Hybrid Session Verification through Endpoint API Generation

Raymond Hu and Nobuko Yoshida

Imperial College London

Abstract. This paper proposes a new hybrid session verification methodology for applying session types directly to mainstream languages, based on generating protocol-specific endpoint APIs from multiparty session types. The API generation promotes static type checking of the behavioural aspect of the source protocol by mapping the state space of an endpoint in the protocol to a family of channel types in the target language. This is supplemented by very light run-time checks in the generated API that enforce a linear usage discipline on instances of the channel types. The resulting hybrid verification guarantees the absence of protocol violation errors during the execution of the session. We have implemented our methodology for Java as an extension to the Scribble framework, and used it to implement compliant clients and servers for real-world protocols such as HTTP and SMTP. The API generation methodology additionally provides a platform for applying further features from session type theory: our implementation supports choice subtyping through branch interface generation, and safe permutation of I/O actions and affine inputs through input future generation.

1 Introduction

Application of session types to practice. Session types [15,16,5] are a type theory for communications programming which can guarantee the absence of communication errors in the execution of a session, such as sending an unexpected message or failing to handle an incoming message, and deadlocks due to mutual input dependencies between the participants. One direction of applying session types to practice has investigated extending existing languages with the necessary features, following the theory, to support static session typing. This includes extensions of Java [19,40] with first-class channel I/O primitives and mechanisms for restricting the aliasing of channel objects, that perform static session type checking as a preprocessor step alongside standard Java compilation. New languages have also been developed from session type concepts. The design of SILL [33,41] is based on a Curry-Howard isomorphism between propositions in linear logic and session types, giving a language with powerful linear and session typing features, but that requires programmers to shape their data structures and algorithms according to this paradigm.

To apply session types more directly to existing languages, another direction has investigated dynamic verification of sessions. In [9], multiparty session types

(MPST) are used as a protocol specification language from which run-time endpoint monitors can be automatically generated. The framework guarantees that each monitor will allow its endpoint to perform only the I/O actions permitted according to the source protocol [1]. Although flexible, dynamic verification loses benefits of static type checking such as compile-time error detection and IDE support. Session types have been also applied through code generation to specific target contexts. [31] develops a framework for MPI programming in C that uses MPST as a language for specifying parallel processing topologies, from which a skeleton implementation of the communication structure using MPI operations is generated. The skeleton is then merged with user supplied functions for the computations around the communicated messages to obtain the final program.

This paper presents a new methodology for applying session types directly to mainstream statically typed languages. There are two main novel elements:

Hybrid session verification. A trend in recent works [13,8,7,2,42] has been the study of explicit relationships between session types and linear types. In this work, we continue in the direction of developing session types as a system for tracking correct communication behaviour, in terms of I/O channel actions, built on top of a linear usage discipline for channel resources (every instance of a channel should be used exactly once). We apply this formulation practically as *hybrid* session verification: we statically verify the behavioural aspect through the native type system of the target language, supplemented by very light run-time checks on linear channel usage.

Endpoint API generation. In this work, we use multiparty session types as a protocol specification language from which we can generate APIs for implementing the endpoints in a statically typed target language. Taking an FSM (finite state machine) representation of the endpoint behaviour in the protocol [11,22], we reify each state as a distinct channel type in the target language that permits only the exact I/O operations in that state according to the source protocol. These *state channels* are linked up as a call-chaining API for the endpoint that returns a new instance of the successor state channel for the action performed. Session type safety is thus ensured by static typing of I/O behaviour on each state channel, in conjunction with run-time checks that every instance of a state channel is used linearly.

Our methodology is a practical compromise that combines benefits from fully static session type systems and code generation approaches. Firstly, this methodology allows many of the safety benefits of session types, such as sending only expected message types and exhaustive handling of potential input types, to be statically checked in mainstream languages like Java, up to the linear channel usage contract of the generated API. Secondly, by directly targeting existing languages, user implementations of session endpoints using the generated API can be readily integrated with existing libraries and IDE support.

We present the implementation of our methodology for Java as an extension to Scribble [39], a practical protocol description language based on multiparty session types. Beyond the basic safety properties of enforcing session type behaviour through endpoint FSMs, we take advantage of our hybrid approach

```
// External data types for message payload formatting
type <java> "java.lang.Integer" from "rt.jar" as Int;

global protocol Adder(role C, role S) {
  choice at C { // Internal choice by C
    Add(Int, Int) from C to S; // Message sig: op(payload)
    Res(Int) from S to C;
    do Adder(C, S); // Recursive protocol def
  } or {
    Bye() from C to S;
    Bye() from S to C; // Protocol end
} }
```

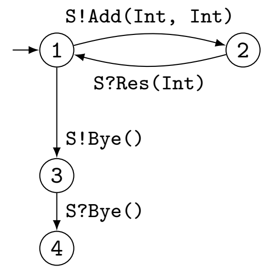

Fig. 1. (a) Scribble global protocol, and (b) Endpoint FSM for C.

to support additional practical features such as value-switched branches and abstraction of nominal state channels as I/O interfaces. API generation also provides a platform for applying further features from session type theory: our implementation supports choice subtyping [12] through branch interface generation, and safe permutation of I/O actions [26,3] and affine inputs [33,25] through input future generation. We have tested our framework by using our API generation to implement compliant clients and servers for real-world protocols such as HTTP and SMTP.

Outline: § **2** describes the Scribble toolchain that this paper builds on, and gives an overview of the new methodology for hybrid session verification through API generation. § **3** presents our implementation of the proposed methodology that generates Java endpoint APIs from Scribble protocol specifications. § **4** discusses SMTP as a use case and extensions to the API generation for asynchronous I/O permutations and affine inputs, and abstraction of nominal Java channel types by generating I/O interfaces. § **5** discusses related and future work.

This work on hybrid session verification through endpoint API generation was first presented at the CoCo:PoPs workshop [4]. The Java tools presented in this paper are publically available as part of the Scribble [39] open source github repository [38]. The first presented version [4] of these tools and example applications can be retrieved from there, e.g., June 2015 [36]; the latest research version can be found at a public fork [37].

2 Overview

The Scribble toolchain. The Scribble methodology starts from specifying a *global protocol*, a description of the full protocol of interaction in a multiparty communication session from a neutral perspective, i.e. all potential and necessary message exchanges between all participants from the start of a session until completion. The communication model for Scribble protocols is designed for asynchronous but reliable message transports with ordered delivery between each pair of participants, e.g. standard Internet applications and Web services that use TCP, HTTP, etc.

Global protocol specification. We use as a first running example a simple client-server protocol for a service that adds two integers, written in Scribble in Fig. 1 (a). The main elements of the protocol specification are as follows.

The *protocol signature* (line 4) declares the name of the protocol (`Adder`) and the abstraction of each participant as a named `role` (`C` and `S`). *Payload format types* (line 2) give an alias (e.g. `Int`) to data type definitions from an external language (`java.lang.Integer`) used to define the wire protocols for message formatting. A *message signature* (e.g. `Add(Int, Int)`) declares an *operator* name (`Add`) as an abstract message identifier (which may correspond concretely to, e.g., a header field), and some number of payload types (a pair of `Int`). *Message passing* (e.g. line 6) is output-asynchronous, i.e. dispatching the message is non-blocking for the sender (`C`). The receiver (`S`) is blocked on the message input. *Located choice* (e.g. line 5) states the subject role (`C`) for which selecting one of the listed protocol blocks to follow is a mutually exclusive internal choice. This decision is an external choice to all other roles involved in each block, which must be appropriately coordinated by explicit messages. *Recursive protocol definitions* (line 8) describe recursive interactions between the roles involved. Non-recursive `do` statements can be used to factor out common subprotocols.

Scribble performs an initial validation on global protocols to assert that the protocol can be correctly realised by a system of independent endpoint processes. In this two-party example, the validation checks that each choice case is indeed communicated by `C` to `S` unambiguously (a simple error would be, e.g., if `C` firstly sends a `Bye` to `S` in both cases).

Local protocol projection and Endpoint FSMs. Following a top-down interpretation of formal MPST systems, Scribble syntactically *projects* [6] a valid source global protocol to a *local protocol* for each role. Projection essentially extracts the parts of the global protocol in which the target role is directly involved, giving the localised behaviour required of each role in order for a session to execute correctly as a whole. See § A.1 for the projection of `Adder` for `C`. A further validation step is performed on each projection of the source protocol for role-sensitive properties, such as reachability of all protocol actions per role. The validation also restricts recursive protocols to tail recursion. A valid global protocol with valid projections for each role is a *well-formed protocol*.

Building on a formal correspondence between syntactic local MPST and communicating FSM [10,22], Scribble can transform the projection of any well-formed protocol for any of its roles to an equivalent *Endpoint FSM* (EFSM). Fig. 1 (b) depicts the EFSM of the projection for `C`. The nodes delineate the state space of the endpoint in the protocol, and the transitions the explicit I/O actions between protocol states. The notation, e.g., `S!Bye()` means output of message `Bye()` to `S`; `?` similarly stands for input. The (tail) recursion in the protocol naturally corresponds to the cycle between states `1` and `2`.

Hybrid session verification through endpoint API generation. This paper proposes a new methodology for applying session types to practice that ensures communication safety through a hybrid verification approach.

Static type checking of I/O behaviour. We consider the EFSMs derived from a source global protocol to represent the *behavioural* aspect of the session type. Our methodology is to generate a protocol-specific endpoint implementation API for a target role by capturing its EFSM via the native type system of a statically typed target language. The key points of the API generation are:

– The Scribble toolchain is used to validate the source global protocol, project to local protocols, and generate the EFSM for the target role.
– Each state in the EFSM is reified as a distinct channel type in the type system of the target language. We refer to channels of these generated types as *state channels*.
– The only I/O operations permitted by a generated channel type are safe actions according to corresponding EFSM state in the protocol.
– The return type of each generated I/O operation is the channel type for the next state following the corresponding transition from the current state. Performing an I/O operation on a state channel returns a new instance of the successor channel type.

Starting from a session channel for the initial state of the protocol, and performing an I/O operation on each state channel returned by the previous operation, the generated API statically ensures that an endpoint implementation is accepted by the encapsulated EFSM and thus observes the protocol. The implicit usage contract of the generated API is thus to use every state channel returned by an API call exactly once up to the end of the session, to respect EFSM semantics in terms of following state transitions linearly up to the terminal state. If respected, the generated API is guaranteed to yield a fully session type safe endpoint implementation.

Run-time checking of linear state channel usage. Due to the lack of static support for linear usage of values or objects in most mainstream languages, we take the practical approach of checking linear usage of state channel instances at run-time. These checks are inlined into the Endpoint API as part of the API generation. There are two cases for state channel linearity to be violated.

Repeat use. Every state channel instance maintains a boolean state value indicating whether an I/O operation has been performed. The generated API guards each I/O operation permitted by the channel type with a run-time check on this boolean to ensure the state channel is not used more than once.

Unused. All state channels for a given session instance share a boolean state value indicating whether the session is complete for the local endpoint. The generated API sets this flag when a *terminal operation*, i.e. an I/O action leading to the terminal EFSM state, is performed. In conjunction with a language mechanism for delimiting the scope of a session implementation, such as standard exception handling constructs, the generated API checks session completion when program execution leaves the scope of the session.

If any state channel remains unused (possibly discarded, e.g. garbage collected) on leaving the scope of a session implementation, then it is not possible for the completion flag to be set.

3 Hybrid Endpoint API generation for Java

The API generation takes as input a Java-based Scribble protocol specification, meaning a well-formed global protocol with Java-defined payload format types. More formally, as explained below, we start from a *global type*, project the target *local type*, and translate it to an Endpoint FSM (EFSM).

Global and local types. Set $\mathbb{R}$ (ranged by $r, r', \ldots$) is the finite set of *roles* and $\mathbb{L}$ (ranged by $l, l', \ldots$) is the finite set of *message labels*. The syntax of global [16,5,22] (ranged by $G, G', \ldots$) and local types (ranged by $L, L', \ldots$) is:

$$G ::= r \to \{r_i(l_i : \vec{T_i}) : G_i\}_{i \in I} \mid \mu X.G \mid X \mid \mathsf{end}$$
$$L ::= !\{r_i(l_i : \vec{T_i}) : L_i\}_{i \in I} \mid r?\{(l_i : \vec{T_i}) : L_i\}_{i \in I} \mid \mu X.L \mid X \mid \mathsf{end}$$

The first global type corresponds to a choice in Scribble where r selects l_i with payload types $\vec{T_i}$ at r_i and becomes G_i. $\mu X.G$ is a recursive type and end denotes termination. In local types, the first type represents a select to r_i by l_i with $\vec{T_i}$, and the second represents a branch from r. The rest is as for G. The relationship between global and local types is defined by the projection function $G \downarrow_r L$ (defined in [16,5,22]) which means that L is a view from r of G. E.g.,

$$r \to \{r_i(l_i : \vec{T_i}) : G_i\}_{i \in I} \downarrow_r !\{r_i(l_i : \vec{T_i}) : L_i\}_{i \in I} \text{ with } G_i \downarrow_r L_i$$

Endpoint FSMs (EFSMs) serve as an interface between source protocol validation (§ **??**) and projection, and the subsequent API generation. Formally, an *Endpoint FSM* E for L is a tuple $(\mathbb{R}, \mathbb{L}, \mathbb{T}, \Sigma, \mathbb{S}, \delta)$ where: $\mathbb{R}$ and $\mathbb{L}$ are sets occurring in L, $\mathbb{T}$ is the finite set of *payload format types* declared by L; the *alphabet* Σ is a finite non-empty set of *actions* $\{\alpha_i\}_{i \in I}$, where α is either output $r!l(\vec{T})$ or input $r?l(\vec{T})$ with $r \in \mathbb{R}, l \in \mathbb{L}, T_i \in \mathbb{T}$; the set of *states* $\mathbb{S}$ is a finite non-empty set of S; and the *transition function* δ is a partial function $\mathbb{S} \times \Sigma \to \mathbb{S}$. We additionally define: $\delta(S) = \{\alpha \mid \exists S' \in \mathbb{S}.\delta(S, \alpha) = S'\}$.

Given L, we obtain a mapping $\mathtt{fsm}(L) = (\mathbb{R}, \mathbb{L}, \mathbb{T}, \Sigma, \mathbb{S}, \delta)$ as follows. Let $\mathbb{X}$ be a map from recursion variables to states $X \mapsto S$, and S_{term} be the unique terminal state (and the only accepting state). We define: $\mathtt{graph}(S, L, \mathbb{X}) =$

$$\begin{cases} \bigcup_{i \in I}[(S, r_i!l_i(\vec{T_i})) \mapsto S'_i) \cup \mathtt{graph}(S'_i, L_i, \mathbb{X})] & L = !\{r_i(l_i : \vec{T_i}) : L_i\}_{i \in I}, S'_i = \mathsf{succ}(L_i, \mathbb{X}) \\ \bigcup_{i \in I}[(S, r?l_i(\vec{T_i})) \mapsto S'_i) \cup \mathtt{graph}(S'_i, L_i, \mathbb{X})] & L = r?\{(l_i : \vec{T_i}) : L_i\}_{i \in I}, S'_i = \mathsf{succ}(L_i, \mathbb{X}) \\ \mathtt{graph}(S, L, \mathbb{X} \cup X \mapsto S) & L = \mu X.L \\ \emptyset & L = X \text{ or } \mathsf{end} \end{cases}$$

$\mathsf{succ}(X, \mathbb{X}) = S$ if $X \mapsto S \in \mathbb{X}$; $\mathsf{succ}(\mathsf{end}, \mathbb{X}) = S_{\mathsf{term}}$; otherwise $\mathsf{succ}(L, \mathbb{X}) =$ fresh S. Then we set $\delta = \mathtt{graph}(S_{\mathsf{init}}, L, \emptyset)$ where S_{init} is the initial state, and Σ and $\mathbb{S}$ are the set of actions and states in δ.

Some properties are guaranteed for any EFSM derived from a well-formed protocol. (1) There is exactly one initial state $S_{\mathsf{init}} \in \mathbb{S}$ such that $\nexists S' \in \mathbb{S}, \alpha \in \Sigma.\delta(S', \alpha) = S_{\mathsf{init}}$. (2) S_{term} is the only $S \in \mathbb{S}$ such that $\delta(S) = \emptyset$. (3) Every $S \in \mathbb{S}$ is one of three kinds: an *output state* $S^!$, *input state* $S^?$, or S_{term}. An output state means $\delta(S) = \{\alpha_i\}_{i \in I}, |I| > 0$ and every $\alpha_{i \in I}$ is an output. Similarly for input states. (4) For each $S^?$ with $\delta(S^?) = \{\alpha_i\}_{i \in I}$, every $\alpha_{i \in I}$ specifies the same r.

Session API. From a given EFSM, our implementation of the Endpoint API generation outputs two main protocol-specific components, the *Session API* and the *State Channel API*. The generated APIs depend on a small collection of abstract protocol-independent base Java classes: `Role`, `Op`, `Session`, `SessionEndpoint` and `Buf`. These are explained below.

The main class of the Session API is a generated final subclass of the base `Session` class with the same name as the source protocol, e.g. `Adder` (Fig. 1 (a)). Its two main purposes are as follows.

Reification of abstract names. Session types make use of abstract names as role and message identifiers in types, that the type system expects to be present in the program to drive the type checking. The Session API reifies these names as singleton Java types. For each role or operator name $n \in \mathbb{R} \cup \mathbb{L}$, we generate the following. (1) A final Java class named n that extends the relevant base class (`Role` or `Op`). The n class has a single private constructor, and a public static final field of type n and with name n, initialised to a singleton instance of this class, e.g. `public static final C C = new C();`. (2) A public static final field of type n and with name n in the main Session Class, that refers to the corresponding field constant in the n class.

The Session API comprises the main Session Class with the role and message name classes.

Session instantiation. As a distributed computing abstraction, a session can be considered a unit of interaction that is an instance of a session type. Following this intuition, the API user starts an endpoint implementation by creating a new instance of the main Session Class. The API uses the Session object to encapsulate static information, such as the source protocol, and run-time state related to the execution of this session, such as the session ID.

A Session object is used to create a `SessionEndpoint<S, R>`, parameterised on the parent Session and target role types, as on lines 1–3 in Fig. 4 (a). The first two constructor arguments are the Session object and the singleton generated for the target role, from which the `SessionEndpoint` type parameters are inferred, and the third is an implementation of the Scribble `MessageFormatter` interface for this endpoint using the Java format types declared in the Scribble specification. The new `SessionEndpoint` object encapsulates the state specific to this endpoint in the session, such as the target role and local network state.

State Channel API. Based on the properties of EFSMs, the core State Channel API is given by generating the channel classes for each EFSM state according to Fig. 2. In the following, we use r, l, etc. to denote both a session type name and its generated Java type (as described above); similarly, S for an EFSM state and its generated Java channel type.

An output state is generated as a `SendSocket` with one `send` method for each outgoing transition action α: the first two parameters are the role r and operator l singleton types, followed by the sequence of Java payload format types, if any (ϵ means no payloads). The return type is `EndSocket` (which supports no session I/O operations) if the successor state is the terminal state, or else the channel class generated for the successor state. Unary and non-unary input states are treated differently. Channel class generation for unary inputs is similar to that for outputs. The main difference is that each payload format type is generated as

State kind	Java channel base class and session operation method signatures
$S^!$	`SendSocket` For each $\alpha = r!l(\vec{T}) \in \delta(S^!)$: T_{ret} `send(`r `role,` l `op` $[\![\vec{T}]\!]^!$`)`
Unary $S^?$	`ReceiveSocket` ($\lvert\delta(S^?)\rvert = 1$) For $\alpha = r?l(\vec{T}) \in \delta(S^?)$: T_{ret} `receive(`r `role,` l `op` $[\![\vec{T}]\!]^?$`)`
$S^?$	`BranchSocket` ($\lvert\delta(S^?)\rvert > 1$) For $\alpha = r?l(\vec{T}) \in \delta(S^?)$: $C_{S^?}$ `branch(`r `role)` where $C_{S^?}$ is the following `CaseSocket` class `CaseSocket` For each $\alpha = r?l(\vec{T}) \in \delta(S^?)$: T_{ret} `receive(`l `op,` $[\![\vec{T}]\!]^?$`)`

where $[\![\vec{T}]\!]^! = \epsilon$ if $|\vec{T}| = 0$, else '`,` T_1 `pay`$_1$`, ...,`T_n `pay`$_n$'
$[\![\vec{T}]\!]^? = \epsilon$ if $|\vec{T}| = 0$, else '`, Buf<? super` T_1`> pay`$_1$`, ..., Buf<? super` T_n`> pay`$_n$'
$T_{ret} = \delta(S, \alpha)$ if $S \neq S_{\mathsf{term}}$, else `EndSocket`

Fig. 2. State channel Java class generation.

Gen. class	Session operation methods	Return
`C_1`	`send(S role, Add op, Integer pay1, Integer pay2)`	`C_2`
	`send(S role, Bye op)`	`C_3`
`C_2`	`receive(S role, Res op, Buf<? super Integer> pay1)`	`C_1`
`C_3`	`receive(S role, Bye op)`	`EndSocket`
`S_1`	`branch(C role)`	`S_1_Cases`
`S_1_Cases`	`receive(Add op, Buf<? super Integer> pay1, Buf<? super Integer> pay2)`	`S_2`
	`receive(Bye op)`	`S_3`
`S_2`	`send(C role, Res op, Integer pay1)`	`S_1`
`S_3`	`send(C role, Bye op)`	`EndSocket`

Fig. 3. Generated state channel Endpoint API for `C` and `S` in `Adder`.

a Scribble `Buf` type with a supertype of the payload type as a type parameter. A Scribble `Buf` is a simple parameterised buffer for a single payload value, written by the generated `receive` code when the message is received. Non-unary inputs are explained in § 3 (Session branches).

Only the channel class corresponding to the initial EFSM state has a public constructor (taking a single argument of type `SessionEndpoint<S, R>`). Every other state channel class is only instantiated internally by the method-chaining API: every session method is generated to return a new instance of the successor state channel. Fig. 3 summarises the channel classes and session I/O methods generated for the `C` and `S` roles of the `Adder` example. The API generation promotes the use of the generated utility types to direct implementations as much as possible. For example, in `C_1`, the two output options are distinguished as `send` methods overloaded on the operator type (as well as the payload types).

```
1  Adder adder = new Adder(); // New session object
2  try (SessionEndpoint<Adder,C> se =
3          new SessionEndpoint<>(adder, C, new AdderFormatter())) {
4    se.connect(S, SocketChannel::new, hostS, portS); // TCP channel
5    Adder_C_1 s1 = new Adder_C_1(se);
6    // State channel implementation of C starting from s1 of state type C_1
7    Buf<Integer> i = new Buf<>(1); // i.val stores the buffer value (Integer)
8    for (int j = 0; i < N; j++)
9      s1 = s1.send(S, Add, i.val, i.val).receive(S, Res, i); // C_1->C_2->C_1
10   s1.send(S, Bye).receive(S, Bye); // C_1->C_3->EndSocket
11 } // Session completion checked at run-time when se is (auto) closed
```

```
Adder_C_3 fib(Adder_C_1 s1, Buf<Integer> i1, Buf<Integer> i2, int i) throws ... {
  return (i < N) ? fib(s1.send(S, Add, i1.val, i1.val = i2.val) // C_1->C_2..
                         .receive(S, Res, i2), i1, i2, i+1) // ..->C_1
                 : s1.send(S, Bye);  } // C_1->C_3
```

```
Adder_S_3 add(Adder_S_1 s1, Buf<Integer> i1, Buf<Integer> i2) throws ... {
  Adder_S_1_Cases cases = s1.branch(C); // Receives message; S_1->S_1_Cases
  switch (cases.op) { // enum field set by API according to the received op
    case Add: return add(cases.receive(Add, i1, i2) // S_1_Cases->S_2..
                              .send(C, Res, i1.val+i2.val), i1, i2); //..->S_1
    case Bye: return cases.receive(Bye); // S_1_Cases->S_3
} }
```

Fig. 4. Using Fig. 3: (a) session initiation and example endpoint implementation for `C`, (b) a Fibonacci client implemented using `C`, and (c) the main loop and branch of `S`.

Hybrid verification of endpoint implementations. Fig. 4 (a) lines 1–5 list a typical preamble in an endpoint implementation using the generated API.

Session initiation and state channel chaining. We create a new `Adder` session instance and a `SessionEndpoint` for role `C`. The `SessionEndpoint se` is used to perform the client-side `connect` to `S` (the first argument) as a standard TCP channel (second argument). The session connection phase is concluded when `se` is given as a constructor argument to create an initial state channel of type `Adder_C_1`, and commence the implementation of the `C` endpoint.

Lines 7–10 give a simple imperative style implementation of `C` that repeatedly adds an integer, stored in the `Buf<Integer> i`, to itself. In each protocol state, given by the channel class, the generated API ensures that any session operation performed is indeed permitted by the protocol, e.g. state channel `s1` permits only a `send(S, Add, int, int)` or a `send(S, Bye)`. The method-chaining API is used as a fluent interface (the implicit state transitions are in comments), chaining the `receive` onto the `send Add`, which returns a new instance of `C_1` following the recursive protocol. The recursion is enacted N times by the for-loop, linearly assigning the new `C_1` to the existing `s1` variable in each iteration, before the final `Bye` exchange after the loop terminates. Naturally, the API also allows the equivalent safe implementation, unfolding the recursion for a fixed N:

```
s1.send(S, Add, i.val, i.val).receive(S, Res, i)..Add/Res chained N−1 more times..
  .send(S, Bye).receive(S, Bye);
```

The flexbility of the Endpoint API as a native language API is demonstrated by the session type safe functional style implementation of a Fibonacci client in

Fig. 4 (b) using the `Adder` service. While the structure of the imperative code in (a) corresponds closely to that of source protocol, the more complicated protocol control flow in this more functional style code demonstrates the value of the session type based Endpoint API in guiding the implementation and promoting safe protocol compliance. The API ensures that the nested `send`-`receive` argument expression returns the endpoint to the `S_1` state in each recursive method call, and that the recursion terminates with the endpoint in the `S_3` state.

State channel linearity. Linear usage of every session channel object in endpoint implementations is enforced by inlining run-time checks into the generated Java API following the two cases of the basic approach in § 2.

Repeat use of a state channel raises a `LinearityException`. The boolean state indicating linear object consumption, and the associated guard method called by every generated session operation method, are inherited from a base `LinearSocket` that is the superclass of channel classes in Fig. 2 (except `EndSocket`).

Session completion is treated by generating the `SessionEndpoint` object to implement the Java `Autocloseable` interface. The Endpoint API requires the user to declare the `SessionEndpoint` in a try-with-resource statement (as in Fig. 4 (a), line 3), allowing the API to check that a terminal session operation has been performed when control flow leaves the try-statement; if not, then an exception is raised. Java IDEs, such as Eclipse, support compile-time warnings when `AutoCloseable` resources are not safely handled in an appropriate try statement.

We observe that certain implementation styles using the generated API, such as fluent method-chaining and functional methods (e.g. above and Fig. 4 (b)), can help avoid linearity bugs by reducing the use of intermediate state variables and potentially bad aliasing through state channel assignments.

Session branches. The theoretical languages for which session types were developed support communication channels as first-class primitives. In particular, session calculi typically feature an explicit branching primitive, e.g. $c\&(r, \{l_i : P_i\}_{i \in I})$ [6], to atomically receive a message on channel c from r and, depending on the label l_i, reduce to the corresponding process continuation P_i. For languages like Java that lack such I/O primitives, the API generation approach enables different options.

The basic option supported by our API generation, intended for standard `switch` patterns (or `if-else` cases, etc.), is to separate the branch input action from the subsequent case analysis on the received message operator (`BranchSocket` and `CaseSocket` for non-unary inputs in Fig. 2). To delimit the cases of a branch state in a type-directed manner, the API generation creates an enum covering the permitted operators in each `BranchSocket` class, e.g. for `S` in `Adder`:

```
enum Adder_S_1_Enum implements OpEnum { Add, Bye }
```

Fig. 4 (c) lists the main loop and branch in an implementation of `S` in `Adder`. The `branch` operation of the `BranchSocket s1` blocks until the message is received, and returns the corresponding `CaseSocket` with the `op` field, of the enum type `Adder_S_1_Ops`, set according to the received operator. Using a switch statement on the `op` enum, the user calls the appropriate `receive` method on the `CaseSocket` to obtain the corresponding state channel continuation. The API raises an ex-

```
global protocol Smtp(role C, role S) {
  220 from S to C; // 220 smtp2.cc.ic.ac.uk ESMTP Exim 4.85 ...
  do Initiation(C, S); // First initiation exchange on plain TCP connection
  do StartTls(C, S); // Negotiate secure connection
  do Initiation(C, S); // Second initiation exchange on secure connection
  ... // Continuation of SMTP session over secure connection
}
global protocol Initiation(role C, role S) {
  Ehlo from C to S; // EHLO user.test.com
  rec X { choice at S { 250d from S to C; // 250-smtp2.cc.ic.ac.uk Hello ...
                        continue X; } // 250-SIZE 26214400, 250-8BITMIME, etc.
                   or { 250 from S to C; } } // 250 HELP (no dash after 250)
}
global protocol StartTls(role C, role S) {
  StartTls from C to S; // STARTTLS
  220 from S to C; // 220 TLS go ahead
}
```

Fig. 5. Simplified excerpt from a Scribble specification of SMTP.

ception if the wrong `receive` is used, similarly to a cast error, thus introducing an additional run-time check to maintain session type safety.

Java IDEs are able to statically check exhaustive coverage of enum cases, and it would also be straightforward to develop a small plugin for, e.g. Eclipse, to statically check correct handling of branch enum cases for the basic patterns. § A.2 discusses alternative API generation of *branch interfaces*, that support branch subtyping, and do not require any run-time checks for session safety.

4 Use case and further Endpoint API generation features

We have used Scribble and our Java API generation to specify and implement standardised Internet applications, such as HTTP and SMTP, as real-world use cases. Using SMTP as an example, we discuss practically motivated extensions to the core Endpoint API generation presented so far.

SMTP [20] is an Internet standard for email transmission. We specified a subset of the protocol in Scribble (§ A.3) that includes establishing a secure connection and conducting the main mail transaction. Using the generated Endpoint API, it was straightforward to implement a compliant client in Java that is interoperable with existing SMTP servers.

For this section, we focus on a simplified excerpt from the opening stages of `Smtp` in Fig. 5 (cf. § A.3: `Quit`, etc. cases omitted). The client (`C`) first creates a plain TCP connection to the server (`S`) and following the Server `220` welcome message, the initiation exchange (client `EHLO`, and the server `250-` and `250` list of service extensions) is performed. The client then starts the negotiation of a secure channel by `StartTls`. When the channel is secured, the client and server conduct the initiation exchange again (the server may now offer different service extensions), and the remainder of the session is conducted over the secure channel. For this running example, we omit the payload types for brevity.

State kind	Additional method generated for `ReceiveSocket` (from Fig. 2).
Unary $S^?$	`ReceiveSocket` ($\lvert\delta(S^?)\rvert = 1$) For $\alpha = r?l(T_{0\leq i\leq n}) \in \delta(S^?)$: T_{ret} `async(`r `role,` l `op, Buf<? super` $F_{S^?}$`> fut)`
where	T_{ret} is as in Fig. 2, and $F_{S^?}$ is the `InputFuture` generated for this state

Fig. 6. API generation for asynchronous input in unary input states using futures.

Asynchronous I/O permutations and affine inputs. An advanced session pattern unsupported by basic session types, but studied in later extensions [26,3], is to take advantage of asynchronous messaging for safe reordering of I/O actions at an endpoint. For illustration, in Fig. 5, the `Ehlo` message in `Initiation` from `C` to `S` is always preceded by a `220` from `S` to `C`. It is safe for `C` to permute these two actions, sending `Ehlo` first, then receiving `220`. (Note the reverse permutation at `S` is unsafe, due to the potential for deadlock from mutual inputs at both ends.)

Our API generation implements support for safe permutations of I/O actions through the generation of message input futures. For each unary input state, the `ReceiveSocket` (from Fig. 2) is generated with an additional `async` method that takes the same role and operator types as the corresponding `receive` method, and an additional parameter for the subclass of `InputFuture` that we generate for this state, e.g. `C_1_Future`. In contrast to the original `receive`, `async` is generated to return immediately, regardless of whether the expected message has arrived, returning instead a new input future for this state, via the supplied `Buf`, and the successor state channel. The user is free to call `sync` on the input future, which blocks the caller until the message is received, at any later point. E.g.

```
// Assume s1 of type Smtp_C_1 is the initial state channel for C
Buf<Smtp_C_1_Future> buf = new Buf<>(); // The generated InputFuture for this state
s1.async(S, _220, buf).send(S, Ehlo); // "Postponed" input; output done first
String pay1 = buf.val.sync().pay1; // Postponed input now performed via the future
```

The `async` operation essentially enables the input *transition* from the local EFSM state to be decoupled from the actual message input *action* in a safe way. Calling `sync` on an input future implicitly triggers all pending prior input futures for the same peer role, safely preserving the FIFO messaging semantics between each pair of roles in a session. Thus any endpoint implementation using the generated input futures retains the same safety properties as implementations using only a regular blocking receive for inputs. (With this extension, `receive` is simply generated to combine `async` and `sync` in one step.) This scheme naturally supports the permutation of inputs between different roles.

Using an input future more than once has no effect, but input futures are not linear objects (cf. state channels). An input future may be discarded unused, treating the input as an affine action [33,25]. In session types (e.g. [16,5]), input actions are typically treated linearly to prevent unread messages in input queues corrupting later inputs. Here, safety is preserved by the implicit completion of pending futures, clearing any potential garbage before the current future itself.

Interface generation for abstract I/O states. The SMTP use case raised a practical issue in generating Java state channel APIs from session types. While

```
// Action Interfaces (message payloads ommitted for brevity)
interface In_S$220<_S extends Succ_In_S$220> { _S receive(S role, _220 op); }
interface Out_S$Ehlo<_S extends Succ_Out_S$Ehlo> { _S send(S role, Ehlo op); }
interface In_S$250d<_S extends Succ_In_S$250d> { _S receive(S role, _250d op); }
interface In_S$250<_S extends Succ_In_S$250> { _S receive(S role, _250 op); }

// Successor State Interfaces (for the "EHLO" and "250-" messages)
interface Succ_Out_S$Ehlo {
  default Branch_S$250d$_250<?,?> to(Branch_S$250d$_250<?,?> cast) { return (..) this; }
}
interface Succ_In_S$250d { ... } // default 'to' cast method as above

// Abstract I/O State Interfaces (for the "250-" and "250 " branches)
interface Branch_S$250d$_250<_S1 extends Succ_In_S$250d, _S2 extends Succ_In_S$250>
      extends Succ_Out_S$Ehlo, Succ_In_S$250d { // Denotes preceding actions
    public static final Branch_S$250d$_250<?, ?> cast = null; // For 'to' casts
    Cases_S$250d$_250<_S1, _S2> branch(S role);
}
// Protocol branches generate a pair of abstract Branch/Case I/O State Interfaces
interface Cases_S$250d$_250<_S1 extends Succ_In_S$250d, _S2 extends Succ_In_S$250>
      extends In_S$250d<_S1>, In_S$250<_S2> { ... } // Denotes available actions

// Concrete state channel classes (for the "250-" and "250 " branches)
class Smtp_C_3 implements Branch_S$250d$_250<Smtp_C_3, Smtp_C_4> { ... }
class Smtp_C_7 implements Branch_S$250d$_250<Smtp_C_7, ...> { ... }
```

Fig. 7. Selected abstract I/O interfaces and channel classes generated for `C` in `Smtp`.

formal session types offer a structural abstraction of communication behaviour by focusing on the I/O actions between protocol states, the API generation reifies these states concretely as nominal Java types.

Although nominal channel types are good as protocol documentation (the default numbering scheme for states can be easily replaced by a user-supplied mapping from states to more meaningful class names), this example shows a situation where the nominal types limit code reuse within a session implementation using the Endpoint APIs generated so far. The repeated initiation pattern is factored out in the Scribble as a subprotocol, but the two exchanges correspond to distinct parts of the EFSM (§ A.3), and are thus generated with distinct channel types, preventing this pattern from being factored out in the implementation.

To address this issue, our approach is to supplement the nominal Java channel types by generating interfaces for abstract I/O states, which we explain through this example. Fig. 7 lists a selection of the generated concrete state channel classes and their I/O interfaces. Together, there are four main elements:

(1) For every I/O action, we generate an *Action Interface* named according to its session type characterisation, e.g. `In_S$220` means input of `220` from `S`. This interface is parameterised on a Successor State Interface (explained next).

(2) For every I/O action, we generate a *Successor Interface* to be implemented by every I/O State Interface (explained next) that succeeds the action, e.g. `Succ_Out_S$Ehlo` is implemented by every state that follows an `Out_S$Ehlo` action. Every Successor Interface is generated with a default `to` "cast" method for each

```
1 Succ_In_S$250 doInitiation(Send_S$Ehlo<?> s) { // Take S!Ehlo chan; return succ(S?250)
2   Branch_S$250d$_250<?, ?> b = s.send(S, Ehlo).to(Branch_S$250d$_250.cast);
3   for (Cases_S$250d$_250<?, ?> c = b.branch(S); true; c = b.branch(S))
4     switch (c.getOp()) {
5       case _250d: { b = c.receive(S, _250d).to(Branch_S$250d$_250.cast); break; }
6       case _250: return c.receive(S, _250);
7 } } // (Message payloads ommitted for brevity)
```

```
doInitiation( // Second init exchange on secure channel
  doInitiation(new Smtp_C_1(se).async(S, _220) // First init exchange on plain TCP
    .to(Send_S$StartTls.cast).send(S, StartTls).to(Receive_S$220.cast).async(S, _220)
    .to(Send_S$Ehlo.cast).wrapClient(S, SSLSocketChannelWrapper::new) // SSL/TLS
)....; // Remainder of session
```

Fig. 8. Using the generated I/O State Interfaces to factor out the initiation exchange.

I/O state that implements it.

(3) For every state, we generate an *I/O State Interface* named according to its session type characterisation, e.g. `Branch_S$250d$_250` is a branch state for the cases of `250d` and `250` from `S`. This interface: (a) extends all the Successor Interfaces for the actions that lead to a state with this I/O characterisation; (b) extends all the Action Interfaces permitted by this state; and (c) is parameterised on each of its possible successors, passed through to the corresponding Action Interface. E.g. the `Branch_S$250d$_250` state interface is: (a) reached by an `Out_S$Ehlo` or an `In_S$250d` action; (b) permits `In_S$250d` and `In_S$250` actions (in its counterpart `Cases` interface); and (c) is succeeded by `_S1` and `_S2`.

(4) Finally, each concrete channel class (e.g. `Smtp_C_3`) implements its I/O State Interface, instantiating its generic parameters with its concrete successors. The other contents of the channel class are generated as previously.

The naming scheme for these generated I/O interfaces is not dissimilar to more formal notations for session types, but restricted to the current state and immediate actions with the continuations captured in the successor type parameters.

Using the state channel API generated for `C` with the I/O interfaces in Fig. 7, we factor out one method to implement both initiation exchanges in Fig. 8 (top). The method accepts any state channel with the `Send_S$Ehlo` interface and performs the `send`. This returns the Successor Interface `Succ_Out_S$Ehlo`, for which the only I/O State Interface (in this example) is `Branch_S$250d$_250`. Hence the call to the generated `to` on line 2, although operationally a run-time type cast on the state channel, is a *safe* cast because the it is guaranteed to be valid for all possible successor states at this point. The cast returns a state channel with this interface, and the branch is implemented using a switch according to the relevant I/O State Interfaces. We directly return the `Succ_In_S$250` Successor Interface after receiving the `250` in the second case.

As the above method is implemented using I/O State Interfaces only, we can reuse it to perform both initiation exchanges as in Fig. 8 (bottom). `doInitiation` returns a `Succ_In_S$250`, which may concretely be either the state after the first initiation exchange (to send `StartTls`) or the second (remainder of session). Although the generated I/O State Interface limits the subsequent `to` cast to these two cases, this cast relies on the run-time check for safety. In summary, our Java API generation offers static safety of casting between abstract I/O states and concrete state channels when successor states share the same I/O State Interface, as in Fig. 8, which we believe corresponds to situations where such factoring under common I/O interfaces is the most useful. Otherwise, safety is preserved by a form of hybrid session type checking via the generated cast.

5 Related and future work

Many programming languages based on session types have been developed in the past decade. See [43] for a recent comprehensive survey. Some of the most closely related work was mentioned in § 1; here we give additional discussions.

Static session type checking. A static MPST system uses local types to type check programs (binary session types can be used directly). An implementation of static session type checking, following standard presentations [15,16,5], typically requires two key elements: (1) a syntactic correspondence between local type constructors and I/O language primitives, and (2) a mechanism, such as linear or uniqueness typing, or restrictions on pointer/reference aliasing, that enables precise tracking of channel endpoint values or objects through the control flow of the program. [19] is an extension of Java for binary session types, and [40] for multiparty session types, along these lines. Both introduce new syntax for declaring session types and special session constructs to facilitate typing, with an additional analysis to deal with aliasing of channels. Without such extensions, it is difficult to perform static session type checking in a language like Java without being extremely conservative in the programs that pass type checking. Our API generation approach confers benefits of session types directly to native Java programming, and can be readily generalised for many other existing languages.

Implementations of static session typing in Haskell [35,34] are able to benefit from powerful typing features (in these works, indexed parameterised monads) to ensure linearity of session types without language extensions. An earlier implementation [28] instead relies on implicit threading of a single channel through a computation to avoid any aliasing that may violate linearity.

Other session-based systems, such as Mungo [27]/Bica [14] based on typestates in Java, Links [23]/Jolie [21] for web services and Pabble [32]/ParTypes [24] based on indexed dependent types for parallel programs also require syntax extensions or annotations to be implemented as static typing for most mainstream languages. We believe our hybrid API generation approach is an interesting alternative option for implementing related forms of behavioural types.

Dynamic session verification by run-time monitoring of I/O actions [9,30,29] is the primary verification method in Scribble [39]. Run-time session monitoring is subject to common trade-offs of dynamic verification (§ 1). Monitoring can be applied directly to existing languages, but endpoint implementations must use a specific API or be instrumented with appropriate hooks for the monitor to intercept the actions. Monitoring also verifies only the observed execution trace, not the implementation itself. Our cheaper hybrid verification approach allows certain benefits of static types to be reclaimed for free, including static protocol error detection, up to the linearity condition on state channels, and other IDE assistance for session programming, such as code generation (e.g. session method completion, branch case enumeration) and partial static checking of linearity (e.g. unused state channel variables, unhandled session resources).

Code generation from session types. The code generation framework in [31] (§ 1) works by targetting a specific context, that is, parallel MPI programs in C. In contrast, our API generation approach uses session types for lighter-weight generation of types, rather than programs. Programming using a generated Java Endpoint API is amenable to varied user implementations in terms of local control flow (e.g. imperative or functional) and concurrency (e.g. multithreaded or event-driven) via standard Java language features and existing libraries.

References

1. L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring Networks through Multiparty Session Types. In *FMOODS/FORTE'13*, volume 7892 of *LNCS*, pages 50–65. Springer, 2013.
2. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.
3. T. Chen, M. Dezani-Ciancaglini, and N. Yoshida. On the preciseness of subtyping in session types. In *PPDP'14*, pages 135–146. ACM, 2014.
4. *CoCo:PoPs: Communication-based Computation: Practicalities of Programming with Sessions*. School of Computing Science, University of Glasgow. 8–9 June, 2015. `http://groups.inf.ed.ac.uk/abcd/cocopops/`.
5. M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida. A gentle introduction to multiparty asynchronous session types. In *SFM-15:MP*, volume 9104 of *LNCS*, pages 146–178. Springer, 2015.
6. M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *MSCS*, 760:1–65, 2015.
7. O. Dardha, E. Giachino, and D. Sangiorgi. Session Types Revisited. In *PPDP'12*, pages 139–150. ACM Press, 2012.
8. R. Demangeon and K. Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR'11*, volume 6901 of *LNCS*, pages 280–296. Springer, 2011.
9. R. Demangeon, K. Honda, R. Hu, R. Neykova, and N. Yoshida. Practical Interruptible Conversations: Distributed Dynamic Verification with Multiparty Session Types and Python. *Formal Methods in System Design*, pages 1–29, 2015.
10. P.-M. Deniélou and N. Yoshida. Multiparty Session Types Meet Communicating Automata. In *ESOP'12*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.
11. P.-M. Deniélou and N. Yoshida. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *ICALP'13*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013.
12. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
13. S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 2009.

14. S. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312. ACM, 2010.
15. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
16. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008. A full version will appear in *JACM*.
17. R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-Safe Eventful Sessions in Java. In *ECOOP'10*, volume 6183 of *LNCS*, pages 329–353. Springer, 2010.
18. R. Hu, R. Neykova, N. Yoshida, and R. Demangeon. Practical interruptible conversations: Distributed dynamic verication with session types and python. In *RV 2013*, volume 8174 of *LNCS*, pages 148–130. Springer, 2013.
19. R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In *ECOOP*, volume 5142, pages 516–541. Springer, 2008.
20. IETF. Simple Mail Transfer Protocol. `https://tools.ietf.org/html/rfc5321`.
21. Jolie homepage. `http://www.jolie-lang.org/`.
22. J. Lange, E. Tuosto, and N. Yoshida. From Communicating Machines to Graphical Choreographies. In *POPL'15*, pages 221–232. ACM Press, 2015.
23. Links homepage. `http://groups.inf.ed.ac.uk/links/`.
24. H. A. Lopez, E. R. B. Marques, F. Martins, N. Ng, C. Santos, V. T. Vasconcelos, and N. Yoshida. Protocol-based verification of message-passing parallel programs. In *OOPSLA'15*. ACM, 2015.
25. D. Mostrous and V. T. Vasconcelos. Affine sessions. In *Coordination*, volume 8459 of *LNCS*, pages 115–130. Springer, 2014.
26. D. Mostrous and N. Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.*, 241:227–263, 2015.
27. Mungo. Mungo Project, 2015. `http://www.dcs.gla.ac.uk/research/mungo/`.
28. M. Neubauer and P. Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.
29. R. Neykova, L. Bocchi, and N. Yoshida. Timed Runtime Monitoring for Multiparty Conversations. In *BEAT'14*, volume 162 of *EPTCS*, pages 19–26, 2014.
30. R. Neykova and N. Yoshida. Multiparty Session Actors. In *COORDINATION'14*, volume 8459 of *LNCS*. Springer, 2014.
31. N. Ng, J. G. Coutinho, and N. Yoshida. Protocols by Default: Safe MPI Code Generation based on Session Types. In *CC'15*, LNCS. Springer, 2015.
32. N. Ng, N. Yoshida, and K. Honda. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *TOOLS'12*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.
33. F. Pfenning and D. Griffith. Polarized substructural session types. In *FoSSaCs'13*, volume 9034 of *LNCS*, pages 3–22. Springer, 2015.
34. R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2008.
35. M. Sackman and S. Eisenbach. Session types in haskell, 2008. draft.
36. Scribble. Java Endpoint API tools (CoCo:PoPs version). `https://github.com/scribble/scribble-java/tree/8788ee40ae8e7614bc50bfea86698febc901c066`.
37. Scribble. Latest version of the Java Endpoint API generation tools. `https://github.com/rhu1/scribble-java/tree/rhu1-research`.

38. Scribble. Open source github repository. https://github.com/scribble/scribble-java.
39. Scribble. Project homepage. www.scribble.org.
40. K. C. Sivaramakrishnan, K. Nagaraj, L. Ziarek, and P. Eugster. Efficient session type guided distributed interaction. In *COORDINATION*, volume 6116 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2010.
41. B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP'13*, volume 7792 of *LNCS*, pages 350–369. Springer, 2013.
42. P. Wadler. Proposition as Sessions. In *ICFP'12*, pages 273–286, 2012.
43. Survey on languages based on behavioural types. http://www.di.unito.it/~padovani/BETTY/BETTY_WG3_state_of_art.pdf.

A Appendix

Future work. As future work, we plan to extend our methodology to handle events [17] and interrupt messages [18], which currently rely on syntax extensions or purely dynamic monitoring.

A.1 Appendix: Overview

The Scribble local protocol projected from `Adder` in Fig. 1 (a) for `C` is:

```
local protocol Adder_C(self C, role S) {
  rec X { // Projected recursive type
    choice at C {
      Add(Int, Int) to S;
      Res(Int) from S;
      continue X; // The recursive case
    } or {
      Bye() to S;
      Bye() from S;
} } }
```

A.2 Appendix: Further API generation features

Choice subtyping. Session subtyping [12] for choices allows a select operation to be safely typed as depending on a superset of the cases actually required, and dually a branch operation to be safely typed as supporting a subset of the cases actually offered. Practically speaking, select subtyping is very natural, allowing an implementation to concretely pursue any one option out of those available according to the protocol specification. Select subtyping is implicitly supported by our API generation since Java typing, in conjunction with linearity checking, allows exactly one `send` method to be selected from those permitted by the `SendSocket`.

The practical benefit of branch subtyping in our setting is to allow safe reuse of session branch code in contexts that depend on a subset of the supported cases. For this purpose, our implementation generates (alongside Fig. 3) for

Additional method generated for BranchSocket Adder_S_1 (from Fig. 3)

Gen. i/face	Abstract session operation methods	Return
S_1	`branch(C role, Adder_S_1_Handler h)`	void
S_1_Handler	`handle(Adder_S_2 s, Add op, Buf<? super Integer> pay1, Buf<? super Integer> pay2)`	void
	`handle(Adder_S_3 s, Bye op)`	void

Fig. 9. Branch handler callback API generation for S in Adder.

each branch state a message handler callback interface with an abstract handler method for each case. Java typing requires the user to implement at least the specified cases, hence this code can be directly reused as an implementation of another branch interface featuring a subset of compatible cases (e.g. if a branch case is deleted from the source protocol specification during development).

For S in the Adder example (Fig. 1 (a)), the API generation for branch handlers generates the method and interface in Fig. 9. The new branch method specifies an additional parameter for the generated Adder_S_1_Handler interface. The user implements this interface by continuing the session in each handle method following the channel type of the first parameter, according to the case that the operator and payload types of the other parameters are received. As before, the new branch method is generated to block until a message is received. The API then calls the handle method for the received operator on the supplied handler object, passing a new instance of the successor state channel for this session endpoint.

This approach essentially reflects the inverse direction of branch subtyping, wrt. select subtyping, in the generated Java types via the inverse control flow of the callback interface. Unlike the branch in § 3, the handler API introduces no additional run-time checks, but requires the user to program in an event-driven style.

A.3 Appendix: SMTP use case

Scribble global protocol for SMTP. The excerpt in Fig. 5 is simplified from the global protocol listed in Fig. 10.

Endpoint FSM for C in Smtp (Fig. 10) is depicted in Fig. 11. The EFSM for the simplified excerpt in Fig. 5 corresponds to states 1–8 without the Quit transitions to the terminal state. The generated channel classes Smtp_C_3 and Smtp_C_7 in Fig. 7 correspond to states 3 and 7 respectively. The I/O state interface Branch_S_250d_250 (Fig. 7), implemented by both these channel classes, is an abstraction of the current state and immediate actions at these two states, parameterised on their continuations.

IDE support for session programming. Fig. 12 shows a screenshot of a implementation of `C` for `Smtp` (Fig. 10) using the generated Endpoint API in Eclipse. There is a protocol error because the input of `250` on line 77 (an `async`) is commented out. This is of course a compile-time error in Java, and reported by Eclipse. This implementation also uses the generated input futures (§ 4, Asynchronous I/O permutations) to safely follow the protocol at this endpoint without actually reading the, e.g., `354` message (this simple implementation is choosing to discard these basic acknowledgements).

A.4 Appendix: Generated Endpoint API Javadoc

Example Javadoc for generated Endpoint API. Fig. 13 shows a screenshot of the API documentation generated by the standard Javadoc tool from the Java Endpoint API generated for `Adder` for `C` (Fig. 3). The automatically generated documentation for an Endpoint API can be read by the user as a target language oriented specification of the source protocol (for the given role), and may often be more concise and clear than common protocol specification formats such as English prose and typically informally used notations such as UML, message sequence charts and BPMN.

```
sig <java> "..." from "..._220.java"
    as 220;
// etc. for 250, 235, 535, 501, ...

sig <java> "..." from "...Ehlo.java"
    as Ehlo;
// etc. for StartTls, Auth, Mail, ...

global protocol SMTP(role S, role C) {
  220 from S to C;
  do Ehlo(S, C);
}

global protocol Ehlo(role S, role C) {
  choice at C {
    Ehlo from C to S;
    rec X {
      choice at S {
        250d from S to C;
        continue X;
      } or {
        250 from S to C;
        do StartTls(S, C);
      } }
  } or {
    Quit from C to S;
} }

global protocol
        StartTls(role S, role C) {
  choice at C {
    StartTls from C to S;
    220 from S to C;
    do SecureEhlo(S, C);
  } or {
    Quit from C to S;
} }

global protocol
        SecureEhlo(role S, role C) {
  choice at C {
    Ehlo from C to S;
    rec X {
      choice at S {
        250d from S to C;
        continue X;
      } or {
        250 from S to C;
        do Auth(S, C);
      } }
  } or {
    Quit from C to S;
} }

global protocol Auth(role S, role C) {
  rec Y {
    choice at C {
      Auth from C to S;
      choice at S {
        235 from S to C;
        do Mail(S, C);
      } or {
        535 from S to C;
        continue Y;
      } or
        ... // 501 Invalid base64 Data, etc.
    } or {
      Quit from C to S;
} } }

global protocol Mail(role S, role C) {
  rec Z1 {
    choice at C {
      Mail from C to S;
      choice at S {
        501 from S to C;
        continue Z1;
      } or {
        250 from S to C;
        rec Z2 {
          choice at C {
            Rcpt from C to S;
            choice at S {
              250 from S to C;
              continue Z2;
            } or
              ...
          } or {
            Data from C to S;
            354 from S to C;
            rec Z3 {
              choice at C {
                DataLine from C to S;
                continue Z3;
              } or {
                Subject from C to S;
                continue Z3;
              } or {
                DataEnd from C to S;
                250 from S to C;
                continue Z1;
      } } } } }
    } or {
      Quit from C to S;
  } }
}
```

Fig. 10. An interoperable subset of SMTP with secure connection establishment and the main mail transaction.

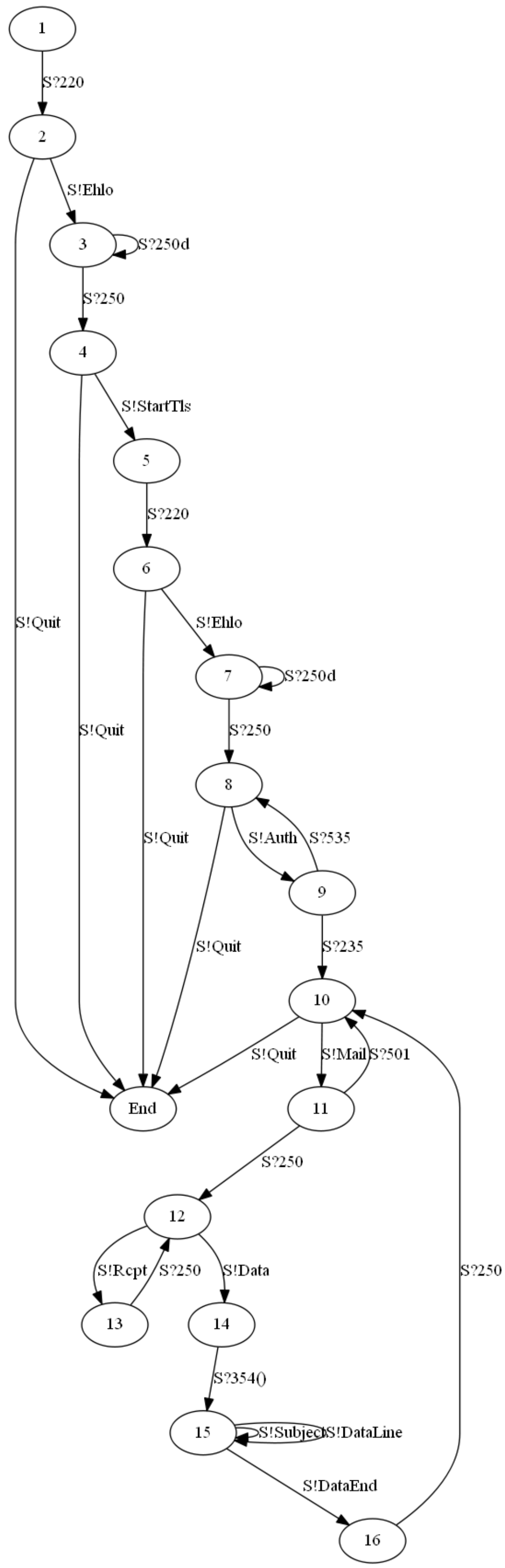

Fig. 11. Endpoint FSM for `C` in `Smtp` (Fig. 10).

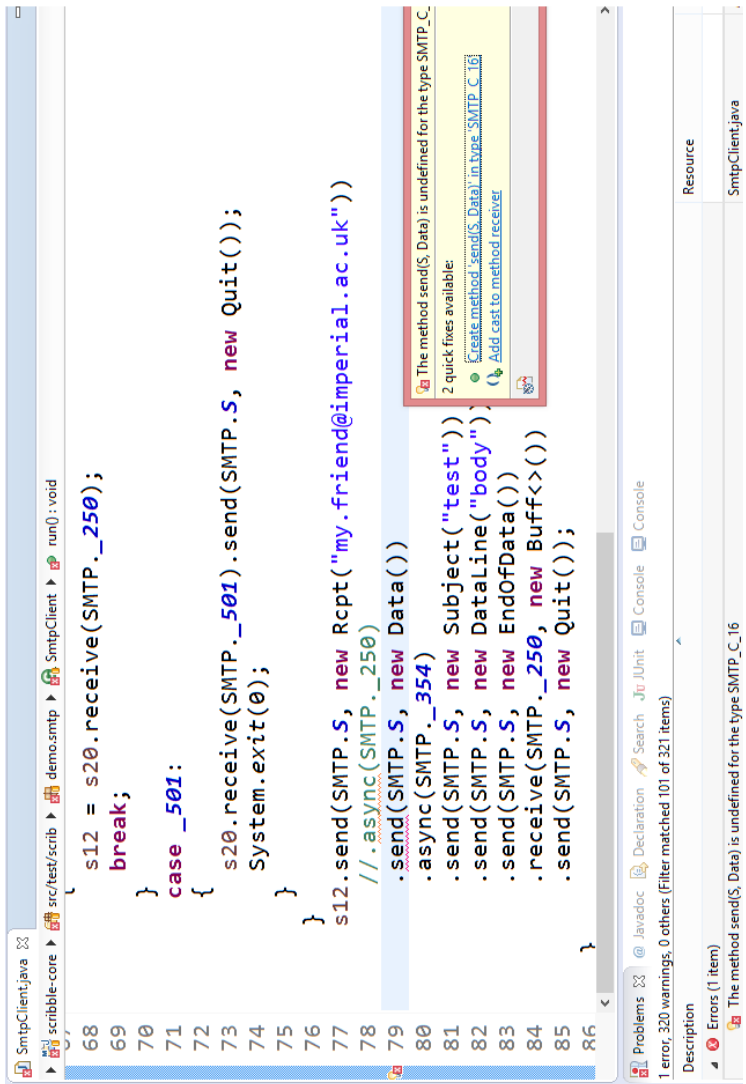

Fig. 12. A statically detected protocol error in an implementation of `C` in `Smtp` (Fig. 10) in Eclipse according to the MPST-generated Endpoint API.

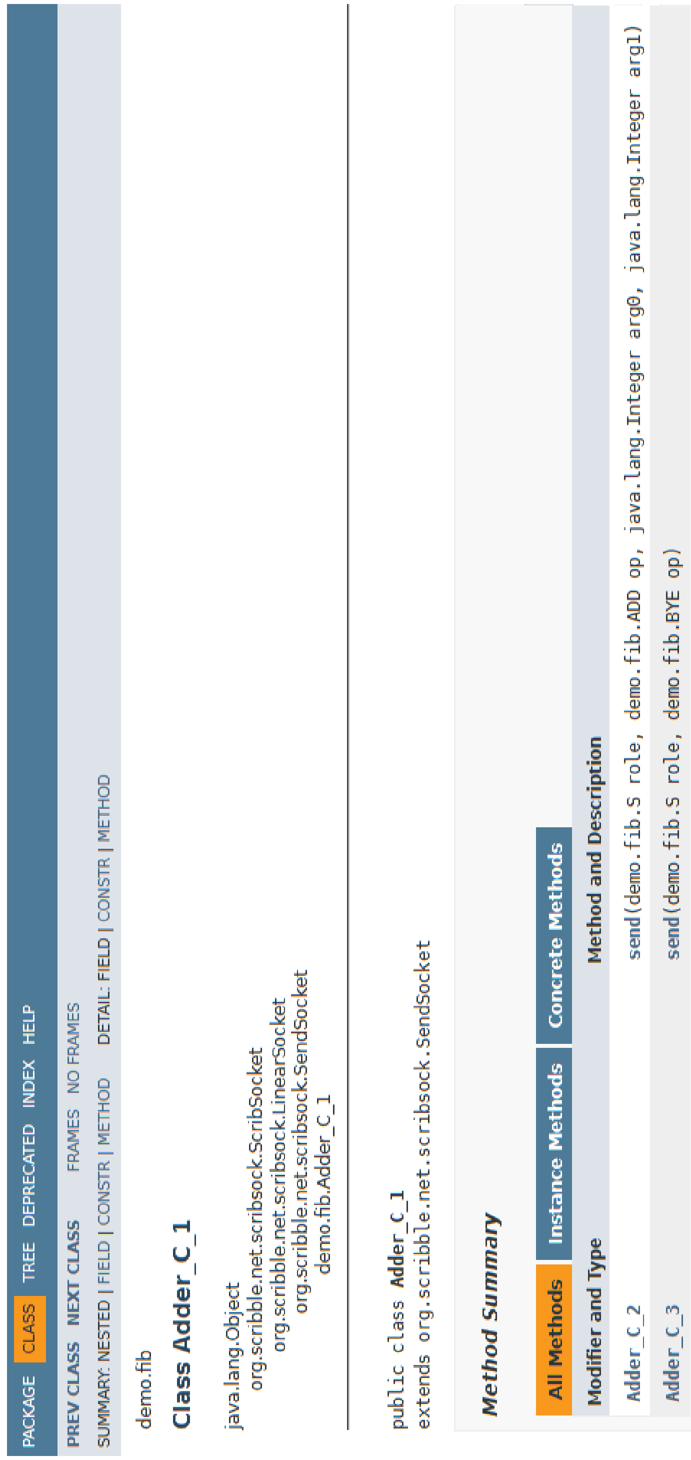

Fig. 13. Javadoc API documentation for the Endpoint API generated for `Adder` for `C` (Fig. 3).