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 implement our methodology for Java as an extension to the Scribble framework, and use it to specify and implement compliant clients and servers for real-world protocols such as HTTP and SMTP.

1 Introduction

Application of session types to practice. Session types [14,15,4] 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 [17,39] 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 [33,40] 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 [8], 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. [30] 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 [12,7,6,2,41] has involved 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 a finite state machine (FSM) representation of the endpoint behaviour in the protocol [10,20], the API generation (i.e. type generation) reifies 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. Our hybrid form of 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 static session type systems with the utility of code generation approaches. First, this methodology allows protocol conformance to be statically checked in mainstream languages like Java, up to the linear channel usage contract of the generated API, by constraining outputs to the specified message types and promoting exhaustive handling of inputs. Second, by directly targetting existing languages, user implementations of session endpoints using generated APIs can be readily integrated with native language features, existing libraries and IDE support.

We present the implementation of our methodology for Java as an extension to Scribble [37], a practical protocol description language based on MPST. Beyond the core safety benefits of regulating session type behaviour through endpoint FSMs, we take advantage of hybrid verification and API generation to support additional practically motivated features for session programming in Java, and to apply further features from session type theory. The former includes value-switched session branching and the abstraction of nominal state channel

```
 1  type <java> "java.lang.Integer" from "rt.jar" as Int;
 2
 3  global protocol Adder(role C, role S) {
 4    choice at C { // Internal choice by C
 5      Add(Int, Int) from C to S; // Message sig: op(payload)
 6      Res(Int) from S to C;
 7      do Adder(C, S); // Recursive protocol def
 8    } or {
 9      Bye() from C to S;
10      Bye() from S to C; // Protocol end
11  } }
```

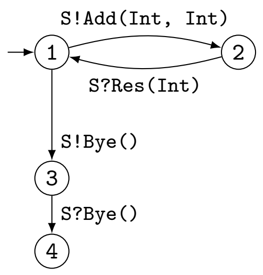

Fig. 1. (a) A Scribble global protocol. (b) The Endpoint FSM for `C`.

types as I/O interfaces. Examples of the latter are the generation of state-specific input futures to support aspects of non-blocking inputs [16], safe permutations of I/O actions [25,3] and affine inputs [33,24]; and the generation of Java subtype hierarchies for I/O interfaces to reflect session subtyping [11]. 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 proposed methodology for hybrid session verification through API generation. § **3** presents our implementation that generates Java endpoint APIs from Scribble protocol specifications. § **4** discusses SMTP as a use case, and practically motivated extensions to the core API generation related to session programming in Java and more advanced session type features. § **5** discusses related work. An extended version of this paper and other resources can be found at `http://www.doc.ic.ac.uk/~rhu/scribble`.

2 Overview

The Scribble toolchain. The Scribble [37,43] framework 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, which encompasses standard Internet applications and Web services that use TCP, HTTP, etc.

Global protocol specification. We use as the 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 3) declares the name of the protocol (`Adder`) and the abstraction of each participant as a named `role` (`C` and `S`). *Payload format types* (line 1) 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 value), and some number of payload types (a pair of `Int`). *Message passing* (e.g. line 5) is output-asynchronous: dispatching the message is non-blocking for the sender (`C`), but the message input is blocking for the receiver (`S`). *Located choice* (e.g. line 4) 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 7) 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 indeed be soundly realised by a system of independent endpoint processes. For instance, in this example, the validation ensures that the two choice cases are 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* 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. Projecting `Adder` for `C` gives: `rec X { choice at C { Add(Int, Int) to S; Res(Int) from S; continue X; } or { Bye() to S; Bye() from S; } }`. A further validation step is performed on each projection of the source protocol for role-sensitive properties, such as reachability of all relevant protocol states 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 FSMs, Scribble can transform the projection of any well-formed protocol for each 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`; `?` dually denotes input.

The core features of the Scribble protocol language are based on and extend those of [5], to which we refer the reader for formal definitions of global and local protocols (i.e. multiparty session types). The global-local projection [5,4] and EFSM transformation [9,20] performed by the Scribble toolchain implement and extend those formalised in the afore-cited works to support the additional features of Scribble (such as located choice, sequencing and subprotocols).

Hybrid session verification through endpoint API generation. This paper proposes a methodology for applying session types to practice that confers 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 Endpoint API generation are:

- The Scribble toolchain is used to validate the source global protocol, project to the local protocol, 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 the 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 state channel of the initial protocol state and performing an I/O operation on each state channel returned by the previous operation, the generated API statically ensures that an endpoint implementation conforms to the encapsulated EFSM and thus observes the protocol. Consequently, the implicit usage contract of the generated API is 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.

Run-time checking of linear state channel usage. Due to the lack of support for statically verifying 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.

Hybrid session safety. Together, a statically typed Endpoint API with run-time state channel linearity checking satisfies the following properties. (1) If state channel linearity is respected by session endpoint implementations, then communication safety (in the sense of e.g. [15, error-freedom]) is statically ensured by the generated API types. (2) Regardless of state channel linearity, any statically type-safe endpoint implementation will never perform a message passing action whose execution trace is not accepted by the EFSM of the generated API.

The latter is because an implementation using an Endpoint API can only attempt a non-conformant messaging action by violating state channel linearity, which the API is generated to guard against. This hybrid form of session verification thus guarantees the absence of protocol violation errors during the execution of a session, up to premature termination (which is always a possibility in practice due to program errors outside of the session code or failures).

3 Hybrid Endpoint API generation for Java

Our implementation of Endpoint API generation for Java takes an Endpoint FSM derived from a Java-based Scribble protocol specification (i.e. a well-formed global protocol with Java-defined payload format types), and outputs two main protocol-specific components, the *Session API* and the *State Channel API*.

Endpoint FSMs (EFSMs) serve as an interface between source protocol validation and projection, and the subsequent API generation. Formally, an EFSM is a tuple $(\mathbb{R}, \mathbb{L}, \mathbb{T}, \Sigma, \mathbb{S}, \delta)$. $\mathbb{R}$ and $\mathbb{L}$ are the sets of role names (ranging over $r, r', ..$) and message operator names $(l, l', ..)$ occurring in the source local protocol, and $\mathbb{T}$ is the set of payload format types $(T, T', ..)$ that it declares. The *alphabet* Σ is a finite set of *actions* $\{\alpha_i\}_{i\in I}$, where α is either an *output* $r!l(\vec{T})$ or an *input* $r?l(\vec{T})$ with $r \in \mathbb{R}$, $l \in \mathbb{L}$ and each $T_i \in \mathbb{T}$. The set of *states* $\mathbb{S}$ is a finite non-empty set of state identifiers ranging over $S, S', ...$ The *transition function* δ is a partial function $\mathbb{S} \times \Sigma \to \mathbb{S}$. We additionally define $\delta(S) = \{\alpha \mid \exists S' \in \mathbb{S}.\delta(S, \alpha) = S'\}$.

Certain properties are guaranteed for any EFSM derived from a well-formed protocol by the Scribble toolchain. (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) There is at most one terminal state $S_{\mathsf{term}} \in \mathbb{S}$ such that $\delta(S_{\mathsf{term}}) = \emptyset$. (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. The generated Endpoint APIs make use of a small collection of protocol-independent base Java classes: `Role`, `Op`, `Session`, `SessionEndpoint` and `Buf`. The first three are abstract classes. We explain them below.

The main class of the Session API (referred to as the Session Class) 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 (i.e. an eagerly initialised singleton pattern). E.g. `public static final C C = new C();`. (2) In the Session Class, a public static final field of type n and with name n that refers to the corresponding field constant in the n class.

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

Session instantiation. As a distributed computing abstraction, a run-time 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 Session Class. The Session object is used by the API 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 2–3 in Fig. 3 (a). The first two constructor arguments are the Session object and the singleton generated for the target role, from which the type parameters are inferred, and the third is an implementation of the Scribble `MessageFormatter` interface for this endpoint using the declared format types for message serialization and deserialization. The `SessionEndpoint` object encapsulates the state specific to this endpoint in the session, such as the local role and networking state.

State Channel API. Based on the aforementioned properties of EFSMs, the core State Channel API is given by generating the channel classes for each EFSM state according to Fig. 2 (a). In the following, we use r, l, etc. to denote both a session type name and its generated Java type (as described above); similarly, we use 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 (ϵ means the empty sequence). 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 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, which is written by the generated `receive` API code when the message is received. Non-unary inputs are explained later (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

State kind	Java state 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`

Gen. class	Session operation methods
`Adder_C_1`	`Adder_C_2 send(S role, Add op, Integer pay1, Integer pay2)`
	`Adder_C_3 send(S role, Bye op)`
`Adder_C_2`	`Adder_C_1 receive(S role, Res op, Buf<? super Integer> pay1)`
`Adder_C_3`	`EndSocket receive(S role, Bye op)`
`Adder_S_1`	`Adder_S_1_Cases branch(C role)`
`Adder_S_1_Cases`	`Adder_S_2 receive(Add op, Buf<? super Integer> pay1, Buf<? super Integer> pay2)`
	`Adder_S_3 receive(Bye op)`
	`Adder_S_1 send(C role, Res op, Integer pay1)`
`Adder_S_3`	`EndSocket send(C role, Bye op)`

Fig. 2. (a) Java state channel class generation. (b) Generated State Channel API for the `C` and `S` roles of `Adder` (using the default channel class naming scheme).

other state channel class is only instantiated internally by the method-chaining API: each session method is generated to return a new instance of the successor state channel. Fig. 2 (b) summarises the channel classes and session I/O methods generated for the `C` and `S` roles of the `Adder` example (Fig.1). The API generation promotes the use of the generated utility types to direct implementations as much as possible. E.g. in `Adder_C_1`, the two output options are distinguished as `send` methods overloaded on the operator type (as well as the payload types).

Hybrid verification of endpoint implementations. Fig. 3 (a) lists an example implementation of `C` using the generated API in Fig. 2 (b).

Session initiation and state channel chaining. Lines 1–5 are a typical preamble. 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` (first argument) as a standard TCP channel (second). The session connection phase is

```
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); // Field i.val stores the buffer value (Integer)
8    for (int j = 0; j < 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_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) { // op enum field set by API according to the received message
    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
} } // Exhaustive handling of enum cases can be generated or checked by an IDE
```

Fig. 3. Examples using the generated APIs from Fig. 2 (b): (a) session initiation and endpoint implementation for `C`, and (b) the main loop and branch of `S`.

concluded when `se` is given as a constructor argument to create an initial state channel of type `Adder_C_1`, to 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 for a fixed N, unfolding the recursion:

```
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 flexibility of the Endpoint API as a native language API is demonstrated by the following Fibonacci client using `Adder` in a different recursive method style.

```
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
```

While the structure of the session code in (a) corresponds quite directly to that of the source protocol, the more obfuscated session control flow here demonstrates the value of the session type based Endpoint API in guiding the implementation and promoting safe protocol conformance. The Java API ensures that the nested `send`-`receive` argument expression safely returns the endpoint to the `S_1` state for each recursive method call, and that the recursion terminates according to the `S_3` return state.

State channel linearity. Linear usage of every session channel object in an endpoint implementation is enforced by inlining run-time checks into the generated Java API following the two cases of the basic approach outlined 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 the `LinearSocket` superclass of all the base channel classes in Fig. 2 (a) (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. 3 (a), line 2), 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 a generated API, taking advantage of fluent method-chaining (e.g. as above), can help avoid linearity bugs by reducing the use of intermediate protocol state variables and state channel aliasing due to assignments.

Session branches. The theoretical languages for which session types were developed typically feature a special-purpose *input branching* primitive, e.g. $c\&(r, \{l_i : P_i\}_{i \in I})$ [5], that atomically inputs a message on a channel c from role r and, according to the received message label l_i, reduces to the corresponding process continuation P_i. For languages like Java that lack such I/O primitives, the API generation approach enables some different options.

The basic option, intended for use in standard `switch` patterns (or `if-else` cases, etc.), separates the branch input action from the subsequent case analysis on the received message operator by generating a pair of `BranchSocket` and `CaseSocket` classes (non-unary inputs in Fig. 2 (a)). 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 } // Generated in Adder_S_1
```

Fig. 3 (b) 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_Enum`, 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 exception if the wrong `receive` is used (like a cast error) thus introducing an additional run-time check to maintain this hybrid form of session type safety. Java IDEs are, however, able to statically check exhaustive enum handling, which could be

```
global protocol Smtp(role C, role S) { // Main protocol decl (start of SMTP)
  220 from S to C; // "220 smtp2.cc.ic.ac.uk ESMTP Exim 4.85 ..."
  do Init(C, S); // First init exchange on plain TCP connection
  do StartTls(C, S); // Negotiate secure connection
  do Init(C, S); // Second init exchange on secure connection
  ... // Remainder of SMTP session over secure connection
}
global protocol Init(role C, role S) { // "Initiation exchange" subprotocol
  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. 4. Simplified excerpt from a Scribble specification of SMTP.

supplemented by developing, e.g., an Eclipse plugin to statically check that the `receive` methods are correctly matched up in basic switch (etc.) patterns.

The alternative option supported by our implementation is the generation of *callback interfaces* for branch states. These confer fully static safety for branch handling, but require the user to program in an event-driven callback style.

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 examples from the SMTP use case, we discuss practically motivated extensions to the core Endpoint API generation methodology presented so far.

SMTP [18] is an Internet standard for email transmission. We have specified a subset of the protocol in Scribble [38] that includes authenticating a secure connection and conducting the main mail transaction. Using the generated Endpoint API, it is straightforward to implement a compliant Java client (e.g. [38]) that is interoperable with existing SMTP servers.

For this section, we use the simplified excerpt from the opening stages of `Smtp` in Fig. 4. On a plain TCP connection, the client (`C`) receives the `220` welcome message from the Server (`S`) and the initiation exchange (client `EHLO`, and the server `250-`/`250` list of service extensions) is performed. The client then starts the negotiation to secure the channel by `StartTls`. Once secured, the client and server perform the initiation exchange again (different service extensions may now be valid), and the remainder of the session is conducted over the secure channel. In this running example, we omit payload types for brevity.

State-specific input futures. There are many works on extending session type theory to support more advanced communication patterns while retaining the desired safety properties. The API generation approach offers a platform for exploring the application of some of these features in practice.

One extension we have implemented to the core API generation is the generation of *state-specific input futures*. For each unary input state, we generate: (1) a subclass of a base `InputFuture` class that performs the input when forced; and (2) an additional `async` method for the `ReceiveSocket` (Fig. 2 (a)) of this state.

$$T_{ret}\ \texttt{async(}r\ \texttt{role,}\ l\ \texttt{op, Buf<? super}\ F_{S?}\texttt{> fut)}$$

The r, l and T_{ret} types are as for the corresponding `receive` method, and $F_{S?}$ is the generated input future class type. In contrast to `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 future is forced, i.e. the input is performed, by a `sync` method, which blocks the caller until the message is received and writes the received payload values to generated fields (e.g. `pay1`) of the future.

Consider the `Ehlo` message in `Init` (Fig. 4) from `C` to `S`, which, in this example, is necessarily preceded by a `220` from `S` to `C` for both occurrences of `Init`. Assuming an initial state channel `s1` of type `Smtp_C_1`, we can implement this exchange at `C` using the input future generated for the `220` (`Smtp_C_1_Future`) by:

```
Buf<Smtp_C_1_Future> buf = new Buf<>(); // For the generated Smtp_C_1 InputFuture
s1.async(S, _220, buf).send(S, Ehlo); // S?220 "postponed"; S!Ehlo done first
String pay1 = buf.val.sync().pay1; // Postponed input done via the Smtp_C_1_Future
```

Calling `sync` on an input future implicitly forces all pending prior futures, in order, for the same peer role. This safely preserves the FIFO messaging semantics between each pair of roles in a session, and endpoint implementations using generated input futures thus retain the same safety properties as implementations using only blocking receives. (With this extension, `receive` is simply generated as `async` and `sync` in one step.) Repeat forcing of an input future has no effect.

Generating input futures captures aspects of several advanced session type features, which we explain by the above example. (1) `async` enables safe *non-blocking input* in session implementations (the key element towards event-driven sessions [16]). `async` essentially allows the input *transition* in the local EFSM to be decoupled in the user program from the actual message input *action* in safe situations. (2) Postponing input actions supports natural communication patterns that exploit asynchronous messaging for *safe permutations* of I/O actions at an endpoint [25,3]. In the example, the input future allows `C` to safely permute the actions: send `Ehlo` first, then receive `220`. (Note the reverse permutation at `S` is unsafe, due to the potential for deadlock by mutual inputs.) (3) Input futures are not linear objects (cf. state channels), so may be discarded unused, treating the input as an *affine* action [33,24]. In session types, input actions are traditionally (e.g. [14,15]) 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 preceding the current future.

Interfaces for abstract I/O states. The SMTP use case raised a practical issue in generating Java State Channel APIs from session types. While formal syntactic session types offer a structural abstraction of communication behaviour by focusing on the I/O actions between implicit protocol states, the API generation reifies these states explicitly as nominal Java types. Nominal channel types can be good for protocol documentation (the default numbering scheme for states can be replaced by a user-supplied mapping to more meaningful class names); this example, however, shows a situation where the nominal types limit code reuse within a session implementation using Endpoint APIs as generated so far. The repeated initiation exchange is factored out in the Scribble as a subprotocol (`Init`), but the two exchanges correspond to distinct parts of the resulting EFSM as a whole, and are thus generated as distinct "unrelated" channel types, preventing this pattern from being factored out in the implementation code.

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 the current example. 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$250` means input of `250` from `S`:

```
interface In_S$250<_S1 extends Succ_In_S$250> { _S1 receive(S role, _250 op); }
```

Each Action Interface is parameterised on a corresponding Successor Interface.

(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.

```
interface Succ_Out_S$Ehlo { // For all I/O States that may succeed an S!Ehlo..
  default Branch_S$250$_250d<?, ?> to(Branch_S$250$_250d<?, ?> c) {
    return (Branch_S$250$_250d<?, ?>) this; // Generated cast
} } // ..i.e. the input branch between 250 and 250d
```

Every Successor Interface is generated with a default `to` "cast" method for each I/O state that implements it: in the above, only `Branch_S$250$_250d` (see next).

(3) For every state, we generate a `Send`, `Receive` or `Branch`/`Case` *I/O State Interface* named according to its session type characterisation, e.g. `Branch_S$250$_250d` is a branch state for the cases of `250` and `250d` from `S` (the action suffixes are ordered lexically). 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.

```
interface Branch_S$250$_250d<_S1 extends Succ_In_S$250,_S2 extends Succ_In_S$250d>
    extends Succ_Out_S$Ehlo, Succ_In_S$250d { // (a) Can succeed S!Ehlo or S?250d
  public static final Branch_S$250d$_250<?, ?> cast = null; // Used for "to" casts
  Case_S$250d$_250<_S1, _S2> branch(S role);
}  // Branch states are generated as a pair of Branch/Case I/O State Interfaces
interface Case_S$250$_250d<_S1 extends Succ_In_S$250, _S2 extends Succ_In_S$250d>
    extends In_S$250<_S1>, In_S$250d<_S2> { ... } // (b) Can do S?250 or S?250d
```

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

```
class Smtp_C_3 implements Branch_S$250$_250d<Smtp_C_4, Smtp_C_3> {..} // Init #1
class Smtp_C_7 implements Branch_S$250$_250d<Smtp_C_8, Smtp_C_7> {..} // Init #2
```

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

Fig. 5. Using the generated I/O Interfaces to factor out the initiation exchange.

The naming scheme for these generated I/O interfaces is not dissimilar to 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`, including the I/O interfaces as above, we factor out one method to implement both initiation exchanges in Fig. 5. The method accepts any state channel with the `Send_S$Ehlo` I/O State Interface and performs the `send`. This returns the Successor Interface `Succ_Out_S$Ehlo`, for which the only I/O State Interface (in this example) is `Branch_S$250$_250d`. Hence the call to the generated `to` on line 2, although operationally a run-time type cast on the state channel reference, is a *safe* cast as 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 first case.

```
doInit( // Second init exchange on secure channel
  doInit(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
```

As `doInit` is implemented using I/O State Interfaces only, it can be reused to perform both initiation exchanges as above. Unfortunately, because the return type of `doInit` is just `Succ_In_S$250`, which may concretely be the state after the first initiation exchange (send `StartTls`) or the second (remainder of session), safety of the immediately subsequent `to` casts relies on the run-time check. However, all `to` casts can in fact be eliminated from both `doInit` and the above by reimplementing `doInit`, leveraging type inference for generics, with the signature:

```
<S1 extends Branch_S$250$_250d<S2, S1>, S2 extends Succ_In_S$250> // S2 is bound..
    S2 doInit(Send_S$Ehlo<S1> s) throws ... //..as the successor of the 250 case
```

5 Related work

Much programming languages research based on session types has been developed in the past decade: see [42] for a 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 are the special case of two-party MPST).

An implementation of static session type checking, following standard presentations [14,15,4], 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 endpoints through the control flow of the program. [17] is an extension of Java for binary session types, and [39] 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 other existing languages.

Other session-based systems that would also require syntax extensions or annotations to be implemented as static typing for most mainstream languages include: Mungo [26] and Bica [13] based on typestates in Java; Links [22,21] and Jolie [19] for Web services; Pabble [31] and ParTypes [23] based on indexed dependent types for parallel programs. We believe our hybrid API generation approach is a practical alternative for applying various forms of behavioural types. Implementations of static session typing in Haskell [35,34] are able to benefit from rich typing features (here, indexed parameterised monads) to ensure session linearity without language extensions, but with various usability tradeoffs. In [27], session code is restricted to a single channel to simplify the treatment of linearity. Outside of API generation, combining static and run-time mechanisms for session safety is being explored in other settings: [32] is an ML library for binary sessions with a focus on type inference, and [36] for actors in Scala.

Dynamic session verification and code generation from session types. Run-time monitoring of I/O actions [8,29,28] is the primary verification method in Scribble [37], and is subject to the common tradeoffs of dynamic verification (§ 1). Monitoring can be applied directly to existing languages, but endpoint implementations must still 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 lightweight hybrid verification approach allows certain benefits of static typing 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).

The code generation framework in [30] (§ 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 final programs. Programming using a generated Endpoint API is amenable to varied user implementations in terms of local control flow style (e.g. imperative or functional) and concurrency (e.g. multithreaded or event-driven) via standard Java language features and existing libraries.

We thank Gary Brown and Matthew Arrott for collaborations, and Julien Lange for comments. This work is partially supported by EPSRC EP/K034413/1, EP/K011715/1, and EP/L00058X/1; and by EU FP7 612985 (UPSCALE).

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 '10*, 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. 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.
5. M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*, 760:1–65, 2015.
6. O. Dardha, E. Giachino, and D. Sangiorgi. Session Types Revisited. In *PPDP '12*, pages 139–150. ACM Press, 2012.
7. 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.
8. 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.
9. P.-M. Deniélou and N. Yoshida. Multiparty Session Types Meet Communicating Automata. In *ESOP '12*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.
10. 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.
11. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
12. S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.
13. S. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL '10*, pages 299–312. ACM, 2010.
14. 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.
15. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL '08*, pages 273–284. ACM, 2008. Full version to appear in *JACM*.
16. 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.
17. R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In *ECOOP '08*, volume 5142, pages 516–541. Springer, 2008.

18. IETF. Simple Mail Transfer Protocol. `https://tools.ietf.org/html/rfc5321`.
19. Jolie homepage. `http://www.jolie-lang.org/`.
20. J. Lange, E. Tuosto, and N. Yoshida. From Communicating Machines to Graphical Choreographies. In *POPL '15*, pages 221–232. ACM Press, 2015.
21. S. Lindley and J. G. Morris. A semantics for propositions as sessions. In *ESOP '15*, volume 9032 of *LNCS*, pages 560–584. Springer, 2015.
22. Links homepage. `http://groups.inf.ed.ac.uk/links/`.
23. 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*, pages 280–298. ACM, 2015.
24. D. Mostrous and V. T. Vasconcelos. Affine sessions. In *COORDINATION '14*, volume 8459 of *LNCS*, pages 115–130. Springer, 2014.
25. D. Mostrous and N. Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Information and Computation*, 241:227–263, 2015.
26. Mungo homepage. `http://www.dcs.gla.ac.uk/research/mungo/`.
27. M. Neubauer and P. Thiemann. An Implementation of Session Types. In *PADL '04*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.
28. R. Neykova, L. Bocchi, and N. Yoshida. Timed Runtime Monitoring for Multiparty Conversations. In *BEAT '14*, volume 162 of *EPTCS*, pages 19–26, 2014.
29. R. Neykova and N. Yoshida. Multiparty Session Actors. In *COORDINATION '14*, volume 8459 of *LNCS*, pages 131–146. Springer, 2014.
30. N. Ng, J. Coutinho, and N. Yoshida. Protocols by Default: Safe MPI Code Generation based on Session Types. In *CC '15*, LNCS, pages 212–232. Springer, 2015.
31. 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.
32. L. Padovani. A Simple Library Implementation of Binary Sessions (Unpublished). `https://hal.archives-ouvertes.fr/hal-01216310`.
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*, pages 25–36. ACM, 2008.
35. M. Sackman and S. Eisenbach. Session Types in Haskell (Unpublished). `http://pubs.doc.ic.ac.uk/session-types-in-haskell/`.
36. A. Scalas and N. Yoshida. Lightweight session types in Scala (Unpublished). `http://www.doc.ic.ac.uk/research/technicalreports/2015/#7`.
37. Scribble homepage. `http://www.scribble.org`.
38. Session types use cases: SMTP (Scribble). `https://github.com/epsrc-abcd/session-types-use-cases/tree/master/Simple%20Mail%20Tranfer%20Protocol/scribble`.
39. K. C. Sivaramakrishnan, K. Nagaraj, L. Ziarek, and P. Eugster. Efficient session type guided distributed interaction. In *COORDINATION '10*, volume 6116 of *LNCS*, pages 152–167. Springer, 2010.
40. 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.
41. P. Wadler. Proposition as Sessions. In *ICFP '12*, pages 273–286, 2012.
42. Survey on languages based on behavioural types. `http://www.di.unito.it/~padovani/BETTY/BETTY_WG3_state_of_art.pdf`.
43. N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble Protocol Language. In *TGC '13*, volume 8358 of *LNCS*, pages 22–41. Springer, 2013.