Multiparty protocol specification and endpoint implementation using Scribble

Raymond Hu

Imperial College London

http://www.doc.ic.ac.uk/~rhu/betty16b.pdf

Aims

- Scribble
  - Implementation and application of MPST to current practices
  - Specify real-world protocols
  - Implement fully interoperable endpoints in mainstream languages

Implementing and applying session types

- 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
StMungo	[PPDP16] Kouzapas, Dardha, Perera, Gay

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

[FTPL13] *Behavioral Types in Programming Languages*. BETTY WG3 (Languages). `http://summerschool2016.behavioural-types.eu/programme/WG3SOAR.pdf`

Implementing and applying session types

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

- Embedding into existing languages, e.g. Haskell

	[PADL04] Neubauer, Thiemann
`sessions`	[ICTR08] 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

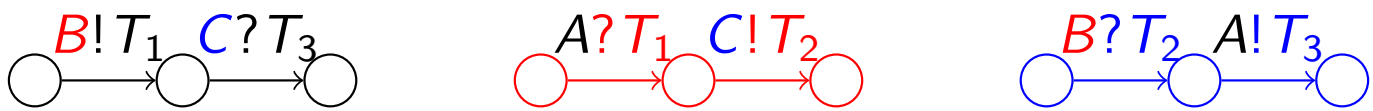
Implementing and applying session types

- 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

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



  - 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

- 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

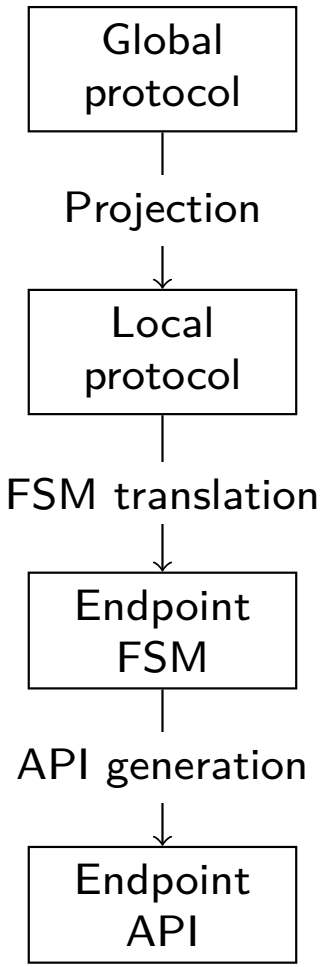
[CC15] Ng, Coutinho, Yoshida
[OOPSLA15] López, Marques, Martins, Ng, Santos, Vasconcelos, Yoshida

Hybrid session verification through Endpoint API generation

- Hybrid session verification
  - A combination of static and run-time checks
  - Directly for mainstream languages (with static typing)
  - 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

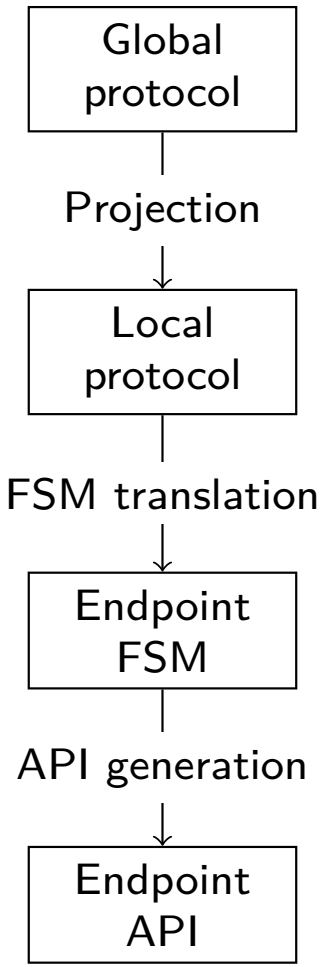
Scribble Endpoint API generation toolchain

- 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



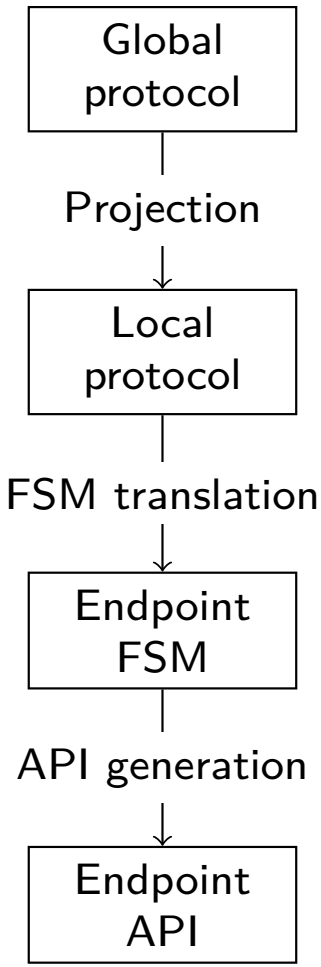
Scribble Endpoint API generation toolchain

- 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



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

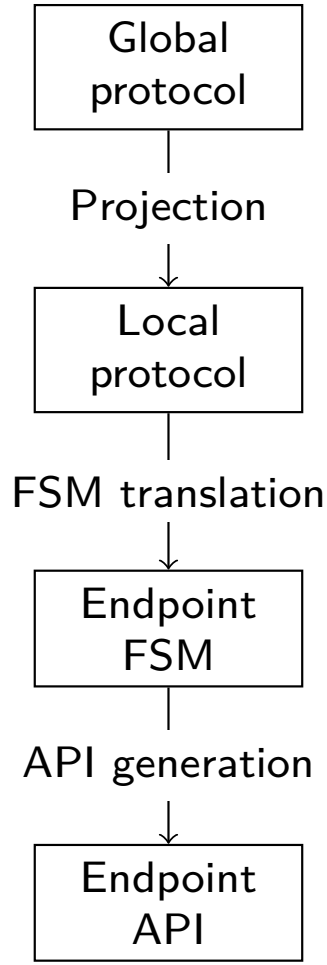


Example: Adder

- 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



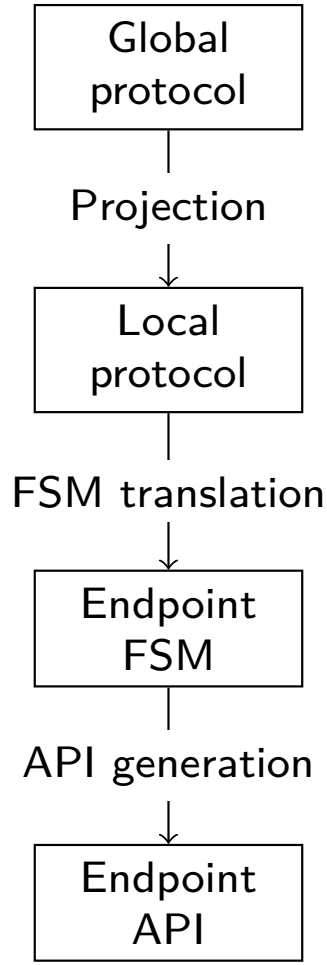
- `https://github.com/rhu1/scribble-java/tree/rhu1-research/modules/core/src/test/scrib/demo/betty16/lec2/adder`

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



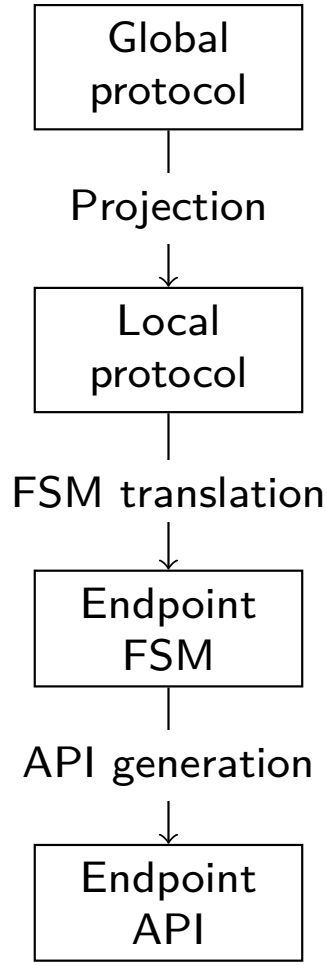
- `https://github.com/rhu1/scribble-java/tree/rhu1-research/modules/core/src/test/scrib/demo/betty16/lec2/adder`

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



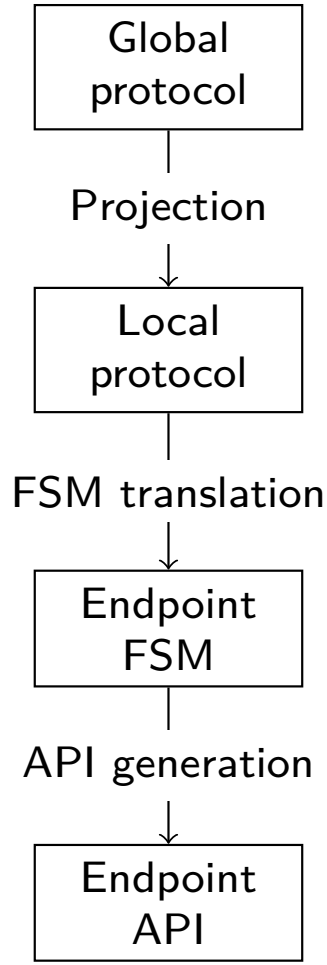
- `https://github.com/rhu1/scribble-java/tree/rhu1-research/modules/core/src/test/scrib/demo/betty16/lec2/adder`

Example: Adder

- Network service for adding two integers

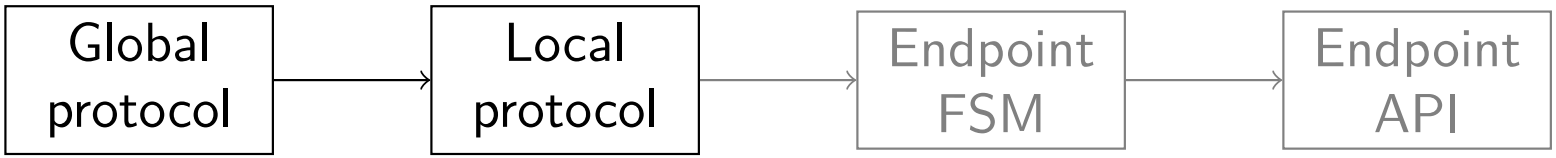
```
global protocol Adder(role C, role S) {
  choice at C {
    Add(Integer, Integer) from C to S;
    Res(Integer) from S to C;
    do Adder(C, S);
  } or {
    Bye() from C to S;
    Bye() from S to C;
  }
}
```

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



- `https://github.com/rhu1/scribble-java/tree/rhu1-research/modules/core/src/test/scrib/demo/betty16/lec2/adder`

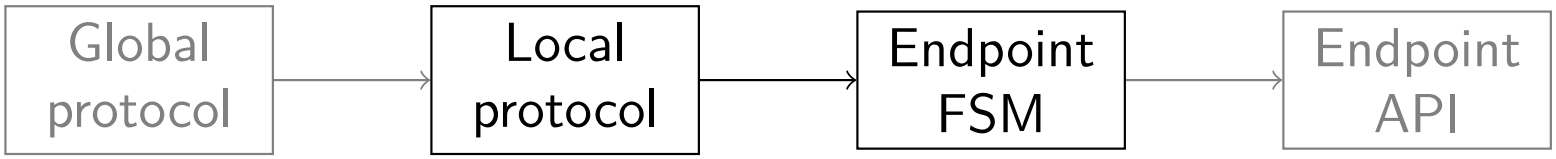
Example: Adder



(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;
  }
}
```

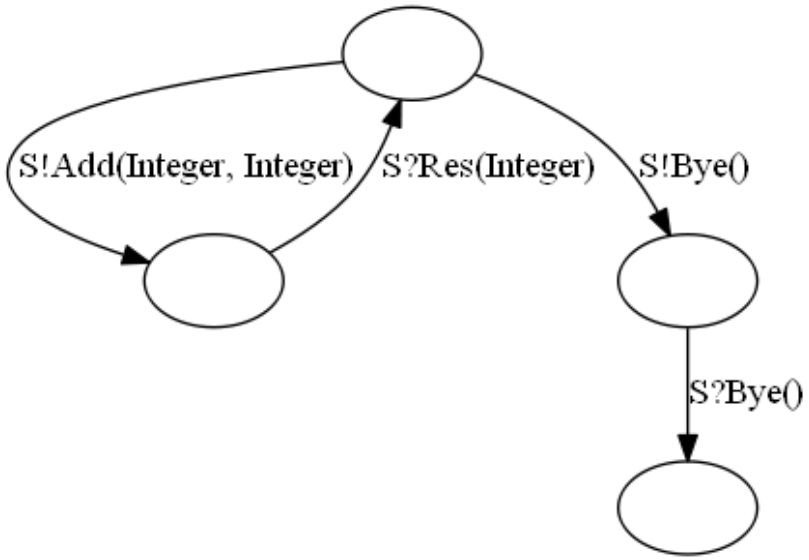
Example: Adder



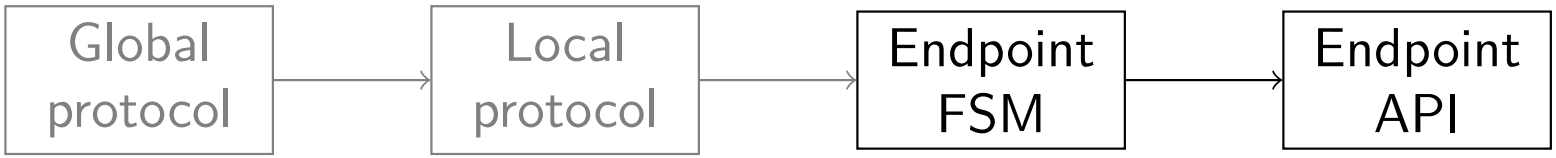
(FSM translation)

```
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

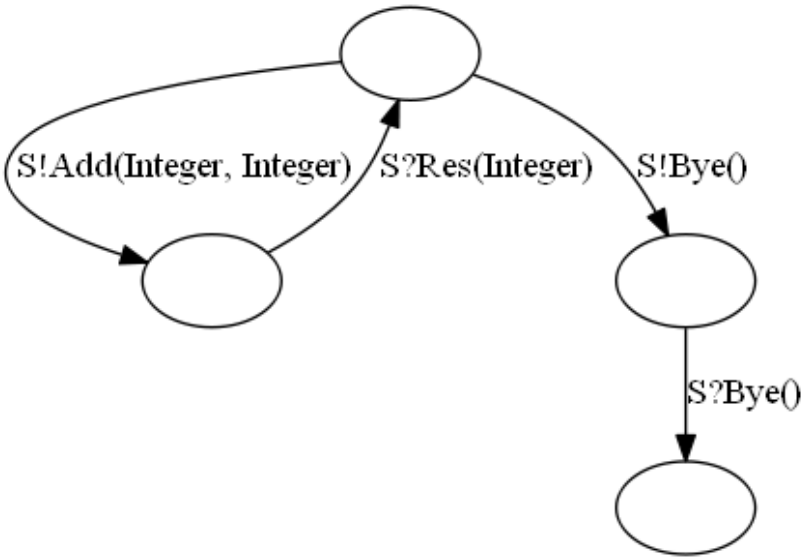


Example: Adder

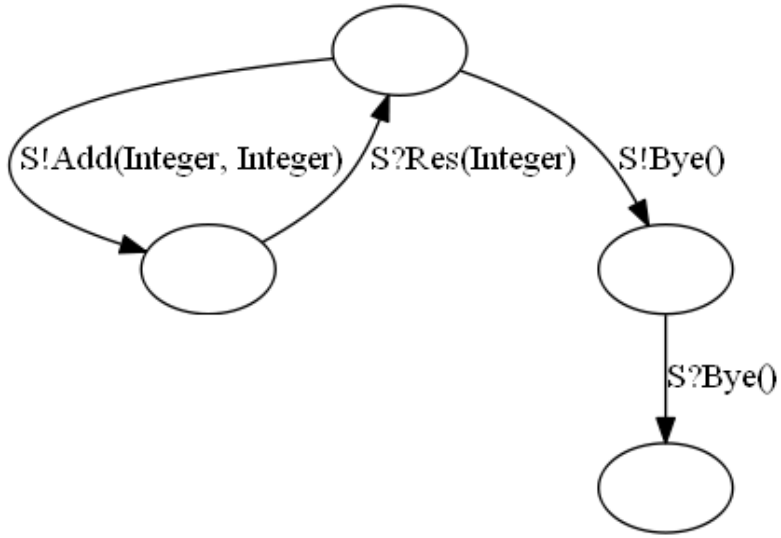


API generation

- 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

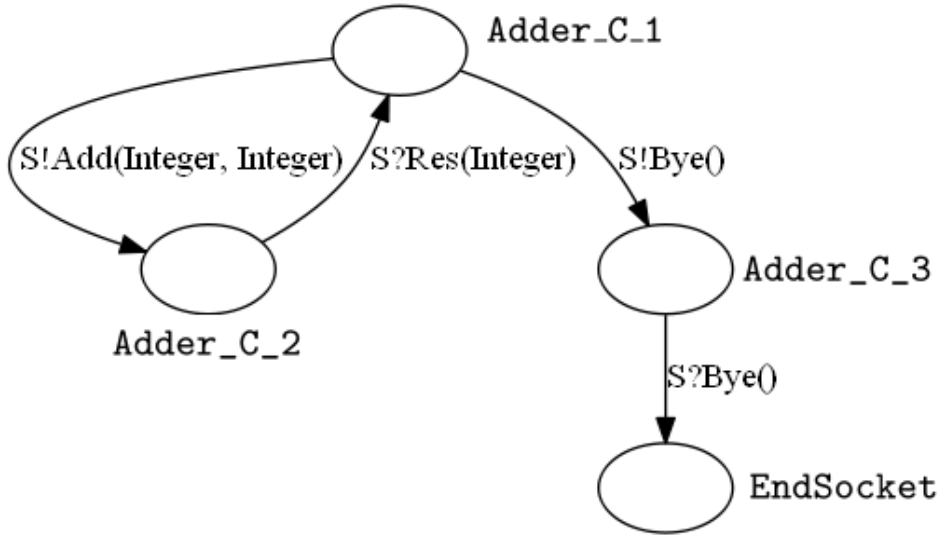


“State Channel” API



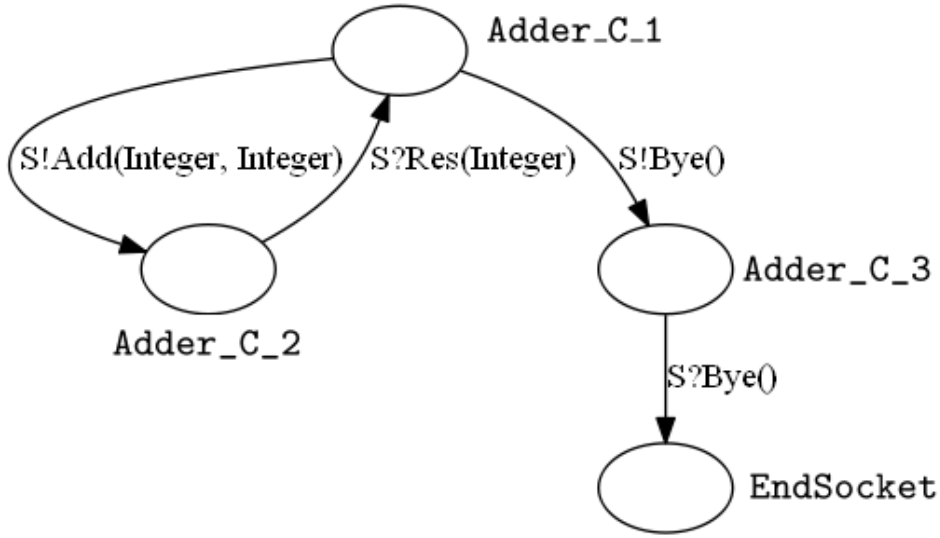
- Protocol states as state-specific channel types
  - Java nominal types: state enumeration as default
  - Generated *state channel* class offers exactly the valid I/O operations for the corresponding protocol state
    - Three state/channel kinds: output, unary input, non-unary input
  - Fluent interface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

“State Channel” API



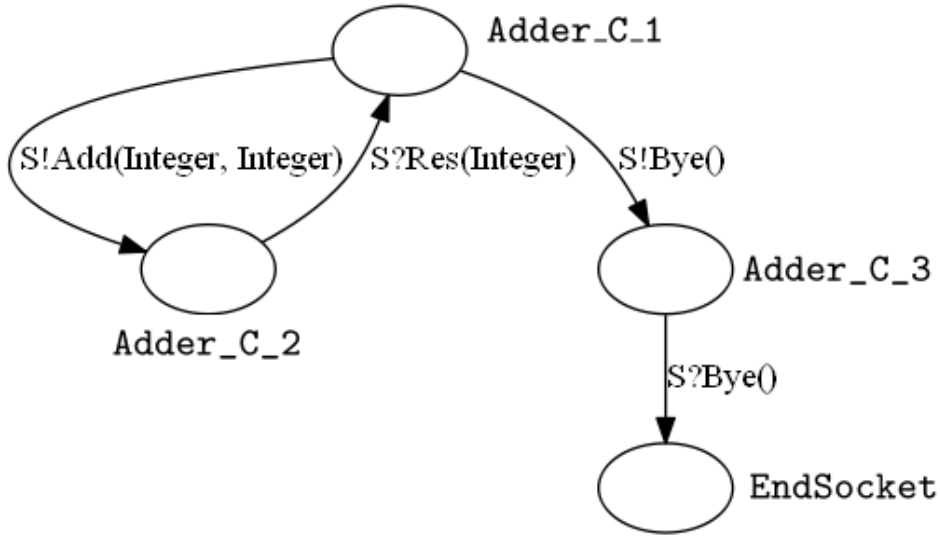
- Protocol states as state-specific channel types
  - Java nominal types: state enumeration as default
  - Generated *state channel* class offers exactly the valid I/O operations for the corresponding protocol state
    - Three state/channel kinds: output, unary input, non-unary input
  - Fluent interface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

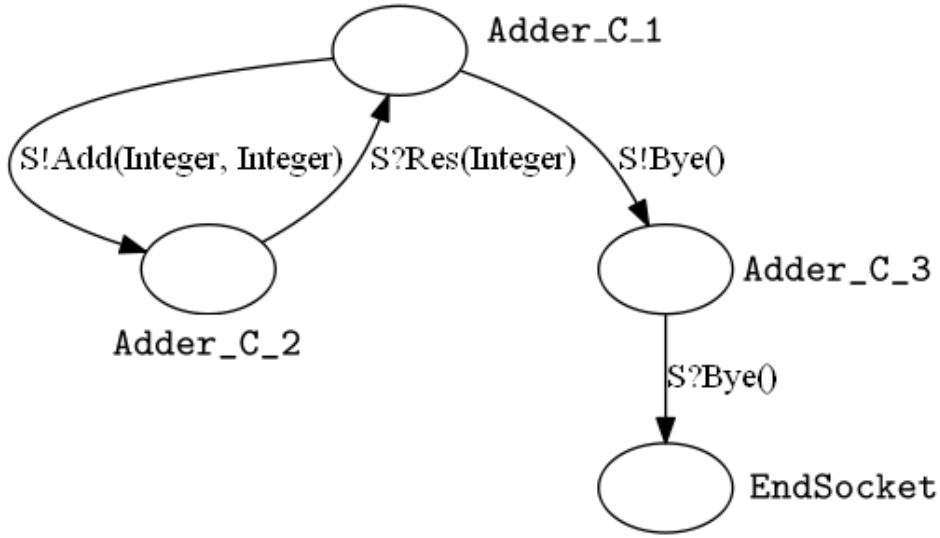
Example: Adder

- 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:

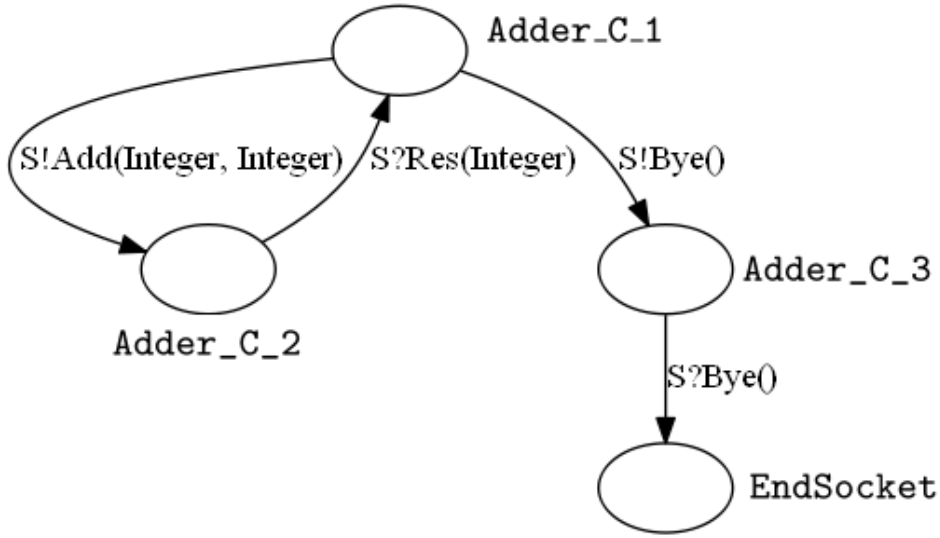
`Out_S_Add_Integer_Integer<Adder_C_2>, Out_S_Bye<Adder_C_3>, Select_C_S_Add_Integer_Integer<Adder_C_2>, Select_C_S_Add_Integer_Integer__S_Bye<Adder_C_2,Adder_C_3>, Select_C_S_Bye<Adder_C_3>, Succ_In_S_Res_Integer`

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 ...
  ```

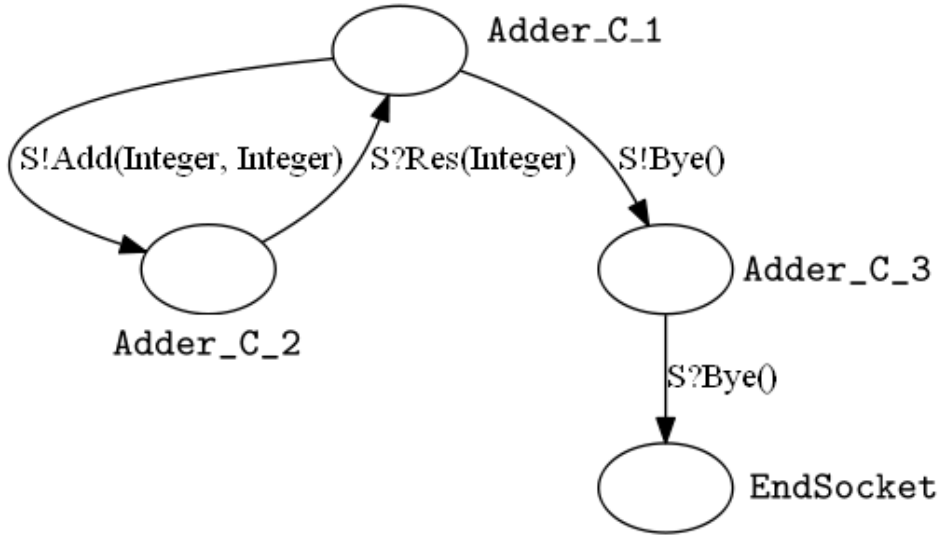
  - Unary input state channel: a `receive` method
  - (Received payload written to a parameterised buffer arg)
  - Recursion: return new instance of a "previous" channel type

- `Adder_C_3`

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

  - `EndSocket` for terminal state

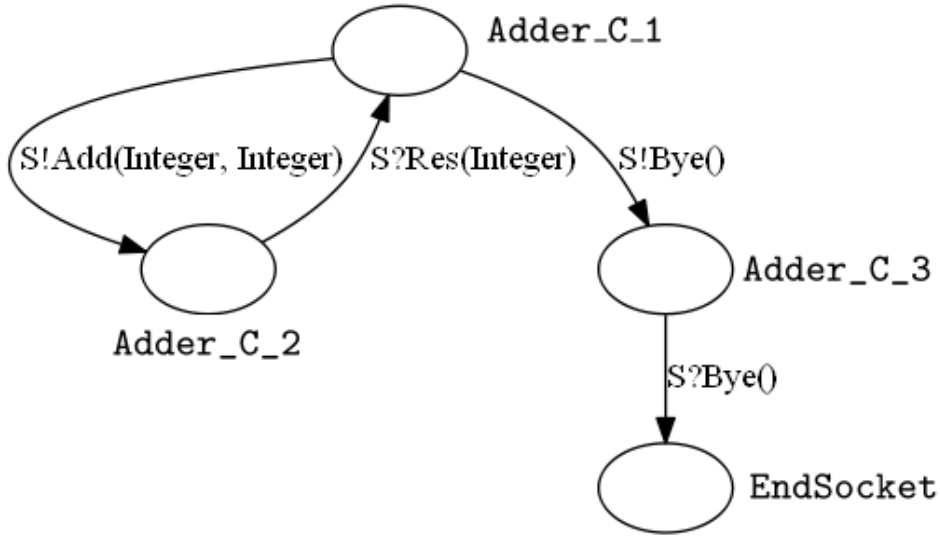
Adder: endpoint implementation for C




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

The value of the local variable c1 is not used

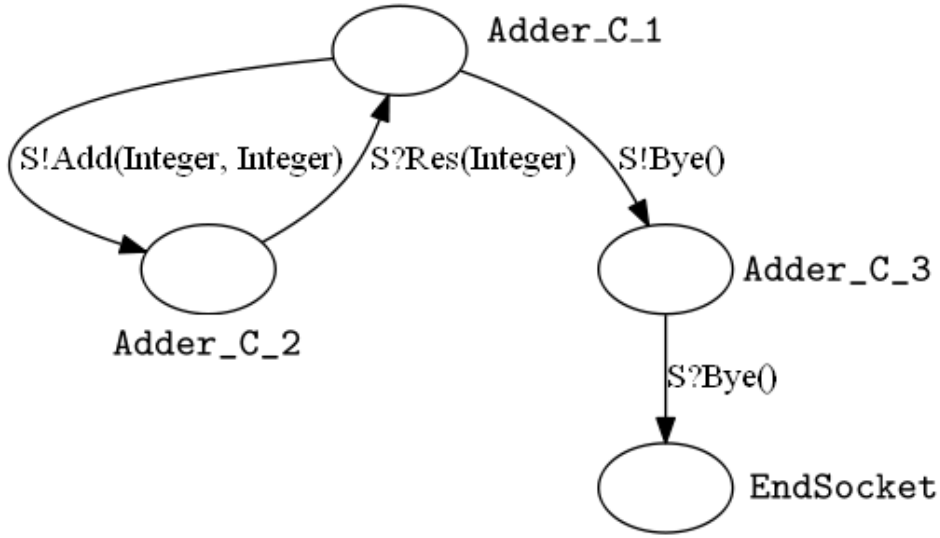
Adder: endpoint implementation for C




```
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

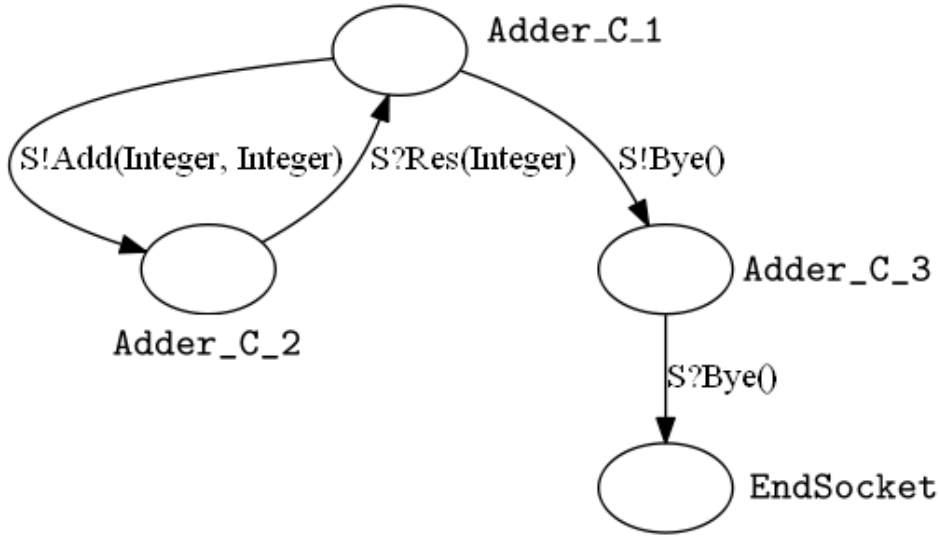
Adder: endpoint implementation for C



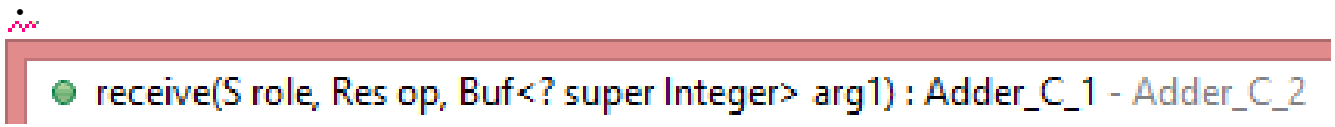

```
Adder_C_1 c1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
c1.send(S, Add, i.val, i.val);
```

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

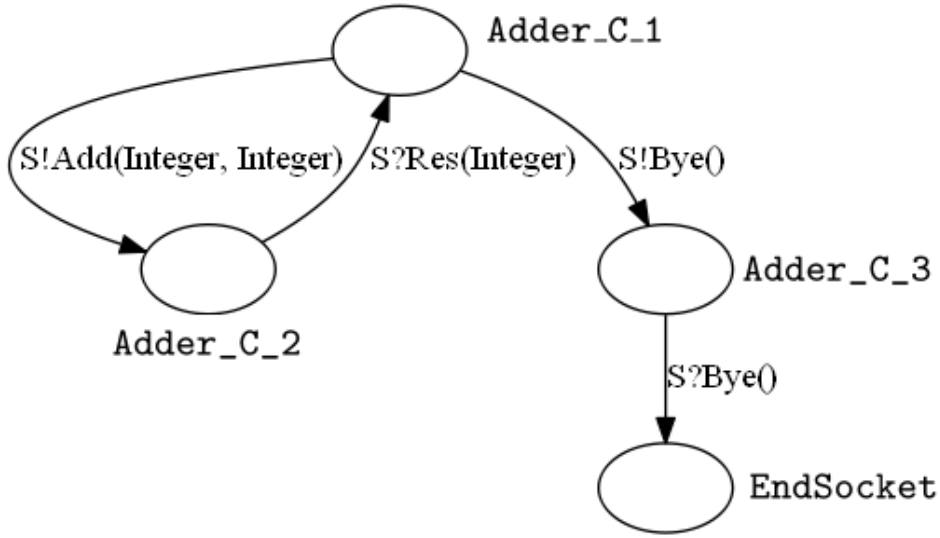
Adder: endpoint implementation for C



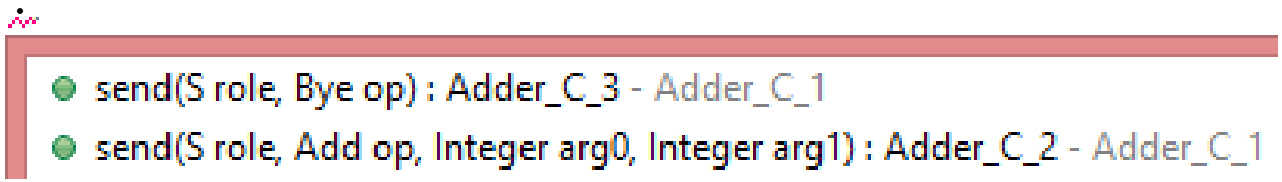

```
Adder_C_1 c1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
c1.send(S, Add, i.val, i.val)
```



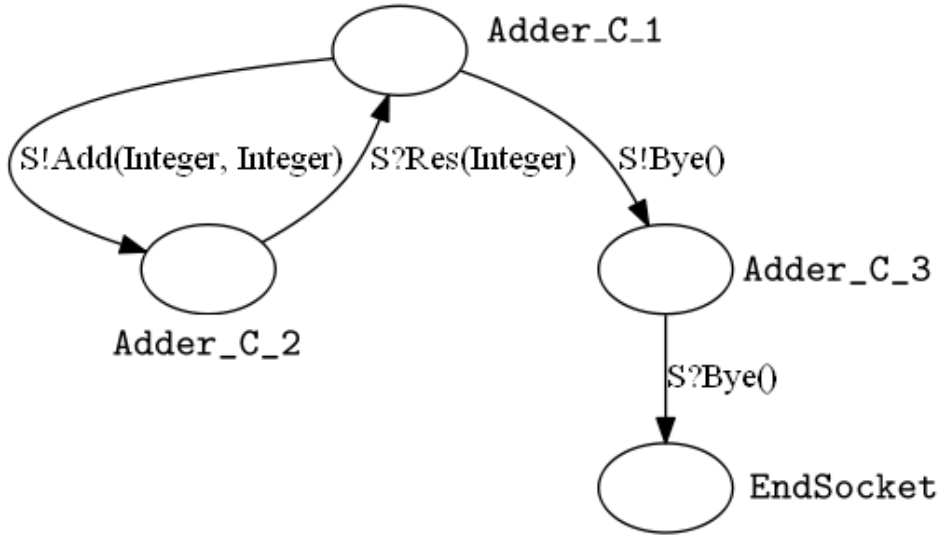
Adder: endpoint implementation for C




```
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)
```



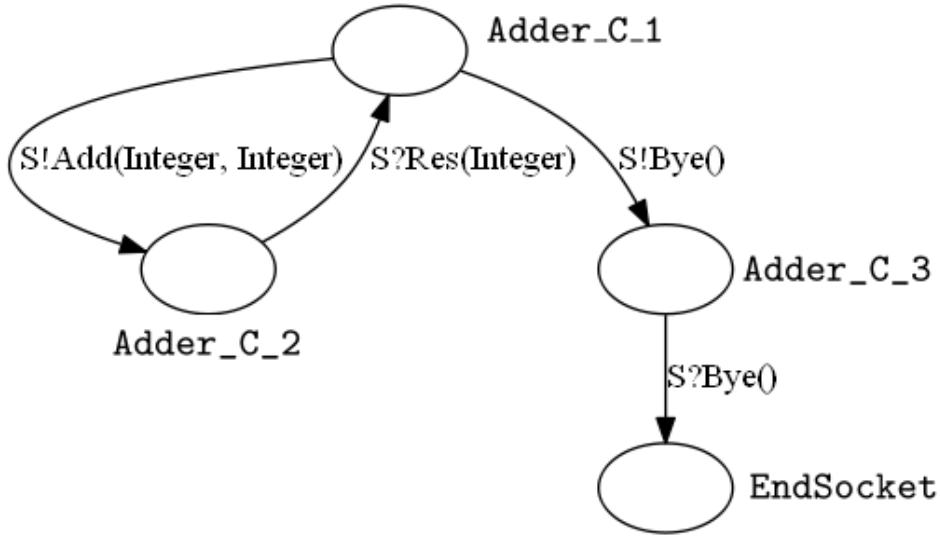
Adder: endpoint implementation for C




```
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

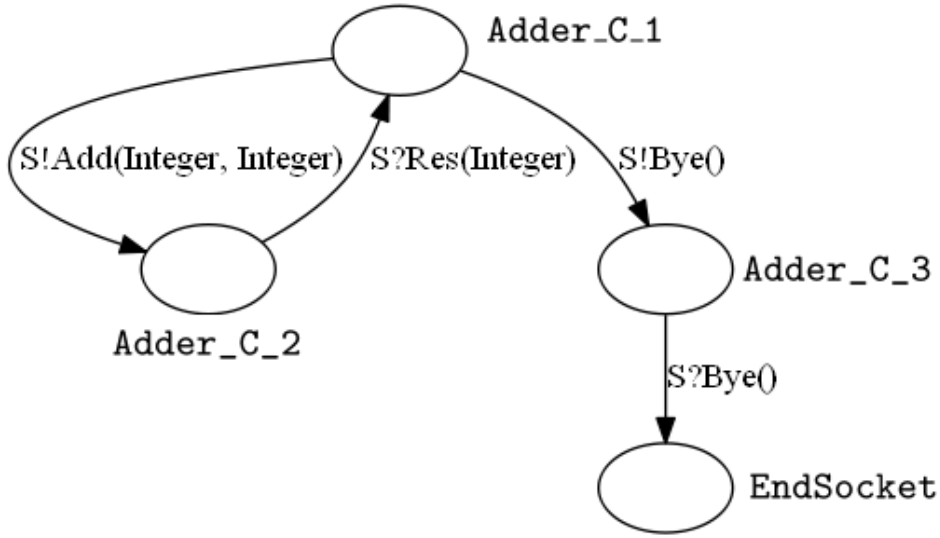



```
Adder_C_1 c1 = new Adder_C_1(...);
Buf<Integer> i = new Buf<>(1);
while (i.val < N)
  c1 = c1.send(S, Add, i.val, i.val).receive(S, Res, i);
c1.send(S, Bye).receive(S, Bye);
```

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

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

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> client
      = new SessionEndpoint<>(adder, C, ...)) {
  client.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 c1 = new Adder_C_1(client);
  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> client
      = new SessionEndpoint<>(adder, C, ...)) {
  client.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 c1 = new Adder_C_1(client);
  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> client
      = new SessionEndpoint<>(adder, C, ...)) {
  client.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 c1 = new Adder_C_1(client);
  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> client
      = new SessionEndpoint<>(adder, C, ...)) {
  client.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 c1 = new Adder_C_1(client);
  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> client
      = new SessionEndpoint<>(adder, C, ...)) {
  client.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 c1 = new Adder_C_1(client);
  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> client
      = new SessionEndpoint<>(adder, C, ...)) {
  client.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 c1 = new Adder_C_1(client);
  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

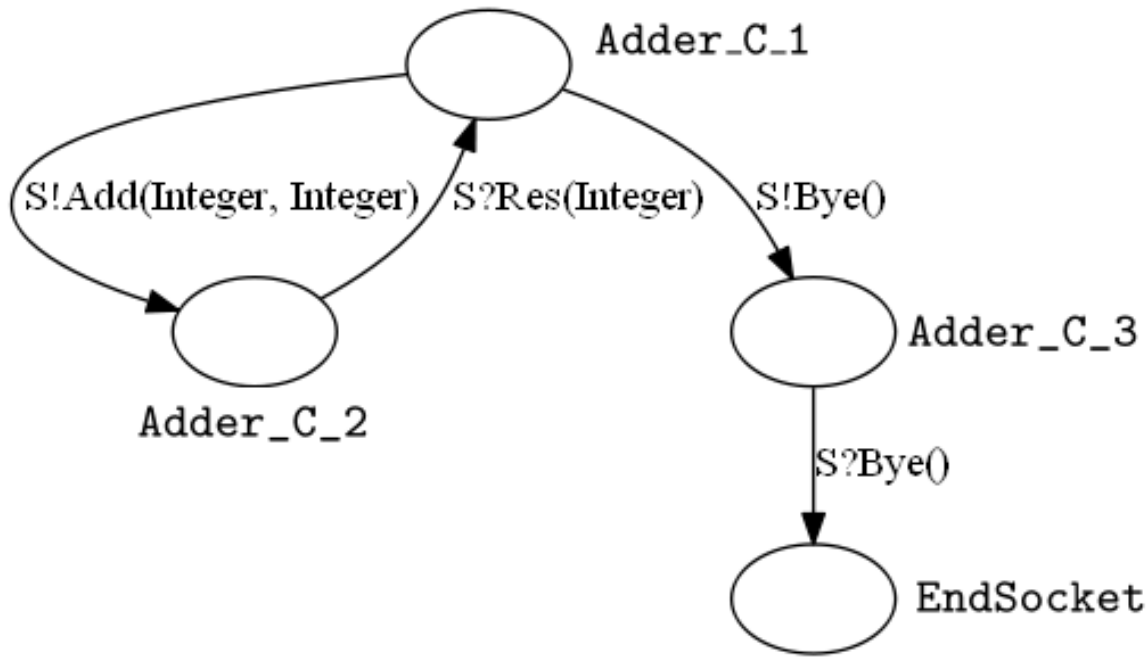
```
Adder adder = new Adder();
try (SessionEndpoint<Adder, C> client
      = new SessionEndpoint<>(adder, C, ...)) {
  client.connect(S, SocketChannelEndpoint::new, host, port);
  Adder_C_1 c1 = new Adder_C_1(client);
  Buf<Integer> i = new Buf<>(1);
  while (i.val < N)
    c1 = c1.send(S, Add, i.val,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

Exercise: recursive Fibonacci client

- `fibo(c1, new Buf<Integer>(0), new Buf<Integer>(1), N);`

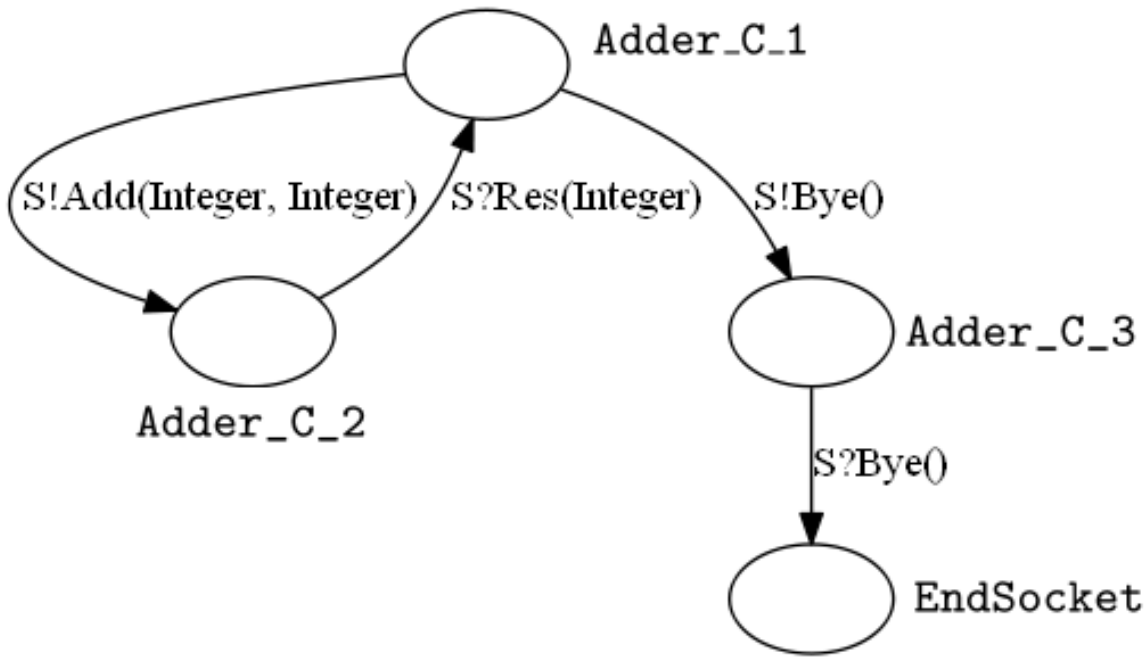
```
// Result: i1.val is the N-th Fib number
EndSocket fibo(Adder_C_1 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {


        ..c1.send(S, Add, i1.val, i1.val=i2.val)..



}
```



Exercise: recursive Fibonacci client

- `fibo(c1, new Buf<Integer>(0), new Buf<Integer>(1), N);`

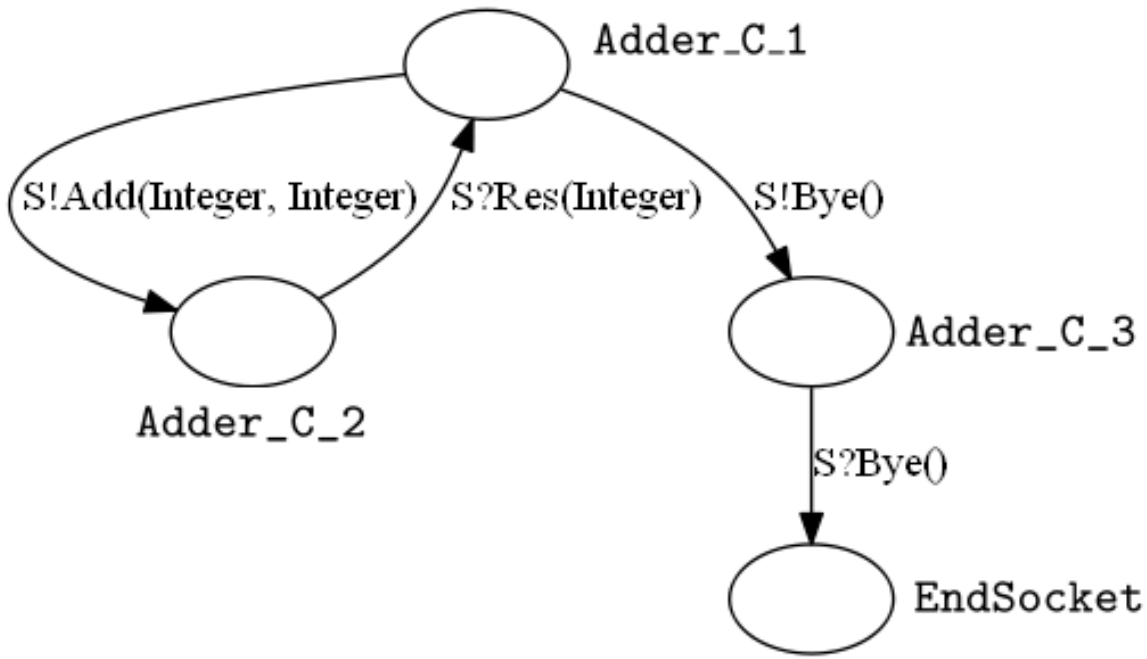
```
// Result: i1.val is the N-th Fib number
EndSocket fibo(Adder_C_1 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fibo(
        c1.send(S, Add, i1.val, i1.val=i2.val)

                  )
    :
}
```



Exercise: recursive Fibonacci client

- `fibo(c1, new Buf<Integer>(0), new Buf<Integer>(1), N);`

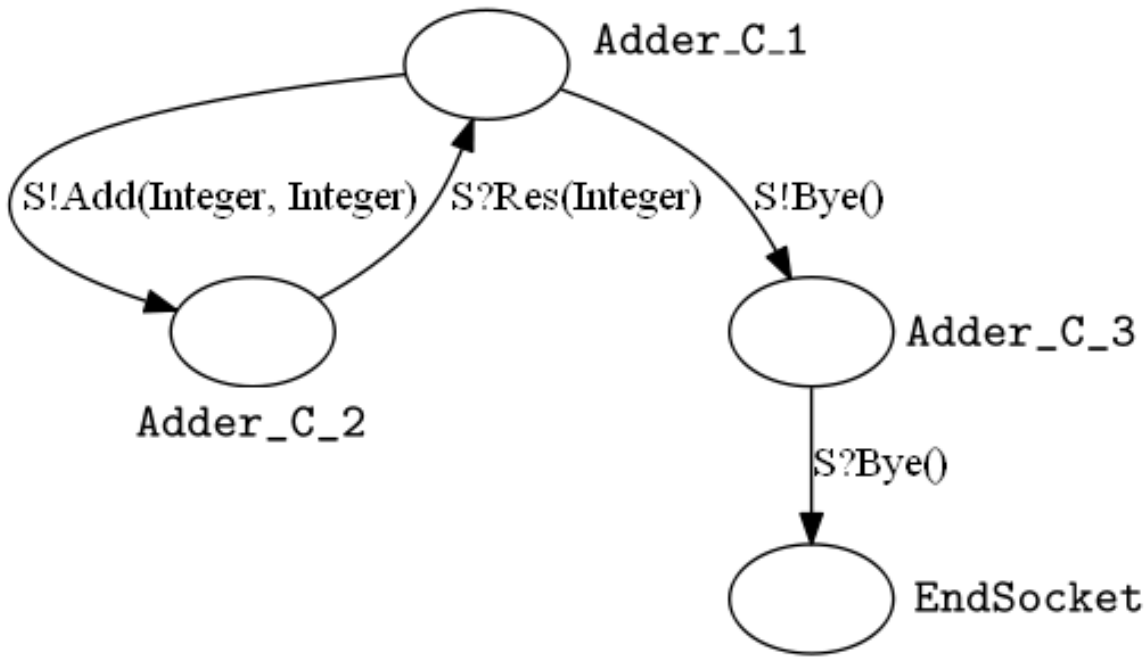
```
// Result: i1.val is the N-th Fib number
EndSocket fibo(Adder_C_1 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fibo(
        c1.send(S, Add, i1.val, i1.val=i2.val)
          .receive(S, Res, i2)
                  )
    :
}
```



Exercise: recursive Fibonacci client

- `fibo(c1, new Buf<Integer>(0), new Buf<Integer>(1), N);`

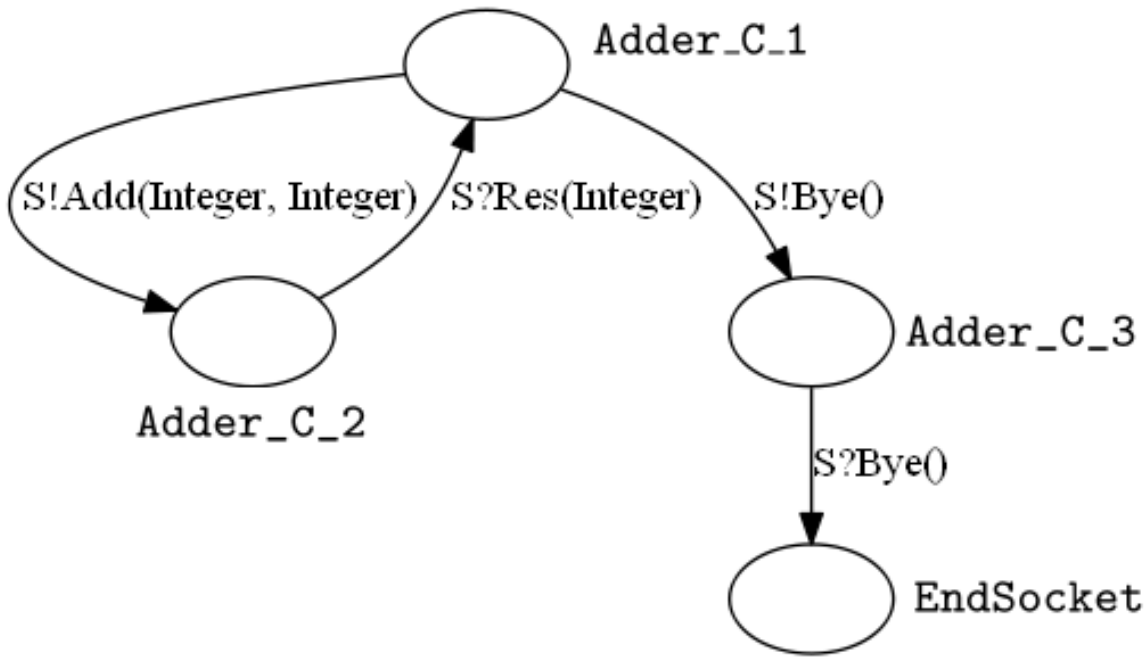
```
// Result: i1.val is the N-th Fib number
EndSocket fibo(Adder_C_1 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fibo(
        c1.send(S, Add, i1.val, i1.val=i2.val)
          .receive(S, Res, i2),
        i1, i2, i-1)
    :
}
```



Exercise: recursive Fibonacci client

- `fibo(c1, new Buf<Integer>(0), new Buf<Integer>(1), N);`

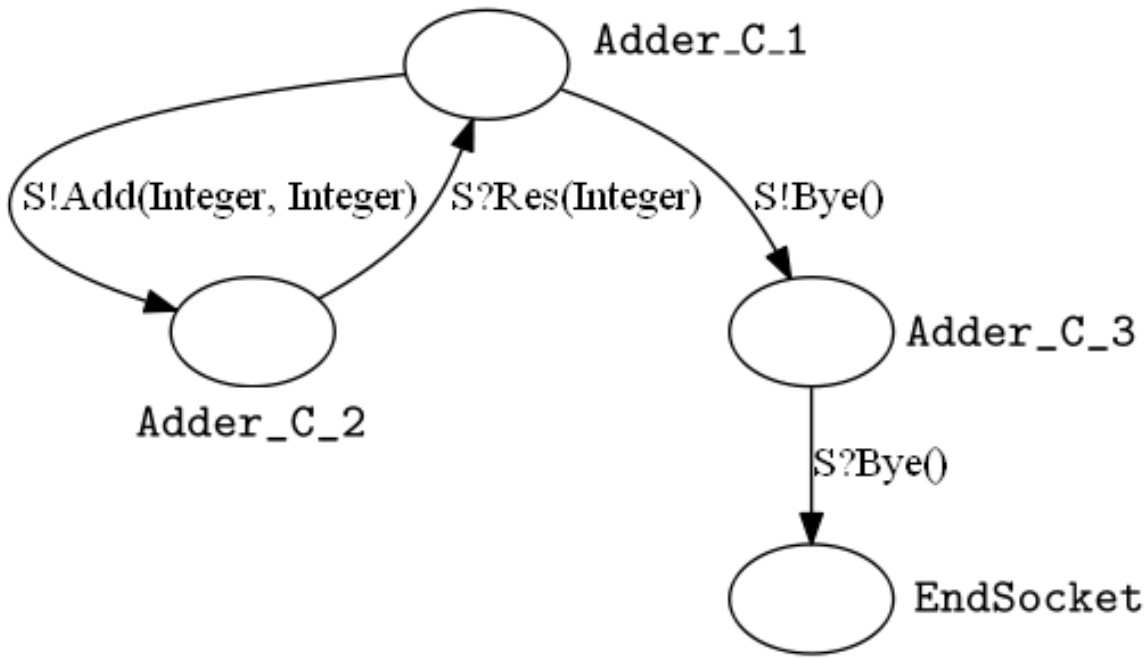
```
// Result: i1.val is the N-th Fib number
EndSocket fibo(Adder_C_1 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fibo(
        c1.send(S, Add, i1.val, i1.val=i2.val)
          .receive(S, Res, i2),
        i1, i2, i-1)
    :
}
```



Exercise: recursive Fibonacci client

- `fibo(c1, new Buf<Integer>(0), new Buf<Integer>(1), N);`

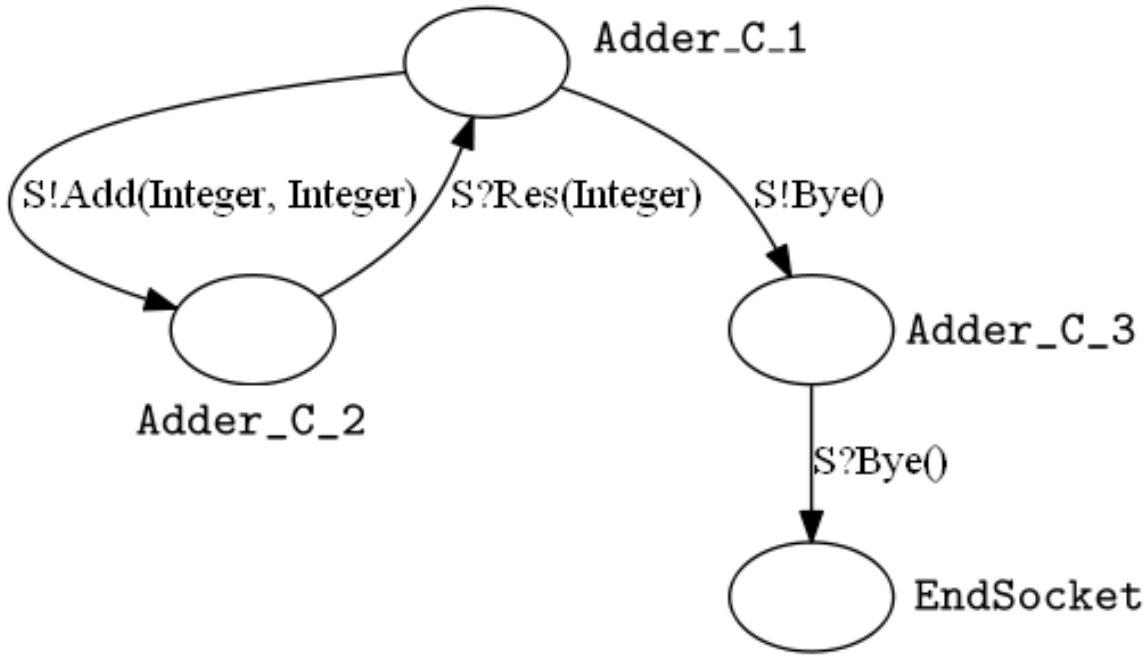
```
// Result: i1.val is the N-th Fib number
EndSocket fibo(Adder_C_1 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fibo(
        c1.send(S, Add, i1.val, i1.val=i2.val)
          .receive(S, Res, i2),
       i1, i2, i-1)
    : c1.send(S, Bye).receive(S, Bye);
}
```



Exercise: recursive Fibonacci client

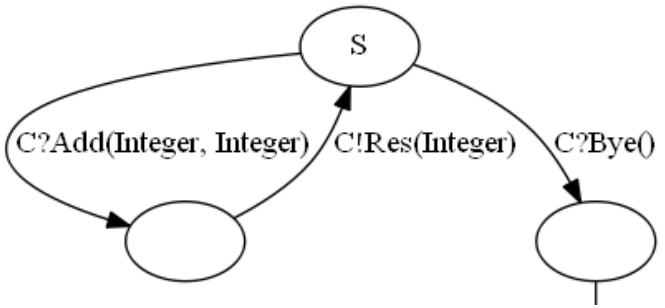
- `fibo(c1, new Buf<Integer>(0), new Buf<Integer>(1), N);`

```
// Result: i1.val is the N-th Fib number
EndSocket fibo(Adder_C_1 c1, Buf<Integer> i1, Buf<Integer> i2, int i)
      throws ... {
  return (i > 0)
    ? fibo(
        c1.send(S, Add, i1.val, i1.val=i2.val)
          .receive(S, Res, i2),
       i1, i2, i-1)
    : c1.send(S, Bye).receive(S, Bye);
}
```



Adder server: 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



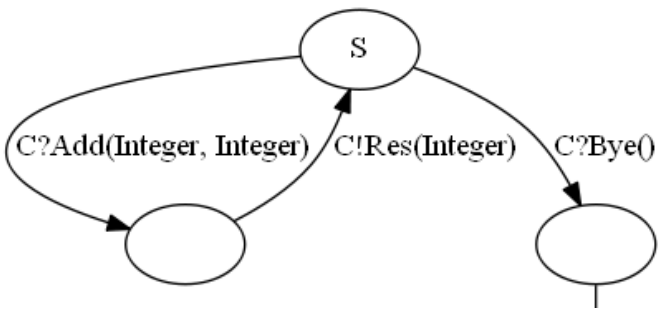
Adder server: session branching

- 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) {
  Adder_S_1_Cases c = s1.branch(C);
  switch (c.op) {
  case Add: s1 = c.receive(Add, i1, i2)
                 .send(C, Res, i1.val+i2.val); break;
  case Bye: c.receive(Bye).send(C, Bye); return;
} }
```

✓ Familiar Java switch (etc.) pattern

× Additional run-time branch continuation "cast"

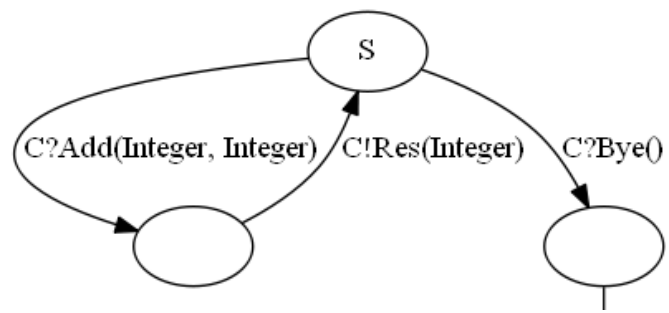


Adder server: session branching

- 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 Server2 implements Adder_S_1_Handler {
  void receive(Adder_S_2 s2, Add op, Buf<Integer> arg1, Buf<Integer> arg2) .. {
    s2.send(C, Res, arg1.val+arg2.val).branch(C, this);
  }
  void receive(Adder_S_3 s3, Bye op, Buf<Integer> arg) throws ... {
    s3.send(C, Bye);
} }
```

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

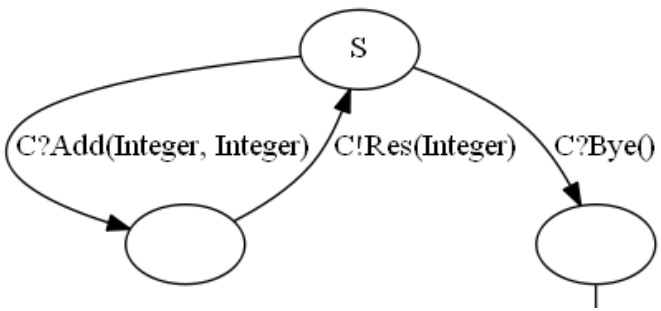


Adder server: session branching

- 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 Server2 implements Adder_S_1_Handler {
  void receive(Adder_S_2 s2, Add op, Buf<Integer> arg1, Buf<Integer> arg2) .. {
    s2.send(C, Res, arg1.val+arg2.val).branch(C, this);
  }
  void receive(Adder_S_3 s3, Bye op, Buf<Integer> arg) throws ... {
    s3.send(C, Bye);
} }
```

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



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] L. Padovani
Scala (CPS, actors)	[ECOOP16] Scalas, Yoshida

SMTP

- Simple Mail Transfer Protocol
  - Internet standard for email transmission (RFC 5321) [SMTP]
  - Rich conversation structure

- `https://github.com/rhu1/scribble-java/tree/rhu1-research/modules/core/src/test/scrib/demo/betty16/lec2/smtp`

[SMTP] `https://tools.ietf.org/html/rfc5321`

APIs for 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`

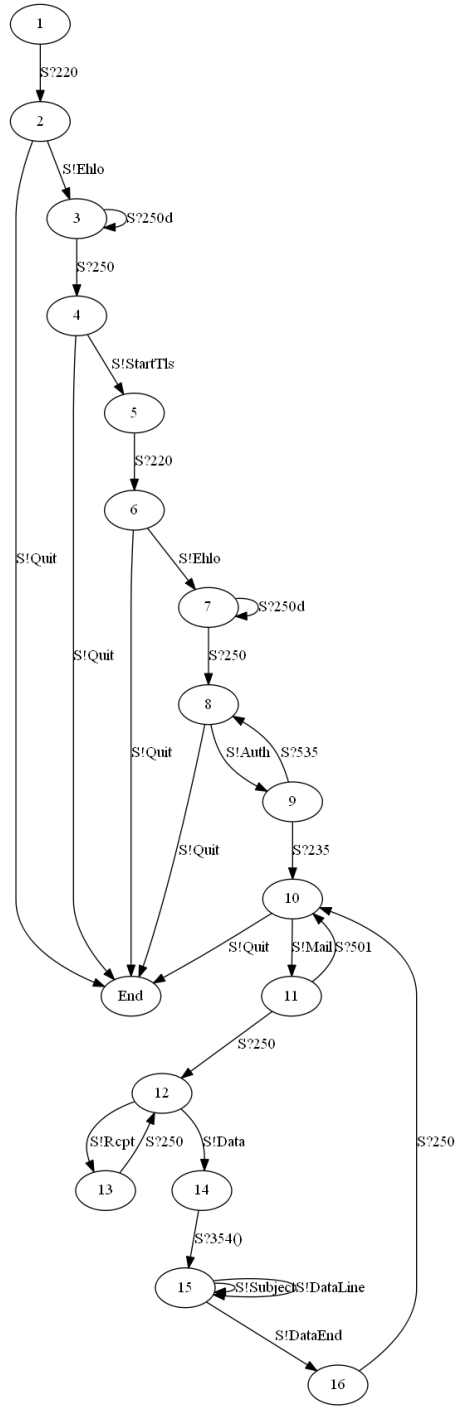
SMTP: global protocol

- "Initiation exchange" done once on plain TCP connection..
  - ..and repeated on secured connection
  - Factor out "subprotocol"

```
global protocol Smtp(role C, role S) {
  220 from S to C;   // "220 smtp2.cc.ic.ac.uk ESMTP Exim 4.85"
  do Init(C, S);
  do StartTls(C, S); // Secure the connection
  do Init(C, S);
  ...                // Main mail exchanges...
}

global protocol Init(role C, role S) {
  Ehlo from C to S;      // "EHLO foo.bar.com"
  rec X {
    choice at S {
      250d from S to C;  // "250-SIZE 26214400", "250-8BITMIME", ...
      continue X;
    } or {
      250 from S to C;   // "250 HELP" (no dash after 250)
} } }
```

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

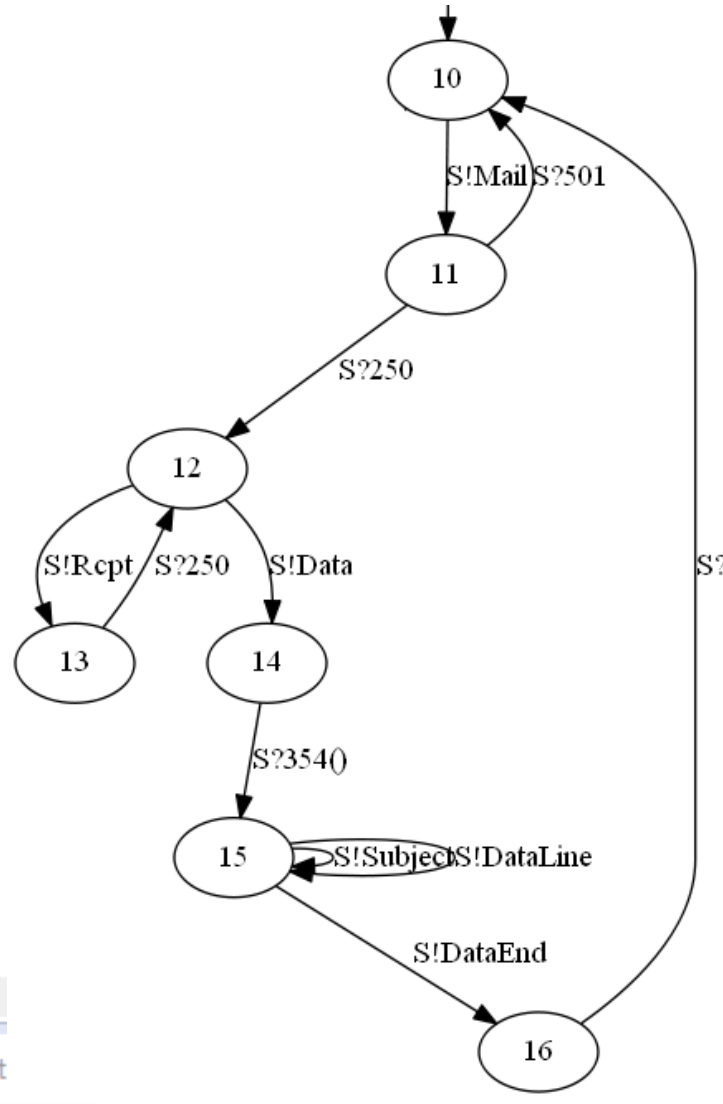
```
83          .send(S, new Mail(mail))
84          .branch(S);
85      switch (cases.getOp())
86      {
87        case _250:
88        {
89          cases.receive(_250)
90            .send(S, new Rcpt(rcpt)).async(S, _250)
91            .send(S, new Data()).async(S, _354)
92            .send(S, new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

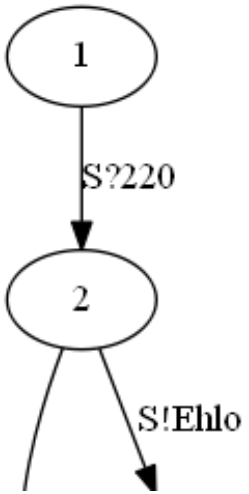
The method receive(S, _250, new Buf<>()) is undefined for the type Smtp_C_15



SMTP: input future generation

- Generation of state-specific (unary) input futures

```
Buf<_220> buf = new Buf<>();

c3 = c1.receive(S, _220, buf)
       .send(S, new Ehlo("..."));
...
```



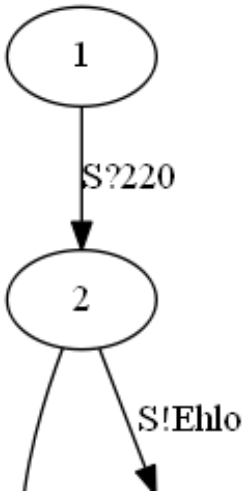
- Safe decoupling of local protocol state transition from message input
  - Non-blocking session input actions, cf. [ECOOP10]
  - "Asynchronous permutation" of I/O actions, cf. [PPDP14]
  - Affine message handling, cf. [FoSSaCS15]

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

SMTP: input future generation

- Generation of state-specific (unary) input futures

```
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
...
```



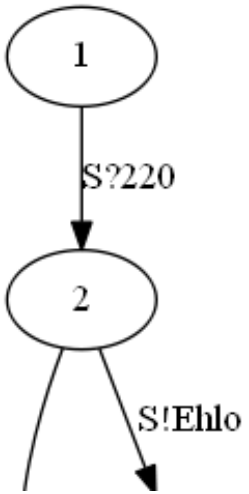
- Safe decoupling of local protocol state transition from message input
  - Non-blocking session input actions, cf. [ECOOP10]
  - "Asynchronous permutation" of I/O actions, cf. [PPDP14]
  - Affine message handling, cf. [FoSSaCS15]

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

SMTP: input future generation

- Generation of state-specific (unary) input futures

```
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
...
```



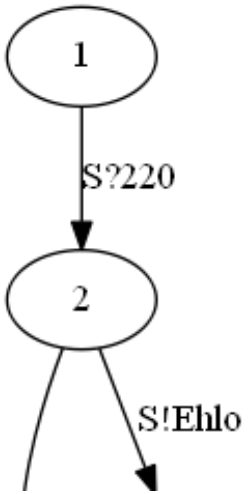
- Safe decoupling of local protocol state transition from message input
  - Non-blocking session input actions, cf. [ECOOP10]
  - "Asynchronous permutation" of I/O actions, cf. [PPDP14]
  - Affine message handling, cf. [FoSSaCS15]

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

SMTP: input future generation

- Generation of state-specific (unary) input futures

```
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
...
```



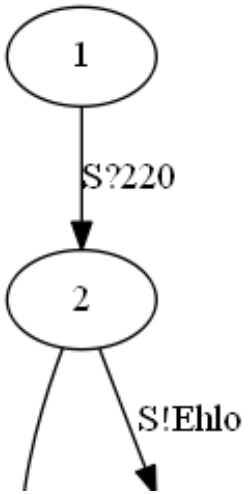
- Safe decoupling of local protocol state transition from message input
  - Non-blocking session input actions, cf. [ECOOP10]
  - "Asynchronous permutation" of I/O actions, cf. [PPDP14]
  - Affine message handling, cf. [FoSSaCS15]

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

SMTP: input future generation

- Generation of state-specific (unary) input futures

```
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
...
```



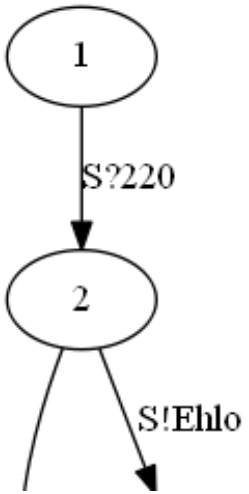
- Safe decoupling of local protocol state transition from message input
  - Non-blocking session input actions, cf. [ECOOP10]
  - "Asynchronous permutation" of I/O actions, cf. [PPDP14]
  - Affine message handling, cf. [FoSSaCS15]

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

SMTP: input future generation

- Generation of state-specific (unary) input futures

```
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
...
```



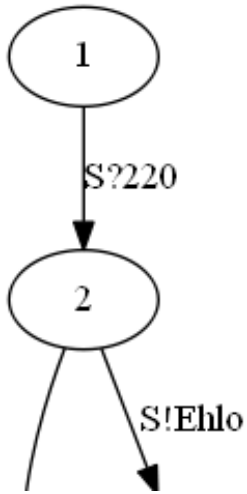
- Safe decoupling of local protocol state transition from message input
  - Non-blocking session input actions, cf. [ECOOP10]
  - "Asynchronous permutation" of I/O actions, cf. [PPDP14]
  - Affine message handling, cf. [FoSSaCS15]

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

SMTP: input future generation

- Generation of state-specific (unary) input futures

```
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
...
```



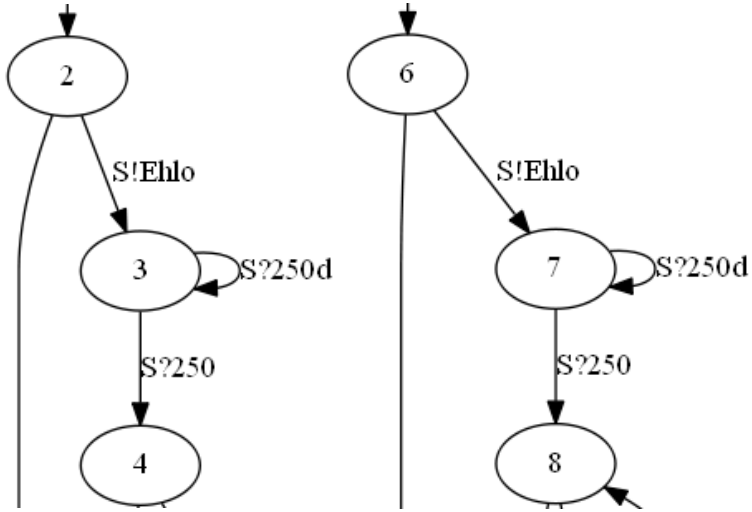
- Safe decoupling of local protocol state transition from message input
  - Non-blocking session input actions, cf. [ECOOP10]
  - "Asynchronous permutation" of I/O actions, cf. [PPDP14]
  - Affine message handling, cf. [FoSSaCS15]

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

SMTP: 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);
  ...;
}
```



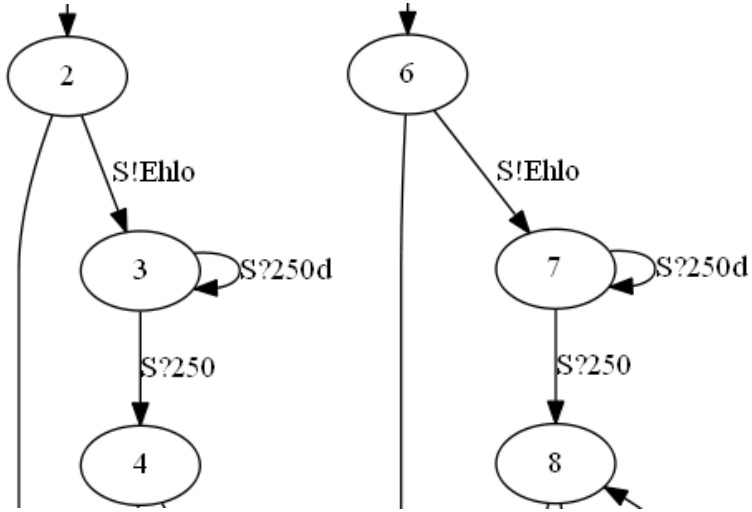
- Basic nominal Java state channel types limit code reuse

```
EndSocket run(Smtp_C_1 c1) throws ...
Smtp_C_4 doInit(Smtp_C_2 c2) throws ...
Smtp_C_6 doStartTls(Smtp_C_4 c4) throws ...
Smtp_C_8 doInit(Smtp_C_6 c2) throws ...
...
```

SMTP: 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);
  ...;
}
```



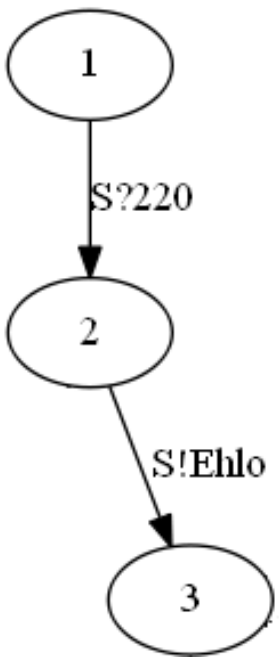
- Basic nominal Java state channel types limit code reuse

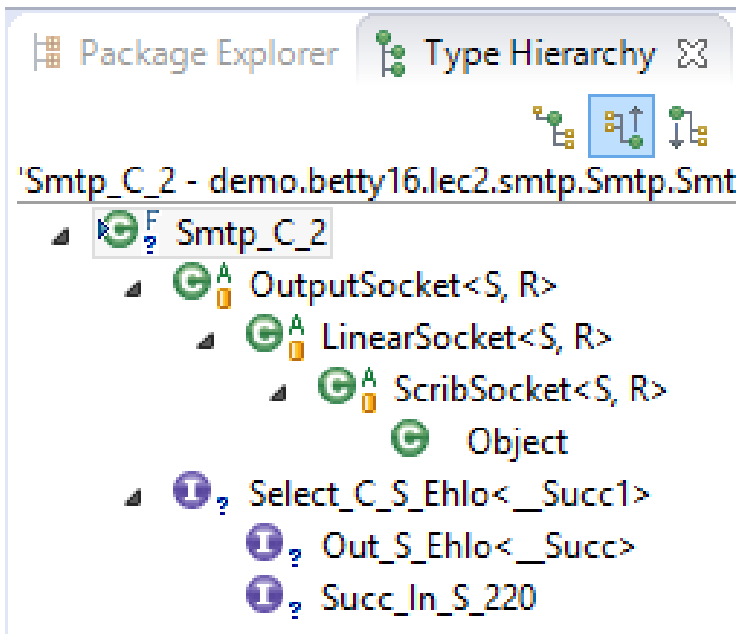
```
EndSocket run(Smtp_C_1 c1) throws ...
Smtp_C_4 doInit(Smtp_C_2 c2) throws ...
Smtp_C_6 doStartTls(Smtp_C_4 c4) throws ...
Smtp_C_8 doInit(Smtp_C_6 c2) throws ...
...
```

SMTP: I/O state interfaces

```
class Smtp_C_2 extends OutputSocket<Smtp, C>
               implements Select_C_S_Ehlo<Smtp_C_3>
```

- (Unary) select interface: `Select_C_S_Ehlo<Smtp_C_3>`
  - Read as: " `C!{ S(Ehlo): Smtp_C_3 }` "
  - Actions parametrised on successors: `Out_S_Ehlo<__Succ>`
    i.e. " `S!Ehlo . __Succ` "
  - Concrete successor: `Smtp_C_3`
- Successor interface: `Succ_In_S_220`





Exercise: refactor SMTP client `doInit` method

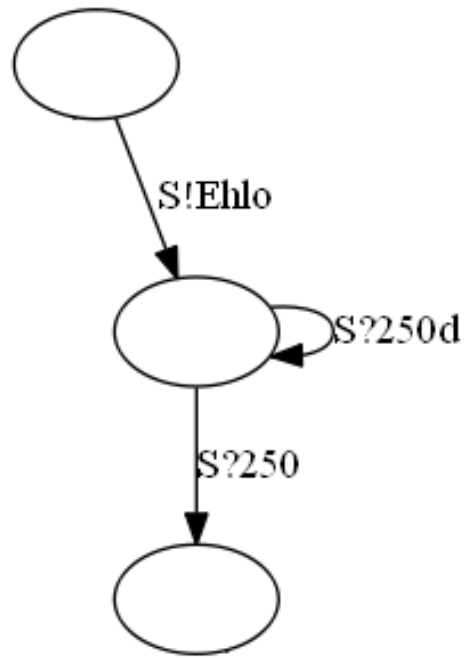
```
EndSocket run(Smtp_C_1 c1) throws ... {
  return
    doInit(
        doStartTls(
            doInit(c1.async(S, _220)))
    )
    .send(S, new Quit());
}
```

- I/O state interfaces built behind the scenes for concrete state channels
  - But may also be used directly..
  - Exercise: factor out a single `doInit` method to be used as above

```
?? doInit(?? c) throws ...
Smtp_C_6 doStartTls(Smtp_C_4 c4) throws ...
```

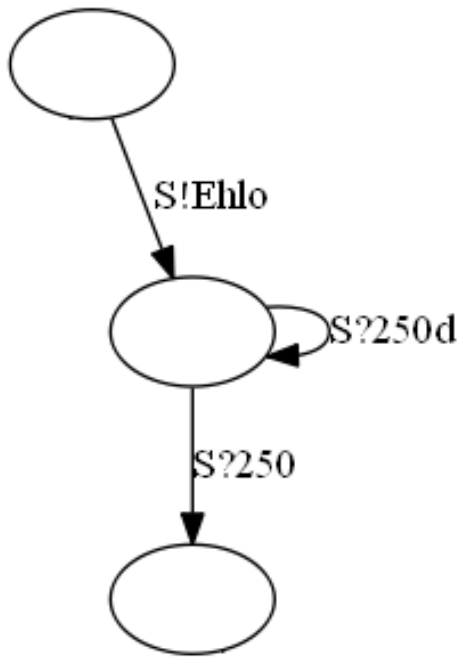
Exercise: refactor SMTP client `doInit` method

```
// S!Ehlo.?? -> Succ(S?250)
Succ_In_S_250 doInit(Select_C_S_Ehlo<? > c) throws ... {




}
```



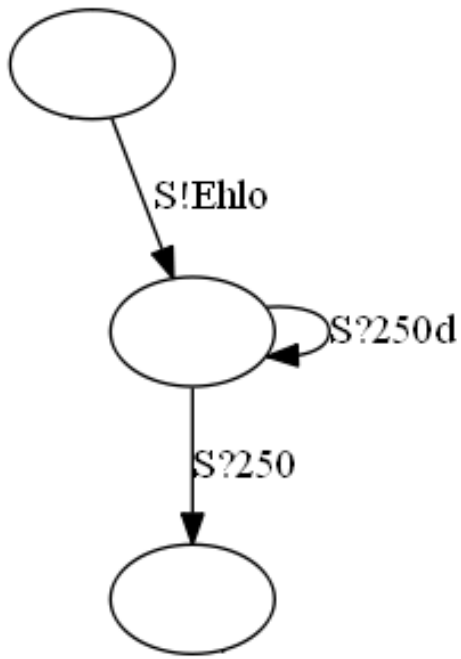
Exercise: refactor SMTP client `doInit` method

```
// S!Ehlo.?? -> Succ(S?250)
Succ_In_S_250 doInit(Select_C_S_Ehlo<? > c) throws ... {
        c.send(S, new Ehlo("..."))




}
```



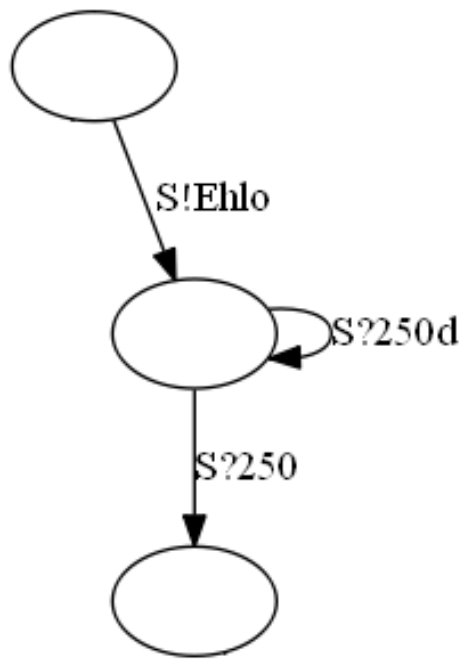
Exercise: refactor SMTP client `doInit` method

```
<
  T1 extends Branch_C_S_250__S_250d<? , ? > // T1 = S?{ 250: ??, 250d: ?? }

>
// S!Ehlo.T1 -> Succ(S?250)
Succ_In_S_250 doInit(Select_C_S_Ehlo<T1> c) throws ... {
  T1 b = c.send(S, new Ehlo("..."))









}
```



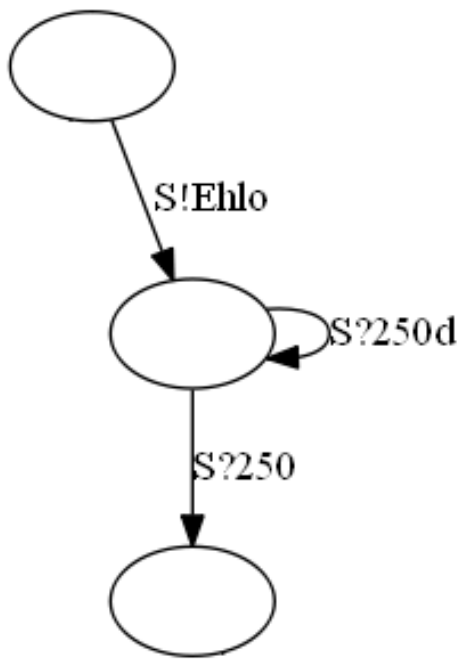
Exercise: refactor SMTP client `doInit` method

```
<
  T1 extends Branch_C_S_250__S_250d<? , ? > // T1 = S?{ 250: ??, 250d: ?? }

>
// S!Ehlo.T1 -> Succ(S?250)
Succ_In_S_250 doInit(Select_C_S_Ehlo<T1> c) throws ... {
  T1 b = c.send(S, new Ehlo("..."))

    Case_C_S_250__S_250d<? , ? > cases = b.branch(S);
    switch (cases.getOp()) {
      case _250d:     cases.receive(_250d);
      case _250:        cases.receive(_250);
    }

}
```



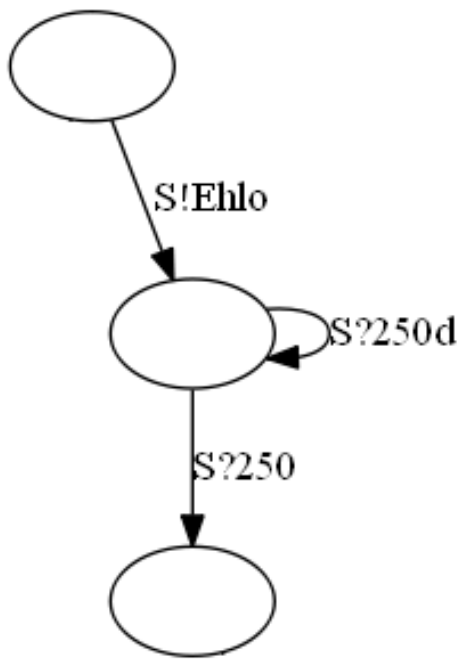
Exercise: refactor SMTP client `doInit` method

```
<
  T1 extends Branch_C_S_250__S_250d<? , ? > // T1 = S?{ 250: ??, 250d: ?? }

>
// S!Ehlo.T1 -> Succ(S?250)
Succ_In_S_250 doInit(Select_C_S_Ehlo<T1> c) throws ... {
  T1 b = c.send(S, new Ehlo("..."))
  while (true) {
    Case_C_S_250__S_250d<? , ? > cases = b.branch(S);
    switch (cases.getOp()) {
      case _250d:    cases.receive(_250d);
      case _250:        cases.receive(_250);
    }
  }
}
```



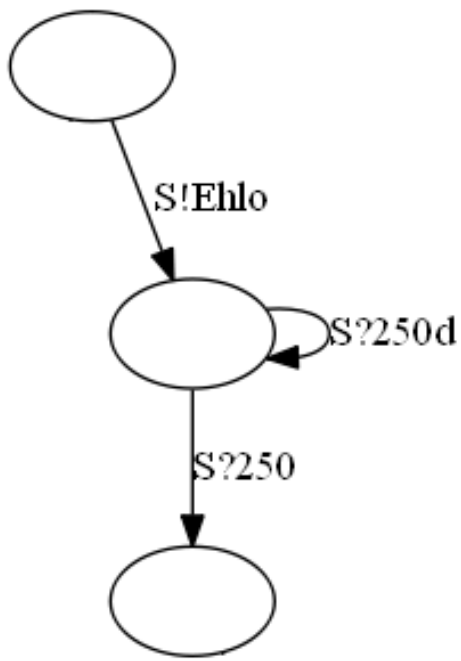
Exercise: refactor SMTP client `doInit` method

```
<
  T1 extends Branch_C_S_250__S_250d<? , T1> // T1 = S?{ 250: ??, 250d: T1 }

>
// S!Ehlo.T1 -> Succ(S?250)
Succ_In_S_250 doInit(Select_C_S_Ehlo<T1> c) throws ... {
  T1 b = c.send(S, new Ehlo("..."))
  while (true) {
    Case_C_S_250__S_250d<? , T1> cases = b.branch(S);
    switch (cases.getOp()) {
      case _250d: b = cases.receive(_250d); break;
      case _250:        cases.receive(_250);
    }
  }
}
```



Exercise: refactor SMTP client `doInit` method

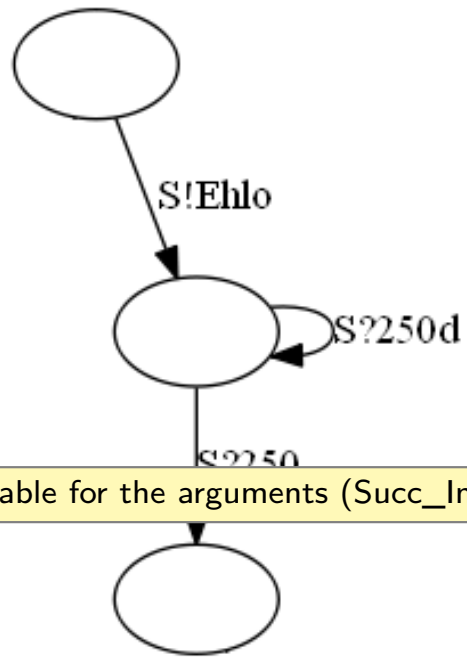
```
<
  T1 extends Branch_C_S_250__S_250d<? , T1> // T1 = S?{ 250: ??, 250d: T1 }

>
// S!Ehlo.T1 -> Succ(S?250)
Succ_In_S_250 doInit(Select_C_S_Ehlo<T1> c) throws ... {
  T1 b = c.send(S, new Ehlo("..."))
  while (true) {
    Case_C_S_250__S_250d<? , T1> cases = b.branch(S);
    switch (cases.getOp()) {
      case _250d: b = cases.receive(_250d); break;
      case _250:  return cases.receive(_250);
    }
  }
}
```



Exercise: refactor SMTP client doInit method

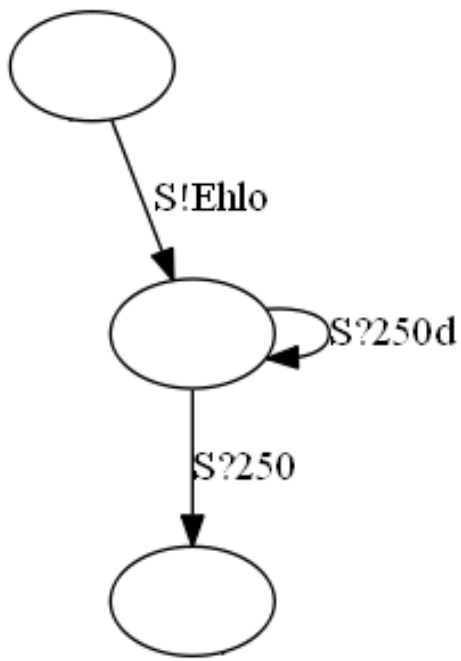
```
<
  T1 extends Branch_C_S_250__S_250d<? , T1> // T1 = S?{ 250: ??, 250d: T1 }

>
// S!Ehlo.T1 -> Succ(S?250)
Succ_In_S_250 doInit(Select_C_S_Ehlo<T1> c) throws ... {
  T1 b = c.send(S, new Ehlo("..."))
  while (true) {
    Case_C_S_250__S_250d<? , T1> cases = b.branch(S);
    switch (cases.getOp()) {
      case _250d: b = cases.receive(_250d); break;
      case _250:  return cases.receive(_250);
    }
  }
}

EndSocket run(Smtp_C_1 c1) throws ... {
  return
    doInit(
        doStartTls(  // Smtp_C_4 -> Smtp_C_6
            doIn
    )
    .send(S, new Quit());
```

The method doStartTls(Smtp_C_4) .. is not applicable for the arguments (Succ_In_S_250)



Exercise: refactor SMTP client `doInit` method

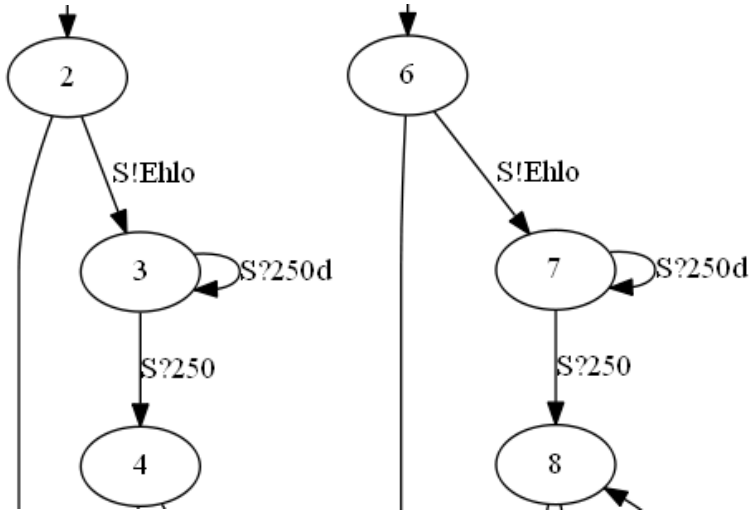
```
<
  T1 extends Branch_C_S_250__S_250d<T2, T1>, // T1 = S?{ 250: T2, 250d: T1 }
  T2 extends Succ_In_S_250                   // T2 = Succ(S?250)
>
// S!Ehlo.T1 -> T2
T2            doInit(Select_C_S_Ehlo<T1> c) throws ... {
  T1 b = c.send(S, new Ehlo("..."));
  while (true) {
    Case_C_S_250__S_250d<T2, T1> cases = b.branch(S)
    switch (cases.getOp()) {
      case _250d: b = cases.receive(_250d); break;
      case _250:  return cases.receive(_250);
    }
  }
}
```



SMTP: 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, subtyping, some (generics) inference

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

```
doInit(
    doStartTls(
        doInit(c1.async(S, _220)))
)
.send(S,
```

<Smtp_C_3, Smtp_C_4> Smtp_C_4 doInit(Select_C_S_Ehlo<Smtp_C_3> c) throws Exception

Session subtyping via I/O interface subtyping

- E.g. add a case in the global protocol

```
global protocol Smtp(role C, role S) {
  220 from S to C;

    do Init(C, S);
    do StartTls(C, S);
    do Init(C, S);
    ...
```

Session subtyping via I/O interface subtyping

- E.g. add a case in the global protocol

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

Session subtyping via I/O interface subtyping

- E.g. add a case in the global protocol

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

  - A previous implementation of `S` (branch) needs the new case..

Session subtyping via I/O interface subtyping

- E.g. add a case in the global protocol

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

  - A previous implementation of `S` (branch) needs the new case..
  - ..but our previous code of `C` should be fine (select subtyping)

    [Acta05] Gay, Hole

Session subtyping via I/O interface subtyping

- E.g. add a case in the global protocol

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

  - A previous implementation of `S` (branch) needs the new case..
  - ..but our previous code of `C` should be fine (select subtyping)

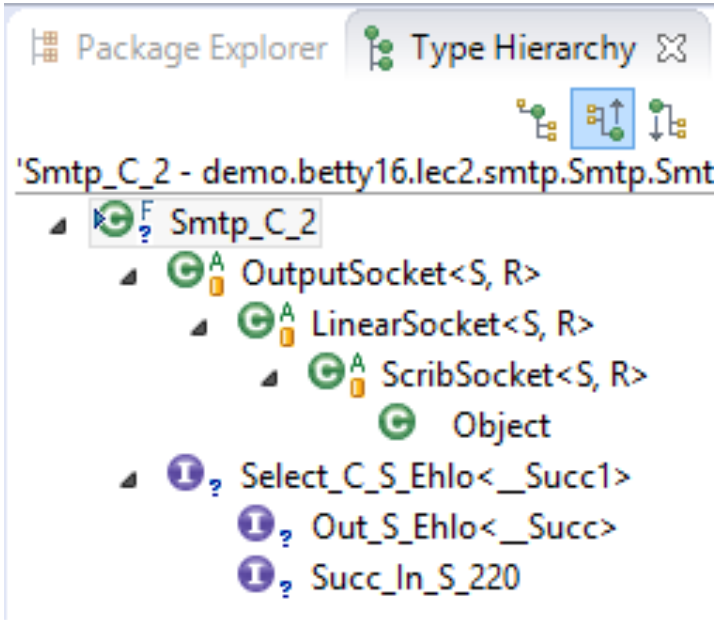
    [Acta05] Gay, Hole

    - Endpoint API generation reflects session subtyping in the Java subtype hierarchy for I/O state interfaces

Session subtyping via I/O interface subtyping

- E.g. add a case in the global protocol

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



  - A previous implementation of `S` (branch) needs the new case..
  - ..but our previous code of `C` should be fine (select subtyping)

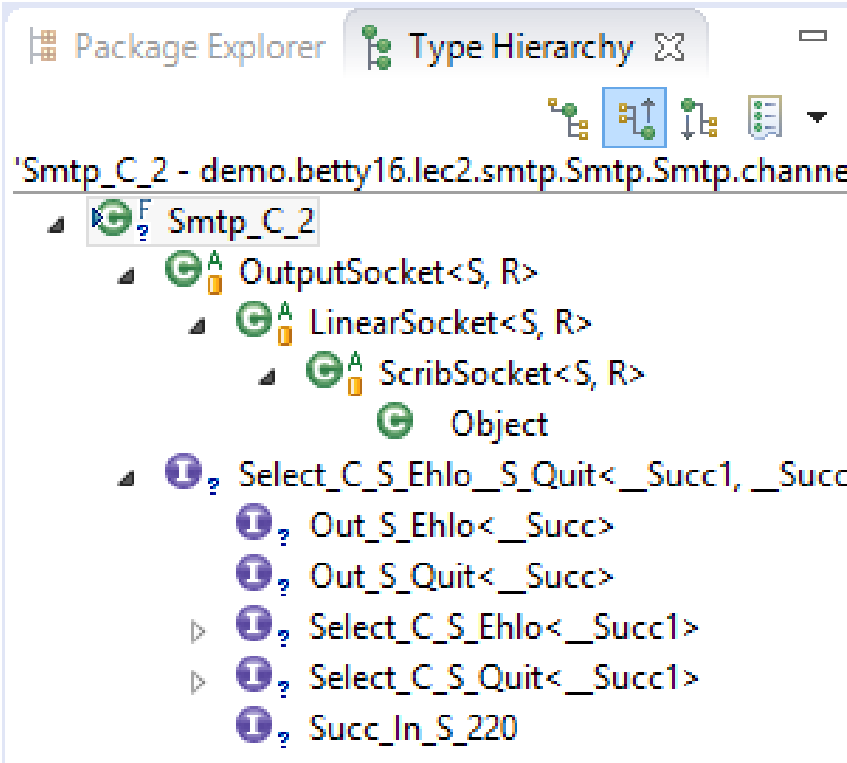
    [Acta05] Gay, Hole

    - Endpoint API generation reflects session subtyping in the Java subtype hierarchy for I/O state interfaces

Session subtyping via I/O interface subtyping

- E.g. add a case in the global protocol

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



  - A previous implementation of `S` (branch) needs the new case..
  - ..but our previous code of `C` should be fine (select subtyping)

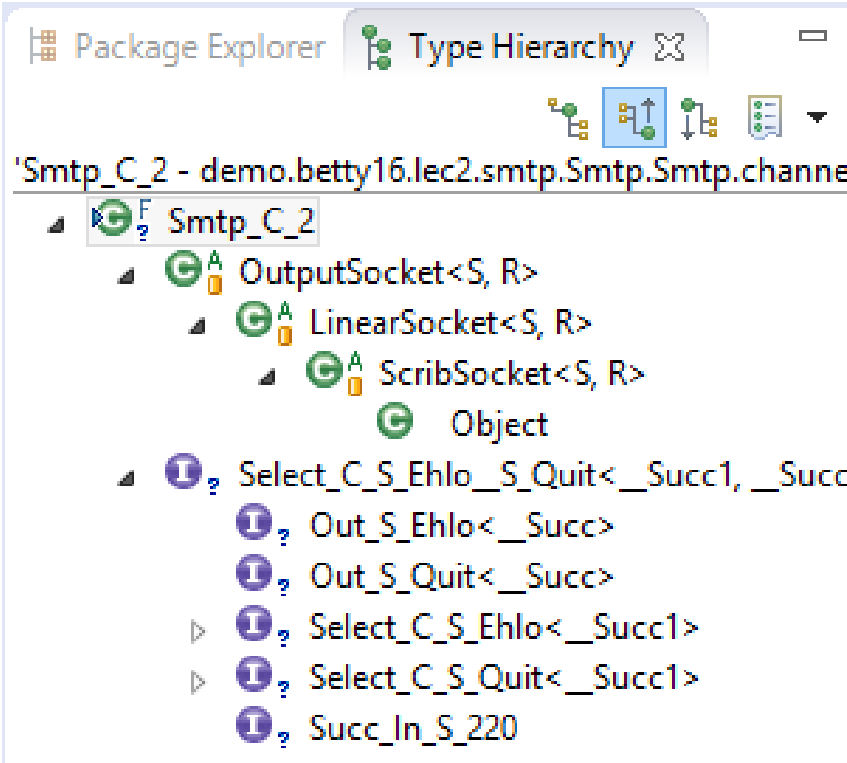
    [Acta05] Gay, Hole

    - Endpoint API generation reflects session subtyping in the Java subtype hierarchy for I/O state interfaces

Session subtyping via I/O interface subtyping

- E.g. add a case in the global protocol

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




```
doInit(
    doStartTls(
        doInit(c1.async(S, _220)))
)
.send(S,
```

<Smtp_C_3, Smtp_C_4> Smtp_C_4 doInit(Select_C_S_Ehlo<Smtp_C_3> c) throws Exception

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.

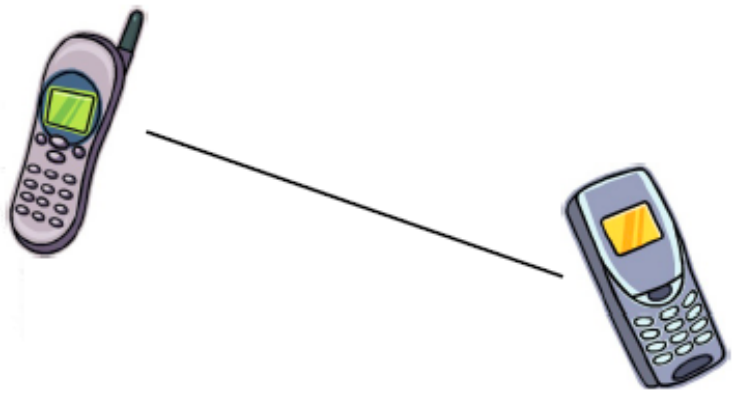
Static type checking backed up by dynamic typecasts/coercions.

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

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

Implementing session delegation

- Type safe connection dynamics
- Transparent to the "passive party"



$$s[\mathrm{p}]!\langle\langle \mathrm{q}, s'[\mathrm{p}']\rangle\rangle.P \mid s : h \longrightarrow P \mid s :$$

$$s[\mathrm{p}]?((\mathrm{q}, y)).P \mid s : (\mathrm{q}, \mathrm{p}, s'[\mathrm{p}']) \cdot h \longrightarrow$$

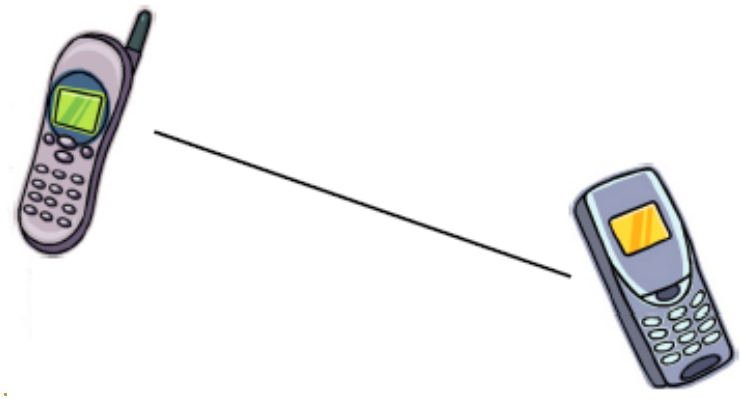
- Asynchrony modelled by decoupling input/output via (global) queue
  - All messages "rerouted" in transit

Implementing session delegation

- Type safe connection dynamics
- Transparent to the "passive party"

$$s[\mathrm{p}]!\langle\langle \mathrm{q}, s'[\mathrm{p}']\rangle\rangle.P \mid s : h \longrightarrow P \mid s :$$

$$s[\mathrm{p}]?((\mathrm{q}, y)).P \mid s : (\mathrm{q}, \mathrm{p}, s'[\mathrm{p}']) \cdot h \longrightarrow$$



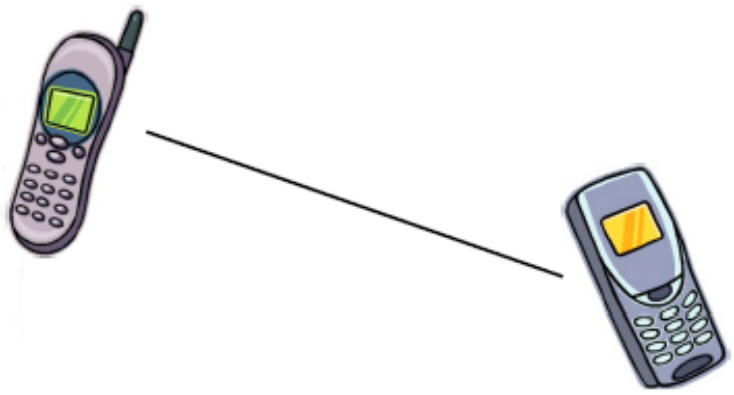
- Asynchrony modelled by decoupling input/output via (global) queue
  - All messages "rerouted" in transit

Implementing session delegation

- Type safe connection dynamics
- Transparent to the "passive party"

$$s[\mathrm{p}]!\langle\langle \mathrm{q}, s'[\mathrm{p}']\rangle\rangle.P \mid s : h \longrightarrow P \mid s :$$

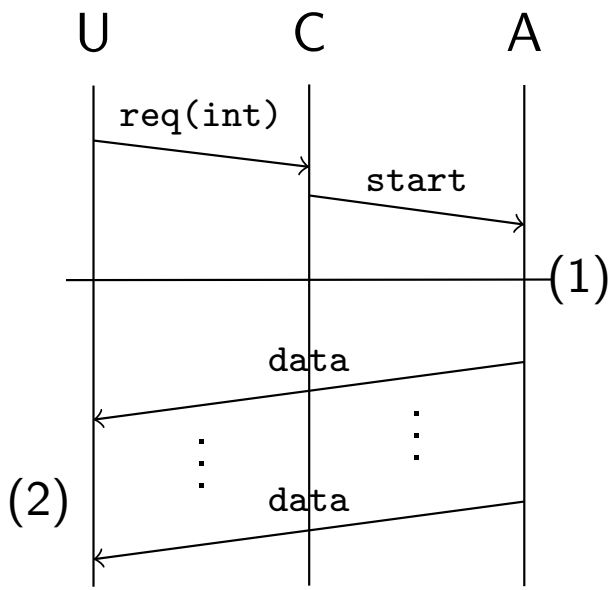
$$s[\mathrm{p}]?((\mathrm{q}, y)).P \mid s : (\mathrm{q}, \mathrm{p}, s'[\mathrm{p}']) \cdot h \longrightarrow$$

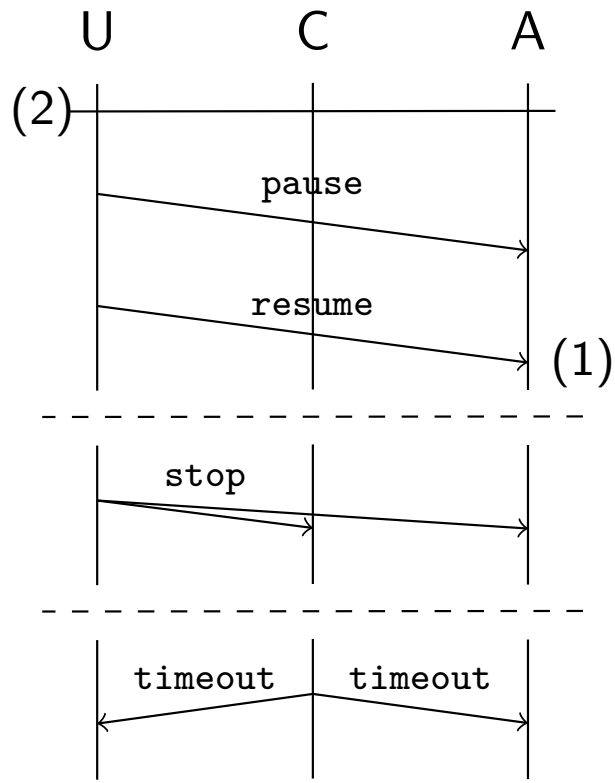


- Asynchrony modelled by decoupling input/output via (global) queue
  - All messages "rerouted" in transit

OOI Resource Usage Control (interruptible)

- **U**ser, Resource **C**ontroller, Instrument **A**gent
- **U** registers with **C** to use a resource (instrument) via **A** for a specified duration (or another metric)



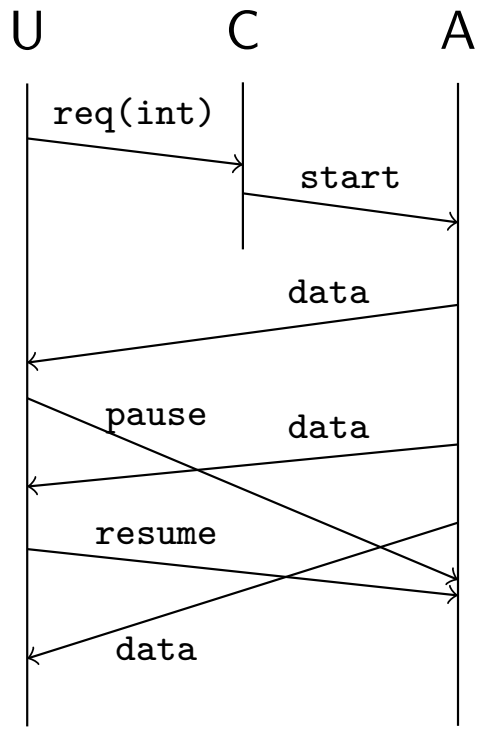


- `https://confluence.oceanobservatories.org/display/CIDev/Resource+Control+in+Scribble`

Extending MPST with interrupts

- Well-formed global types traditionally rule out any protocol control flow ambiguities between roles
  - Sent messages are expected by receiver and vice versa
  - No messages lost or redundant

- Asynchronous interrupts: inherent "communication races"
  - Interruptible is a mixed choice
    And optional
  - Concurrent and nested interrupts
  - Asynchronous entry/exit of interruptible blocks by roles

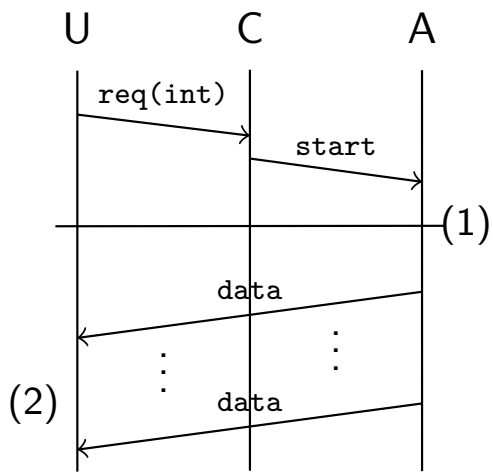


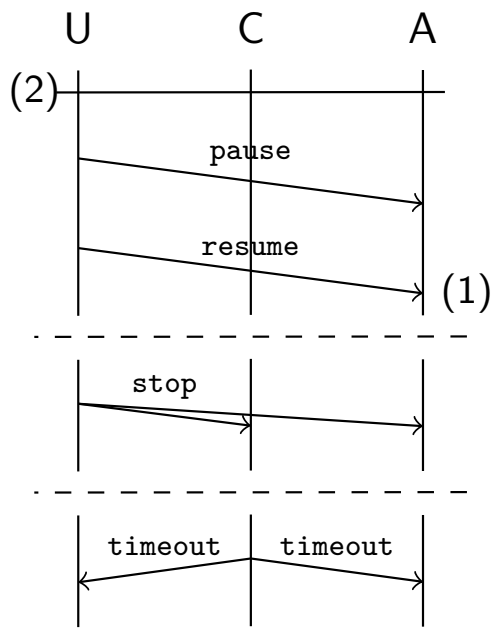
A valid trace

[FMSD15] *Practical interruptible conversations*. Demangeon, Honda, Hu, Neykova, Yoshida.

Resource Control in Scribble

```
// U = User, C = Controller, A = Instrument Agent
global protocol ResourceControl(role U, role C, role A) {
  req(int) from U to C;
  start() from C to A;
  interruptible {
    rec X {
      interruptible {
        rec Y {
          data() from A to U;
          continue Y;
        }
      } with {
        pause() by U;
      }
      resume() from U to A;
      continue X;
    }
  } with {
    stop() by U;
    timeout() by C;
  }
}
```

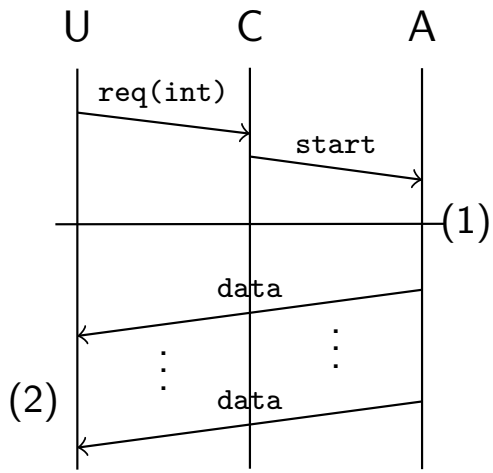


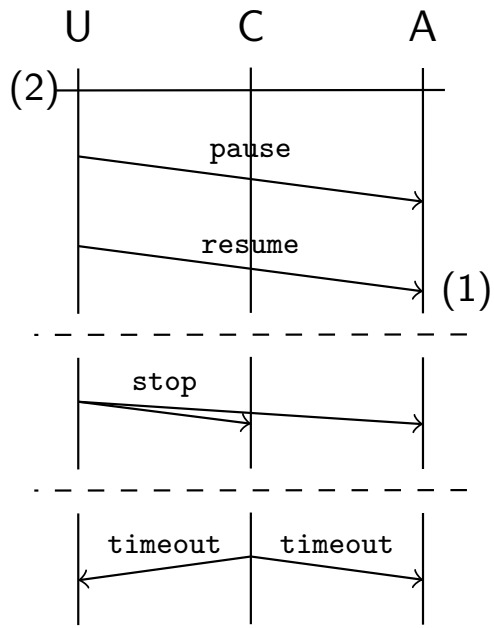


Resource Control in Scribble

```
// U = User, C = Controller, A = Instrument Agent
global protocol ResourceControl(role U, role C, role A) {
  req(int) from U to C;
  start() from C to A;
  interruptible {
    rec X {
      interruptible {
        rec Y {
          data() from A to U;
          continue Y;
        }
      } with {
        pause() by U;
      }
      resume() from U to A;
      continue X;
    }
  } with {
    stop() by U;
    timeout() by C;
  }
}
```

- Continuous stream by `A`..

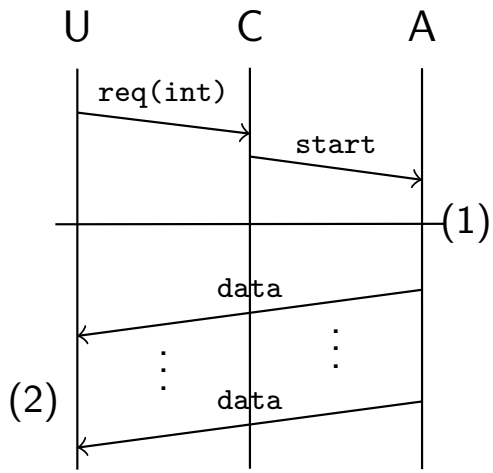


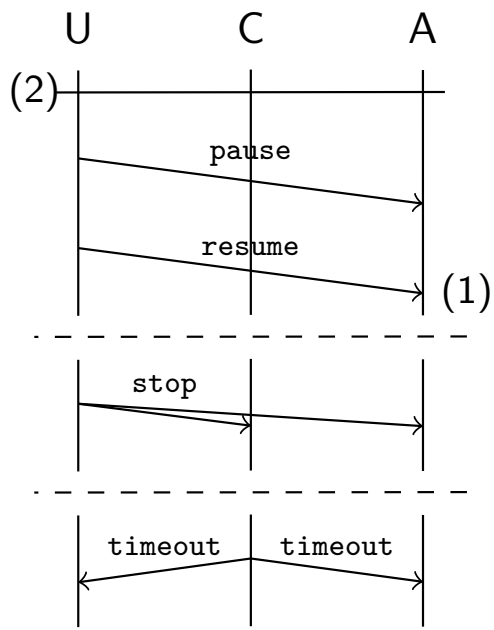


Resource Control in Scribble

```
// U = User, C = Controller, A = Instrument Agent
global protocol ResourceControl(role U, role C, role A) {
  req(int) from U to C;
  start() from C to A;
  interruptible {
    rec X {
      interruptible {
        rec Y {
          data() from A to U;
          continue Y;
        }
      } with {
        pause() by U;
      }
      resume() from U to A;
      continue X;
    }
  } with {
    stop() by U;
    timeout() by C;
  }
}
```

- Continuous stream by A..
  - ..interrupted by U

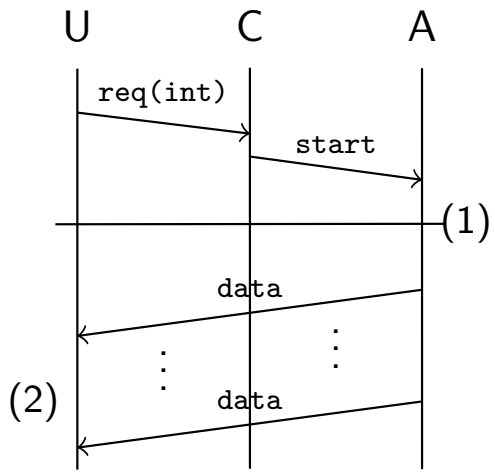


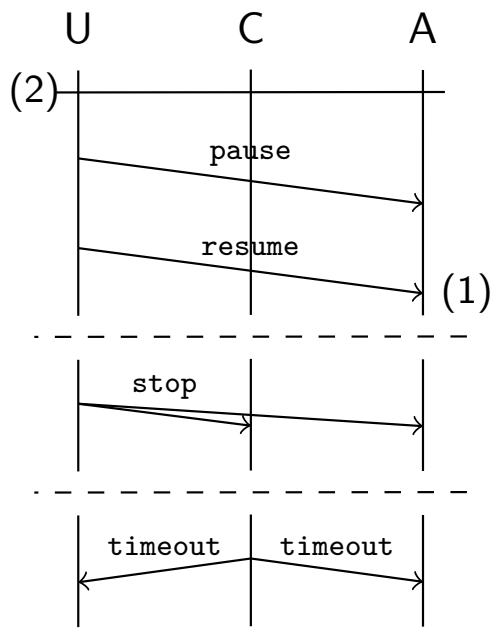


Resource Control in Scribble

```
// U = User, C = Controller, A = Instrument Agent
global protocol ResourceControl(role U, role C, role A) {
  req(int) from U to C;
  start() from C to A;
  interruptible {
    rec X {
      interruptible {
        rec Y {
          data() from A to U;
          continue Y;
        }
      } with {
        pause() by U;
      }
      resume() from U to A;
      continue X;
    }
  } with {
    stop() by U;
    timeout() by C;
  }
}
```

- Continuous stream by `A`..
  - ..interrupted by `U`

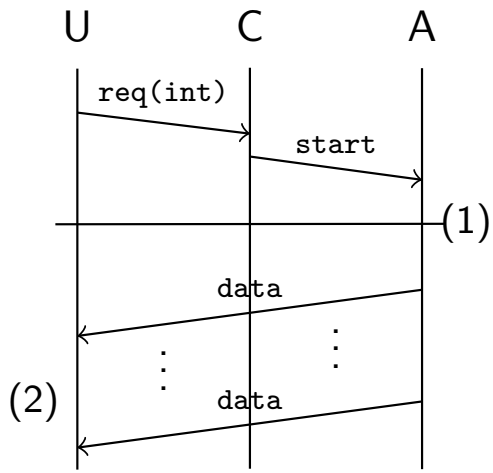


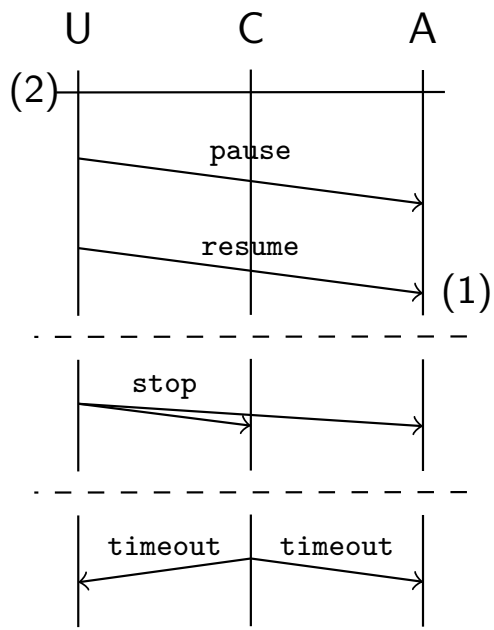


Resource Control in Scribble

```
// U = User, C = Controller, A = Instrument Agent
global protocol ResourceControl(role U, role C, role A) {
  req(int) from U to C;
  start() from C to A;
  interruptible {
    rec X {
      interruptible {
        rec Y {
          data() from A to U;
          continue Y;
        }
      } with {
        pause() by U;
      }
      resume() from U to A;
      continue X;
    }
  } with {
    stop() by U;
    timeout() by C;
  }
}
```

- Continuous stream by A..
  - ..interrupted by U and C

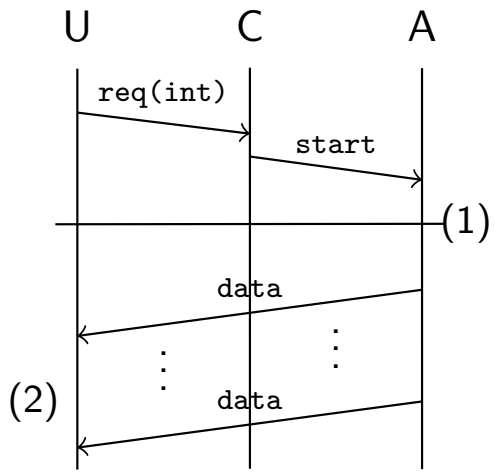


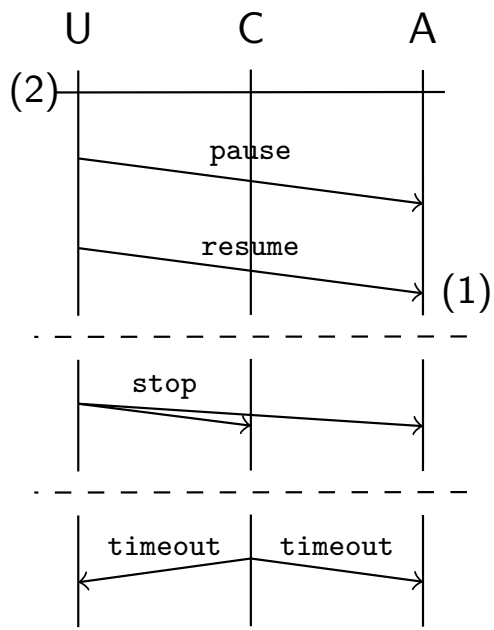


Resource Control in Scribble

```
// U = User, C = Controller, A = Instrument Agent
global protocol ResourceControl(role U, role C, role A) {
  req(int) from U to C;
  start() from C to A;
  interruptible {
    rec X {
      interruptible {
        rec Y {
          data() from A to U;
          continue Y;
        }
      } with {
        pause() by U;
      }
      resume() from U to A;
      continue X;
    }
  } with {
    stop() by U;
    timeout() by C;
  }
}
```

- Continuous stream by A..
  - ..interrupted by U and C

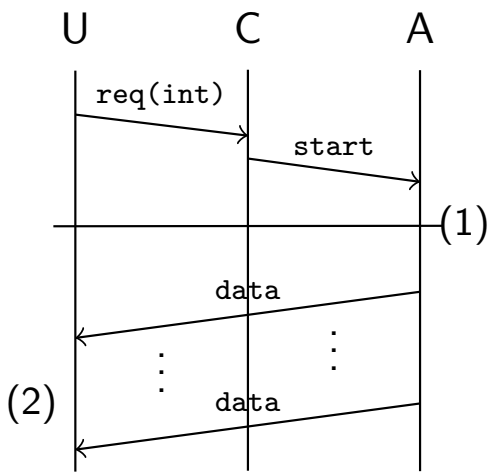


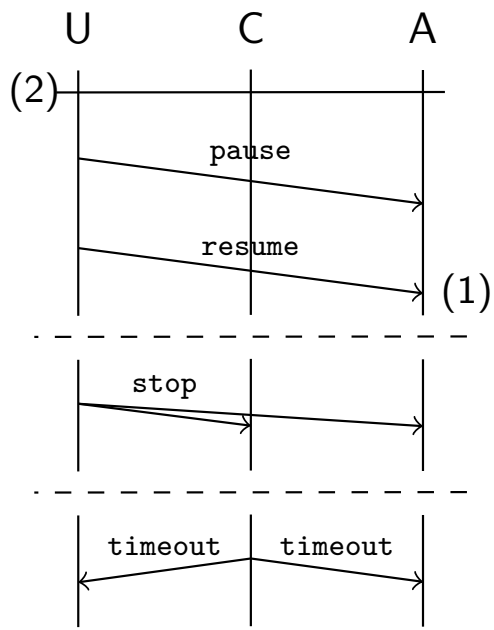


Resource Control in Scribble

```
// U = User, C = Controller, A = Instrument Agent
global protocol ResourceControl(role U, role C, role A) {
  req(int) from U to C;
  start() from C to A;
  interruptible _1 {
    rec X {
      interruptible _2 {
        rec Y {
          data() from A to U;
          continue Y;
        }
      } with {
        pause() by U;
      }
      resume() from U to A;
      continue X;
    }
  } with {
    stop() by U;
    timeout() by C;
  }
}
```

- Session "scopes" reified
- Messages carry scope information
- Endpoints use scopes to independently correct control flow discrepancies

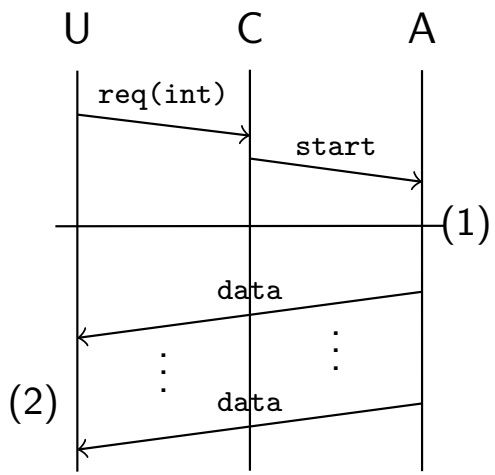


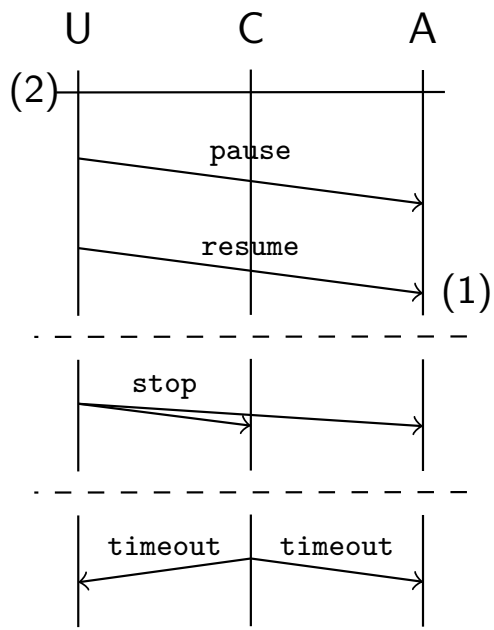


Resource Control in Scribble

```
// U = User, C = Controller, A = Instrument Agent
global protocol ResourceControl(role U, role C, role A) {
  req(int) from U to C;
  start() from C to A;
  interruptible _1 {
    rec X {
      interruptible _2 {
        rec Y {
          data() from A to U;
          continue Y;
        }
      } with {
        pause() by U;
      }
      resume() from U to A;
      continue X;
    }
  } with {
    stop() by U;
    timeout() by C;
  }
}
```

- Session "scopes" reified
- Messages carry scope information
- Endpoints use scopes to independently correct control flow discrepancies

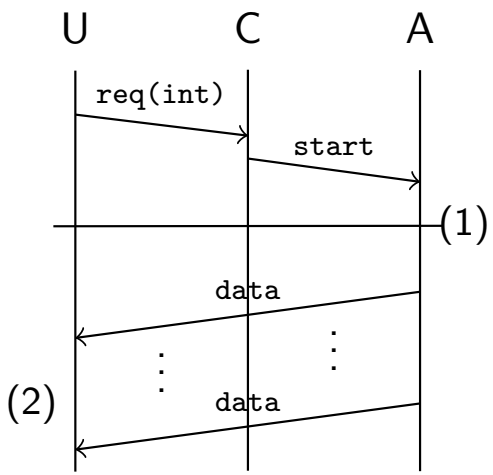


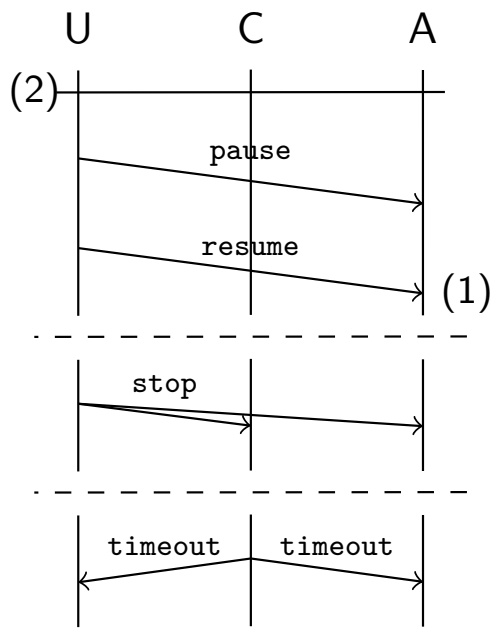


Resource Control in Scribble

```
// U = User, C = Controller, A = Instrument Agent
global protocol ResourceControl(role U, role C, role A) {
  req(int) from U to C;
  start() from C to A;
  interruptible _1 {
    rec X {
      interruptible _2 {
        rec Y {
          data() from A to U;
          continue Y;
        }
      } with {
        pause() by U;
      }
      resume() from U to A;
      continue X;
    }
  } with {
    stop() by U;
    timeout() by C;
  }
}
```

- Session "scopes" reified
- Messages carry scope information
- Endpoints use scopes to independently correct control flow discrepancies





Resource Control: User

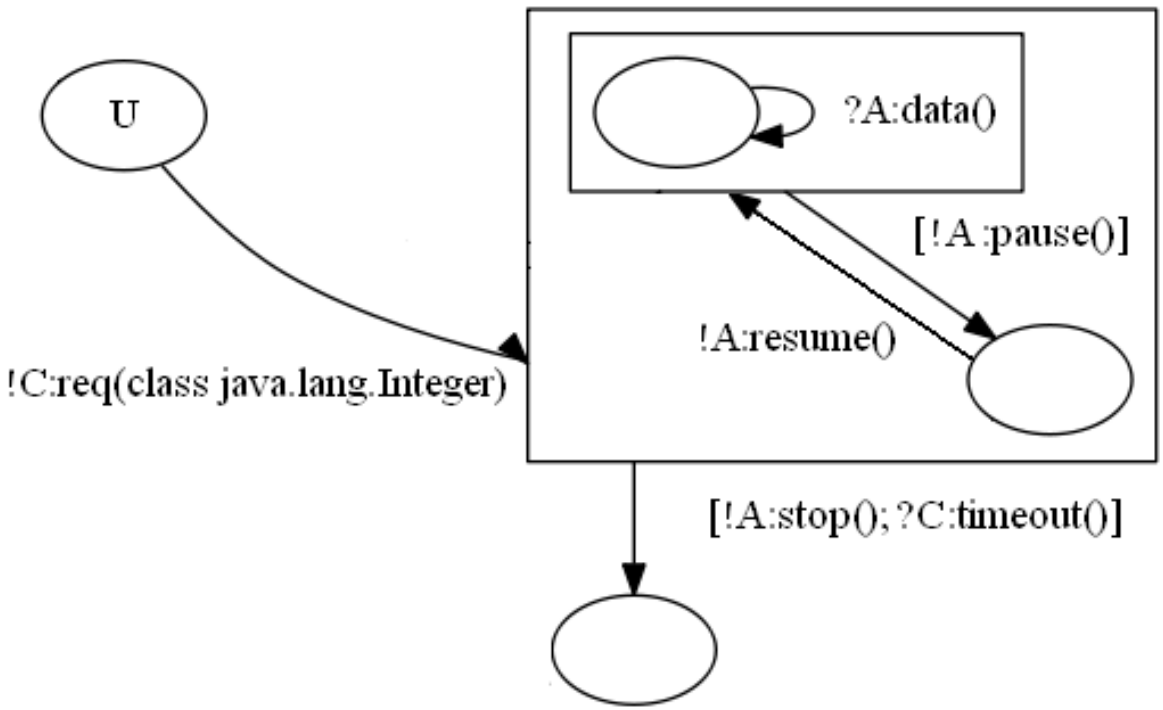
```
req(int) to C;
// start() from C to A;
interruptible _1 {
  rec X {
    interruptible _2 {
      rec Y {
        data() from A;
        continue Y;
      }
    } with {
       throws pause() to A;
    }
    resume() to A;
    continue X;
  }
} with {
  throws stop() to A, C;
  catches timeout() from C;
}
}
```

Resource Control: User

```
    req(int) to C;
    // start() from C to A;
    interruptible _1 {
      rec X {
        interruptible _2 {
          rec Y {
            data() from A;
            continue Y;
          }
        } with {
           throws pause() to A;
        }
        resume() to A;
        continue X;
      }
    } with {
      throws stop() to A, C;
      catches timeout() from C;
    }
}
```

Resource Control: User

```
req(int) to C;
// start() from C to A;
interruptible _1 {
  rec X {
    interruptible _2 {
      rec Y {
        data() from A;
        continue Y;
      }
    } with {
       throws pause() to A;
    }
    resume() to A;
    continue X;
  }
} with {
  throws stop() to A, C;
  catches timeout() from C;
}
}
```



Resource Control: User implementation

```
class UserApp(BaseApp):
    def start(self):
        self.buffer = buffer(MAX_SIZE)
        conv = Conversation.create('RACProtocol', 'config.yml')
        c = conv.join(user, 'alice')
        # Request 1 hour access
        c.send(controller, 'req', 3600)
        with c.scope('timeout', 'stop') as c_x:
            while not self.should_stop():
                with c_x.scope('pause') as c_y:
                    while not self.buffer.is_full():
                        data = c_y.recv(agent)
                        self.buffer.append(data)
                    c_y.send_interrupt('pause')
                use_data(self.buffer)
                self.buffer.clear()
                c_x.send(agent, 'resume')
            c_x.send_interrupt('stop')
        c.close()
```

Session session type

```
// A = speaker, B,C = other participants... role parameterisation?
global protocol Session(role A, role B, role C) {
  interruptible {
    rec X {
      interruptible {
        rec Y {
          talk() from A to B, C;
          continue Y;
        }
      } with {
        question() by B;
        question() by C;
      }
      answer() from A to B;
      continue X;
    }
  } with {
    thanks() by A;
  }
}
```

Session session type

```
// A = speaker, B,C = other participants... role parameterisation?
global protocol Session(role A, role B, role C) {
  interruptible {
    rec X {
      interruptible {
        rec Y {
          talk() from A to B, C;
          continue Y;
        }
      } with {
        question() by B;
        question() by C;
      }
      answer() from A to B;
      continue X;
    }
  } with {
    thanks() by A;
  }
}
```