Explicit Connection Actions in Multiparty Session Types

Raymond Hu[1] and Nobuko Yoshida[1]

Imperial College London

Abstract. This work extends asynchronous multiparty session types (MPST) with *explicit connection actions* to support protocols with optional and dynamic participants. The actions by which endpoints are connected and disconnected are a key element of real-world protocols that is not treated in existing MPST works. In addition, the use cases motivating explicit connections often require a more relaxed form of multiparty choice: these extensions do not satisfy the conservative restrictions used to ensure safety in standard syntactic MPST. Instead, we develop a modelling-based approach to validate MPST safety and progress for these enriched protocols. We present a toolchain implementation, for distributed programming based on our extended MPST in Java, and a core formalism, demonstrating the soundness of our approach. We discuss key implementation issues related to the proposed extensions: a practical treatment of choice subtyping for MPST progress, and multiparty correlation of dynamic binary connections.

1 Introduction

Multiparty session types (MPST) is a type systems theory for verifying message passing concurrent processes, originally developed in the π-calculus [21]. A standard top-down presentation of syntactic MPST systems consists of three layers: (1) a global specification of an asynchronous message passing protocol as a *global type*, with the participants abstracted as *roles*; (2) a syntactic *projection* to a localised view of the protocol for each role as a *local type*; which are in turn used to (3) type check the endpoint *processes* implementing the roles. A well-typed system of session endpoint implementations is guaranteed free from communication safety errors, such as unexpected message receptions and deadlocks.

In our view, the central design point of practical languages and tools based on session types is: (a) to identify a class of protocols, through the constraints of the type syntax and accompanying well-formedness conditions; such that MPST safety is indeed guaranteed by (b) (independent) verification of endpoint programs against their local projections. Much research, both multiparty and the special case of *binary* sessions, has focused on addressing (b) in various ways: extending existing languages to support static session typing (e.g., Links [32]) via pre-processing tools (Java [25,45]), embedding into existing languages via encodings (Haskell [40,26], Rust [27]), dynamic session typing by runtime monitoring (Python [15], Erlang [19]), hybrid (part static, part dynamic) approaches (Java [24], Scala [42], ML [36]), and code generation (MPI/C [35]).

Regarding (a), the multiparty works of the above mostly follow the core theoretical systems [22,12], where protocol well-formedness is directly derived from syntactic restrictions in conjunction with various simplifying assumptions. Unfortunately, these restrictions are too conservative for many useful patterns found in practice. An important example of such a pattern is an interaction between two session participants that, in *some* cases, leads to the later involvement of a third party in the session. By contrast, the standard MPST notion of session initiation is assumed to be a single, atomic synchronisation between *all* parties (as in all of the above works), which inherently rules out any instance of this pattern. Standard MPST basically do not support protocols with *dynamic joining/leaving* of participants during a session, nor *optional* participation.

This paper. We develop an MPST toolchain to address limitations w.r.t. to (a) as discussed above, that can be readily integrated with some of the existing approaches for (b). There are two main contributions.

One is to extend MPST to support *explicit connection actions* in protocol specifications, in a manner that is closely guided by the practical motivations. Rather than a globally interconnected structure between a fixed number of participants, we consider a multiparty session as a dynamically evolving configuration of *binary* bidirectional connections that are established and closed (and possibly re-established) as the session progresses. Concretely, we extend an existing MPST-based protocol description language, Scribble [44,47]. The following is an instance of the pattern from above in our extended Scribble:

```
explicit global protocol OptionalDynamicThirdParty(role A, role B, role C) {
  hello() connect A to B; // A connects to B; sends a message labelled hello
  goodday() from B to A;  // B replies to A on the established connection
  choice at A { opt1() from A to B; // A has two choices: send opt1 or opt2 to B
                greetings() connect B to C; } // B connects to C; sends greetings
           or { opt2() from A to B; } }       // Session ends without involving C
```

(The syntax is explained more in § 2.) Explicit connection actions allow MPST to better fit real-world use cases from domains such as Internet applications and Web services, where multiparty systems are often implemented over binary transports like TCP and HTTP. As we shall see in examples, many patterns involving explicit connections also require a more relaxed form of choice than in standard MPST, with mixed action kinds and destination roles.

The second aspect relates to global type *validation* in our extended MPST. The proposed extensions do not satisfy the conservative restrictions used to ensure safety in standard syntactic MPST: they allow writing additional use cases, but also introduce the potential for errors that were previously precluded.

```
/* Standard MPST: all roles interconnected
 * on session init. (Scribble default) */
global protocol
       P1(role A, role B, role C) {
  choice at A { 1() from A to B; }
           or { 2() from A to C; }
  do P1(A, B, C);
}
```

```
// Explicit connection actions
explicit global protocol
       P2(role A, role B, role C) {
  choice at A { 1() connect A to B;
                disconnect A and B; }
           or { 2() connect A to C;
                disconnect A and C; }
  do P2(A, B, C); }
```

The minimal examples above illustrate some of the issues at hand. P1 features a choice involving only A and B in one case, and A and C in the other (which is not permitted in [22,12,17,18]), that is repeated continuously by the recursion (not permitted in [18]). However, P1 *does* satisfy the intuitive notion of MPST *safety* (e.g., no reception errors or deadlocks); and under an assumption of *output choice fairness*, i.e., provided A does not starve B or C of messages, P1 also satisfies MPST *progress* (otherwise, if, e.g., A talks only to B, then C remains in the session but never progresses). Using explicit connection actions, this pattern can be rewritten in P2 to satisfy both safety and progress *without* such an assumption.

Our approach is to develop a modelling-based validation for MPST protocols. Specifically, we derive a model of a global type from the *1-bounded* execution of the induced multiparty session, i.e., where the capacity of each dynamically established, asynchronous channel is limited to one message; and explicitly check the model is free of the traditional MPST safety and progress errors, as well as the additional kinds of errors introduced by our extensions, such as unexpected or duplicate (dis)connections. The key to this approach is that the characteristics of syntactic MPST can be leveraged to serve the soundness of the bounded validation; as opposed to solely relying on syntactic restrictions for outright safety. We treat output choice fairness by a structural transformation in the model construction, that reflects the underlying issue of session subtyping [37]; e.g., our validation accepts P1 (above) only if fairness is assumed.

Techniques based on "minimal asynchrony" have been employed for various purposes in related theoretical works (§ 5); e.g., to show the decidability of choreography realisability [4], classifying session types in the context of communicating FSMs [18], and the study of properties of half-duplex binary systems [11]. The advance of this work is to formulate the 1-bounded validation for our extended MPST; and its application in a practical toolchain, from the validation of our extended Scribble specifications to safe implementations of distributed Java endpoints. We believe that such an approach may offer a practical, uniform validation methodology for MPST-based protocols, towards incorporating further MPST extensions (e.g., [6,15,5,46,29]) together in an integrated toolchain.

2 Use Case and Overview

2.1 Use Case: Travel Agency Web Service (Revisited)

Travel Agency is one of the widely-used examples in session types literature, based on a W3C Web services choreography use case;[1] we follow the version in [1]. The basic scenario starts by a *Client* (C) initiating a session with the *Travel Agent* (A) to negotiate a product quote. The client may eventually choose to reject all quotes, ending the session; or to accept one, leading to a payment transaction between the client and a third-party *Service* (S). Although this is a natural multiparty use case, it is not actually fully supported by standard MPST. To see the potential problems, consider the following fragment from the latter part of the protocol:

[1] https://www.w3.org/TR/2004/WD-ws-chor-reqs-20040311/ § 3.1.1

```
1  explicit global protocol TravelAgency
2        (role C, role A, role S) {
3    connect C to A;
4    do Nego(C, A, S);
5  }
6  // aux subprotocols
7  aux global protocol Nego
8        (role C, role A, role S) {
9    choice at C {
10     query(Str) from C to A;
11     quote(Int) from A to C;
12     do Nego(C, A, S);
13   } or {
14     do Pay(C, A, S);
15 } }
```

```
17 // So far, only C and A are connected
18 aux global protocol Pay
19       (role C, role A, role S) {
20   choice at C {
21     // C connects to S, sends pay info
22     pay(Str) connect C to S;
23     // S returns a payment reference
24     confirm(Int) from S to C;
25     // C forwards the payref to A
26     accpt(Int) from C to A;
27   } or {
28     reject() from C to A;
29   }
30 } // End of protocol
31
```

Fig. 1. The Travel Agency choreography use case[1] using explicit connection actions.

```
choice at C { pay(Str) from C to S; confirm(Int) S to C; accpt(Int) from C to A;}
          or { reject() from C to A; } // S not involved                          [i]
```

In standard MPST, the execution model is that all three roles are synchronised on session initiation, and there are no further implicit messages (e.g., no session termination handshake). Under these assumptions, the above fragment is unsafe because, in the second case, there is no way for an implementation of S to *locally* determine that the session is finished. Consequently, specifications in existing MPST use workarounds that are less rigorous (e.g., decomposing the protocol into separate global types, losing some of the message causalities) or less realistic/efficient (e.g., by introducing extra messages, or *delegation* [12]).

The above fragment is also not permitted as a standard MPST choice due to the *directed choice* restriction: the messages from a branch point must be sent to the *same* role in all cases (e.g., r' in the type grammar $r \to r' : \{l_i.G_i\}_{i \in I}$ [22,12]; similarly in automata-based works [18,17]) as a conservative element towards ensuring safety. The superficial quick fix by simply moving the accpt message to the start of the first case is not possible in this example, because the Int payload of this message is intended to be the value (the payment reference Int) received by C in the preceding confirm message.

Explicit connection actions allow this use case to be safely captured as a single global type, as given by TravelAgency and its two subprotocols (aux) in our extended Scribble in Fig. 1. Line 1 declares the root protocol with the three roles C, A and S. The new explicit modifier means that every inter-role connection used for message passing must first be established by explicitly specified connection actions. A session starts by C connect to A (line 3), creating a bidirectional channel (e.g., TCP) between client C and server A.

We then enter the Nego subprotocol by the do-statement, with the do argument roles playing the target parameter roles (given the same names in this example). The choice at C on line 9 means C makes an *internal* choice between the two cases (the `or`-separated blocks), to be explicitly communicated as an *external* choice to other roles as appropriate. In the first case, a message of *signature* `query(Str)` (a message with header/label `query`, and one payload value of type `Str`) is sent `from C to A`. `A` replies with a `quote(Int)`, and the choice is repeated by the recursive `do` on line 12. `A` and `C` thus perform the `query`/`quote` exchange some number of times (possibly zero, in this simplified version). Finally, in `Pay`, `C` has two further options. `C` may `connect` to `S`, thereby *dynamically* bringing `S` into the session: `C` exchanges payment details `pay(Str)` for a payment reference `confirm(Int)` with `S`, and forwards the reference to `A`. Otherwise, `C` sends a `reject` to `A`, and the session ends without involving `S`. Note that these syntactically nested choices actually amount to a single choice at `C`, between *mixed* kinds of actions to *different* roles: the connect to `S`, and the sends to `A`.

Extending MPST with explicit connection actions allows such protocols because, e.g., the `connect from C to S`, serves to delimit the scope of `S`'s involvement to the relevant choice case only. From `S`'s view, the *whole* session starts and ends, by interactions with `C`, in this one case, if the session indeed proceeds this way at run-time—while `S` remains unconnected, we can consider it as "inactive" with regards to session safety and progress. At the same time, this solution reduces the gap between MPST-based descriptions and real protocols, like Internet application RFCs, by recognising that the client/server connection actions are as important in a rigorous specification as the message passing (e.g., the `STARTTLS` "re-connection" in SMTP [28], and FTP's active/passive modes [39]).

The communication model promoted by our extended MPST is *at most* one (as opposed to *exactly* one) connection between any pair of roles. Consider the following `explicit` protocol with roles `A`, `B` and `C`:

```
connect A to B; rec X {                                                     [ii]
choice at A { 1() from A to B; 2() connect B to C; disconnect B and C;continue X;
       } or { 3() from A to B; } }
```

The `disconnect` is necessary, inside the *recursion* `rec X { ...continue X; }`, to ensure there is never more than one connection between `B` and `C` (similarly in `P2` in § 1). We can assume implicit `disconnect` actions at the end of a protocol.

2.2 Overview of 1-Bounded Global Type Validation and Examples

The restrictions employed in standard MPST are convenient for reasoning about the MPST safety properties. Aside from surface syntax details, systems like [12] ensure safety by essentially requiring pairwise *syntactic* duality of per-role views at all points in a protocol (called *consistency* [12] or *coherence* [22]). By contrast, our proposed extensions allow additional safe protocols, but also (syntactically) allow protocols with errors that were previously precluded. E.g., consider the choice from `P1` in § 1, where it is safe, but now without the recursion: either `B` or `C` is unsafely left hanging at the end of a session.

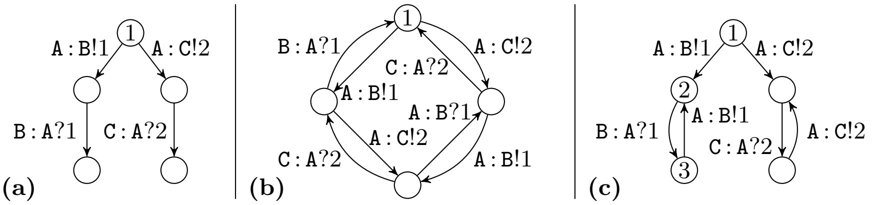
```
choice at A { 1() from A to B; } or { 2() from A to C; }                   [iii]
```

To deal with such additional errors, and those related to explicit connection actions, we validate global types by (1) a lighter set of syntactic conditions, in comparison to standard MPST; complemented by (2) explicit error checking on a *1-bounded model* of the protocol. The key conditions of (1) are:

Role enabling. For any given `choice`, we consider the subject (the `at` role) to be *enabled* by default; other roles become enabled after receiving a message. Only enabled roles may connect or send messages to other roles. Role enabling checks that this transitive propagation of the enabled status is respected.

Consistent external choices. Every potentially incoming message in an *input* choice (i.e., either accept or receive) must be directed *from* the *same* role.

These basic conditions, in conjunction with the inherent pairing of role-to-role actions in global types, serve the soundness of (2) in the presence of asynchrony and recursion (in general, the state space of an MPST protocol may be unbounded; e.g., `P1` in § 1). We note that the latter condition is implicitly imposed by the standard projection in existing MPST [12,22] (and by projections extended with *merging* [46,18]), with the additional restriction of directed choice to *send* every *output* choice message to the same role.



(a) (b) (c)

We first demonstrate the validation by illustrating some models used by our tool for some previous examples; the details will be covered in § 3 and § 4. Initial states are labelled 1; the notation, e.g., `A : B!1` means `A` performs the local send `B!1`. **(a)** is for Ex. [iii]: the two terminals are *unfinished role* errors (§ 3.2), where the system is terminated but either `B` or `C` is not locally terminated. **(b)** is for `P1` from § 1 assuming output choice fairness, i.e., that both the `B!1` and `C!2` options are always viable; this model passes the validation. **(c)** is the contrary view for `P1`, where `A` commits exclusively to a single choice case after the first selection. Our tool additionally constructs this variant to expose such *role progress* violations (§ 3.2), where an unfinished role never progresses along some infinite execution, e.g., `C` does not progress in the cycle between 2 and 3. **(a)** is not affected by the fairness assumption, as there is no recursion.

The "unfair" model for `P2` (not shown) has the same structure as **(c)**, but with connects/disconnects in place of the sends/receives. It would *not* violate progress because either `B` or `C` remains in a local connection-accept "guard" state, which is not considered unfinished (rather, "inactive"). `TravelAgency` satisfies progress (i.e., wrt. `S`) regardless of output choice fairness for the same reason.

3 MPST with Explicit Connection Actions

3.1 Global Types, Local Types and Sessions

Syntax. A core syntax of *global types* G and *local types* L is defined in Fig. 2. Global types have *guarded choices* $\mathbf{\Sigma}_{i\in I}\pi_i.G_i$, with *connection* $r \twoheadrightarrow r'{:}l$, *messaging*

Roles	$A, B, \ldots \in \mathbb{R}$	ranged over by	$r, r', \ldots$
Message labels	$1, 2, 3, \ldots \in \mathbb{L}$		$l, l', \ldots$
Recursion variables	$X, Y, \ldots \in \mathbb{X}$		$X, Y, \ldots$
Paired interactions	$(\mathbb{R} \times \{\rightarrow, \twoheadrightarrow\} \times \mathbb{R} \times \mathbb{L}) \cup (\mathbb{R} \times \{\#\} \times \mathbb{R})$		$\pi, \pi', \ldots$
Localised actions	$\mathbb{A} \subseteq (\mathbb{R} \times \{!, ?, !!, ??\} \times \mathbb{L}) \cup (\mathbb{R} \times \{\#\})$		$\alpha, \alpha', \ldots$

$$G ::= \mathbf{\Sigma}_{i\in I}\pi_i.G_i \mid \mu X.G \mid X \mid \mathsf{end} \qquad L ::= \mathbf{\Sigma}_{i\in I}\alpha_i.L_i \mid \mu X.L \mid X \mid \mathsf{end} \qquad |I| \geq 1$$

$$r_1 \dagger r_2 : l.G \upharpoonright_{\Delta} r = \begin{cases} r_2 \llbracket \dagger \rrbracket_{\bullet} l.(G \upharpoonright_{\emptyset} r) & r = r_1 \\ r_1 \llbracket \dagger \rrbracket_{\circ} l.(G \upharpoonright_{\emptyset} r) & r = r_2 \\ G \upharpoonright_{\Delta} r & r \notin \{r_1, r_2\} \end{cases} \qquad r_1 \# r_2.G \upharpoonright_{\Delta} r = \begin{cases} r'\#.(G \upharpoonright_{\emptyset} r) & r, r' \in \{r_1, r_2\}, r \neq r' \\ G \upharpoonright_{\Delta} r & \text{otherwise} \end{cases}$$

$$\llbracket \rightarrow \rrbracket_{\bullet} = ! \quad \llbracket \rightarrow \rrbracket_{\circ} = ? \quad \llbracket \twoheadrightarrow \rrbracket_{\bullet} = !! \quad \llbracket \twoheadrightarrow \rrbracket_{\circ} = ??$$

$$\underset{(|I|>1)}{\mathbf{\Sigma}_{i\in I} G_i \upharpoonright_{\Delta} r} = \begin{cases} X \quad (\text{resp. } \mathsf{end}) & \forall i \in I.G_i \upharpoonright_{\Delta} r = X \quad (\text{resp. } \forall i \in I.G_i \upharpoonright_{\Delta} r = \mathsf{end}) \\ \mathbf{\Sigma}_{j\in J\subseteq I}(L_j = G_j \upharpoonright_{\Delta} r) & |J| > 0, \forall k \in I \setminus J(G_k \upharpoonright_{\Delta} r = \mathsf{end} \text{ or } X \in \Delta), \text{ and} \\ & \text{either} \begin{cases} \forall j \in J.L_j = \alpha_j^{\bullet}.L'_j \\ \exists r' \forall j \in J.L_j = \alpha_j^{\circ}.L'_j \wedge \mathsf{subj}(\alpha_j^{\circ}) = r' \end{cases} \end{cases}$$

$$\mu X.G \upharpoonright_{\Delta} r = \begin{cases} \mathsf{end} & G \upharpoonright_{\Delta \cup \{X\}} r = X' \text{ or } \mathsf{end} \\ \mu X.(G \upharpoonright_{\Delta \cup \{X\}} r) & \text{otherwise} \end{cases} \qquad \begin{array}{c} X \upharpoonright_{\Delta} r = X \\ \mathsf{end} \upharpoonright_{\Delta} r = \mathsf{end} \end{array}$$

Fig. 2. Core syntax and global-to-local type projection.

$r \rightarrow r' : l$ and *disconnection* $r \# r'$ actions; *recursion* $\mu X.G$ and X; and *termination* end. As an example, `TravelAgency` from Fig. 1 may be written (assuming an empty label `nil` for the initial connect, and "flattening" the nested `choices`):

$$\begin{array}{l} \mathtt{C} \twoheadrightarrow \mathtt{A} : \mathtt{nil}.\mu\mathtt{TravelAgency}.\,(\mathtt{C} \rightarrow \mathtt{A} : \mathtt{query}.\mathtt{A} \rightarrow \mathtt{C} : \mathtt{quote}.\mathtt{TravelAgency} \\ \qquad + \mathtt{C} \twoheadrightarrow \mathtt{S} : \mathtt{pay}.\mathtt{S} \rightarrow \mathtt{C} : \mathtt{confirm}.\mathtt{C} \rightarrow \mathtt{A} : \mathtt{accpt}.\mathsf{end} + \mathtt{C} \rightarrow \mathtt{A} : \mathtt{reject}.\mathsf{end}) \end{array}$$

Local types are the same except for localised actions: *connect* $r!!l$, *accept* $r??l$, *send* $r!l$, *receive* $r?l$, and *disconnect* $r\#$. For a local action $\alpha = r \dagger l$, the annotation α° means $\dagger \in \{?, ??\}$; and $\alpha^{\bullet}$ means either α with $\dagger \in \{!, !!\}$ or an action $r\#$. We sometimes omit end.

The *projection of* G *onto* r, written $G \upharpoonright r$, is the L given by $G \upharpoonright_{\emptyset} r$ in Fig. 2, where the Δ is a set $\{X_i\}_{i\in I}$. Our projection is more "relaxed" than in standard MPST, in that we seek only to regulate some basic conditions to support the later validation (see below). Δ is simply used to prune X that become unguarded in choices during projection onto r, when the recursive path does not involve r; e.g., projecting `TravelAgency` onto `S`: $\mathtt{C}??\mathtt{pay}.\mathtt{C}!\mathtt{confirm}.\mathsf{end}$). Note: this core formulation simplifies and omits certain features of the Scribble implementation, e.g., we omit payload types and flattening of nested choice projections [23].

We assume some basic constraints (typical to MPST) on any given G. **(1)** For all $\pi_{\dagger} = r \dagger r' : l$, $\dagger \in \{\rightarrow, \twoheadrightarrow\}$, and all $\pi_{\#} = r\#r'$, we require $r \neq r'$. We then define: $\mathsf{subj}(\pi_{\dagger}) = \{r\}, \mathsf{obj}(\pi_{\dagger}) = \{r'\}, \mathsf{lab}(\pi_{\dagger}) = l$; and $\mathsf{subj}(\pi_{\#}) = \{r, r'\}, \mathsf{obj}(\pi_{\#}) = \emptyset$. **(2)** G is closed, i.e., has no free recursion variables. **(3)** G features only deterministic choices in its projections. We write: $r \in G$ to mean r occurs in G; and $\alpha \in L$ to mean $L' = \mathbf{\Sigma}_{i\in I}\alpha_i.L_i$, where L' is obtained from L

$$\text{(Sessions)} \quad S ::= (P, Q) \quad P ::= \{L_r\}_{r \in \mathbb{R}} \quad Q : (\mathbb{R} \times \mathbb{R}) \mapsto \{\bot\} \cup \vec{l}$$

$$[\textsc{Conn}] \frac{\exists i' \in I, j' \in J \quad \alpha_{i'} = r'!!l \quad \alpha'_{j'} = r??l \quad Q(r, r') = Q(r', r) = \bot}{(\{(\mathbf{\Sigma}_{i \in I} \alpha_i.L_i)_r, (\mathbf{\Sigma}_{j \in J} \alpha'_j.L'_j)_{r'}\} \cup P, Q) \rightarrow_k (\{(L_{i'})_r, (L'_{j'})_{r'}\} \cup P, Q[r, r' \mapsto \epsilon][r', r \mapsto \epsilon])}$$

$$[\textsc{Send}] \frac{\exists j \in I \quad \alpha_j = r'!l \quad Q(r, r') \neq \bot \quad Q(r', r) = \vec{l} \quad |\vec{l}| < k}{(\{\mathbf{\Sigma}_{i \in I} \alpha_i.L_i\}_r \cup P, Q) \rightarrow_k (\{L_j\}_r \cup P, Q[r', r \mapsto \vec{l} \cdot l])}$$

$$[\textsc{Recv}] \frac{\exists j \in I \quad \alpha_j = r'?l \quad Q(r, r') = l \cdot \vec{l}}{(\{\mathbf{\Sigma}_{i \in I} \alpha_i.L_i\}_r \cup P, Q) \rightarrow_k (\{L_j\}_r \cup P, Q[r, r' \mapsto \vec{l}])}$$

$$[\textsc{Dis}] \frac{Q(r, r') = \epsilon}{(\{r' \#.L\}_r \cup P, Q) \rightarrow_k (\{L\}_r \cup P, Q[r, r' \mapsto \bot])} \qquad [\textsc{Rec}] \frac{(\{L[\mu X.L/X]\}_r \cup P, Q) \rightarrow_k (P', Q')}{(\{\mu X.L\}_r \cup P, Q) \rightarrow_k (P', Q')}$$

Fig. 3. Sessions (pairs of participants and message queues), and session reduction.

by some number (possibly zero) of recursion unfoldings, with $\alpha = \alpha_i$ for some i. (Unfolding is the substitution on recursions $\mathsf{unf}(\mu X.G) = G[\mu X.G/X]$; $\mathsf{unf}(G) = G$ otherwise.) We use $\mathbb{R}_G$ to denote $\{r \mid r \in G\}$, omitting the subscript G where clear from context.

Well-formed global type. For a given G, let $\varphi(G)$ be the global type resulting from the *once-unfolding* of every recursion $\mu X.G'$ occurring within G (defined by $\varphi(\mu X.G) = \varphi(G[\mathsf{end}/X])$, and homomorphic for the other constructors). *Role enabling* (outlined in § 2) on global types $R \vdash G$, $R \subseteq \mathbb{R}$, is defined by $R \vdash \mathsf{end}$ for any R, and:

$$\frac{\mathsf{subj}(\pi) \subseteq R \quad R \cup \mathsf{obj}(\pi) \vdash G}{R \vdash \pi.G} \qquad \frac{|I| > 1 \;\; \exists r \in R \, \forall i \in I. \mathsf{subj}(\pi_i) = \{r\} \wedge \{r\} \cup \mathsf{obj}(\pi_i) \vdash G_i}{R \vdash \mathbf{\Sigma}_{i \in I} \pi_i.G_i}$$

A global type G is *well-formed*, $\mathsf{wf}(G)$, if $\mathbb{R}_G \vdash \varphi(G)$, and for all $r \in \mathbb{R}_G$, $G \upharpoonright r$ *is defined*. A consequence is that disconnects are not prefixes in non-unary choices. Also, every local choice in a projection of a $\mathsf{wf}(G)$ comprises only $\alpha^\bullet$ or α° actions, with a consistent subject r in all cases of the latter.

Sessions (Fig. 3) are pairs of a set of *participant* local types P and inter-role *message queues* Q. $\bot$ designates a *disconnected* queue. We use the notation $Q[K \mapsto V]$ to mean Q' where $Q'(K) = V$, and $Q'(K') = Q(K')$ for $K \neq K'$. *Session reduction* (Fig. 3), $S \rightarrow_k S'$, is parameterised on a maximum queue size $k \in \mathbb{N}_1 \cup \{\omega\}$. If two roles are mutually disconnected, [Conn] establishes a connection, synchronising on a common label l. If both sides are connected, [Send] asynchronously appends a message to destination queue if there is space. If the local queue is still connected: [Recv] consumes the first message, if any; and [Dis] disconnects the queue if it is empty.

For a $\mathsf{wf}(G)$ with roles $\mathbb{R}$, we define: **(1)** $\rightarrow_k^*$ is the reflexive and transitive closure of $\rightarrow_k$; **(2)** the *k-reachable set* of a session S for some k is $RS_k(S) = \{S' \mid S \rightarrow_k^* S'\}$; we say $S' \in RS_k(S)$ *is k-reachable from S*; **(3)** the *initial session* is the session $S_0 = (\{G \upharpoonright r\}_{r \in \mathbb{R}}, Q_{\mathbb{R}0})$, where $Q_{\mathbb{R}0} = \{r, r' \mapsto \bot \mid r, r' \in \mathbb{R}\}$; and

(4) a *k-final session* S is such that $\nexists S'(S \rightarrow_k S')$. We may annotate a reduction step $S \xrightarrow{r}_k S'$ by a *subject role* r of the step: in Fig. 3, in [Send], [Recv] and [Dis] the subject is r; in [Conn], both r and r' are subjects. Given S, r and k, $S \xrightarrow{r}_k$ stands for $\exists S'(S \xrightarrow{r}_k S')$. For $k = \omega$, we often omit the ω.

3.2 MPST Safety and Progress

The following defines MPST safety errors and progress for this formulation. Assume a $\mathsf{wf}(G)$ with initial session S_0 and $S \in RS_k(S_0)$ for some k. For $r \in \mathbb{R}_G$, we say: r *is inactive in* S, where $S = (P, Q)$ and $L_r \in P$, if **(1)** $L_r = \mathsf{end}$; or **(2)** $L_r = G \upharpoonright r = \mathbf{\Sigma}_{i \in I} r'??l_i.L_i$. Otherwise, r *is active in* S.

Then, session $S = (P, Q)$ is a *k-safety error*, k-Err, if:

(i) $L_r \in P$ and any of the following holds:

(*Reception error*) $L_r = \mathbf{\Sigma}_{i \in I} r'?l_i.L_i$, $Q(r, r') = l \cdot \vec{l}$ and $l \notin \{l_i\}_{i \in I}$;
(*Connection error*) r is active in S, $r'??l \in L_r$ and $Q(r, r') \neq \bot$;
(*Disconnect error*) $r'\# \in L_r$ and $Q(r, r') \neq \epsilon$;
(*Unconnected error*) $r'?l \in L_r$ and $Q(r, r') = \bot$;
(*Synchronisation error*) $r'!!l \in L_r$, $(\mathbf{\Sigma}_{i \in I} r??l_i.L_i)_{r'} \in P$, and $l \notin \{l_i\}_{i \in I}$;

or *(ii)* S is either:

(*Orphan message*) $r \in G$ is inactive in S and $\exists r'(Q(r, r') \notin \{\epsilon, \bot\})$;
(*Unfinished role*) S is k-final and $r \in G$ is active in S.

Session S *satisfies k-progress* if, for all $S' = (P, Q) \in RS_k(S)$, we have: (*Role progress*) for all $r \in \mathbb{R}$, if r is active in S', then $S' \rightarrow_k^* \xrightarrow{r}_k$; (*Eventual connection*) if $L_r \in P$ and $r'!!l \in L_r$, then $S' \xrightarrow{\sigma}_k (P', Q')$ where $L_{r'} \in P'$, $r'??l \in L_{r'}$ and $r \notin \mathsf{subj}(\sigma)$; and (*Eventual reception*) $S' \xrightarrow{\sigma}_k (P', Q')$ such that $\forall r, r' \in \mathbb{R}_G.Q'(r, r') \in \{\epsilon, \bot\}$ and $r' \notin \mathsf{subj}(\sigma)$. A session S *is k-safe* if $\nexists k$-$\mathsf{Err} \in RS_k(S)$. We simply say session S *is safe* if it is ω-safe; and S *satisfies progress* if it satisfies ω-progress.

The following establishes the soundness of our framework. Our approach is to adapt the CFSM-based methodology of [18,6], by reworking the notion of *multiparty compatibility* developed there, in terms of our syntactic and explicitly checked 1-bounded conditions. See § B for the remaining definitions and proofs.

Theorem 1. *(Soundness of 1-bounded validation). Let S_0 be the initial session of a $\mathsf{wf}(G)$ that is 1-safe and satisfies 1-progress. Then S_0 is safe and satisfies progress.*

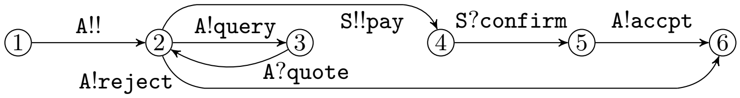
4 Implementation

4.1 Modelling MPSTs by CFSMs with Dynamic Connections

We have developed a prototype implementation [43] that adapts the preceding formulation by constructing and checking explicit state models of our extended global types, based on a correspondence between MPST and communicating FSMs (CFSMs) [18,15,30]. In this setting, our extensions correspond to CFSMs with *dynamic connection actions*. An *Endpoint FSM* (EFSM) for a role is:

(EFSM) $M = (\mathbb{S}, \mathbb{R}, s_0, \mathbb{L}, \delta)$ (States) $s, s', \ldots \in \mathbb{S}$ (Transitions) $\delta \subseteq \mathbb{S} \times \mathbb{A} \times \mathbb{S}$

where s_0 is the *initial state*; $\mathbb{R}$, $\mathbb{L}$ and $\mathbb{A}$ are as defined in Fig. 2. We write $\delta(s)$ to denote $\{\alpha \mid \exists s'.\delta(s,\alpha) = s'\}$. EFSMs are given by a (straightforward) translation from local types, for which we omit the full details [23]: an EFSM essentially captures the structure of the syntactic local type with recursions reflected as cycles. E.g., for `C` in `TravelAgency` (Fig. 1), omitting payload types:



The execution of EFSM systems is adapted from basic CFSMs [9] following Fig. 3 in the expected way [23]. Then, assuming an *initial configuration* c_0 (the system with all endpoints in their initial EFSM states and *unconnected*) for a $\mathsf{wf}(G)$, the (base) *model of* G is the set of configurations that can be reached by *1-bounded* execution from c_0. We remark that the model of a $\mathsf{wf}(G)$ is finite.

Based on § 3.2, G can be validated by its model as follows. The MPST safety errors pertain to individual configurations: this allows to simply check each configuration by adapting the Err-cases to this setting. E.g., an *unfinished role* error is a terminal configuration where role r is in a non-terminal state s_r, and s_r is not an accept-guarded initial state. MPST progress for potentially non-terminating sessions can be characterised on the finite model in terms of closed subsets of mutually reachable configurations (sometimes called *terminal sets*). E.g., a *role progress* violation manifests as such a closure in which an active role is not involved in any transition (e.g., configs. 2 and 3, wrt. `C`, in **(c)** on p. 6).

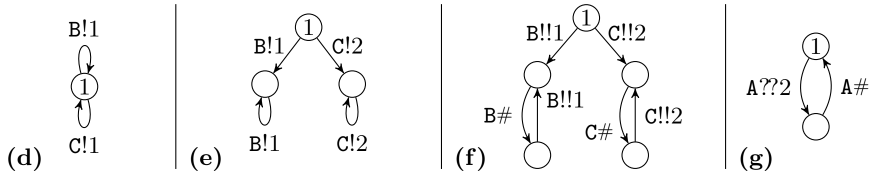
Choice subtyping vs. progress. A projected local choice is either an output choice (connects, sends) or an input choice (accepts, receives). While input choices are driven by the received message, output choices are driven by *process*-level procedures that global and local types abstract from. The notion of *session subtyping* [20,13] was developed to allow more flexible implementations against a local type. E.g., the projection of `P1` from § 1 onto `A` is $\mu X.(\mathtt{B}!\mathtt{1}.X + \mathtt{C}!\mathtt{2}.X)$ which says `A` repeatedly has the choice of sending `1` to `B` or `2` to `C`: intuitively, it is *safe* here to implement an `A` that always opts to send `1` (e.g., a process $P(x) = x \oplus \langle \mathtt{B}, \mathtt{1}\rangle.P\langle x\rangle$, where x is `A`'s session channel, $\oplus$ is the select primitive [12]). For our relaxed form of multiparty choice, however, such an (naive) interpretation of subtyping raises the possibility of *progress* errors (in this case, for `C`).

To allow our validation approach to be integrated with the various methods of verifying local types in mainstream languages, we consider this issue from the perspective of two basic assumptions on implementations of output choices. One is to simply assume *output choice fairness* (the basic interpretation that an infinite execution of an output choice selects each recursive case infinitely many times), which corresponds to the model construction as presented so far.

The other interpretation is developed as a "worst case" view, where we do not assume any direct support for session typing or subtyping (fair or otherwise) in the target language (e.g., native Java), and allow the implementation of every recursive output choice to be reduced to only ever following one particular case. Our tool implements this notion as a transformation on each EFSM, by refining the continuations of output choices such that the *same* case is always selected if that choice is repeated in the future. We outline the transformation below (see [23] for the definition):

– For each non-unary output choice $s^\bullet$, we clone the subgraph reachable via an action $\alpha{\in}\delta(s^\bullet)$ in each case that $s^\bullet$ is reachable via α, i.e., if $s^\bullet{\in}RS(\delta(s^\bullet, \alpha))$.
– In each subgraph cloned via α, all $\alpha'{\in}\delta(s^\bullet)$ edges, s.t. $\alpha'{\neq}\alpha$, are pruned from the clone of $s^\bullet$. We redirect the α-edge from $s^\bullet$ to the clone of its successor $\delta(s^\bullet, \alpha)$ in the cloned subgraph. (States no longer connected are discarded.)
– This transformation is applied recursively on the cloned subgraphs, until every recursive output choice is reduced to a single action.

This transformation reflects endpoint implementations that push output choice subtyping to exercise a *minimum* number of different recursive cases along a path. To expose progress violations under subtyping when fairness is not assumed, our tool uses the transformed EFSMs to additionally construct and check the "unfair" 1-bounded global model in the same manner as above.



We illustrate some examples. **(d)** is the base EFSM, i.e., assuming output choice fairness, for `A` in `P1` from § 1. **(e)** is the transformed EFSM: if `A` starts by selecting the `1` case it will continue to select this case only; similarly for `2`. (The transformation does not change `B` or `C`.) Using **(e)** gives the global model for `P1` in **(c)** on p. 6, raising the role progress violations for `B` and `C`. By contrast, **(f)** is the transformed EFSM for `A` in `P2` from § 1: as in **(e)**, `A` commits exclusively to whichever case is selected first. However, `P2` does not violate progress, despite the transformation of `A` in **(f)**, because the involvement of `C` is guarded by the initial connection-accept actions in **(g)**; similarly for `B`.

4.2 Type-Checking Endpoint Programs by Local Type Projections

Java endpoint implementation via API generation. We demonstrate an integration of the above developments with an existing approach for using local types to verify endpoint programs. Concretely, we extend the approach of [24], to *generate* Java APIs for implementing each role of a global type, including explicit connection actions, via the translation of projections to EFSMs. The idea is to reify each EFSM state as a Java class for a *state-specific* channel, offering methods for exactly the permitted I/O actions. These channel classes are linked by setting the return type of each method to its successor state. Session safety is assured by static (Java) typing of the I/O method calls, combined with run-time checks (built into the API) that each instance of a channel class is used *exactly once*, for the linear aspect of session typing. An endpoint implementation thus proceeds, from a channel instance of the initial state, by calling one I/O method on the current channel to obtain the next, up to the end of the session (if any).

```
 1  TravelAgency sess = new TravelAgency();                   // Generated session class
 2  try (ExplicitEndpoint<TravelAgency, C> ep = new ExplicitEndpoint<>(sess, C) {
 3    Buf<Integer> b = new Buf<>();
 4    TravelAgency_C_2 C2 = new TravelAgency_C_1(ep)       // Generated channel classes
 5        .connect(A, SocketChannelEndpoint::new, host_A, port_A);       // TCP client
 6    for (int i = 0; i < queries.length; i++)             // Assume queries: String[]
 7      C2 = C2.send(A, query, queries[i]).receive(A, quote, b);
 8    C2.connect(S, SocketChannelEndpoint::new, host_S, port_S,         // TCP client
 9          pay, "..paymentInfo..").receive(S, confirm, b)
10      .send(A, accpt, b.val);                // C simplified to always accept the quote
11  }                                                          // (reject option unused)
```

Fig. 4. Safe Java implementation of `C` in `TravelAgency` (Fig. 1) using generated APIs.

Fig. 4 illustrates the incorporation of explicit connect, accept and disconnect actions from local types into the API generated for `C` in `TravelAgency`; this code can be compared against the EFSM on p. 10. `TravelAgency_C_1` is the initial state channel (cf. EFSM state 1), for which the *only* permitted I/O method is the `connect` to `A`; attempting any other session operation is simply a Java type error. (The various constants, such as `A` and `query`, are *singleton type values* in the API.) The `connect` returns a new instance of `TravelAgency_C_2`, offering exactly the mixed choice between the non-blocking `query` (line 7) or `reject` (*unused*, cf. § 4.1, output choice subtyping) to `A`, or the blocking `connect` to `S` (line 8).

If the programmer respects the linear channel usage condition of the generated API, as in Fig. 4, then Java typing statically ensures the session code (I/O actions and message types) follows its local type. The only way to violate the protocol is to violate linearity, in which case the API will raise an exception without actually performing the offending I/O action. Our toolchain, from validated global types to generated APIs, thus assures safe executions of endpoint implementations up to premature termination.

Correlating dynamic binary connections in multiparty sessions. Even aside from explicit connections, session initiation is one aspect in which applications of session type theory, binary and multiparty, to real distributed systems raises some implementation issues. The standard π-calculus theory assumes a so-called *shared channel* used by all the participants for the initiation synchronisation.[2] The formal typing checks, on a "centralised" view of the entire process system, that each and every role is played by a compliant process, initiated via the shared channel. These assumptions transfer to our distributed, binary-connection programs as relying on correct host and port argument values in, e.g., the `connect` calls in `C` in Fig. 4 (lines 5 and 8); similarly for the arguments to the `SocketChannelServer` constructor and `accept` call in the `A` and `S` programs [23].

Existing π-calculus systems could be naively adapted to explicit connection actions by assigning a (binary) shared channel to each accept-point in the session, since the type for any given point in a protocol is fixed. Unfortunately,

[2] E.g., a in $a[1](y).P_1 \,|\, \ldots \,|\, a[n-1](y).P_{n-1} \,|\, \bar{a}[n](y).P_n$, initiating a session between n processes [12].

reusing a shared channel for dynamic accepts across *concurrent* sessions may lead to incorrect *correlation* of the underlying binary connections. E.g., consider $A \twoheadrightarrow B..A \twoheadrightarrow C..B \twoheadrightarrow C..$, where the C process uses multithreading to concurrently serve multiple sessions: if the *same* shared channel is used to accept *all* connections from the `A`'s, and likewise for `B`'s, there is no inherent guarantee that the connection accepted from a `B` by a given server thread will belong to the same *session* as the earlier connection from `A`, despite being of the expected type.

In practice, the correlation of connections to sessions may be handled by various mechanisms, such as passing session identifiers or port values. Consider the version of the `Pay` subprotocol (from Fig. 1), modified to use port passing (cf. FTP [39]), on the left:

```
choice at C { accpt() from C to A;
              connect A to S;
              port(Int) from S to A;
              port(Int) from A to C;
              pay(Str) connect C to S;
              confirm(Int) from S to C;
       } or { reject() from C to A; }
```

```
// Extended Scribble annotations:     [iv]
//   S opens a (fresh) Int port for C
port(p:Int) from S to A; @open=p:C
port(p) from A to C; // A forwards p
pay(Str) connect C to S; @port=p
  // C connects using p as the port
```

`C` sends `accpt` to `A`, and then `A` connects to `S`; `S` sends `A` an `Int` port value, which `A` forwards to `C`; `C` then connects to `S` at that port. To capture this intent explicitly, we adapt an extension of Scribble with assertions [34] to support the specification on the right. In general, *value*-based constraints, like forwarding and connecting to `p`, can be generated into the API as implicit run-time Java assertions. However, we take advantage of the API generation approach to directly generate *statically* safe operations for these actions. *N.B., in the following,* `port` *is simply the message label API constant*; assigning, sending and using the actual port *value* is safely handled *internally* by the generated operations.

```
In S: S.send(A, port).accept(C, pay, b).. // 'port' is the msg label (API const.)
     // API internally opens fresh port p, and sends value; accepts conn. on p
In A: A.receive(S, port).send(C, port)...
     // API internally caches the received value of p; and forwards that value
In C: C.receive(A, port).connect(S, SocketChanEndpoint::new, host_S, pay, "..")..
     // API internaly caches the value of p; and connects using p as the port
```

This combination of explicit connection actions, assertions, and typed API generation is essentially a practical realisation of (private) *shared channel passing* from session π-calculi for our binary connection setting in Java.

To facilitate integration with some existing implementations of session typed languages, our toolchain also supports an optional syntactic restriction on types where: each projection of a Scribble protocol may contain at most one *accept*-choice constructor, and only as the top-most choice constructor (cf. the commonly used replicated-server process primitives in process calculus works). This constraint allows many useful explicit connection action patterns, including nested connects and recursive accepts, while ruling out correlation errors; apart from Ex. [iv], all of the examples in this paper satisfy this constraint.

5 Related Work and Concluding Remarks

Dynamic participants in typed process calculi and message sequence charts. To our knowledge, this paper is the first session types work that allows a single session to have optional roles, and dynamic joining and leaving of roles.

[16] presents a version of session types where a role designates a dynamic *set* of one or more participant processes. Their system does not support optional nor dynamic roles (every role is played by at least one process; the number of processes varies, but the set of *active roles* is fixed). It relies on a special-purpose run-time locking mechanism to block dynamically joining participants until some safe entry point, hindering its use in existing applications. Implementations of sessions in Python [15] and Erlang [19] have used a notion of *subsession* [14] as a coarse-grained mechanism for dynamically introducing participants. The idea is to launch a *separate* child session, by the heavyweight atomic multiparty initiation, involving a subset of the current participants along with other new participants; unlike this paper, where additional roles enter the same, running session by the connect and accept actions between the two relevant participants.

The *conversation calculus* [10] models conversations between dynamic *contexts.* A behavioural typing ensures error-freedom by characterising processes more directly; types do not relate to roles, as in MPST. Their notion of dynamic joining is more abstract (akin to standard MPST initiation), allowing a context n to interact with all other conversation members after a single atomic join action by n; whereas our explicit communication actions are designed to map more closely to concrete operations in standard network APIs.

Dynamic message sequence charts (DMSCs) in [31] support fork-join patterns with potentially unbounded processes. Model checking against a monadic second order logic is decidable, but temporal properties are not studied. [7] studies the implementability of *dynamic communication automata* (DCA) [8] against MSCs as specifications. The focus of study of DCA and DMSCs is more about dynamic *process spawning*; whereas we target dynamic *connections* (and disconnects) between a set of roles with specific concern for MPST safety and progress. Our implementation goes another "layer" down from the automata (i.e., local type) model, applying the validated session types to Java programs with consideration of issues such as choice subtyping and connection correlation.

Well-formedness of session types and choreographies. Various techniques involving bounded executions have been used for multiparty protocols and choreographies. [4,41,3] positions choreography realisability in terms of *synchronisability*, an equivalence between a synchronous global model and the 1-bounded execution of local FSMs; this reflects a stricter perspective of protocol compliance, demanding stronger causality between global steps than session type safety. Their communication model has a single input queue per endpoint, while asynchronous session types has a separate input queue per peer: certain patterns are not synchronisable in the former while valid in the latter. [2] develops more general realisability conditions (in the single-queue model) than the above by determining an upper-bound on queue sizes w.r.t. equivalent behaviours. Our validation of MPST with explicit connection actions remains within a 1-bounded model.

[18] characterises standard MPST w.r.t. CFSMs by *multiparty compatibility*, a well-formedness condition expressed in terms of 1-reachability; it corresponds to the syntactic restrictions of standard MPST in ensuring safety. This paper relaxes some of these restrictions with other extensions, by our 1-bounded validation, to support our use cases. [30] develops a bottom-up synthesis of graphical choreographies from CFSMs via a correspondence between synchronous global models and local CFSMs. These works and the above works on choreographies: (1) do not support patterns with optional or dynamic participants; and (2) study single, pre-connected sessions in isolation without consideration of certain issues of implementing endpoint programs in practice (type checking, subtyping, concurrent connection correlation).

Advanced subtyping of local types with respect to liveness is studied theoretically in [37]. Our present work is based on a coarser-grained treatment of fairness in the global model, to cater for applications to existing (mainstream) languages where it may be difficult to precisely enforce a particular subtyping for sessions via the native type system. We plan to investigate the potential for incorporating their techniques into our approach in future work.

Implementations of session types. The existing version of Scribble [47,24] follows the established theory through syntactic restrictions to ensure safety (e.g., the same set of roles must be involved in every choice case, precluding optional participation). [24] concerns only the use of local types for API generation; it has no formalism, and does not discuss global type validation or projection. This paper is motivated by use cases to relax existing restrictions and add explicit connection actions to types. [38] develops a tool for checking or testing compatibility, adapted from [18], in a local system of abstract concurrent objects. It does not consider global types nor endpoint programs.

Recent implementation works [42,24,32,36,27,35,45,26,40,25], as discussed in § 1, have focused more on applying standard session types, rather than developing session types to better support real use cases. This is in contrast to the range of primarily theoretical extensions (e.g, time [6,33], asynchronous interrupts [15], nested subsessions [14], assertions [5], role parameterisation [46], event handling [29], multi-process roles [16], etc.), which complicates tool implementation because each has its own specific restrictions to treat the subtleties of its setting. The approach of this paper, shifting the emphasis from outright syntactic well-formedness to a more uniform validation of the types, may be one way to help bring some of these scattered features (and those in this paper) together in practical MPST implementations. We plan to investigate such directions in future work, in addition to closer integrations of MPST tools, that treat concepts like role projections, endpoint program typing, subtyping and channel passing, with established model checking tools and optimisations.

Acknowledgements. We thank Gary Brown and Steve Ross-Talbot for collaborations, and Rumyana Neykova and Julien Lange for discussions. This work is partially supported by EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, and EP/N028201/1; and by EU FP7 612985 (UPSCALE).

References

1. D. Ancona et al. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.
2. S. Basu and T. Bultan. Automatic verification of interactions in asynchronous systems with unbounded buffers. In *ASE '14*, pages 743–754. ACM, 2014.
3. S. Basu and T. Bultan. Automated choreography repair. In *FASE '16*, volume 9633 of *LNCS*, pages 13–30. Springer, 2016.
4. S. Basu, T. Bultan, and M. Ouederni. Deciding choreography realizability. In *POPL '12*, pages 191–202. ACM, 2012.
5. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR '10*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
6. L. Bocchi, J. Lange, and N. Yoshida. Meeting deadlines together. In *CONCUR '15*, volume 42 of *LIPIcs*, pages 283–296. Schloss Dagstuhl, 2015.
7. B. Bollig, A. Cyriac, L. Hélouët, A. Kara, and T. Schwentick. Dynamic communicating automata and branching high-level MSCs. In *LATA '13*, volume 7810 of *LNCS*, pages 177–189. Springer, 2013.
8. B. Bollig and L. Hélouët. Realizability of dynamic MSC languages. In *CSR '10*, volume 6072 of *LNCS*, pages 48–59. Springer, 2010.
9. D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30:323–342, April 1983.
10. L. Caires and H. T. Vieira. Conversation types. *Theor. Comput. Sci.*, 411(51-52):4399–4440, 2010.
11. G. Cécé and A. Finkel. Verification of programs with half-duplex communication. *Inf. Comput.*, 202(2):166–190, 2005.
12. 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.
13. 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.
14. R. Demangeon and K. Honda. Nested protocols in session types. In *CONCUR '12*, volume 7454 of *LNCS*, pages 272–286. Springer, 2012.
15. 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.
16. P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *POPL '11*, pages 435–446. ACM, 2011.
17. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP '12*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.
18. 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.
19. S. Fowler. An erlang implementation of multiparty session actors. In *ICE '16*, volume 223 of *EPTCS*, pages 36–50, 2016.
20. S. Gay and M. Hole. Subtyping for session types in the pi-calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
21. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL '08*, pages 273–284. ACM, 2008.

22. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016.
23. R. Hu and N. Yoshida. Explicit Connection Actions in Multiparty Session Types (Long Version). `https://www.doc.ic.ac.uk/~rhu/scribble/explicit.html`.
24. R. Hu and N. Yoshida. Hybrid session verification through endpoint API generation. In *FASE '16*, volume 9633 of *LNCS*, pages 401–418. Springer, 2016.
25. R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In *ECOOP '08*, volume 5142, pages 516–541. Springer, 2008.
26. K. Imai, S. Yuen, and K. Agusa. Session type inference in haskell. In *PLACES*, volume 69 of *EPTCS*, pages 74–91, 2010.
27. T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. Session types for rust. In *WGP '15*, pages 13–22. ACM, 2015.
28. J. Klensin. IETF RFC 5321 Simple Mail Transfer Protocol. `https://tools.ietf.org/html/rfc5321`.
29. D. Kouzapas, N. Yoshida, R. Hu, and K. Honda. On asynchronous eventful session semantics. *Mathematical Structures in Computer Science*, 26(2):303–364, 2016.
30. J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In *POPL '15*, pages 221–232. ACM, 2015.
31. M. Leucker, P. Madhusudan, and S. Mukhopadhyay. Dynamic message sequence charts. In *FSTTCS '02*, volume 2556 of *LNCS*, pages 253–264. Springer, 2002.
32. S. Lindley and J. G. Morris. Lightweight Functional Session Types. `http://homepages.inf.ed.ac.uk/slindley/papers/fst-draft-february2015.pdf`.
33. R. Neykova, L. Bocchi, and N. Yoshida. Timed runtime monitoring for multiparty conversations. In *BEAT '14*, volume 162 of *EPTCS*, pages 19–26, 2014.
34. R. Neykova, N. Yoshida, and R. Hu. SPY: Local verification of global protocols. In *RV '13*, volume 8174 of *LNCS*, pages 363–358. Springer, 2013.
35. 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.
36. L. Padovani. FuSe homepage. `http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html`.
37. L. Padovani. Fair subtyping for multi-party session types. *Mathematical Structures in Computer Science*, 26(3):424–464, 2016.
38. R. Perera, J. Lange, and S. J. Gay. Multiparty compatibility for concurrent objects. In *PLACES '16*, volume 211 of *EPTCS*, pages 73–82, 2016.
39. J. Postel and J. Reynolds. IETF RFC 959 File Transfer Protocol. `https://tools.ietf.org/html/rfc959`.
40. R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In *Haskell '08*, pages 25–36. ACM, 2008.
41. G. Salaün, T. Bultan, and N. Roohi. Realizability of choreographies using process algebra encodings. *IEEE Trans. Services Computing*, 5(3):290–304, 2012.
42. A. Scalas and N. Yoshida. Lightweight session programming in scala. In *ECOOP '16*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl, 2016.
43. Scribble. GitHub repository. `https://github.com/scribble/scribble-java`.
44. Scribble homepage. `http://www.scribble.org`.
45. K. C. Sivaramakrishnan, M. Qudeisat, L. Ziarek, K. Nagaraj, and P. Eugster. Efficient sessions. *Sci. Comput. Program.*, 78(2):147–167, 2013.
46. N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In *FoSSaCs' 10*, volume 6014 of *LNCS*, pages 128–145. Springer, 2010.
47. 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.

A Appendix: § 2

A.1 Local Type Projections for Travel Agency

The global type (from § 3.1) is:

$$
\begin{aligned}
G = \mathtt{C} \twoheadrightarrow \mathtt{A} : \mathtt{nil}.\mu\mathtt{TravelAgency}.\ (\ & \mathtt{C} \to \mathtt{A} : \mathtt{query}.\mathtt{A} \to \mathtt{C} : \mathtt{quote}.\mathtt{TravelAgency} \\
+ \ & \mathtt{C} \twoheadrightarrow \mathtt{S} : \mathtt{pay}.\mathtt{S} \to \mathtt{C} : \mathtt{confirm}.\mathtt{C} \to \mathtt{A} : \mathtt{accpt}.\mathsf{end} \\
+ \ & \mathtt{C} \to \mathtt{A} : \mathtt{reject}.\mathsf{end}\ \ )
\end{aligned}
$$

Projection onto $\mathtt{C}$, $G \upharpoonright \mathtt{C}$:

$$
\begin{aligned}
\mathtt{A}!!\mathtt{nil}.\mu\mathtt{TravelAgency}.\ (\ & \mathtt{A}!\mathtt{query}.\mathtt{A}?\mathtt{quote}.\mathtt{TravelAgency} \\
+ \ & \mathtt{S}!!\mathtt{pay}.\mathtt{C}?\mathtt{confirm}.\mathtt{A}!\mathtt{accpt}.\mathsf{end} \\
+ \ & \mathtt{A}!\mathtt{reject}.\mathsf{end}\ \ )
\end{aligned}
$$

Projection onto $\mathtt{A}$, $G \upharpoonright \mathtt{A}$:

$$
\begin{aligned}
\mathtt{C}??\mathtt{nil}.\mu\mathtt{TravelAgency}.\ (\ & \mathtt{C}?\mathtt{query}.\mathtt{C}!\mathtt{quote}.\mathtt{TravelAgency} \\
+ \ & \mathtt{C}?\mathtt{accpt}.\mathsf{end} \\
+ \ & \mathtt{C}?\mathtt{reject}.\mathsf{end}\ \ )
\end{aligned}
$$

Projection onto $\mathtt{S}$, $G \upharpoonright \mathtt{S}$:

$$\mathtt{C}??\mathtt{pay}.\mathtt{C}!\mathtt{confirm}.\mathsf{end}$$

B Appendix: § 3

B.1 1-bounded Multiparty Compatibility with Explicit Connection Actions

Definition B1. (*Session actions, action sequences and alternations*).

- *(Session actions)* $t ::= r : r'!l \mid r : r'?l \mid r \twoheadrightarrow r' : l \mid r : r'\#$

  We define: $\mathsf{subj}(r : r' \dagger l) = \{r\}$, for $\dagger \in \{!, ?\}$; $\mathsf{subj}(r \twoheadrightarrow r' : l) = \{r, r'\}$; $\mathsf{subj}(r : r'\#) = \{r\}$.
- *(Annotated reduction)* We may annotate session reduction (Fig. 3) by actions, writing $S \xrightarrow{t}_k S'$, as follows: [Conn] $S \xrightarrow{r \twoheadrightarrow r' : l}_k S'$; [Send] $S \xrightarrow{r : r'!l}_k S'$; [Recv] $S \xrightarrow{r : r'?l}_k S'$; [Dis] $S \xrightarrow{r : r'\#}_k S'$; and [Rec] $(\{L[\mu X.L/X]\}_r \cup P, Q) \xrightarrow{t}_k (P', Q') \Rightarrow (\{\mu X.L\}_r \cup P, Q) \xrightarrow{t}_k (P', Q')$.
- *(Session action sequences)* We write σ to denote a sequence of actions $\vec{t}$. We write ϵ for the empty sequence.

  We write $S \xrightarrow{\sigma}_k S'$, where $\sigma = t_{1..n}$, if $S \xrightarrow{t_1}_k \ldots \xrightarrow{t_n}_k S'$. We write $t_j \in (t_i)_{1..n}$ when $j \in 1..n$, and define $\mathsf{subj}(\sigma) = \{r \mid t \in \sigma \wedge r \in \mathsf{subj}(t)\}$.
- *(Alternations)* An *alternation* is a sequence $(t_i)_{1..n}$ such that $t_i = r : r'!l \Rightarrow t_{i+1} = r' : r?l$ for $i \in 1..n-1$, and $t_n \neq r : r'!l$.

Proposition B2. (*Basic syntactic properties*). Assume a $\mathsf{wfg}(G)$. Let $\mathcal{I} = \{?, ??\}$ and $\mathcal{O} = \{!, !!\}$. For all $r \in \mathbb{R}_G$, every occurrence of a local choice $\mathbf{\Sigma}_{i \in I} \alpha_i.L_i$ in $G \upharpoonright r$ is such that:

(i) for $i, j \in I$, if $\alpha_i = r_i \mathcal{O}_i l_i$ and $i \neq j$, then $\alpha_j = r_j \mathcal{O}_j l_j$, and $l_i \neq l_j$;
(ii) for $i, j \in I$, if $\alpha_i = r' \mathcal{I} l_i$ and $i \neq j$, then $\alpha_j = r' \mathcal{I} l_j$ and $l_i \neq l_j$;
(iii) if $\alpha_1 = r'\#$, then $|I| = 1$.

Definition B3. (*One-step diamond property*). Let S be a session such that $S \xrightarrow{t_1}_1 S_1$ and $S \xrightarrow{t_2}_1 S_2$, $t_1 \neq t_2$. *S satisfies the one-step diamond property w.r.t. t_1 and t_2*, if there exists S' such that $S_1 \xrightarrow{t_2}_1 S'$ and $S_2 \xrightarrow{t_1}_1 S'$.

We may simply say *S satisfies 1-diamond w.r.t. t_1 and t_2*, for short.

Lemma B4. Assume a $\mathsf{wf}(G)$ with initial session S_0, and $S_0 \to_1^* S$ such that $S \xrightarrow{t_1}_1 S_1$ and $S \xrightarrow{t_2}_1 S_2$. If $t_1 = t_2$ then $S_1 = S_2$. Otherwise:

1. if $t_1 = r_1 : r'_1 ! l_1$, then one of the following holds:
   (i) $t_2 = r_2 : r'_2 ! l_2$ and either:
      (a) $r_1 = r_2$ and ($r'_1 \neq r'_2$ or $l_1 \neq l_2$);
      (b) $r_1 \neq r_2$ and S satisfies 1-diamond w.r.t. t_1 and t_2.
   (ii) $t_2 = r_2 : r'_2 ? l_2$ and:
      $r_1 \neq r_2$, ($r_1 \neq r'_2$ or $r'_1 \neq r_2$), and S satisfies 1-diamond w.r.t. t_1 and t_2.
   (iii) $t_2 = r_2 \twoheadrightarrow r'_2 : l_2$ and either:
      (a) $r_1 = r_2$ and $r'_1 \neq r'_2$;
      (b) $r_1 \neq r_2$, $r_1 \neq r'_2$ and S satisfies 1-diamond w.r.t. t_1 and t_2.
   (iv) $t_2 = r_2 : r'_2 \#$ and either:
      (a) $r_1 = r'_2$ and $r_2 = r'_1$;
      (b) $r_1 \neq r_2$, ($r_1 \neq r'_2$ or $r'_1 \neq r_2$) and S satisfies 1-diamond w.r.t. t_1 and t_2.
2. If $t_1 = r_1 : r'_1 ? l_1$, then one of the following holds:
   (i) $t_2 = r_2 : r'_2 ! l_2$ and case 1(ii) holds.
   (ii) $t_2 = r_2 : r'_2 ? l_2$ and:
      $r_1 \neq r_2$ and S satisfies 1-diamond w.r.t. t_1 and r_2.
   (iii) $t_2 = r_2 \twoheadrightarrow r'_2 : l_2$ and:
      $r_1 \notin \{r_2, r'_2\}$ and S satisfies 1-diamond w.r.t. t_1 and t_2.
   (iv) $t_2 = r_2 : r'_2 \#$ and:
      $r_1 \neq r_2$ and S satisfies 1-diamond w.r.t. t_1 and t_2.
3. If $t_1 = r_1 \twoheadrightarrow r'_1 : l_1$, then one of the following holds:
   (i) $t_2 = r_2 : r'_2 ! l_2$ and case 1(iii) holds.
   (ii) $t_2 = r_2 : r'_2 ? l_2$ and case 2(iii) holds.
   (iii) $t_2 = r_2 \twoheadrightarrow r'_2 : l_2$ and either:
      (a) $r_1 = r_2$ and ($r'_1 \neq r'_2$ or $l_1 \neq l_2$);
      (b) $r_1 \notin \{r_2, r'_2\}$ and S satisfies 1-diamond w.r.t. t_1 and t_2.
   (iv) $t_2 = r_2 : r'_2 \#$ and:
      $r_2 \notin \{r_1, r'_1\}$ and S satisfies 1-diamond w.r.t. t_1.

Proof. By cases, following the definition of session reduction (Fig. 3) and using Prop. B2. □

Definition B5. (*1-bounded causality*). We write $t_1 \lhd t_2$ (t_2 *depends on* t_1) if one of the following holds:

(i) $\mathsf{subj}(t_1) \subseteq \mathsf{subj}(t_2)$;
(ii) $t_1 = r_1 : r_2 ! l$ and $t_2 = r_2 : r_1 ? l'$;
(iii) $t_1 = r_1 : r_2 ? l$ and $t_2 = r_2 : r_1 ! l'$.

We write $t_1 \not\lhd t_2$ otherwise.

Lemma B6. (*1-bounded permutation*). Assume a $\mathsf{wf}(G)$ with initial session S_0. Let $S \in RS_1(S_0)$, and $S \xrightarrow{t_1}_1 \xrightarrow{t_2}_1 S'$ where $t_1 \not\lhd t_2$. Then $S \xrightarrow{t_2}_1 \xrightarrow{t_1}_1 S'$.

Proof. By cases on t_1 and t_2, using Lem. B4. □

Definition B7. (*Alternation of an action sequence*). We say σ *is an alternation of* σ' if σ is an alternation that can be derived from σ' by zero or more applications of Lem. B6.

Proposition B8. If $\sigma = \sigma_1 \cdot t \cdot \sigma_2$, $r \in \mathsf{subj}(t)$, $r \notin \mathsf{subj}(\sigma_1)$ and $S \xrightarrow{\sigma'}_1 S'$ where σ' is an alternation of σ, then $\sigma' = \sigma'_1 \cdot t \cdot \sigma'_2$ and $r \notin \mathsf{subj}(\sigma'_1)$.

Definition B9. (*Stable session*). For a given G, we say a session (P, Q) is *stable* if $\forall r, r' \in \mathbb{R}_G . Q(r, r') \in \{\epsilon, \bot\}$.

Lemma B10. (*1-bounded globally-paired interactions*). Assume a $\mathsf{wf}(G)$ with initial session S_0 that is 1-safe and satisfies 1-progress. Let $S_0 \rightarrow_1^* S \xrightarrow{\sigma}_1 S'$ where S and S' are stable. If $\sigma \neq \epsilon$, then:

(i) there exist σ', σ'' and t_1 where $\sigma = \sigma' \cdot t_1 \cdot \sigma''$, and one of the following holds:
  - there exist $\sigma''_{1,2}$ and t_2 where $\sigma'' = \sigma''_1 \cdot t_2 \cdot \sigma''_2$ such that (1) $t_1 = r : r' ! l$ where $\nexists t \in \sigma' . t \lhd t_1$, and (2) $t_2 = r' : r ? l$ where $\nexists t \in \sigma' \cdot \sigma'' . t \lhd t_2$;
  - $t_1 = r \twoheadrightarrow r' : l$ and $r, r' \notin \mathsf{subj}(\sigma')$;
  - $t_1 = r : r' \#$ and $r \notin \mathsf{subj}(\sigma')$.

(ii) $S \xrightarrow{\sigma'}_1 S'$ where σ' is an alternation of σ;

We extend the CFSM-based *multiparty compatibility* well-formedness condition from [18,6] to our setting with explicit connection actions.

Definition B11. (*Multiparty compatibility*). A session S *satisfies compatibility* if, for all $S' = (P', Q') \in RS_1(S)$ such that S' is stable and $L_r \in P'$, the following hold:

1. if $r' ! l {\in} L_r$, then there exists $t{=}r' : r ? l$ and an alternation σ such that $S_1 \xrightarrow{\sigma \cdot r : r' ! l \cdot t}_1$ and $r \notin \mathsf{subj}(\sigma)$;
2. if $r' ? l \in L_r$, then there exists $r' ? l' \in L_r$, $t = r : r' ? l'$ and an alternation $\sigma \cdot t$ such that $S' \xrightarrow{\sigma \cdot t}_1$ and $r \notin \mathsf{subj}(\sigma)$;
3. if $r' !! l \in L_r$, then there exists $t = r \twoheadrightarrow r' : l$ and an alternation σ such that $S' \xrightarrow{\sigma \cdot t}_1$ and $r \notin \mathsf{subj}(\sigma)$;

4. if r is active in S and $r'??l \in L_r$, then there exists $r'??l' \in L_r$, $t = r' \twoheadrightarrow r : l'$ and an alternation σ such that $S' \xrightarrow{\sigma \cdot t}_1$ and $r \notin \mathsf{subj}(\sigma)$;
5. if $r'\,\# \in L_r$, then $S' \xrightarrow{r:r'\#}_1$.

The following establishes compatibility of a well-formed initial session from the explicitly checked 1-bounded properties.

Lemma B12. (*Scribble endpoint compatibility*). Assume a $\mathsf{wf}(G)$ with initial session S_0 that is 1-safe and satisfies 1-progress. Then S_0 *satisfies compatibility.*

Proof. Let $S_0 \to_1^* S = (P, Q)$ where S is stable. Note that S is 1-safe and satisfies 1-progress.

Case 1 (final). S is final. By (role progress), for all $r \in \mathbb{R}$, r is inactive in S. By Def. B11, S trivially satisfies compatibility.

Case 2 (send). S is not final, $L_r \in P$ and $r'!l \in L_r$. By [Send], $S \xrightarrow{r:r'!l}_1 S_1$ where $Q_1(r', r) = l$. By (eventual reception), $S_1 \xrightarrow{\sigma}_1 S_2$ where $\sigma = \sigma_1 \cdot r' : r?l \cdot \sigma_2$, $r \notin \mathsf{subj}(\sigma)$ and S_2 is stable. By Lem. B10(ii) and Prop. B8, $S \xrightarrow{\sigma'}_1 S_2$ where $\sigma' = \sigma_1' \cdot r : r'!l \cdot \sigma_2'$ is an alternation of $r : r'!l \cdot \sigma$ and $r \notin \mathsf{subj}(\sigma_1')$. Thus, $S \xrightarrow{\sigma_1' \cdot r : r'!l \cdot r' : r?l}_1$.

Case 3 (receive). S is not final, $L_r \in P$ and $r'?l \in L_r$. By (role progress), we have $S \xrightarrow{\sigma}_1 S_1 \xrightarrow{t}_1$ and $r \in \mathsf{subj}(t)$. Let $r \notin \mathsf{subj}(\sigma)$. By Prop. B2, $t = r : r'?l'$. By [Recv], $Q_1(r, r') = l'$. By (eventual reception), $S_1 \xrightarrow{\sigma'}_1 S_2$ where $\sigma' = \sigma_1' \cdot r : r'?l' \cdot \sigma_2'$ and S_2 is stable. By Prop. B2, let $r \notin \sigma_1'$. By Lem. B10(ii) and Prop. B8, $S \xrightarrow{\sigma''}_1 S_2$ where $\sigma'' = \sigma_1'' \cdot r : r'?l'' \cdot \sigma_2''$ is an alternation of $\sigma \cdot \sigma'$ and $r \notin \mathsf{subj}(\sigma_1'')$. Thus, $S \xrightarrow{\sigma_1'' \cdot r : r'?l''}_1$ where $\sigma_1'' \cdot r' : r?l''$ is an alternation.

Case 4 (connect). S is not final, $L_r \in P$ and $r'!!l \in L_r$. By (eventual connection), we have $S \xrightarrow{\sigma \cdot r \twoheadrightarrow r' : l}_1 S_1$ and $r \notin \mathsf{subj}(\sigma)$. By (eventual reception), $S_1 \xrightarrow{\sigma'}_1 S_2$ where S_2 is stable. By Lem. B10(ii) and Prop. B8, $S \xrightarrow{\sigma''}_1 S_2$ where $\sigma'' = \sigma_1'' \cdot r \twoheadrightarrow r' : l'' \cdot \sigma_2''$ is an alternation of $\sigma \cdot r \twoheadrightarrow r' : l \cdot \sigma'$ and $r \notin \mathsf{subj}(\sigma_1'')$. Thus, $S \xrightarrow{\sigma_1'' \cdot r \twoheadrightarrow r' : l}_1$ where $\sigma_1'' \cdot r \twoheadrightarrow r' : l$ is an alternation.

Case 5 (accept). S is not final, $L_r \in P$, L_r is active in S and $r'??l \in L_r$. By (role progress), we have $S \xrightarrow{\sigma}_1 S_1 \xrightarrow{t}_1 S_2$ and $r \in \mathsf{subj}(t)$. Let $r \notin \mathsf{subj}(\sigma)$. By Prop. B2, $t = r' \twoheadrightarrow r : l'$.

Case 5-1. S_2 is stable. By Lem. B10(ii) and Prop. B8, $S \xrightarrow{\sigma'}_1 S_2$ where $\sigma' = \sigma_1' \cdot r' \twoheadrightarrow r : l' \cdot \sigma_2'$ is an alternation of $\sigma \cdot t$ and $r \notin \mathsf{subj}(\sigma_1')$. Thus, $S \xrightarrow{\sigma_1' \cdot r' \twoheadrightarrow r : l'}_1$ where $\sigma_1' \cdot r' \twoheadrightarrow r : l'$ is an alternation.

Case 5-2. S_2 is not stable. By (eventual reception), $S_2 \xrightarrow{\sigma'}_1 S_3$ where S_3 is stable. By Lem. B10(ii) and Prop. B8, $S \xrightarrow{\sigma''}_1 S_3$ where $\sigma'' = \sigma_1'' \cdot r' \twoheadrightarrow r : l' \cdot \sigma_2''$ is an alternation of $\sigma \cdot t \cdot \sigma'$ and $r \notin \mathsf{subj}(\sigma_1'')$. Thus, $S \xrightarrow{\sigma_1'' \cdot r' \twoheadrightarrow r : l'}_1$ where $\sigma_1'' \cdot r' \twoheadrightarrow r : l'$ is an alternation.

Case 6 (disconnect). S is not final, $L_r \in P$ and $r'\,\# \in L_r$. By (role progress), we have $S \xrightarrow{\sigma}_1 S_1 = (P_1, Q_1) \xrightarrow{t}_1$ and $r \in \mathsf{subj}(t)$. Let $r \notin \mathsf{subj}(\sigma)$. By Prop. B2, $t = r : r'\#$. Thus, $Q(r, r') = Q_1(r, r')$, and $S \xrightarrow{t}_1$. □

From compatibility, we follow the approach of [6] to show the following.

Lemma B13. (*Stable property*). Assume a $\mathsf{wf}(G)$ with initial session S_0 that is 1-safe and satisfies 1-progress. If $S_0 \to^* S$, then $S \to^* S'$ such that $S_0 \to_1^* S'$ and S' is stable.

B.2 Soundness of 1-bounded Validation

We annotate safety errors to designate the role to which the error pertains.

Definition B14. (*Annotated errors*). A safety error may be annotated Err_r, where r is as specified in the definition of the relevant error kind (§ 3.2).

The following identifies a *common instance* of a *persistent* safety error between two sessions.

Definition B15. (*Persistent errors*). A safety error Err_r is a *persistent error at* r if it is any of the error kinds listed in the following definition. Let $\mathsf{Err}_r = (P_1, Q_1)$ and $S = (P_2, Q_2)$. Then we define $\mathsf{Err}_r \succ_r S$ by:

- (*Reception error*) $L_r = \mathbf{\Sigma}_{i \in I} r'?l_i.L_i \in P_{1,2}$, $Q_1(r,r') = l \cdot \vec{l}$, $Q_2(r,r') = l \cdot \vec{l} \cdot \vec{l'}$, and $l \notin \{l_i\}_{i \in I}$.
- (*Connection error*) $L_r = \mathbf{\Sigma}_{i \in I} r'??l_i.L_i \in P_{1,2}$, $Q_1(r,r') = \vec{l}$, and $Q_2(r,r') = \vec{l} \cdot \vec{l'}$.
- (*Disconnect error*) $L_r = r'\#.L' \in P_{1,2}$, and either (i) $Q_1(r,r') = Q_2(r,r') = \bot$ or (ii) $Q_1(r,r') = l \cdot \vec{l}$ and $Q_2(r,r') = l \cdot \vec{l} \cdot \vec{l'}$.
- (*Unconnected error*) $L_r = \mathbf{\Sigma}_{i \in I} r'?l_i.L_i \in P_{1,2}$ and $Q_1(r,r') = Q_2(r,r') = \bot$.

Proposition B16. $\succ_r$ is reflexive and transitive.

Lemma B17. (*Error preservation*). Let S_0 be the initial session of a $\mathsf{wf}(G)$. If $S_0 \to^* \mathsf{Err}_r$, where Err_r is a *persistent* error, and $\mathsf{Err}_r \xrightarrow{\sigma} S$, then $\mathsf{Err}_r \succ_r S$.

Proof. By induction on the length of σ. The case of $\sigma = \epsilon$ holds trivially by Prop. B16. This includes the case where Err_r is an (unfinished role).

Assume $S_0 \to^* \mathsf{Err}_r \xrightarrow{\sigma^n} S$ and $\mathsf{Err}_r \succ_r S$ holds for σ^n of length n. We proceed by cases on the remaining kinds of Err_r and on the last action in a sequence of length $n+1$. Let $S = (P, Q) \xrightarrow{t} S' = (P', Q')$.

We illustrate the cases for (reception error) and (connection error); the remaining cases are similar.

Case 1 (reception error). By Def. B15, $L_r \in P$, $L_r = \mathbf{\Sigma}_{i \in I} r'?l_i.L_i$, $Q(r,r') = l \cdot \vec{l}$ and $l \notin \{l_i\}_{i \in I}$.
Case 1-1 (send). $t = r_1 : r_2!l'$. By $\mathsf{wf}(G)$, $r_1 \neq r_2$. By [Send], $r_1 \neq r$.
Case 1-1-1 ($r_2 = r$ and $r_1 = r'$). By [Send], $L_{r'} = \mathbf{\Sigma}_{i \in I} \alpha_i.L_i \in P$ and $r!l' = \alpha_j$ where $j \in I$. By [Send], $P' = P \setminus L_{r'} \cup \{L_j\}$ and $Q' = Q[r, r' \mapsto l \cdot \vec{l} \cdot l']$. By Def. B15, $S \succ_r S'$. By Prop. B16, $\mathsf{Err}_r \succ_r S'$, and we have the result by the induction hypothesis.
Case 1-1-2 (Otherwise). By [Send], $L_{r_1} = \mathbf{\Sigma}_{i \in I} \alpha_i.L_i \in P$ and $r_2!l' = \alpha_j$ where $j \in I$. By [Send], $P' = P \setminus L_{r_1} \cup \{L_j\}$ and $Q' = Q[r_2, r_1 \mapsto Q(r_2, r_1) \cdot l']$. By Def. B15, $S \succ_r S'$. By Prop. B16, $\mathsf{Err}_r \succ_r S'$, and we have the result by the induction hypothesis.
Case 1-2 (receive). $t = r_1 : r_2?l'$. By $\mathsf{wf}(G)$, $r_1 \neq r_2$. By [Recv], $r_1 \neq r$. Then $L_r \in P'$ and $Q'(r,r') = Q(r,r')$. By Def. B15, $S \succ_r S'$. By Prop. B16, $\mathsf{Err}_r \succ_r S'$, and we have the result by the induction hypothesis.

Case 1-3 (connect). $t = r_1 \twoheadrightarrow r_2 : l'$. By [Conn], $r \notin \{r_1, r_2\}$. By [Conn], $L_r \in P'$ and $Q'(r,r') = Q(r,r')$. By Def. B15, $S \succ_r S'$. By Prop. B16, $\mathsf{Err}_r \succ_r S$, and we have the result by the induction hypothesis.
Case 1-4 (disconnect). $t = r_1 : r_2\#$. By [Dis], $r_1 \neq r$. By [Dis], $L_r \in P'$ and $Q'(r,r') = Q(r,r')$. By Def. B15, $S \succ_r S'$. By Prop. B16, $\mathsf{Err}_r \succ_r S$, and we have the result by the induction hypothesis.

Case 2 (connection error). By Def. B15, $L_r \in P$, $L_r = \mathbf{\Sigma}_{i \in I} r'??l_i.L_i$ and $Q(r,r') \neq \bot$. By $\mathsf{wf}(G)$, $r_1 \neq r_2$.
Case 2-1 (send). $t = r_1 : r_2!l'$. By [Send], $r_1 \neq r$. By $\mathsf{wf}(G)$, $r_1 = r' \Rightarrow r_2 \neq r$. By [Send], $L_{r_1} = \mathbf{\Sigma}_{i \in I} \alpha_i.L_i \in P$ and $r_2!l' = \alpha_j$ where $j \in I$. By [Send] $P' = P \setminus L_{r_1} \cup \{L_j\}$ and $Q' = Q[r_2, r_1 \mapsto Q(r_2, r_1) \cdot l']$. By Def. B15, $S \succ_r S'$. By Prop. B16, $\mathsf{Err}_r \succ_r S'$, and we have the result by the induction hypothesis.
Case 2-2 (receive). $t = r_1 : r_2?l'$. By [Recv], $r_1 \neq r$. Then $L_r \in P'$ and $Q'(r,r') = Q(r,r')$. By Def. B15, $S \succ_r S'$. By Prop. B16, $\mathsf{Err}_r \succ_r S'$, and we have the result by the induction hypothesis.
Case 2-3 (connect). $t = r_1 \twoheadrightarrow r_2 : l$. By [Conn], $r \notin \{r_1, r_2\}$. By [Conn], $L_r \in P'$ and $Q'(r,r') = Q(r,r')$. By Def. B15, $S \succ_r S'$. By Prop. B16, $\mathsf{Err}_r \succ_r S$, and we have the result by the induction hypothesis.
Case 2-4 (disconnect). $t = r_1 : r_2\#$. By [Dis], $r_1 \neq r$. By [Dis], $L_r \in P'$ and $Q'(r,r') = Q(r,r')$. By Def. B15, $S \succ_r S'$. By Prop. B16, $\mathsf{Err}_r \succ_r S'$, and we have the result by the induction hypothesis. □

Theorem 1. (*Soundness of 1-bounded validation*). Let S_0 be the initial session of a $\mathsf{wf}(G)$ that is 1-safe and satisfies 1-progress. Then *(i)* S_0 is safe and *(ii)* S_0 satisfies progress.

Proof. *(i)* is by contradiction in each case of Err_r. Say $S_0 \xrightarrow{\sigma} \mathsf{Err}_r = (P,Q)$ and $\mathsf{Err}_r \notin RS_1(S_0)$. By Lem. B13, $\mathsf{Err}_r \xrightarrow{\sigma} S$, S is stable and $S_0 \rightarrow_1^* S$.

Case 1 (persistent errors). By Lem. B17, $\mathsf{Err}_r \succ_r S$. However, S is 1-safe—contradiction.

Case 2 (orphan message). r is inactive in Err_r and $\exists r'(Q(r,r') = l \cdot \vec{l})$. Since S is stable, it must be the case that $\mathsf{Err}_r \rightarrow^* \xrightarrow{r:r'?l} \rightarrow^* S$. However, $L_r \in P$ is either end or $\mathbf{\Sigma}_{i \in I} r'??l_i.L_i$—contradiction.

Case 3 (unfinished role). $\sigma = \epsilon$. Then $\mathsf{Err}_r = S$, and $S_0 \rightarrow_1^* \mathsf{Err}_r$—contradiction.

(ii) follows from Lem. B13 and 1-progress. Let $S_0 \rightarrow^* S = (P,Q)$.

Case 1 (role progress). Let r be active in S. By Lem. B13, $S \xrightarrow{\sigma} S' = (P',Q')$ where S' is stable and $S_0 \rightarrow_1^* S'$.
Case 1-1 ($t \in \sigma$ and $r \in \mathsf{subj}(t)$). (Role progress) is satisfied by definition.
Case 1-2 (otherwise). $L_r \in P = L'_r \in P'$, so r is active in S'. Since S' satisfies 1-progress, then $S \xrightarrow{\sigma} \rightarrow_1^* \xrightarrow{r}_1$.

Case 2 (eventual reception). By definition of Lem. B13. □

C Appendix: § 4

C.1 Translation of Local Types to Endpoint FSMs

Fig. 5 defines the main part of the translation from local types to Endpoint FSMs (§ 4.1). In $\mathtt{graph}(L, s_1, s_2, f)$, f is a map $\mathbb{X} \mapsto \mathbb{S}$.

$$\begin{cases} \mathtt{graph}(\alpha.\mathsf{end}, s, s', f) & = \{(s, \alpha, s')\} \\ \mathtt{graph}(\alpha.X, s, s', f) & = \{(s, \alpha, f(X))\} \\ \mathtt{graph}(\alpha.L', s, s', f) & = \{(s, \alpha, s'')\} \cup \mathtt{graph}(L, s'', s', f) \quad L' \text{ not } \mathsf{end} \text{ or } X, \text{fresh } s'' \\ \mathtt{graph}(\mathbf{\Sigma}_{i \in I} \alpha_i.L_i, s, s', f) & = \cup_{i \in I} \mathtt{graph}(\alpha_i.L_i, s, s', f) \qquad |I| > 1 \\ \mathtt{graph}(\mu X.L', s, s', f) & = \mathtt{graph}(L', s, s', f[X \mapsto s]) \end{cases}$$

Fig. 5. Translation of local types to Endpoint FSMs.

Let $M = (\mathbb{S}, \mathbb{R}, s_0, \mathbb{L}, \delta)$ in each of the following:

- $\mathsf{subfsm}(M, s) = (\mathbb{S}', \mathbb{R}, s, \mathbb{L}, \delta')$, where $s \in \mathbb{S}$, $\mathbb{S}' = \{s\} \cup RS(s)$ and
  $(s_1, \alpha, s_2) \in \delta \wedge \{s_1, s_2\} \subseteq \mathbb{S}' \implies (s_1, \alpha, s_2) \in \delta'$
- $\mathsf{prune}(M, \alpha, s) = M'$, where:
  $$M' = \begin{cases} (\{s_{\mathsf{end}}\}, \mathbb{R}, s_{\mathsf{end}}, \mathbb{L}, \emptyset) & s = s_{\mathsf{end}} \\ (\mathbb{S}', \mathbb{R}, s_0, \mathbb{L}, \delta') & s \neq s_{\mathsf{end}}, \text{ such that} \\ \quad (s_1, \alpha', s_2) \in \delta \implies \begin{cases} (\dot{s_1}, \alpha', \dot{s_2}) \in \delta' & s_1 = s, \alpha' = \alpha, s_2 = s_0 \\ (\dot{s_1}, \alpha', \dot{s_2}) \in \delta' & s_1 \neq s \end{cases} \\ \quad \text{and } \mathbb{S}' = \{\dot{s} \mid s \in \mathbb{S} \wedge \dot{s} \in \delta'\}. \end{cases}$$
- $\mathsf{unfair}(M) = (\mathbb{S}', \mathbb{R}, s_0, \mathbb{L}, \delta')$, where for each $\alpha \in \delta(s_0)$:
  let $s_\alpha = \delta(s_0, \alpha)$ and $M'' = \mathsf{subfsm}(M, s_\alpha)$ in
  $$M_\alpha = (\mathbb{S}_\alpha, \mathbb{R}, s_\alpha, \mathbb{L}, \delta_\alpha) = \begin{cases} \mathsf{prune}(M'', \alpha, s_0) & s_0 \in RS(s_\alpha) \\ M'' & \text{otherwise} \end{cases}$$
  then $\mathbb{S}' = \{s_0\} \cup \cup_{\alpha \in \delta(s_0)} \mathbb{S}_\alpha$ and $\delta' = \cup_{\alpha \in \delta(s_0)} (\{(s_0, \alpha, s_\alpha)\} \cup \delta_\alpha)$.

Fig. 6. EFSM transformation based on "unfair" output choice subtyping.

Assume a $\mathsf{wf}(G)$, $r \in G$, $L = G \upharpoonright r$ and fresh states s_0 and s_{end}. Then the *EFSM of r in G* is $\mathsf{fsm}(G, r) = (\mathbb{S}, \mathbb{R}_G, s_0, \mathbb{L}_G, \delta)$, where $\delta = \mathtt{graph}(L, s_0, s_{\mathsf{end}}, \emptyset)$, and $\mathbb{S}$ is the set of states occuring in δ.

For a given $M = (\mathbb{S}, \mathbb{R}, s, \mathbb{L}, \delta)$, we define: (1) a *path* σ is a sequence $s_1..s_n$, $n \geq 1$, such that $\forall i \in 1..n-1.(s_i, \alpha_i, s_{i+1}) \in \delta$; and (2) $RS(s) = \{s' \mid \exists \sigma = s..s'\}$, for $s \in \mathbb{S}$ and $n \geq 1$.

C.2 EFSM Transformation for "Unfair" Output Choice Subtyping

We reflect the potential use of *local* type subtyping in endpoint implementations that may adversely affect role progress, by a transformation on the EFSM for each role.

The main elements of the transformation are defined in Fig. 6. (1) $\mathsf{subfsm}(M, s)$ extracts the subgraph rooted at s. (2) In $\mathsf{prune}(M, \alpha, s)$, the notation $\dot{s}$ means a fresh state identifier derived from s, with the exception $\dot{s}_{\mathsf{end}} = s_{\mathsf{end}}$. prune clones M but also prunes all *non-α* edges from s_0 to s (and discards states that are no longer connected). (3) $\mathsf{unfair}(M)$ uses prune to clone the subgraph reachable from the successor state s_α for each action $\alpha \in \delta(s_0)$, in the cases that the initial state s_0 is again reachable from the s_α, with all non-α cases discarded when (the clone of) s_0 is revisited. The α-edge from s_0 to s_α is replaced by an α-edge to the clone of s_α.

Then, the overall transformation on a given EFSM M proceeds by replacing, for all non-unary $s^\bullet \in \mathbb{S}_M$, the subgraph rooted at $s^\bullet$ by applying unfair until every recursive internal choice has been reduced to a single action.

C.3 Travel Agent Endpoint Implementations in Java

Fig. 7 demonstrates Java implementations of A and S in TravelAgency (from Fig. 1) using the APIs generated from their EFSMs. Note that the exception handling concerns only failures, such as broken connections, and linearity errors; i.e., if a session method is called on a state channel more than once, or if a state channel is unused (and control flow leaves the endpoint-try). Aside from these, the validated API statically ensures by types that the session implementation is safe from I/O errors.

D Appendix: Microservices Use Case

We demonstrate a microservices use case from our industry collaborations. A current project involves using Scribble to generate XML schema for a third-party FSM-based callback engine in a Docker deployment. The aim is to allow a developer to get a service up and running by providing mainly the Scribble protocols, and the structured, application-specific implementation of the generated callback interfaces. Microservices is a setting where explicit connection actions seem to be crucial for adequate specifications by MPSTs, due to applications being formed mainly by optional and dynamic invocations between a numerous set of services.

The scenario of this example is as follows. A *client* first authenticates with a *login service*. If successful, the client disconnects from the login service (to release it from the session) before proceeding to main protocol. In the main loop, the client submits a query request to an *authorisation service*, selecting between information from either a *suppliers service* or a *contracts service*, which may be denied or approved. If approved, in both cases the information the authorisation service uses a *filter service* to process the information before returning it to the client.

The protocol is specified in Scribble in Figs. 8 and 9.

```
void run() throws Exception {
  try (ScribServerSocket ss = new SocketChannelServer(port_A)) {
        // TCP server
    Buf<Object> b = new Buf<>();
    while (true) {
      TravelAgency sess = new TravelAgency();
      try (ExplicitEndpoint<TravelAgency, A> se =
                new ExplicitEndpoint<>(sess, A)) {
        aux(new TravelAgency_A_1(se).accept(C, ss).branch(C));
}  } } }

EndSocket aux(TravelAgency_A_2_Cases A2, Buf<Object> b)
      throws Exception {
  switch (A2.op) {
        // A2.op is a generated state-specific enum for this input choice
    case query: aux(A2.receive(query, b)
                      .send(C, quote, getQuote(b.val)).branch(C))
    case accpt:   return A2.receive(accpt, b);
    case reject:  return A2.receive(reject, b);
    default:      throw new RuntimeException("Won't get in here.");
} }
```

```
public void run() throws Exception {
  try (ScribServerSocket ss = new SocketChannelServer(port_S)) {
    Buf<Object> b = new Buf<>();
    while (true) {
      TravelAgency sess = new TravelAgency();
      try (ExplicitEndpoint<TravelAgency, S> se
           = new ExplicitEndpoint<>(sess, S)) {
        new TravelAgency_S_1(se)
          .accept(C, ss)
          .receive(C, pay, b).send(C, confirm, getPayRef(b.val));
} } } }
```

Fig. 7. Safe Java implementations of `A` and `S` in `TravelAgency` (Fig. 1) using Scribble-generated APIs.

```
type <xsd> "UserName" from "AUTH.xsd" as UserName;
type <xsd> "Password" from "AUTH.xsd" as password;
type <xsd> "UUID" from "AUTH.xsd" as UUID;
type <xsd> "SupplierDetails" from "Retailer.xsd" as SupplierDetails;
type <xsd> "ContractDetails" from "Retailer.xsd" as ContractDetails;
type <xsd> "UserContext" from "Filter.xsd" as UserContext;
type <xsd> "Filters" from "Filter.xsd" as Filters;


explicit global protocol InfoAuth
     (role LoginSvc, role Client, role AuthSvc, role Filtersvc,
      role SupplierSvc, role ContractSvc)
{
  connect Client to LoginSvc;
  login(UserName, password) from Client to LoginSvc;
  choice at LoginSvc {
    loginfailure() from LoginSvc to Client;
  } or {
    loginsuccess() from LoginSvc to Client;
    disconnect Client and LoginSvc;
    connect Client to AuthSvc;
    do Main(Client, AuthSvc, Filtersvc, SupplierSvc, ContractSvc);
  }
}

aux global protocol Main
     (role Client, role AuthSvc, role Filtersvc,
      role SupplierSvc, role ContractSvc) {
  choice at Client {
    getsuppliers(UUID) from Client to AuthSvc;
    do SuppInfo(Client, AuthSvc, Filtersvc, SupplierSvc);
  } or {
    getcontracts() from Client to AuthSvc;
    do ContractInfo(Client, AuthSvc, Filtersvc, ContractSvc);
  }
  do Main(Client, AuthSvc, Filtersvc, SupplierSvc, ContractSvc);
}
```

Fig. 8. Authorised Supplier/Contract Info use case (part 1 of 2).

```
aux global protocol SuppInfo
      (role Client, role AuthSvc, role Filtersvc, role SupplierSvc) {
  choice at AuthSvc {
    deny() from AuthSvc to Client;
  } or {
    connect AuthSvc to SupplierSvc;
    getsuppliers() from AuthSvc to SupplierSvc;
    suppliers() from SupplierSvc to AuthSvc;
    disconnect AuthSvc and SupplierSvc;
    do FilterInfo
        <Filtersuppliers(UserContext, Filters, SupplierDetails)>
        (AuthSvc, Filtersvc);
    suppliers() from AuthSvc to Client;
  }
}

aux global protocol ContractInfo
      (role Client, role AuthSvc, role Filtersvc, role ContractSvc) {
  choice at AuthSvc {
    deny() from AuthSvc to Client;
  } or {
    connect AuthSvc to ContractSvc;
    getcontracts() from AuthSvc to ContractSvc;
    contracts() from ContractSvc to AuthSvc;
    disconnect AuthSvc and ContractSvc;
    do FilterInfo
        <filterContracts(UserContext, Filters, ContractDetails)>
        (AuthSvc, Filtersvc);
    contracts() from AuthSvc to Client;
  }
}

aux global protocol FilterInfo
      <sig Query>
      (role AuthSvc, role Filtersvc)
{
  Query connect AuthSvc to Filtersvc;
  filtered() from Filtersvc to AuthSvc;
  disconnect AuthSvc and Filtersvc;
}
```

Fig. 9. Authorised Supplier/Contract Info use case (part 2 of 2).