statically typed) languages

[RV13] *Practical interruptible conversations*. Hu, Neykova, Yoshida, Demangeon and Honda.
[FMOODS13] *Monitoring networks through multiparty session types*. Bocchi, Chen, Demangeon, Honda and Yoshida.
[ESOP12] *Multiparty session types meet communicating automata*. Deniélou and Yoshida.

- Code/assertion generation from session types
  - For a specific target context: generate I/O stubs/skeletons, etc.
  - e.g. MPI/C [CC15]: weaves user computation with interaction skeleton

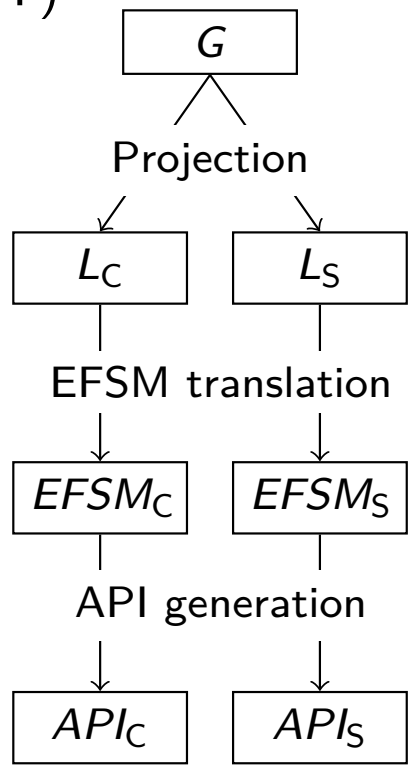
[CC15] *Safe MPI code generation based on session types*. Ng, Coutinho and Yoshida.
[OOPSLA15] *Protocol-based verification of message-passing parallel programs*. López, Marques, Martins, Ng, Santos, Vasconcelos and Yoshida.

Hybrid session verification through Endpoint API generation

- Application of session types to practice:
  - Hybrid (combined static and run-time) session verification
    - Directly for mainstream (statically typed) languages
    - Leverage existing static typing support
  - Endpoint API generation
    - Promote integration with existing language features, libraries and tools
    - Protocol specification: Scribble (asynchronous MPST)
    - Endpoint APIs: Java
- Result: rigorously generated APIs for implementing distributed protocols
  - Cf. ad hoc endpoint implementation from informal specifications

Scribble toolchain

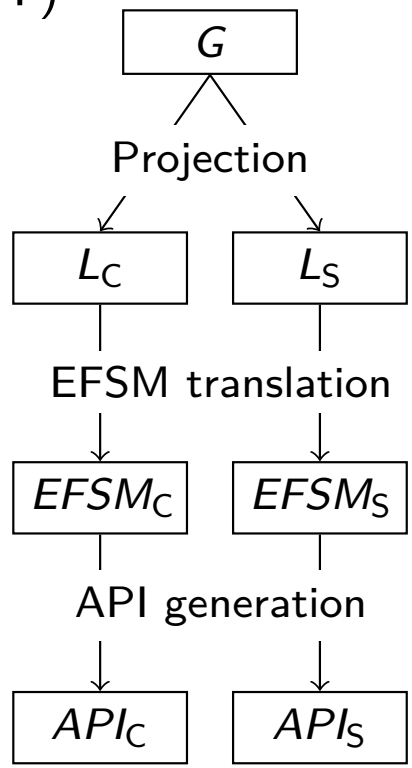
- Protocol spec. as Scribble global protocol (async. MPST)
  - Global protocol validation
    (safely distributable asynchronous protocol)



- Java APIs for implementing the endpoints

Scribble toolchain

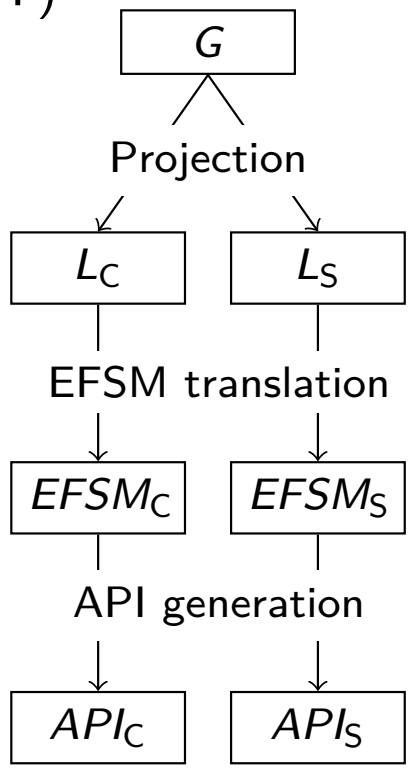
- Protocol spec. as Scribble global protocol (async. MPST)
  - Global protocol validation
    (safely distributable asynchronous protocol)
  - Syntactic projection to local protocols
    (static session typing if supported)



- Java APIs for implementing the endpoints

Scribble toolchain

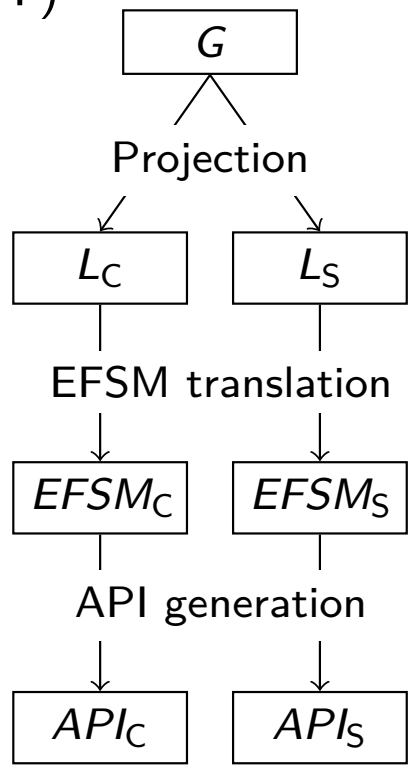
- Protocol spec. as Scribble global protocol (async. MPST)
  - Global protocol validation
    (safely distributable asynchronous protocol)
  - Syntactic projection to local protocols
    (static session typing if supported)
  - Endpoint FSM (EFSM) translation
    (dynamic session typing by monitors)



- Java APIs for implementing the endpoints

Scribble toolchain

- Protocol spec. as Scribble global protocol (async. 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

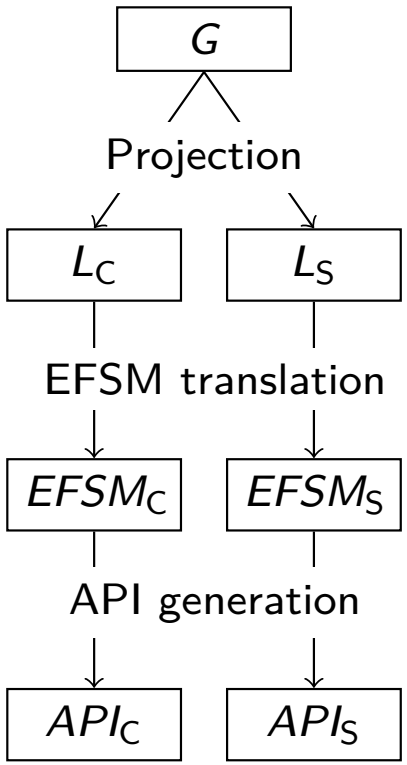


Example: Adder

- Network service for adding two integers

```
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;
  }
}
```

- Scribble global protocol (asynchronous MPST)
  - Role-to-role message passing
  - Located choice

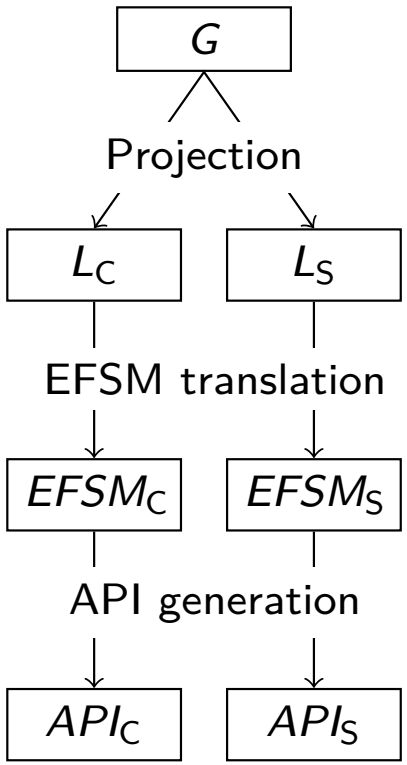


Example: Adder

- Network service for adding two integers

```
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;
  }
}
```

- Scribble global protocol (asynchronous MPST)
  - Role-to-role message passing
  - Located choice

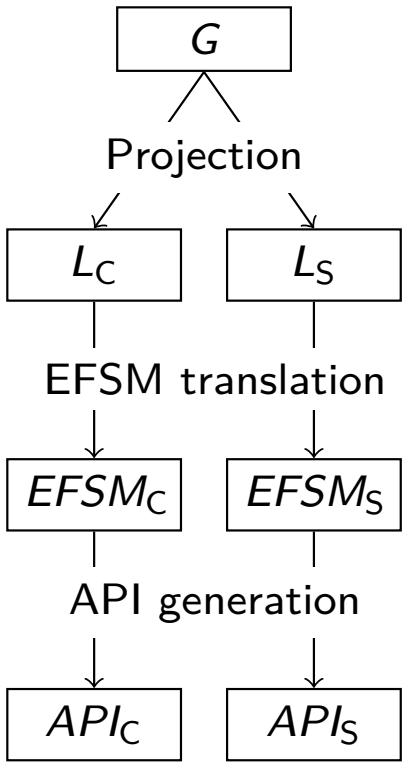


Example: Adder

- Network service for adding two integers

```
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;
  }
}
```

- Scribble global protocol (asynchronous MPST)
  - Role-to-role message passing
  - Located choice

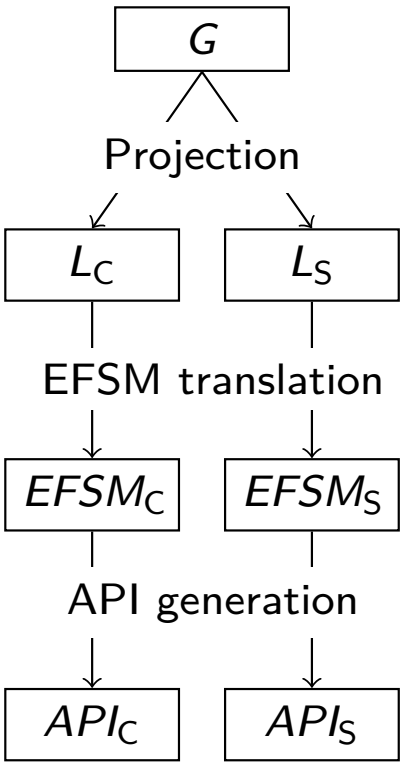


Example: Adder

- Network service for adding two integers

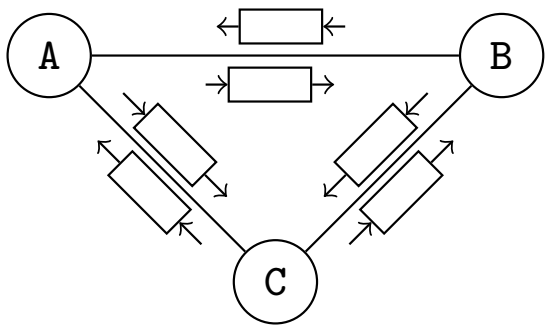
```
global protocol Adder(role C, role S) {
  choice at C {
    Add(Integer, Integer) from C to S;
    Res(Integer) from S to C;
    do Adder(C, S);
  } or {
    Bye() from C to S;
    Bye() from S to C;
  }
}
```

- Scribble global protocol (asynchronous MPST)
  - Role-to-role message passing
  - Located choice



Scribble protocol description language (background)

- Adapts and extends formal MPST for explicit specification and engineering of multiparty message passing protocols
  - Syntax based on [MSCS15]
  - Communication model: asynchronous, reliable, role-to-role ordering




```
1() from A to B;
2() from A to C;
3() from C to B;
```

  - Protocol = message types + interaction structure
    - Fully explicit: no implicit messages needed to conduct a session
  - Collaboration between researchers (Imperial College London) and industry (Red Hat) developers

[TGC13] *The Scribble Protocol Language*. Yoshida, Hu, Neykova and Ng.
[COB12] *Structuring communication with session types*. Honda et al.
[Scribble] Scribble GitHub repo: `https://github.com/scribble`

Global protocol validation (interlude)

- Ensure source global protocol is valid for endpoint projection
  - i.e. protocol can be safely realised via asynchronous message passing between independent endpoints

- Ambiguous choice

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

- Race condition of choice

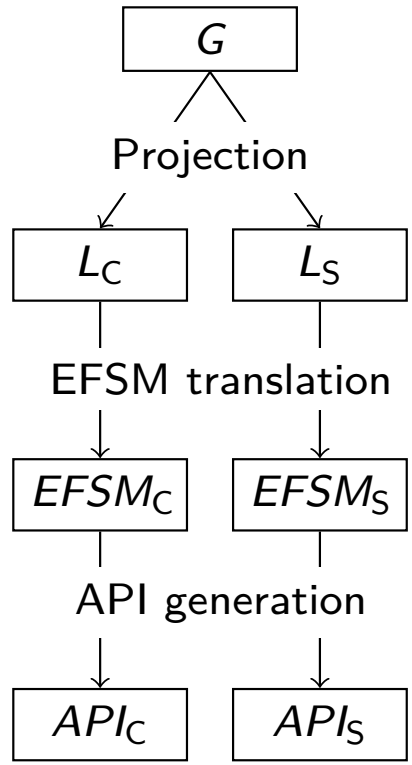
```
choice at A {
  1() from A to B;
  2() from A to C;
  3() from B to C;
  4() from C to B;
} or {
  5() from A to B;
  3() from B to C;
  6() from C to B;
}
```

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;
  }
}
```

- Syntactic projection to local protocol (for C)

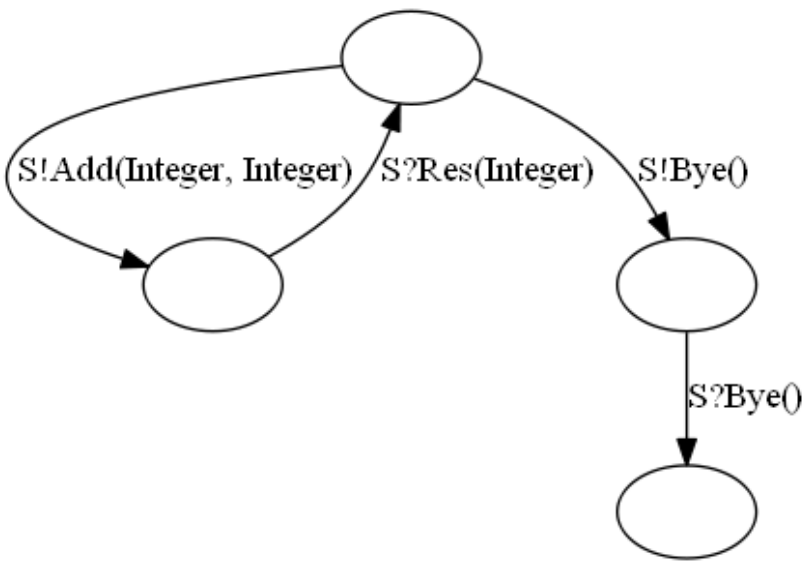
```
local protocol Adder_C(self C, role S) {
  choice at C {
    Add(Integer, Integer) to S;
    Res(Integer) from S;
    do Adder_C(C, S);
  } or {
    Bye() from C to S;
    Bye() from S to C;
  }
}
```

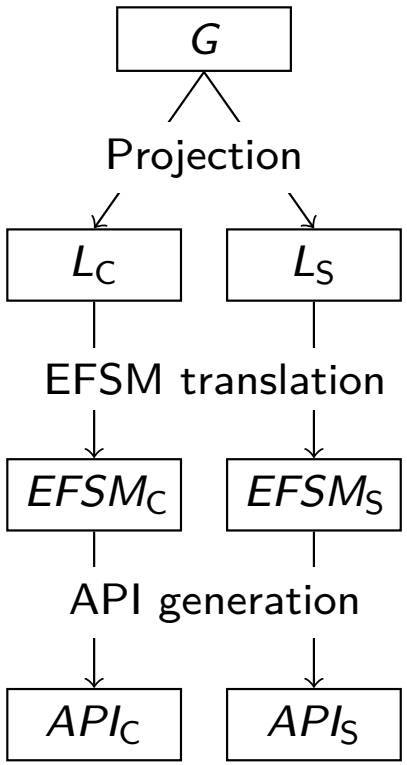


Example: Adder

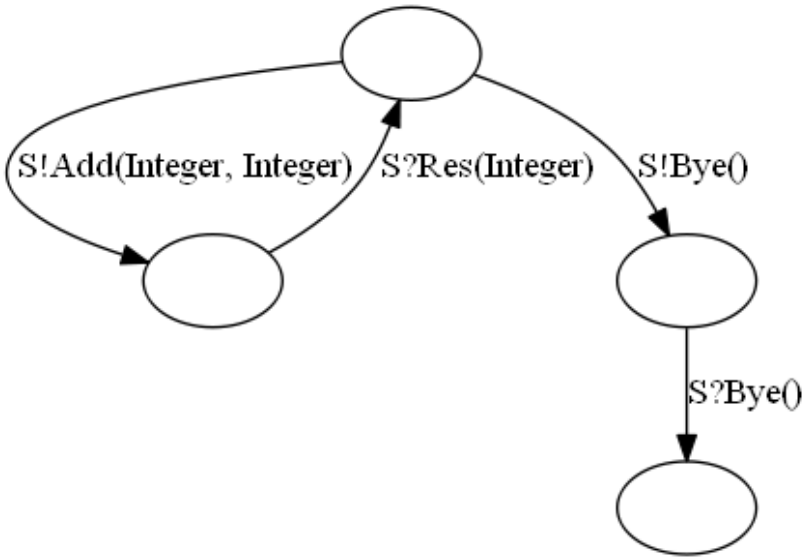
```
local protocol Adder_C(self C, role S) {
  choice at C {
    Add(Integer, Integer) to S;
    Res(Integer) from S;
    do Adder_C(C, S);
  } or {
    Bye() from C to S;
    Bye() from S to C;
  }
}
```

- Endpoint FSM for `C`

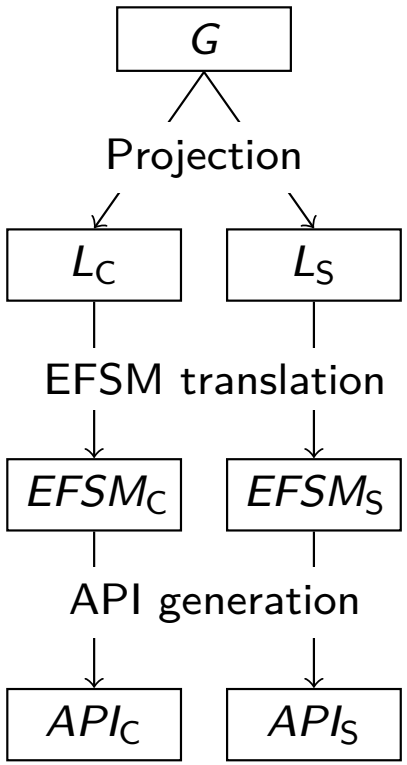




Example: Adder



- Endpoint API generation
  - Session API:
    Reify session type names as singleton types



Adder: Session API

- Reify session type names as Java types (eager singleton pattern)

```
public final class C extends Role {
  public static final C C = new C();
  ...
  private C() {
    super("C");
  }
```

- Main "Session" class

```
public final class Adder extends Session {
  public static final C C = C.C;
  public static final S S = S.S;
  public static final Add Add = Add.Add;
  public static final Bye Bye = Bye.Bye;
  public static final Res Res = Res.Res;
  ...
```

- Instances represent sessions of this type in execution
  - Encapsulates source protocol info, run-time session ID, etc.

Adder: Session API

Class Adder

java.lang.Object
    org.scribble.net.session.Session
        demo.fase.adder.Adder.Adder.Adder

```
public final class Adder
extends org.scribble.net.session.Session
```

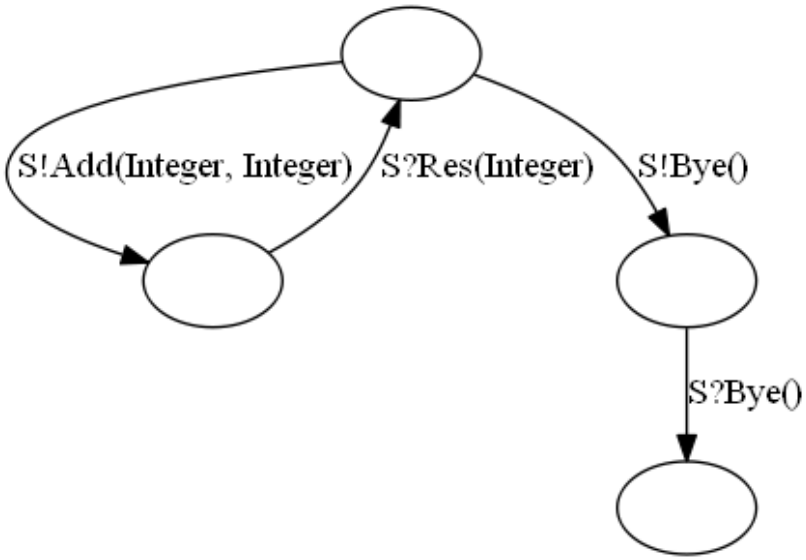
Field Summary

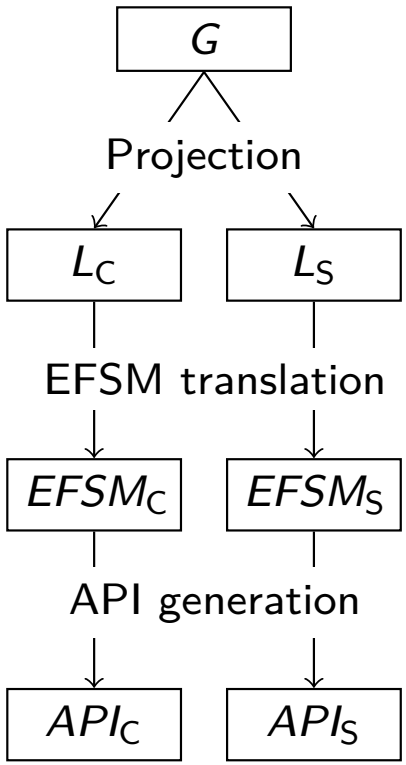
Fields

Modifier and Type	Field and Description
static Add	Add
static Bye	Bye
static C	C

⋮

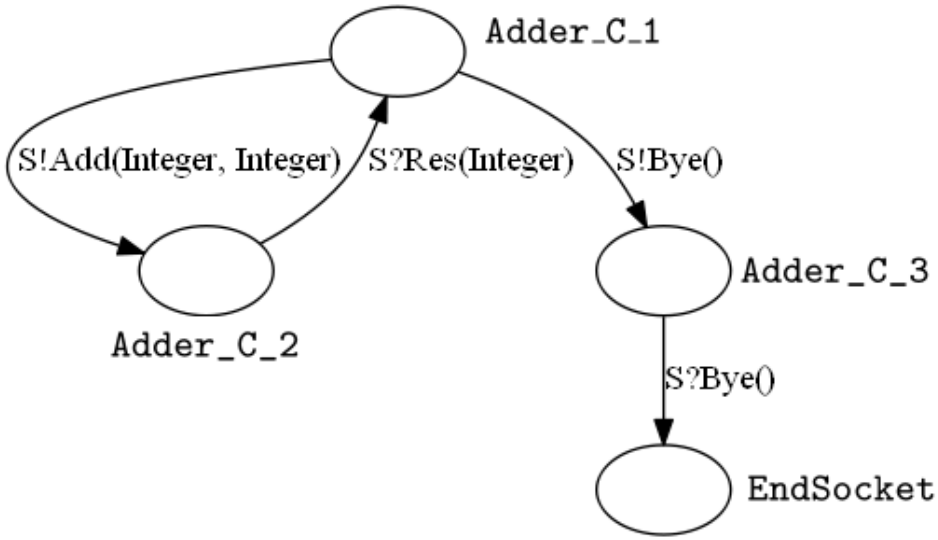
Example: Adder





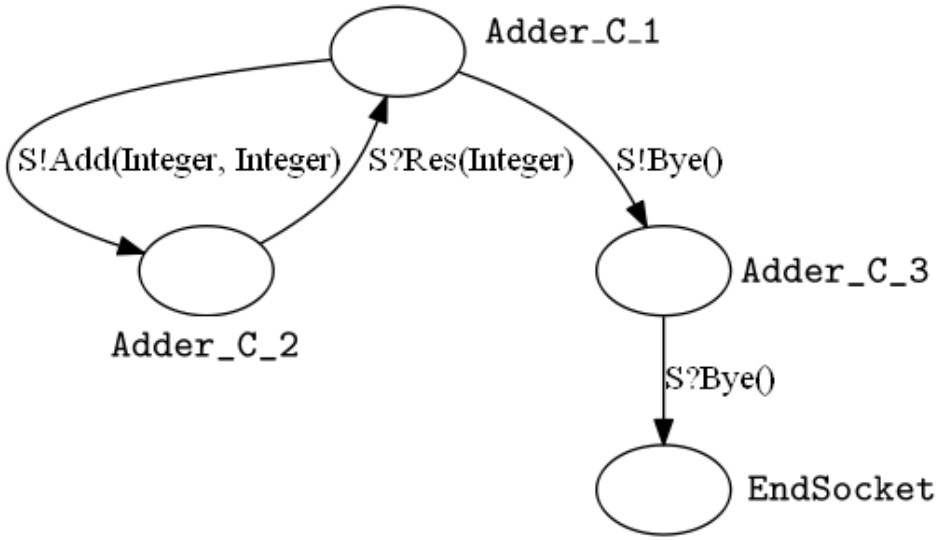
- Endpoint API generation
  - Session API:
    Reify session type names as singleton types
  - State Channel API:
    EFSM represents the endpoint "I/O behaviour"
    - Capture this I/O structure in the type system of the target language

State Channel API



- Protocol states as state-specific channel types
  - Java nominal types: state enumeration as default channel naming scheme
    - Three state/channel kinds: output, unary input, non-unary input
  - Generated *state channel* class offers exactly the valid I/O operations for the corresponding protocol state
  - Fluent interface for chaining channel operations through successive states
    - Only the initial state channel class offers a public constructor

Adder: State Channel API for C



- `Adder_C_1` (output state)
  - Output state has `send` methods

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

    - Parameter types: message recipient, operator and payload
    - Return type: successor state (state channel chaining)
    - Ouput choices via method overloading (session I/O operations directed by the generated utility types)

Adder: State Channel API for C

Class Adder_C_1

java.lang.Object
    org.scribble.net.scribsock.ScribSocket<S,R>
        org.scribble.net.scribsock.LinearSocket<S,R>
            org.scribble.net.scribsock.SendSocket<Adder,C>
                demo.fase.adder.Adder.Adder.channels.C.Adder_C_1

All Implemented Interfaces:

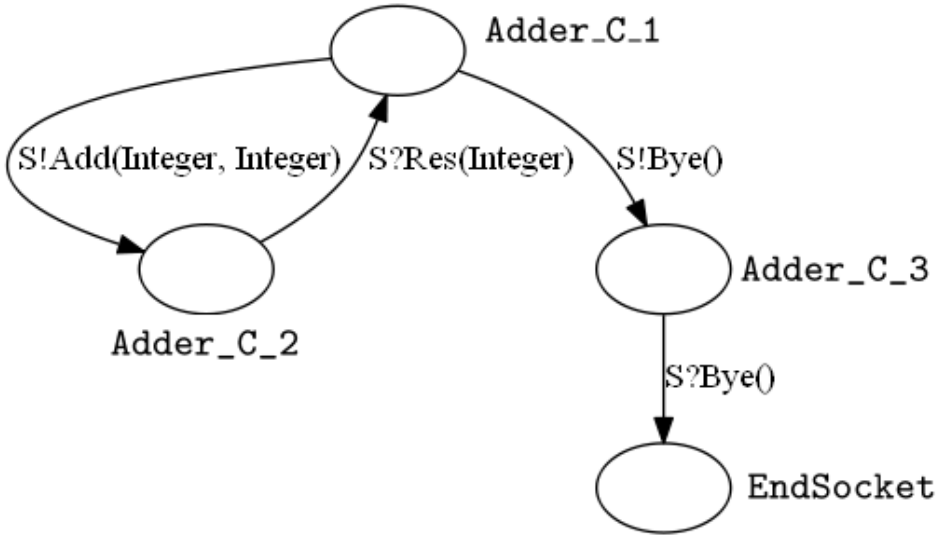
`Out_S_Add_Integer_Integer<Adder_C_2>, Out_S_Bye<Adder_C_3>, Select_C_S_Add_Integer_Integer<Adder_C_2>, Select_C_S_Add_Integer_Integer__S_Bye<Adder_C_2,Adder_C_3>, Select_C_S_Bye<Adder_C_3>, Succ_In_S_Res_Integer`

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`Adder_C_2`	`send(S role, Add op, java.lang.Integer arg0, java.lang.Integer arg1)`
`Adder_C_3`	`send(S role, Bye op)`

Adder: State Channel API for C



- `Adder_C_2` (unary input state)

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

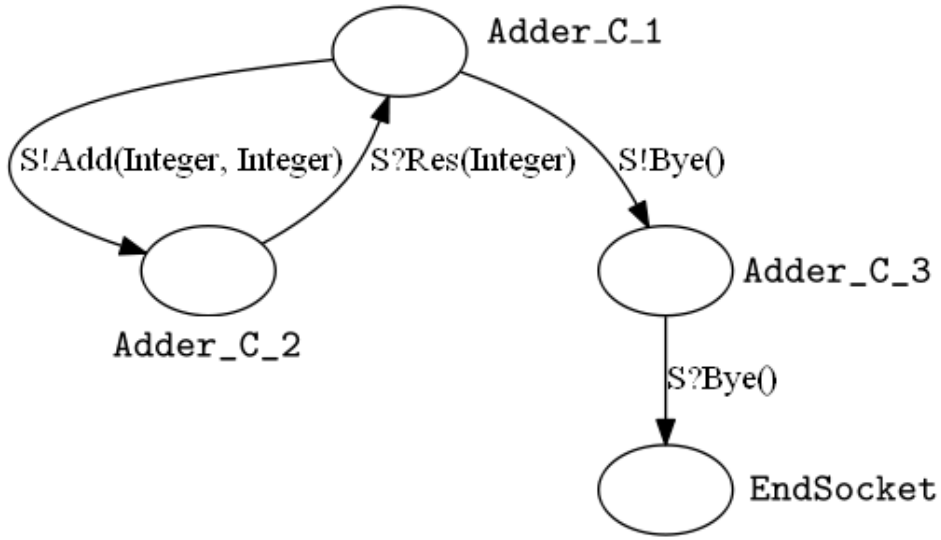
  - Unary input state has a `receive` method
  - Received payloads written to a typed buffer argument
  - (Tail) recursion: return a new instance of a "previous" state channel

- `Adder_C_3` (unary input state)

```
EndSocket receive(S role, Bye op) throws ...
```

  - `EndSocket` for terminal state

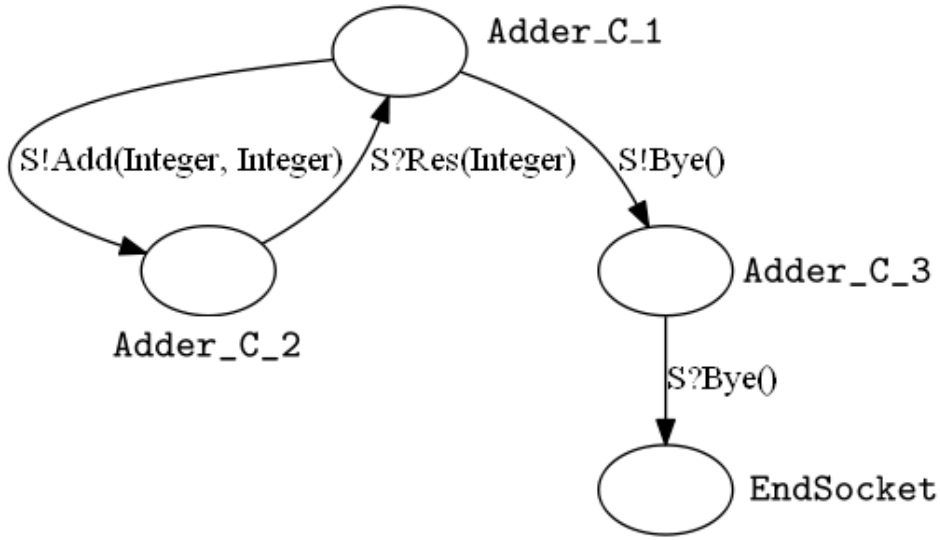
Adder: endpoint implementation for C




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

The value of the local variable c1 is not used

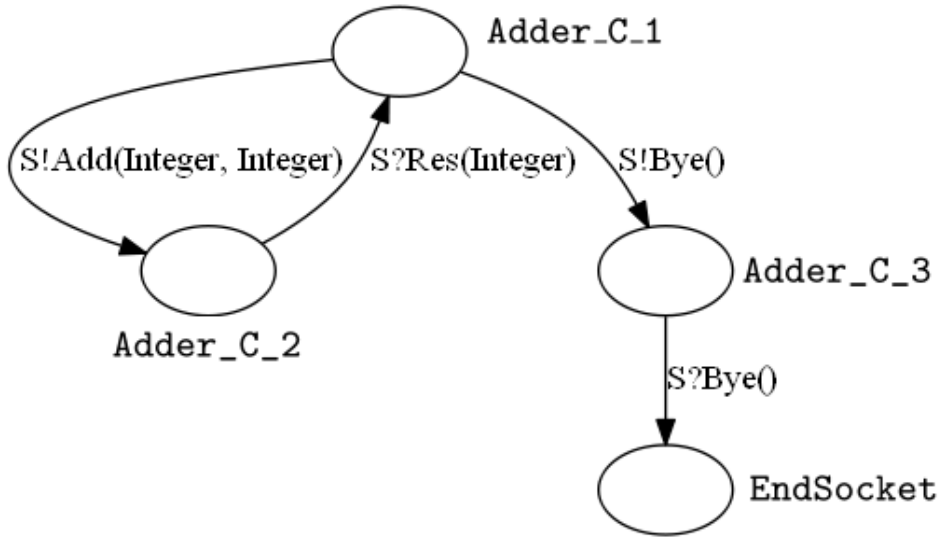
Adder: endpoint implementation for C




```
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

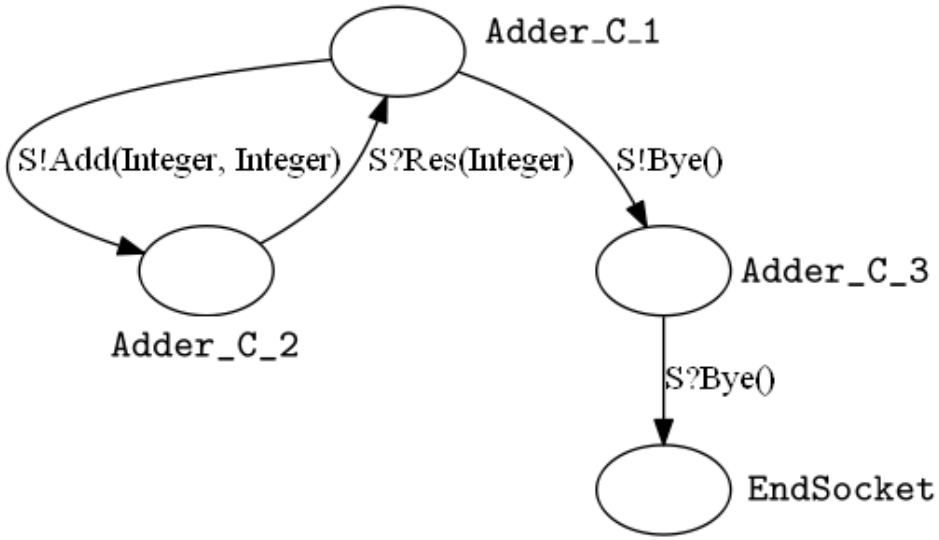
Adder: endpoint implementation for C



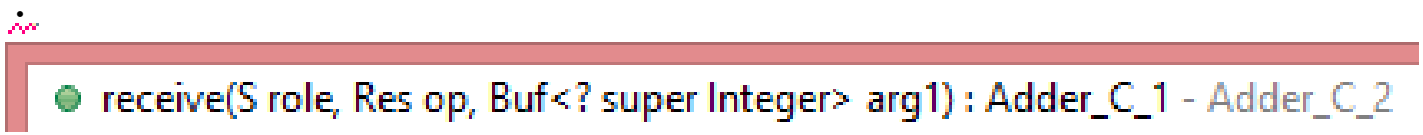

```
Adder_C_1 c1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
c1.send(S, Add, i.val, i.val);
```

Adder_C_2 demo.fase.adder.Adder.Adder.channels.C.Adder_C_1.send(S role, Add op, Integer arg0, Integer arg1) throws ScribbleRuntimeException, IOException

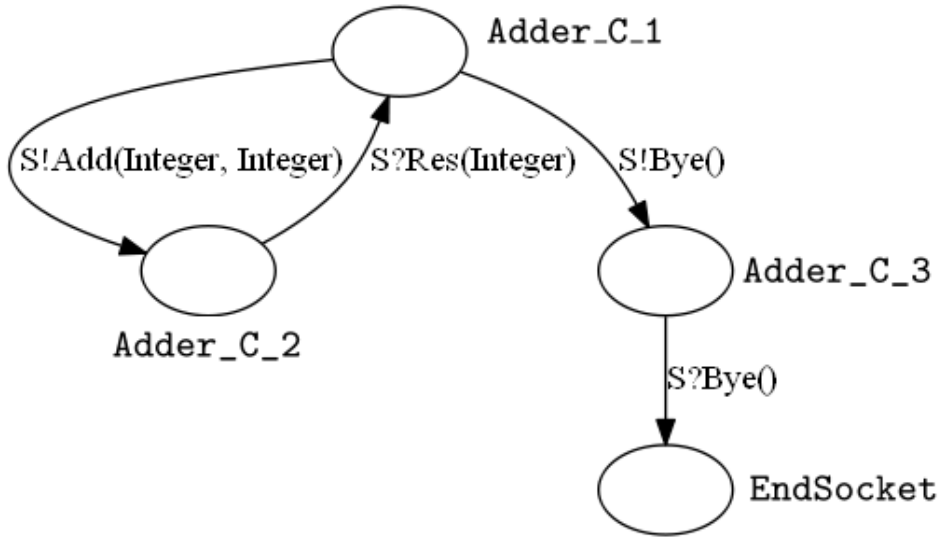
Adder: endpoint implementation for C



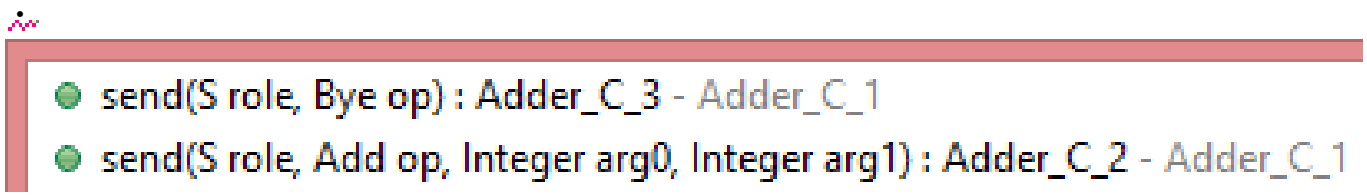

```
Adder_C_1 c1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
c1.send(S, Add, i.val, i.val)
```



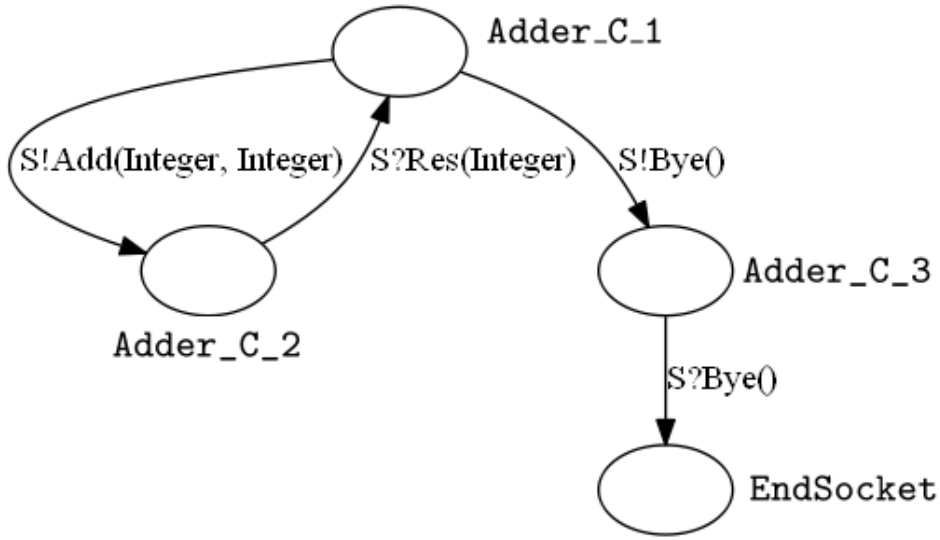
Adder: endpoint implementation for C




```
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)
```



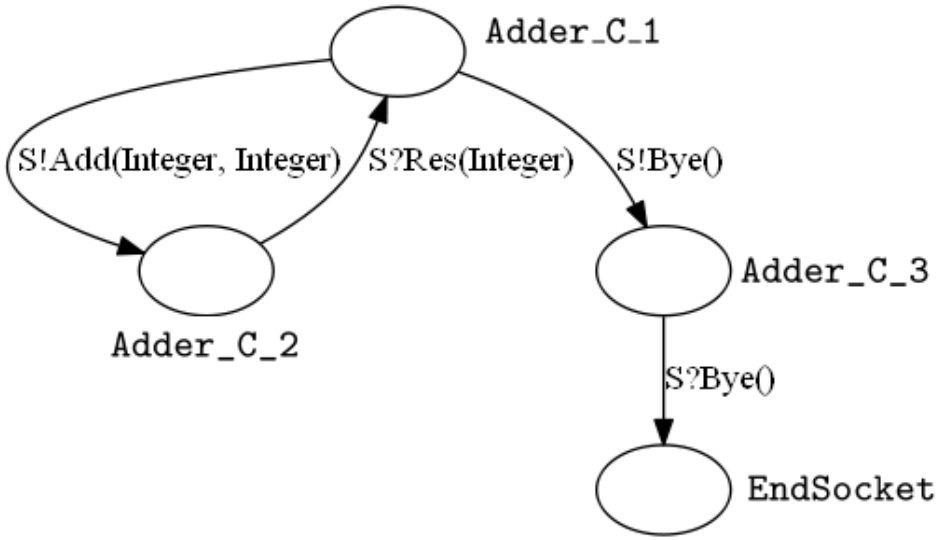
Adder: endpoint implementation for C




```
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

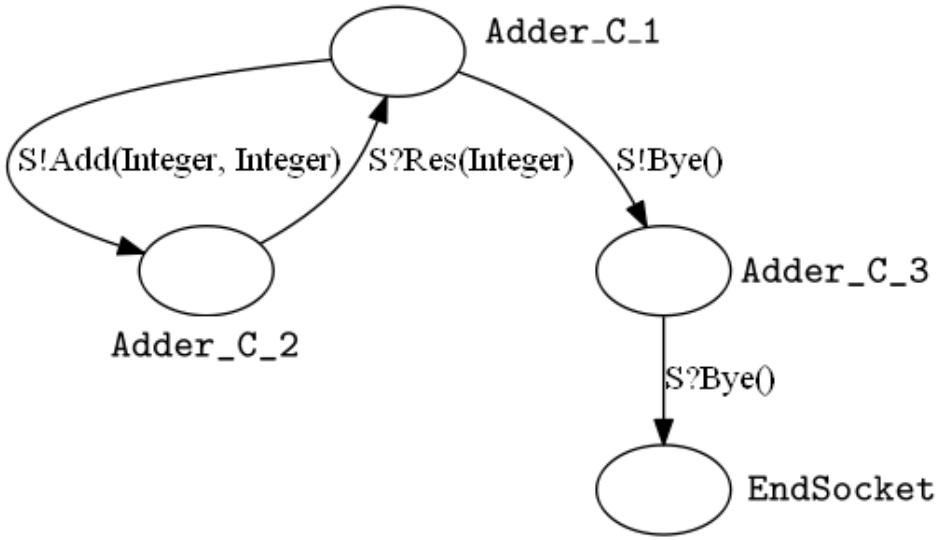
Adder: endpoint implementation for C




```
Adder_C_1 s1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
for (int j = 0; j < N; j++)
  s1 = s1.send(S, Add, i.val, i.val).receive(S, Res, i);
s1.send(S, Bye).receive(S, Bye);
```

EndSocket demo.fase.adder.Adder.Adder.channels.C.Adder_C_3.receive(S role, Bye op
ScribbleRuntimeException, IOException, ClassNotFoundException

Adder: endpoint implementation for C




```
Adder_C_1 s1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
for (int j = 0; j < N; j++)
  s1 = s1.send(S, Add, i.val, i.val).receive(S, Res, i);
s1.send(S, Bye).receive(S, Bye);
```

- Implicit API usage contract:
  - Use each state channel instance exactly once
    - Hybrid session verification:
      Linear channel instance usage checked at run-time by generated API

Hybrid session verification

```
Adder adder = new Adder();
try (SessionEndpoint<Adder, C> se
      = new SessionEndpoint<>(adder, C, ...)) {
  se.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 s1 = new Adder_C_1(se);
  Buf<Integer> i = new Buf<>(1);
  for (int j = 0; j < N; j++)
    s1 = s1.send(S, Add, i.val, i.val).receive(S, Res, i);
  s1.send(S, Bye).receive(S, Bye);
}
```

- Static typing of session I/O actions as State Channel API methods
- Run-time checks on linear usage of state channel instances
  - At most once
    - "Used" flag per channel instance checked and set by I/O actions

Hybrid session verification

```
Adder adder = new Adder();
try (SessionEndpoint<Adder, C> se
      = new SessionEndpoint<>(adder, C, ...)) {
  se.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 s1 = new Adder_C_1(se);
  Buf<Integer> i = new Buf<>(1);
  for (int j = 0; j < N; j++)
    s1 = s1.send(S, Add, i.val, i.val).receive(S, Res, i);
  s1.send(S, Bye).receive(S, Bye);
}
```

- Static typing of session I/O actions as State Channel API methods
- Run-time checks on linear usage of state channel instances
  - At most once
    - "Used" flag per channel instance checked and set by I/O actions

Hybrid session verification

```
Adder adder = new Adder();
try (SessionEndpoint<Adder, C> se
      = new SessionEndpoint<>(adder, C, ...)) {
  se.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 s1 = new Adder_C_1(se);
  Buf<Integer> i = new Buf<>(1);
  for (int j = 0; j < N; j++)
    s1 = s1.send(S, Add, i.val, i.val).receive(S, Res, i);
  s1.send(S, Bye).receive(S, Bye);
}
```

- Static typing of session I/O actions as State Channel API methods
- Run-time checks on linear usage of state channel instances
  - At most once
    - "Used" flag per channel instance checked and set by I/O actions

Hybrid session verification

```
Adder adder = new Adder();
try (SessionEndpoint<Adder, C> se
      = new SessionEndpoint<>(adder, C, ...)) {
  se.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 s1 = new Adder_C_1(se);
  Buf<Integer> i = new Buf<>(1);
  for (int j = 0; j < N; j++)
    s1 = s1.send(S, Add, i.val, i.val).receive(S, Res, i);
  s1.send(S, Bye).receive(S, Bye);
}
```

- Static typing of session I/O actions as State Channel API methods
- Run-time checks on linear usage of state channel instances
  - At most once
    - "Used" flag per channel instance checked and set by I/O actions

Hybrid session verification

```
Adder adder = new Adder();
try (SessionEndpoint<Adder, C> se
      = new SessionEndpoint<>(adder, C, ...)) {
  se.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 s1 = new Adder_C_1(se);
  Buf<Integer> i = new Buf<>(1);
  for (int j = 0; j < N; j++)
    s1 = s1.send(S, Add, i.val, i.val).receive(S, Res, i);
  s1.send(S, Bye).receive(S, Bye);
}
```

- Static typing of session I/O actions as State Channel API methods
- Run-time checks on linear usage of state channel instances
  - At most once
    - "Used" flag per channel instance checked and set by I/O actions
  - At least once
    - "End" flag per endpoint instance set by terminal action
    - Checked via `try` on `AutoCloseable SessionEndpoint`

Hybrid session verification

```
Adder adder = new Adder();
try (SessionEndpoint<Adder, C> se
      = new SessionEndpoint<>(adder, C, ...)) {
  se.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 s1 = new Adder_C_1(se);
  Buf<Integer> i = new Buf<>(1);
  for (int j = 0; j < N; j++)
    s1 = s1.send(S, Add, i.val, i.val).receive(S, Res, i);
  s1.send(S, Bye).receive(S, Bye);
}
```

- Static typing of session I/O actions as State Channel API methods
- Run-time checks on linear usage of state channel instances
  - At most once
    - "Used" flag per channel instance checked and set by I/O actions
  - At least once
    - "End" flag per endpoint instance set by terminal action
    - Checked via `try` on `AutoCloseable SessionEndpoint`
- Hybrid communication safety
  - If state channel linearity respected:
    Communication safety (e.g. [JACM16] Error-freedom) satisfied
  - Regardless of linearity: non-compliant I/O actions are never executed

Another Adder client example

- A recursive Fibonacci client

```
// Result: i1.val is the Nth Fib number
Adder_C_3 fib(Adder_C_1 s1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fib(
        s1.send(S, Add, i1.val, i1.val=i2.val)
          .receive(S, Res, i2),
       i1, i2, i-1)
    : s1.send(S, Bye);
}
```

```
...
fib(s1, new Buf<Integer>(0), new Buf<Integer>(1), N).receive(S, Bye);
...
```

Another Adder client example

- A recursive Fibonacci client

```
// Result: i1.val is the Nth Fib number
Adder_C_3 fib(Adder_C_1 s1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fib(
          s1.send(S, Add, i1.val, i1.val=i2.val)
            .receive(S, Res, i2),
        i1, i2, i-1)
    : s1.send(S, Bye);
}


  ...
  fib(s1, new Buf<Integer>(0), new Buf<Integer>(1), N).receive(S, Bye);
  ...
```

Another Adder client example

- A recursive Fibonacci client

```
// Result: i1.val is the Nth Fib number
Adder_C_3 fib(Adder_C_1 s1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fib(
          s1.send(S, Add, i1.val, i1.val=i2.val)
            .receive(S, Res, i2),
        i1, i2, i-1)
    : s1.send(S, Bye);
}


  ...
  fib(s1, new Buf<Integer>(0), new Buf<Integer>(1), N).receive(S, Bye);
  ...
```

Another Adder client example

- A recursive Fibonacci client

```
// Result: i1.val is the Nth Fib number
Adder_C_3 fib(Adder_C_1 s1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fib(
          s1.send(S, Add, i1.val, i1.val=i2.val)
            .receive(S, Res, i2),
        i1, i2, i-1)
    : s1.send(S, Bye);
}


  ...
  fib(s1, new Buf<Integer>(0), new Buf<Integer>(1), N).receive(S, Bye);
  ...
```

Another Adder client example

- A recursive Fibonacci client

```
// Result: i1.val is the Nth Fib number
Adder_C_3 fib(Adder_C_1 s1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fib(
          s1.send(S, Add, i1.val, i1.val=i2.val)
            .receive(S, Res, i2),
        i1, i2, i-1)
    : s1.send(S, Bye);
}
```

```
...
fib(s1, new Buf<Integer>(0), new Buf<Integer>(1), N).receive(S, Bye);
...
```

Another Adder client example

- A recursive Fibonacci client

```
// Result: i1.val is the Nth Fib number
Adder_C_3 fib(Adder_C_1 s1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fib(
          s1.send(S, Add, i1.val, i1.val=i2.val)
            .receive(S, Res, i2),
        i1, i2, i-1)
    : s1.send(S, Bye);
}


  ...
  fib(s1, new Buf<Integer>(0), new Buf<Integer>(1), N).receive(S, Bye);
  ...
```

Hybrid session verification through Endpoint API generation

- MPST-based generation of rigorous APIs for distributed protocols
  - I/O behaviour of session type role captured by State Channel API
    - Via projected Endpoint FSMs: protocol states as state-specific channels
  - Hybrid verification of state channel usage
    - Native static typing of session I/O actions via state channels methods
    - Supported by run-time checks on linear usage of state channel instances
  - Endpoint API is itself a form of "formal" protocol documentation

- Effective combination of static guidance and run-time checks
  - Practical compromise between safety and flexibility
    - Readily integrates with existing language features and libraries
  - Allows certain benefits of static session typing to be recovered
    - Good value from existing language features, tools and IDE support
  - Methodology can be readily applied to other statically typed languages

- Other hybrid approaches to (binary) ST outside of API generation:

[ML] *A simple library implementation of sessions in ML.* Padovani. `https://hal.archives-ouvertes.fr/hal-01216310/`

[SCALA] *Lightweight sessions in Scala.* Scalas and Yoshida. `www.doc.ic.ac.uk/research/technicalreports/2015/`

SMTP: global protocol

- Simple Mail Transfer Protocol
  - Internet standard for email transmission (RFC 5321)
  - Rich conversation structure
  - Interoperability between "typed" and "untyped" components

```
global protocol Smtp(role S, role C) {
  220 from S to C;
  do Init(C, S);
  do StartTls(C, S);
  do Init(C, S);
  ... // Main mail exchanges
}

global protocol Init(role C, role S) {
  Ehlo from C to S;
⋮
```

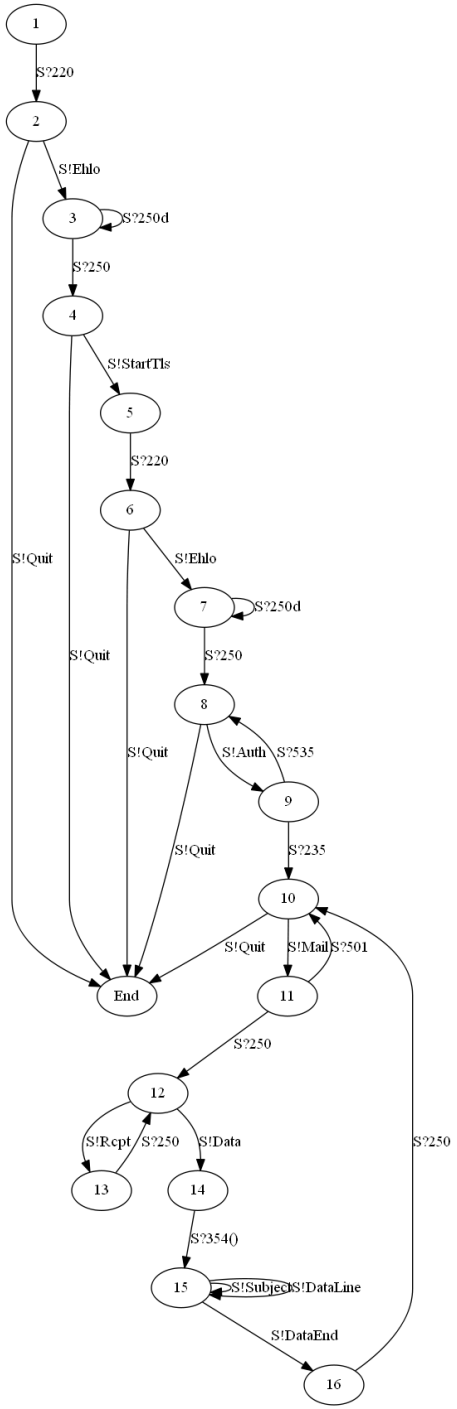
```
⋮
  rec X {
    choice at S {
     250d from S to C;
       continue X;
     } or {
       250 from S to C;
} } }

global protocol StartTls(...) {
⋮
```


[SMTPa] SMTP (IETF RFC 5321). https://tools.ietf.org/html/rfc5321

[SMTPb] SMTP Scribble subset. https://github.com/scribble/scribble-java/blob/master/modules/core/src/test/scrib/demo/smtp/Smtp.scr

SMTP: Client EFSM

- Subset of full SMTP
  - (This EFSM is for a slightly larger fragment than on the previous slide)



SMTP: example protocol implementation error

- Main mail exchange: send a single simple mail
  - Implemented as a trace through the EFSM
  - Protocol violation: missing "end of data" msg

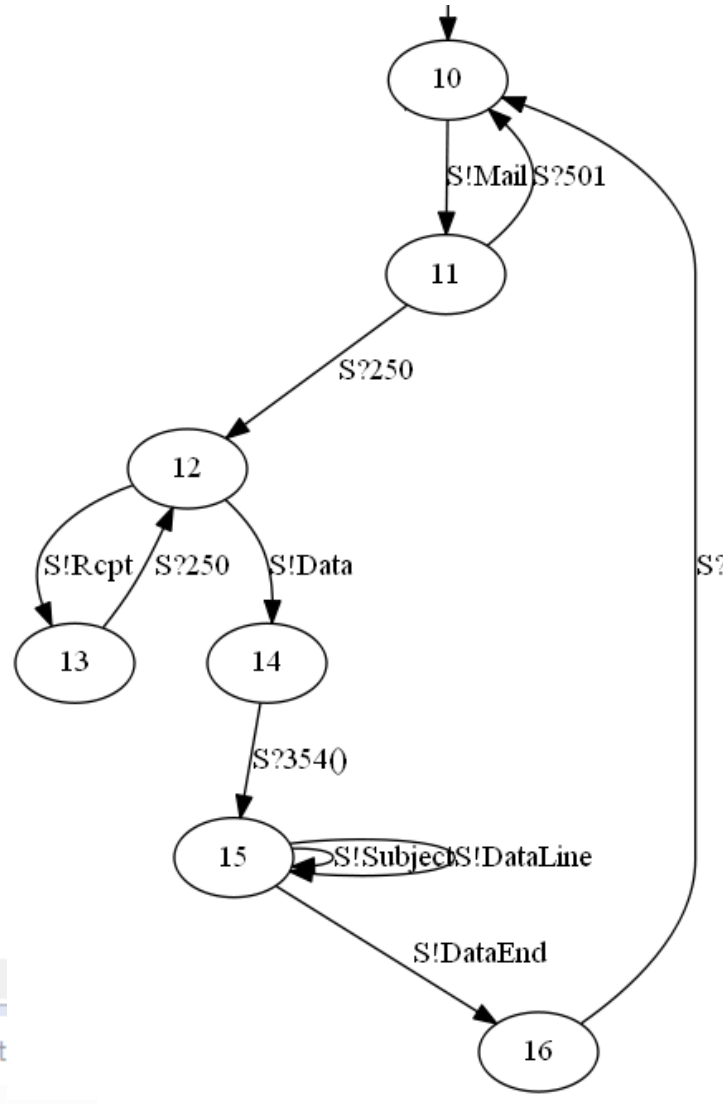
```
83          .send(S, new Mail(mail))
84          .branch(S);
85      switch (cases.getOp())
86      {
87        case _250:
88        {
89          cases.receive(_250)
90            .send(S, new Rcpt(rcpt)).async(S, _250)
91            .send(S, new Data()).async(S, _354)
92            .send(S, new Subject(subj))
93            .send(S, new DataLine(body))
94            //.send(S, new EndOfData())
95            .receive(S, _250, new Buf<>())
96            .send(S, new Quit());
97          break;
98        }
99        case  501:
```

Problems | Javadoc | Declaration | Search | Progress | JUnit

Description

Errors (1 item)

The method receive(S, _250, new Buf<>()) is undefined for the type Smtp_C_15



APIs for programming distributed protocols (background)

- Distributed programming with message passing over channels
  - "Untyped" and unstructured, e.g. `java.net.Socket`

```
int read(byte[] b) // java.io.InputStream
void write(byte[] b) // java.io.OutputStream
```

  - Typed messages but unstructured, e.g. JavaMail API (`com.sun.mail.smtp`)

```
// com.sun.mail.smtp.SMTPTransport implements javax.mail.Transport
protected boolean ehlo(String domain)
protected void mailFrom()
...
```

> Note also that **THERE IS NOT SUFFICIENT DOCUMENTATION HERE TO USE THESE FEATURES**!!! You will need to read the appropriate RFCs mentioned above to understand what these features do and how to use them. Don't just start setting properties and then complain to us when it doesn't work like you expect it to work. **READ THE RFCs FIRST**!!!

[JAVASOCK] Java Socket API. `https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html`

[JAVAMAIL] JavaMail API. `https://javamail.java.net/nonav/docs/api/com/sun/mail/smtp/package-summary.html`

SMTP: session branching

- Non-unary input choice
- API generation approach enables a range of options

SMTP: session branching

- Non-unary input choice
- API generation approach enables a range of options
  - Generate branch-specific enums for standard switch (etc.) patterns
  - Branch performed as separate message input and enum case steps
    - ✓ Familiar (imperative) Java patterns
    - × Additional run-time branch case "cast" check

```
while (true) {
  Smtp_C_3_Cases c = s3.branch(Smtp.S);
  switch (c.op) {
  case _250: Smtp_C_4 s4 = c.receive(_250, buf); return s4;
  case _250d: s3 = c.receive(_250d, buf); break;
} }
```

SMTP: session branching

- Non-unary input choice
- API generation approach enables a range of options
  - Generate branch-specific enums for standard switch (etc.) patterns
  - Branch performed as separate message input and enum case steps
    - ✓ Familiar (imperative) Java patterns
    - × Additional run-time branch case "cast" check

```
while (true) {
  Smtp_C_3_Cases c = s3.branch(Smtp.S);
  switch (c.op) {
  case _250: Smtp_C_4 s4 = c.receive(_250, buf); return s4;
  case _250d: s3 = c.receive(_250d, buf); break;
} }
```

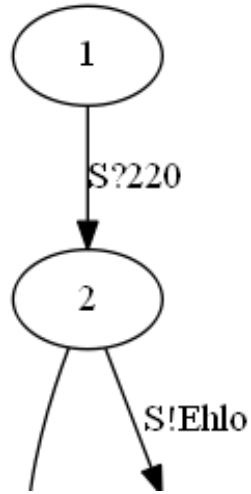
  - Generate branch-specific callback interfaces
    - ✓ Statically safe (up to basic channel linearity)
    - × Requires programming in an "inverted" callback style

```
class MySmtpC3Handler implements Smtp_C_3_Handler {
  void receive(Smtp_C_3 s3, _250d op, Buf<_250d> arg) throws ... {
    s3.branch(S, this);
  }
  void receive(Smtp_C_4 s4, _250 op, Buf<_250> arg) throws ... {
    s4.send(S, new StartTls())
    ...
} }
```

SMTP: input future generation

- Generation of futures for unary input states

```
Buf<Smtp_C_1_Future> f1 = new Buf<>();
...
s3 = s1.async(S, _220, f1)
        .send(S, new Ehlo("..."));
_220 foo = f1.val.sync().msg; // Optional
...
```



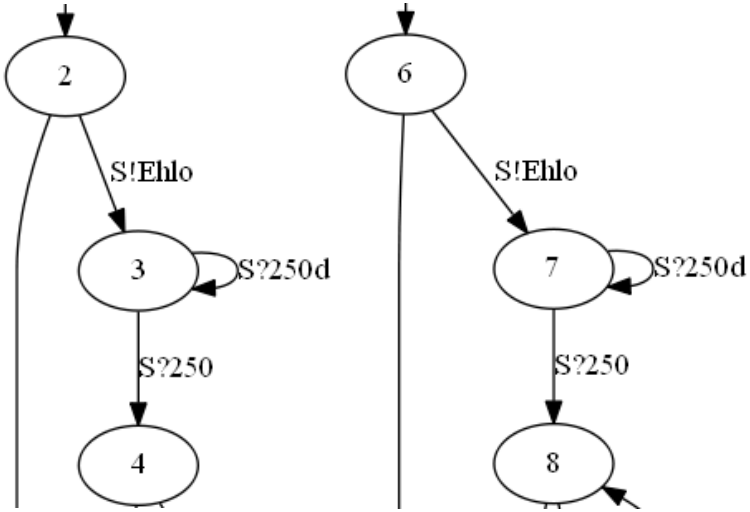
- Safe decoupling of local protocol state transition from message input
  - Non-blocking session input actions, cf. [ECOOP10]
  - Affine "message handling", cf. [FoSSaCS15]
  - "Asynchronous permutation" of I/O actions, cf. [PPDP14]

[ECOOP10] *Type-safe eventful sessions in java*. Hu, Kouzapas, Pernet, Yoshida and Honda.
[FoSSaCS15] *Polarized substructural session types*. Pfenning and Griffith.
[PPDP14] *On the preciseness of subtyping in session types*. Chen, Dezani-Ciancaglini and Yoshida.

SMTP: abstract I/O state interfaces

- Factoring of interaction patterns at the type level

```
global protocol Smtp(role S, role C) {
  220 from S to C;
  do Init(C, S);
  do StartTls(C, S);
  do Init(C, S);
  ...;
}
```



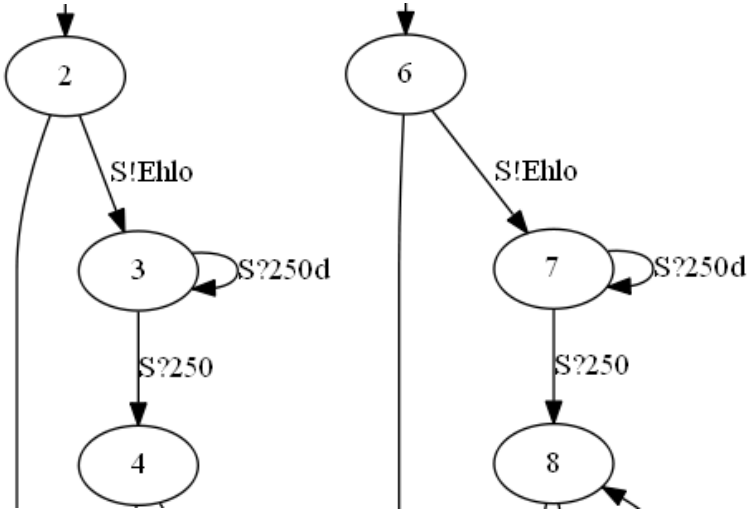
- Basic nominal Java state channel types limit code reuse

```
Smtp_C_4 doInit(Smtp_C_2 s2) throws ...

Smtp_C_8 doInit(Smtp_C_6 s2) throws ...
```

SMTP: abstract I/O state interfaces

- Factoring of interaction patterns at the type level

```
global protocol Smtp(role S, role C) {
  220 from S to C;
  do Init(C, S);
  do StartTls(C, S);
  do Init(C, S);
  ...;
}
```



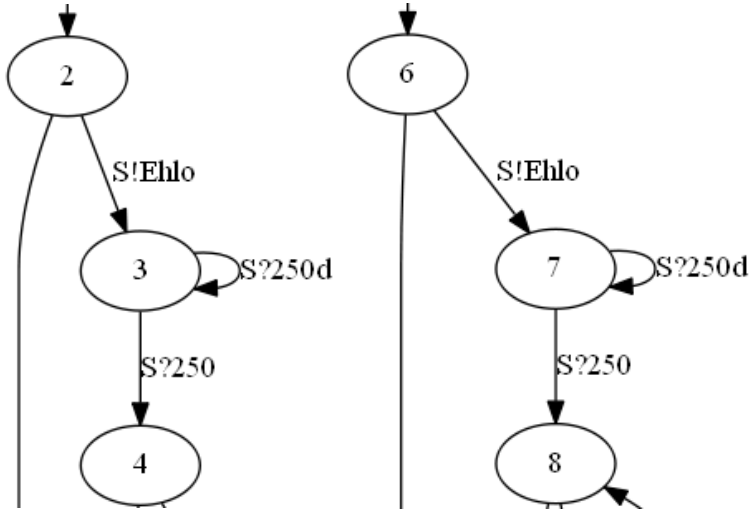
- I/O state interfaces: code factoring, generics inference, subtyping

```
<S1 extends Branch_S$250$_S$250d<S2, S1>, S2 extends Succ_In_S$250>
    S2 doInit(Select_S$Ehlo<S1> s) throws ...
```

SMTP: abstract I/O state interfaces

- Factoring of interaction patterns at the type level

```
global protocol Smtp(role S, role C) {
  220 from S to C;
  do Init(C, S);
  do StartTls(C, S);
  do Init(C, S);
  ...;
}
```



- I/O state interfaces: code factoring, generics inference, subtyping

```
<S1 extends Branch_S$250$_S$250d<S2, S1>, S2 extends Succ_In_S$250>
    S2 doInit(Select_S$Ehlo<S1> s) throws ...
```

```
doInit(
  LinearSocket.wrapClient(
    doInit(s1.async(S, _220, b1))
      .send(S, new StartTls())
      .async(S,  220)
  , S, SSLSo
)
.send(S, new
```

<Smtp_C_3, Smtp_C_4> Smtp_C_4 demo.fase.smtp.FaseClient.doInit (Select_C_S_Ehlo<Smtp_C_3> s) throws Exception

Future work

- make session types good for practice: Extensions to MPST (Scribble)
  - explicit connections
  - Paradigms other than direct message passing channels?
    e.g. actor model, REST, . . . – api gen
  - more properties may want to check (at run-time) – hybrid

- Application of further session types features to practice:
  - events – apigen
  - Explore hybrid verification of further properties: assertions vs. (run-time) dependent types, time ...
  - Augment/combine session types with more advanced constraints
    e.g. message value assertions (HTTP Content-Length), time, . . .