Hybrid session verification through Endpoint API generation

Raymond Hu and Nobuko Yoshida

Imperial College London

Outline

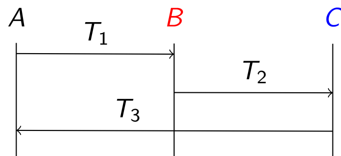
- Background: multiparty session types (MPST)
  - Implementations and applications of session types
- Hybrid session verification through Endpoint API generation
  - Use MPST to generate APIs for implementing distributed protocol endpoints
  - Safety by a combination of static and run-time checks
  - Practical MPST-based (Scribble) toolchain
    - Simple example: Adder service
    - Real-world example: Simple Mail Transfer Protocol (SMTP)

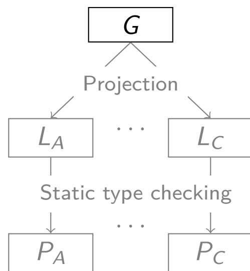
Multiparty session types (background)

- Programming distributed applications
  - Specification: message passing protocol
    - e.g. natural language, sequence diagrams, ...
  - Implentation: endpoint programs
    - Each endpoint performs its respective role in the protocol
  - Potential errors
    - × Communication mismatch
    - × Deadlock
    - × Protocol violation
  - MPST safety properties
    - ✓ Communication safety
    - ✓ Deadlock-freedom (or progress)
    - ✓ Protocol fidelity

Multiparty session types (background)

- Types for specification and verification of message passing programs
  - Originally developed for the π-calculus [POPL08] Honda, Yoshida, Carbone

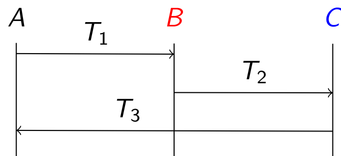


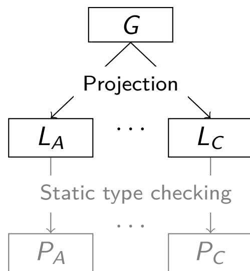


$$G = A \to B : T_1.B \to C : T_2.C \to A : T_3.\text{end}$$

Multiparty session types (background)

- Types for specification and verification of message passing programs
  - Originally developed for the π-calculus [POPL08] Honda, Yoshida, Carbone



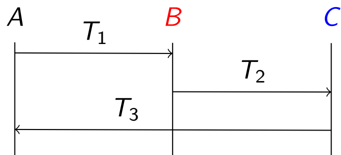


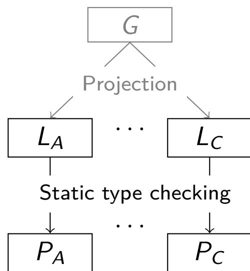
$$G = A \to B : T_1 . B \to C : T_2 . C \to A : T_3 . \mathsf{end}$$

$$L_A = !\langle B, T_1 \rangle . ?(C, T_3) . \mathsf{end}$$

Multiparty session types (background)

- Types for specification and verification of message passing programs
  - Originally developed for the π-calculus [POPL08] Honda, Yoshida, Carbone





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

$$L_A = !\langle B, T_1\rangle.?(C, T_3).\mathsf{end}$$

$$P_A = \bar{a}[A](x).x!\langle B, t_1\rangle.x?(C, u_3).\mathbf{0}$$

Multiparty session types (background)

- Programming distributed applications
  - Specification: message passing protocol
    - e.g. natural language, sequence diagrams, ...
  - Implentation: endpoint programs
    - Each endpoint performs its respective role in the protocol
  - Potential errors
    - × Communication mismatch
    - × Deadlock
    - × Protocol violation
  - MPST safety properties
    - ✓ Communication safety
    - ✓ Deadlock-freedom (or progress)
    - ✓ Protocol fidelity

Implementing and applying session types (related work)

- Static session typing
- Run-time session monitoring
- Code generation from session types

- Extending existing languages, e.g.

SJ (Java)	[ECOOP08] Hu, Yoshida, Honda
Session C	[TOOLS12] Ng, Yoshida, Honda
STING (Java)	[SCP13] Sivaramakrishnan, Ziarek, Nagaraj, Eugster
Links	[ESOP15] Lindley, Morris

  - Need language support for tractability
    - First-class channel I/O primitives
    - Aliasing/linearity control of channel endpoints

Implementing and applying session types (related work)

- Static session typing
- Run-time session monitoring
- Code generation from session types

- Embedding into existing languages, e.g. Haskell

	[PADL04] Neubauer, Thiemann
`sessions`	[ICTechRep08] Sackman, Eisenbach
`simple-sessions`	[HASKELL08] Pucella, Tov
`full-sessions`	[PLACES10] Imai, Yuen, Agusa
`effect-sessions`	[POPL16] Orchard, Yoshida

  - Varying tradeoffs in expressiveness and usability

- New languages, e.g.

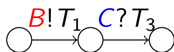
Sing#	[EuroSys06] Fähndrich et al.
SePi (Session Pi)	[BEAT13] Franco, Vasconcelos
SILL (sessions in linear logic)	[ESOP13] Toninho, Caires, Pfenning

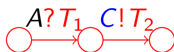
Implementing and applying session types (related work)

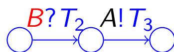
- Static session typing
- Run-time session monitoring
- Code generation from session types

- E.g. generate protocol-specific I/O monitors from MPST
  [RV13] Hu, Neykova, Yoshida, Demangeon, Honda

$$A \rightarrow B : T_1 . B \rightarrow C : T_2 . C \rightarrow A : T_3 .\mathrm{end}$$







  - Direct application of ST to existing (and non-statically typed) languages
  - Run-time verification tradeoffs
- Further refs:

  [ESOP12] Deniélou, Yoshida
  [FMOODS13] Bocchi, Chen, Demangeon, Honda, Yoshida
  [POPL16] Jia, Gommerstadt, Pfenning

Implementing and applying session types (related work)

- Static session typing
- Run-time session monitoring
- Code generation from session types

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

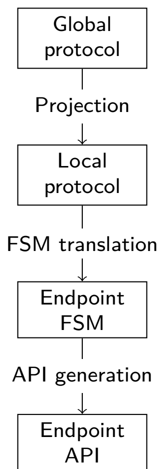
[CC15] Ng, Coutinho, Yoshida
[OOPSLA15] López, Marques, Martins, Ng, Santos, Vasconcelos, Yoshida

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 API for implementing each endpoint of a distributed protocol
  - Cf. ad hoc endpoint implementation from informal specifications

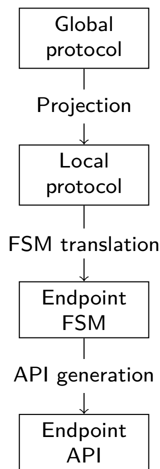
Scribble Endpoint API generation toolchain

- 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



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

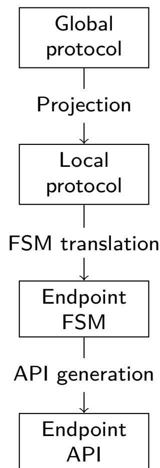


Example: Adder

- 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
  - Role-to-role message passing
  - “Located” choice
  - (Tail) recursive protocols

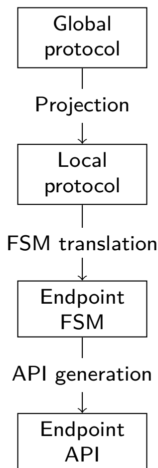


Example: Adder

- 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
  - Role-to-role message passing
  - “Located” choice
  - (Tail) recursive protocols

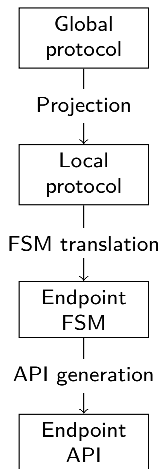


Example: Adder

- 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
  - Role-to-role message passing
  - "Located" choice
  - (Tail) recursive protocols

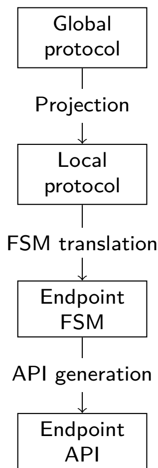


Example: Adder

- Network service for adding two integers

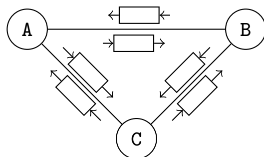
```
global protocol Adder(role C, role S) {
  choice at C {
    Add(Integer, Integer) from C to S;
    Res(Integer) from S to C;
    do Adder(C, S);
  } or {
    Bye() from C to S;
    Bye() from S to C;
  }
}
```

- Scribble global protocol
  - Role-to-role message passing
  - “Located” choice
  - (Tail) recursive protocols



Scribble protocol description language (background)

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

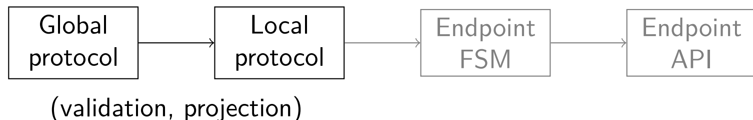



```
1() from A to B;
2() from A to C;
3() from C to B;
```

- Collaboration between researchers (Imperial College London) and industry (Red Hat) developers

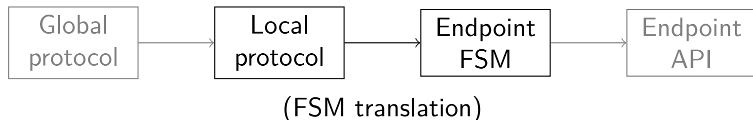
[Scribble] https://github.com/scribble

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;
  }
}
```

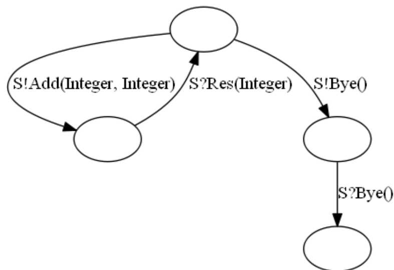
Example: Adder



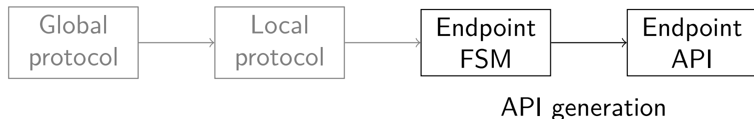
(FSM translation)

```
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;
  }
}
```

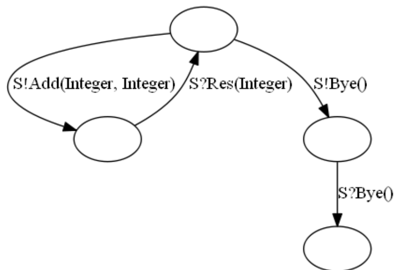
- Projected Endpoint FSM (EFSM) for `C`



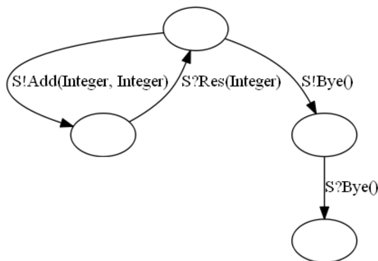
Example: Adder



- EFSM represents the endpoint "I/O behaviour"
  - Capture this I/O structure via the type system of the target language
- Projected Endpoint FSM (EFSM) for C

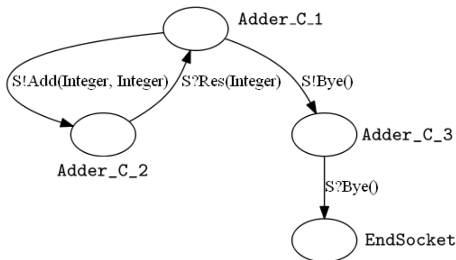


“State Channel” API



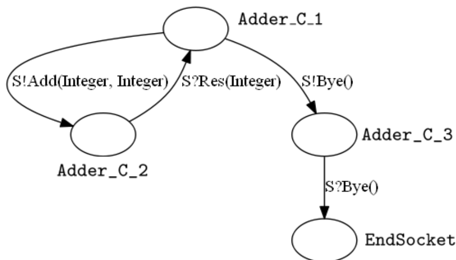
- Protocol states as state-specific channel types
  - Java nominal types: state enumeration as default
  - Generated *state channel* class offers exactly the valid I/O operations for the corresponding protocol state
    - Three state/channel kinds: output, unary input, non-unary input
  - Fluent interface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

"State Channel" API



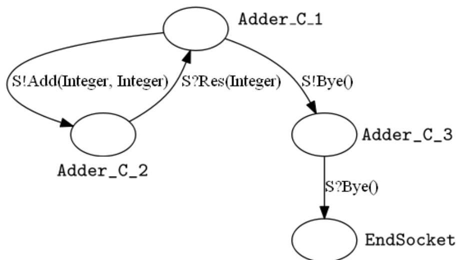
- Protocol states as state-specific channel types
  - Java nominal types: state enumeration as default
  - Generated *state channel* class offers exactly the valid I/O operations for the corresponding protocol state
    - Three state/channel kinds: output, unary input, non-unary input
  - Fluent interface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

“State Channel” API



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

“State Channel” API



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

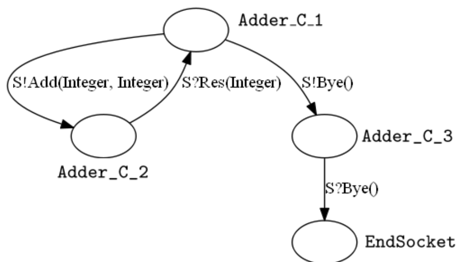
Example: Adder

- Reify session type names as Java singleton types
- 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 run-time sessions of this (initial) type in execution
    - Encapsulates source protocol info, run-time session ID, etc.

Adder: State Channel API for C



- Adder_C_1
  - Output state channel: (overloaded) `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

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:

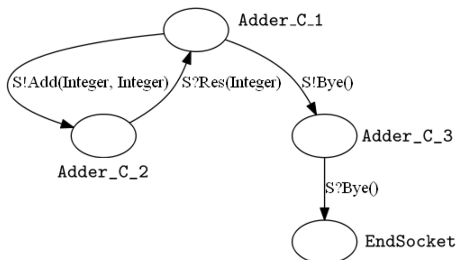
Out_S_Add_Integer_Integer<Adder_C_2>, Out_S_Bye<Adder_C_3>, Select_C_S_Add_Integer_Integer<Adder_C_2>, Select_C_S_Add_Integer_Integer__S_Bye<Adder_C_2,Adder_C_3>, Select_C_S_Bye<Adder_C_3>, Succ_In_S_Res_Integer

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

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

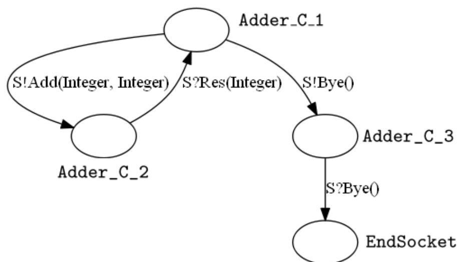
  - Unary input state channel: a `receive` method
  - (Received payload written to a parameterised buffer arg)
  - Recursion: return new instance of a "previous" channel type

- Adder_C_3

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

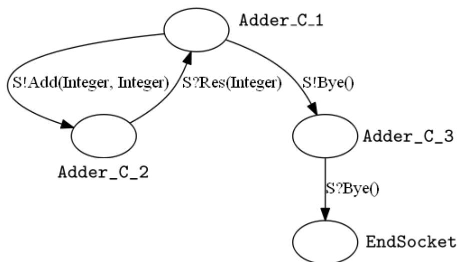
  - `EndSocket` for terminal state

Adder: endpoint implementation for C




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

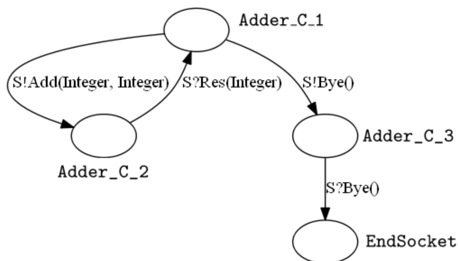
Adder: endpoint implementation for C




```
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

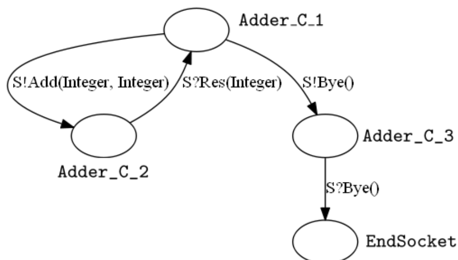
Adder: endpoint implementation for C




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

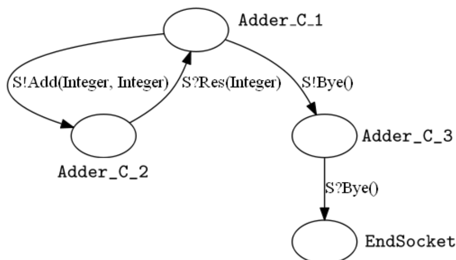
Adder: endpoint implementation for C




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

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

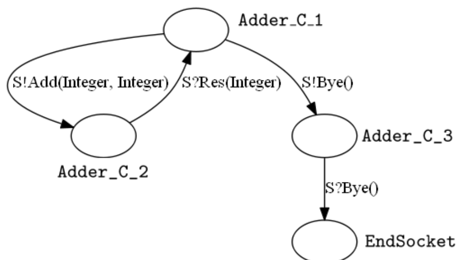
Adder: endpoint implementation for C



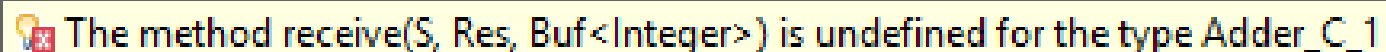

```
Adder_C_1 c1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
c1.send(S, Add, i.val, i.val)
  .receive(S, Res, i)
  .send(S, Add, i.val, i.val)
  .receive(S, Res, i)
  .send(S, Add, i.val, i.val)
  .receive(S, Res, i)
```

```
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
```

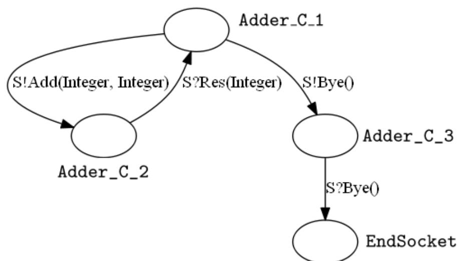
Adder: endpoint implementation for C




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



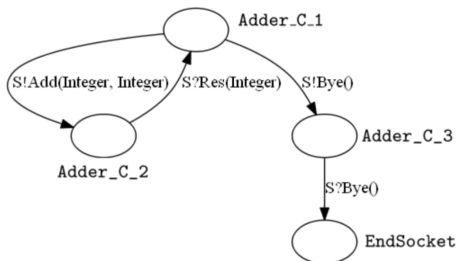
Adder: endpoint implementation for C




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

EndSocket Adder_C_3.receive(S role, Bye op) throws ScribbleRuntimeException, IOException, ClassNotFoundException

Adder: endpoint implementation for C




```
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);
```

- 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> 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);
}
```

- Static typing: session I/O actions as State Channel API methods
- Run-time checks: linear usage of state channel instances

Hybrid session verification

```
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);
}
```

- Static typing: session I/O actions as State Channel API methods
- Run-time checks: linear usage of state channel instances
  - At most once
    - "Used" flag per *channel* instance checked and set by I/O actions
  - At least once
    - "Complete" 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> 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);
}
```

- Static typing: session I/O actions as State Channel API methods
- Run-time checks: linear usage of state channel instances
  - At most once
    - "Used" flag per *channel* instance checked and set by I/O actions
  - At least once
    - "Complete" 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> 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);
}
```

- Static typing: session I/O actions as State Channel API methods
- Run-time checks: linear usage of state channel instances
  - At most once
    - "Used" flag per *channel* instance checked and set by I/O actions
  - At least once
    - "Complete" 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> 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);
}
```

- Static typing: session I/O actions as State Channel API methods
- Run-time checks: linear usage of state channel instances
  - At most once
    - "Used" flag per *channel* instance checked and set by I/O actions
  - At least once
    - "Complete" 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> 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);
}
```

- Static typing: session I/O actions as State Channel API methods
- Run-time checks: linear usage of state channel instances
  - At most once
    - "Used" flag per *channel* instance checked and set by I/O actions
  - At least once
    - "Complete" 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> 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);
}
```

- Static typing: session I/O actions as State Channel API methods
- Run-time checks: linear usage of state channel instances
- Hybrid communication safety
  - If state channel linearity respected:
    Communication safety (e.g. [JACM16] Error-freedom) satisfied
  - Regardless of linearity: non-compliant I/O actions 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 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
     throws ... {
  return (i > 0)
    ? fib(
        c1.send(S, Add, i1.val, i1.val=i2.val)
          .receive(S, Res, i2),
      i1, i2, i-1)
    : c1.send(S, Bye);
}
```

```
  ...
  fib(c1, 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 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fib(
          c1.send(S, Add, i1.val, i1.val=i2.val)
            .receive(S, Res, i2),
        i1, i2, i-1)
    : c1.send(S, Bye);
}


  ...
  fib(c1, 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 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fib(
        c1.send(S, Add, i1.val, i1.val=i2.val)
          .receive(S, Res, i2),
      i1, i2, i-1)
    : c1.send(S, Bye);
}


  ...
  fib(c1, 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 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fib(
        c1.send(S, Add, i1.val, i1.val=i2.val)
          .receive(S, Res, i2),
      i1, i2, i-1)
    : c1.send(S, Bye);
}


  ...
  fib(c1, 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 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fib(
        c1.send(S, Add, i1.val, i1.val=i2.val)
          .receive(S, Res, i2),
      i1, i2, i-1)
    : c1.send(S, Bye);
}
```

```
  ...
  fib(c1, 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 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fib(
        c1.send(S, Add, i1.val, i1.val=i2.val)
          .receive(S, Res, i2),
      i1, i2, i-1)
    : c1.send(S, Bye);
}


  ...
  fib(c1, new Buf<Integer>(0), new Buf<Integer>(1), N).receive(S, Bye);
  ...
```

SMTP: global protocol

- Simple Mail Transfer Protocol
  - Internet standard for email transmission (RFC 5321)
  - Rich conversation structure
  - Interoperability between "typed" and (good) "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;
⋮
```

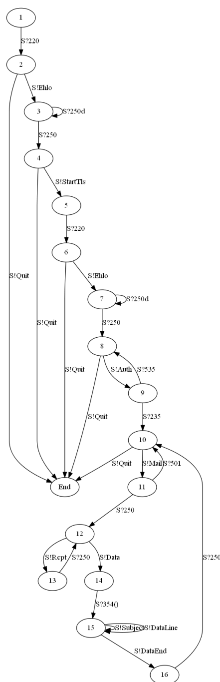
```
⋮
  rec X {
    choice at S {
     250d from S to C;
       continue X;
      } or {
       250 from S to C;
} } }

global protocol StartTls(...) {
⋮
```


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

[SMTPb] SMTP (subset) in Scribble. 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

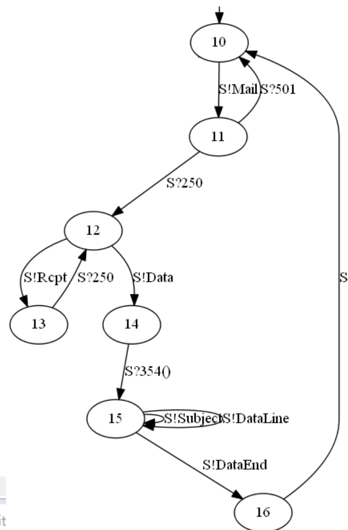
```
83            .send(S, new Mail(mail))
84            .branch(S);
85        switch (cases.getOp())
86        {
87          case _250:
88          {
89            cases.receive(_250)
90              .send(S, new Rcpt(rcpt)).async(S, _250)
91              .send(S, new Data()).async(S, _354)
92              .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

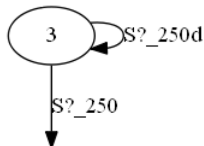
Hybrid session verification through Endpoint API generation

- MPST for rigorous generation of APIs for distributed protocols
  - Static: I/O behaviour (EFSM) of role via State Channel API
  - Run-time: linear state channel usage
- Effective combination of static guidance and run-time checks
  - Recovers certain benefits of static session typing
    - Good value from existing language features, tools and IDE support
  - Generated API as "formal" protocol documentation
  - Methodology can be readily applied to other languages
- Other hybrid approaches to (binary) ST outside of API generation

Inference in ML	[HAL15] Padovani
(Actors in) Scala	[ICTechRep15] Scalas, Yoshida

SMTP: session branching

- Non-unary input choice
- API generation approach enables various options
  - Branch-specific enums.
    User conducts branch as two steps: msg input, then case on enum
  - Branch-specific callback interfaces

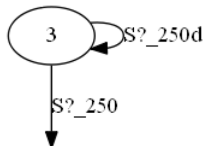


SMTP: session branching

- Non-unary input choice
- API generation approach enables various options
  - Branch-specific enums.
    User conducts branch as two steps: msg input, then case on enum
  - Branch-specific callback interfaces

```
while (true) {
  Smtp_C_3_Cases c = c3.branch(S);
  switch (c.op) {
  case _250: Smtp_C_4 c4 = c.receive(_250, buf); return c4;
  case _250d: c3 = c.receive(_250d, buf); break;
} }
```

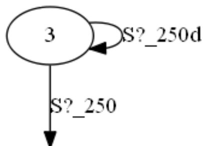
✓ Familiar Java switch (etc.) pattern
× Additional run-time branch continuation “cast”



SMTP: session branching

- Non-unary input choice
- API generation approach enables various options
  - Branch-specific enums.
    User conducts branch as two steps: msg input, then case on enum
  - Branch-specific callback interfaces

```
class MySmtpC3Handler implements Smtp_C_3_Handler {
  void receive(Smtp_C_3 c3, _250d op, Buf<_250d> arg) throws ... {
    c3.branch(S, this);
  }
  void receive(Smtp_C_4 c4, _250 op, Buf<_250> arg) throws ... {
    c4.send(S, new StartTls())
    ...
} }
```

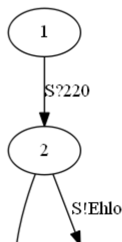


- ✓ Statically safe (up to state channel linearity)
- "Inverted" callback style API

SMTP: input future generation

- Generation of input futures for unary input states

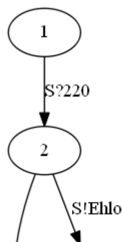
```
Buf<Smtp_C_1_Future> fut1 = new Buf<>();
...
c3 = c1.async(S, _220, fut1)
       .send(S, new Ehlo("..."));
_220 msg = fut1.val.sync().msg; // (Optional)
...
```



SMTP: input future generation

- Generation of input futures for unary input states

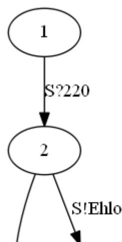
```
Buf<Smtp_C_1_Future> fut1 = new Buf<>();
...
c3 = c1.async(S, _220, fut1)
       .send(S, new Ehlo("..."));
_220 msg = fut1.val.sync().msg; // (Optional)
...
```



SMTP: input future generation

- Generation of input futures for unary input states

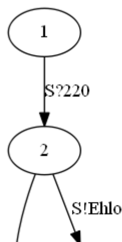
```
Buf<Smtp_C_1_Future> fut1 = new Buf<>();
...
c3 = c1.async(S, _220, fut1)
       .send(S, new Ehlo("..."));
_220 msg = fut1.val.sync().msg; // (Optional)
...
```



SMTP: input future generation

- Generation of input futures for unary input states

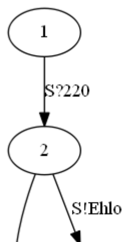
```
Buf<Smtp_C_1_Future> fut1 = new Buf<>();
...
c3 = c1.async(S, _220, fut1)
       .send(S, new Ehlo("..."));
_220 msg = fut1.val.sync().msg; // (Optional)
...
```



SMTP: input future generation

- Generation of input futures for unary input states

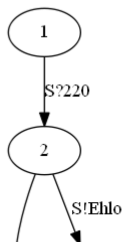
```
Buf<Smtp_C_1_Future> fut1 = new Buf<>();
...
c3 = c1.async(S, _220, fut1)
       .send(S, new Ehlo("..."));
_220 msg = fut1.val.sync().msg; // (Optional)
...
```



SMTP: input future generation

- Generation of input futures for unary input states

```
Buf<Smtp_C_1_Future> fut1 = new Buf<>();
...
c3 = c1.async(S, _220, fut1)
       .send(S, new Ehlo("..."));
_220 msg = fut1.val.sync().msg; // (Optional)
...
```



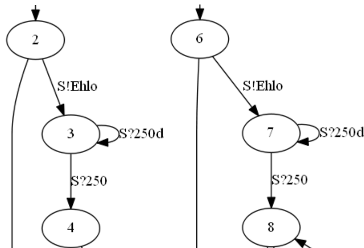
- Safe decoupling of local protocol state transition from message input
  - Non-blocking session input actions, cf. [ECOOP10]
  - "Asynchronous permutation" of I/O actions, cf. [PPDP14]
  - "Affine message handling", cf. [FoSSaCS15]

[ECOOP10] Hu, Kouzapas, Pernet, Yoshida, Honda
[PPDP14] Chen, Dezani-Ciancaglini, Yoshida
[FoSSaCS15] Pfenning, Griffith

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);
  ...;
}
```



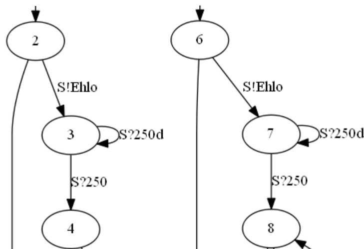
- Basic nominal Java state channel types limit code reuse

```
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);
  ...;
}
```



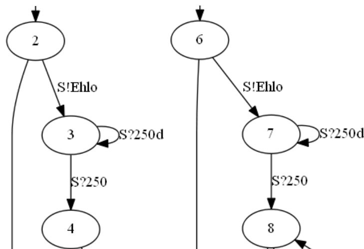
- I/O state interfaces: code factoring, generics inference, subtyping

```
<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 FaseClient.doInit (Select_C_S_Ehlo<Smtp_C_3> s) throws Exception

Future work

- Application of further session types features to practice
  - API generation for hybrid event-driven sessions
  - Hybrid verification for further properties:

    Protocol assertions [CONCUR10] Bocchi, Honda, Tuosto, Yoshida
    Value dependencies [WADLER16] Toninho, Yoshida
    Time [CONCUR15] Bocchi, Lange, Yoshida
    ...

- Practically motivated extensions to MPST (Scribble)
  - Explicit connection actions
  - Paradigms beyond basic message passing channels
    e.g. actors, REST, ...

Further relevant works...

[PPDP12] *Session Types Revisited*. Dardha, Giachino and Sangiorgi.

Encoding between binary typed session-calculus and linear typed π-calculus.

[WGP15] *Session Types for Rust*. Jespersen, Munksgaard and Larsen.

Adapts `simple-sessions` [HASKELL08] interface for affine types in Rust.

[ESOP10] *Stateful Contracts for Affine Types*. Tov and Pucella.

Adapts behavioural contracts [ICFP02] to mediate between affine and conventional typed code.

[TOPLAS14] *Foundations of Typestate-Oriented Programming*. Garcia, Tanter, Wolff and Aldrich.

Type-state oriented programming with gradual typing.

[TOPLAS10] *Hybrid Type Checking*. Knowles and Flanagan.

Static type checking backed up by dynamic typecasts/coercions.

[ICFP02] *Contracts for Higher-Order Functions*. Findler and Felleisen.

Higher-order assertion contracts checked upon application (execution).

Thanks!

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;
  }
}
```