

Multiparty Session Types for Safe Runtime Adaptation in an Actor Language

Paul Harvey ✉ 


Rakuten Mobile Innovation Studio

Simon Fowler ✉ 

School of Computing Science, University of Glasgow

Ornela Dardha ✉ 

School of Computing Science, University of Glasgow

Simon J. Gay ✉ 

School of Computing Science, University of Glasgow

Abstract

Human fallibility, unpredictable operating environments, and the heterogeneity of hardware devices are driving the need for software to be able to *adapt* as seen in the Internet of Things or telecommunication networks. Unfortunately, mainstream programming languages do not readily allow a software component to sense and respond to its operating environment, by *discovering*, *replacing*, and *communicating* with components that are not part of the original system design, while maintaining static correctness guarantees. In particular, if a new component is discovered at runtime, there is no guarantee that its communication behaviour is compatible with existing components.

We address this problem by using *multiparty session types with explicit connection actions*, a type formalism used to model distributed communication protocols. By associating session types with software components, the discovery process can check protocol compatibility and, when required, correctly replace components without jeopardising safety.

We present the design and implementation of EnsembleS, the *first* actor-based language with adaptive features and a static session type system, and apply it to a case study based on an adaptive DNS server. We formalise the type system of EnsembleS and prove the safety of well-typed programs, making essential use of recent advances in *non-classical* multiparty session types.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages

Keywords and phrases Concurrency, session types, adaptation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.12

Related Version A full version of the paper is available at <https://arxiv.org/abs/2105.06973>.

Funding Supported by EPSRC grants EP/T014628/1 (STARDUST), EP/K034413/1 (ABCD), EP/L01503X/1 (CDT in Pervasive Parallelism), ERC Consolidator Grant Skye (682315), and by the EU HORIZON 2020 MSCA RISE project 778233 (BehAPI).

Acknowledgements Thanks to Phil Trinder for helpful comments and discussions, and to the anonymous reviewers for exceptionally detailed reviews.

1 Introduction

The era of single monolithic stand-alone computers has long been replaced by a landscape of heterogeneous and distributed computers and software applications. Technologies such as the IoT [56], self-driving cars [55], or autonomous networks [7] bring the new challenge of needing to successfully operate in face of ever-changing environments, technologies, devices, and human errors, necessitating the need to *adapt*.



© Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay;
licensed under Creative Commons License CC-BY 4.0

35th European Conference on Object-Oriented Programming (ECOOP 2021).

Editors: Manu Sridharan and Anders Møller; Article No. 12; pp. 12:1–12:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Here, we define *dynamic self-adaptation*—hereafter referred to as *adaptation*—as the ability of a software component to sense and respond to its operating environment, by *discovering*, *replacing*, and *communicating* with other software components at runtime that are not part of the original system design [6, 52]. There are many examples of adaptive systems, as well as the mechanisms of adaptation they leverage, such as discovery [37], modularisation [27], dynamic code loading and migration [12, 24]. Commercially, Steam’s in-home streaming system¹ enables video games to dynamically transfer their input/output across a range of devices. Academically, RE^X [50] enables software to self-assemble predefined components, using machine learning to reconfigure the software in response to environmental changes.

Despite strong interest in adaption and substantial work on the mechanisms of adaptation, current programming languages either lack the capabilities to ensure that adaptation can be achieved safely and correctly, or they check correctness dynamically, resulting in runtime overheads which may not be acceptable for resource-constrained devices.

Specifically, if an adaptive system discovers new software components at runtime, these components must interact with the system in a purposeful manner. In concurrent and distributed systems, such interaction goes beyond a simple function call / return expressed with standard types and type systems: interaction involves complex *communication protocols* that constrain the sequence and type of data exchanged. For example, knowing that two components communicate integers and strings does not describe if or when they will be sent or received. In spite of growing interest in the topic, for example, the recent formation of the United Nations group considering *creative adaptation*², mainstream programming languages do not support the *specification* and *verification* of communication protocols in concurrent and distributed systems. In turn, errors are discovered late in the development process and potentially after deployment.

Even where all components are known statically, communication safety cannot be guaranteed: as an example, the RE^X system’s programming language specifies sequential call / return interfaces for components, but not communication protocols for concurrent components. The adaptation in the Steam in-home streaming system is even more limited, being restricted to detection of input/output devices from a set of compatible possibilities. In both cases, the adaptive aspects of the software have been defined and designed ahead of time, as opposed to being composed *on-demand* at runtime, leaving no scope for extending the system via runtime discovery and replacement.

This situation brings us to a key research question:

RQ: Can a programming language support *static (compile-time) verification* of safe runtime dynamic self-adaptation, i.e., *discovery, replacement and communication*?

The problem of static verification of safe communication is addressed by *multiparty session types* [29, 30, 31]. Multiparty session types (MPSTs) are a type formalism used to specify the *type*, *direction* and *sequence* of communication actions between two or more participants. Session types guarantee that software conforms to predefined communication protocols, rather than risking errors manifesting themselves at runtime.

There is already some work in the literature on adaptation and session types, but it does not answer our research question. We discuss related work in §6, but in brief, the state-of-the-art has some combination of the following limitations: theory for a formal model such as the π -calculus [13, 11, 19, 18], rather than a real-world programming language; omission of some

¹ <http://store.steampowered.com/streaming/>

² <https://www.itu.int/en/ITU-T/focusgroups/an/Pages/default.aspx>

Global protocol

```

1 global protocol Bookstore
2   (role Sell, role Buy1, role Buy2) {
3   book(string) from Buy1 to Sell;
4   book(int) from Sell to Buy1;
5   quote(int) from Buy1 to Buy2;
6   choice at Buy2 {
7     agree(string) from Buy2 to Buy1, Sell;
8     transfer(int) from Buy1 to Sell;
9     transfer(int) from Buy2 to Sell;
10  } or {
11    quit(string) from Buy2 to Buy1, Sell;
12  } }

```

Local protocol for Sell

```

1 local protocol Bookstore_Sell
2   (self Sell, role Buy1, role Buy2) {
3   book(string) from Buy1;
4   book(int) to Buy1;
5   choice at Buy2 {
6     agree(string) from Buy2;
7     transfer(int) from Buy1;
8     transfer(int) from Buy2;
9   } or {
10    quit(string) from Buy2;
11  } }

```

■ **Figure 1** Global and local protocols for Bookstore

aspects of adaptation, such as runtime discovery [32]; or verification by runtime monitoring [47, 49, 21], as opposed to static checking.

To answer our research question, we implement EnsembleS, the first actor language leveraging MPSTs to provide compile-time verification of safe dynamic runtime adaptation: we can statically guarantee that a discovered actor will comply with a communication protocol, and guarantee that replacing an actor’s behaviour (e.g., to fix a bug) will not jeopardise communication safety. Key to our approach is the combination of the actor paradigm [28], for its process addressability and explicit message passing, with *explicit connection actions* [32] in multiparty session types, which allow discovered actors to be invited into a session.

Contributions. The overarching contribution of this work is the design, implementation, and formalisation of a language which supports dynamic self-adaptation while guaranteeing communication safety. We achieve this through a novel integration of an actor-based language and multiparty session types with explicit connection actions. Specifically, we introduce:

1. **EnsembleS and its compiler** (§3): we present an actor language, EnsembleS, which supports safe adaptable applications using MPSTs. Our framework supports:
 - MPST specifications, both standard and using explicit connection actions (§3.3);
 - MPSTs to provide guarantees of protocol compliance in runtime discovery (§3.4);
 - automatic generation of application code from MPSTs (§3.2)
2. **An adaptive DNS case study** (§4): using MPSTs and runtime discovery to show safe dynamic self-adaptation can be achieved in a non-trivial software service
3. **A core calculus for EnsembleS** (§5): we formalise EnsembleS and prove type safety and progress.

The formalism makes several technical contributions: it is the first actor-based calculus with statically-checked MPSTs; and it is the first *calculus* to provide a language design and semantics for explicit connection actions, which had previously only been explored at the type level. Our design requires exception handling in the style of Mostrous & Vasconcelos [45] and Fowler *et al.* [22], and the metatheory makes essential (and novel) use of *non-classical* multiparty session types [54].

The implementation and examples are available in the paper’s companion artifact.

```

1 explicit global protocol OnlineStore
2   (role Customer, role Store, role Courier) {
3   login(string) connect Customer to Store;
4   do Browse(Customer, Store, Courier);
5   }
6
7 aux global protocol Deliver
8   (role Customer, role Store, role Courier) {
9   address(string) from Customer to Store;
10  deliver(string) connect Store to Courier;
11  ref(int) from Courier to Store;
12  disconnect Courier and Store;
13  ref(int) from Store to Customer;
14  disconnect Store and Customer;
15  }

```

```

1 aux global protocol Browse
2   (role Customer, role Store, role Courier) {
3   item(string) from Customer to Store;
4   price(int) from Store to Customer;
5   choice at Customer {
6     do Browse(Customer, Store, Courier);
7   } or {
8     do Deliver(Customer, Store, Courier);
9   } or {
10    quit() from Customer to Store;
11    disconnect Store and Customer;
12  }
13 }

```

■ **Figure 2** Global protocol for OnlineStore

2 Multiparty Session Types

Multiparty session types [31] are a type formalism used to describe communication protocols in concurrent and distributed systems. An MPST describes communication among multiple software components or participants, by specifying the *type* and the *direction* of data exchanged, which is given as a sequence of send and receive actions.

We first introduce MPSTs (formalised in §5) via *Scribble* [57], a specification language for communicating protocols based on the theory of multiparty session types. We start with a *global type*, which describes the interactions among all communicating participants. Using the Scribble tool, a global protocol can be *validated*, guaranteeing its correctness, and then *projected* for each participant. Projection returns a *local type*, which describes communication actions from the viewpoint of that participant.

Bookstore example. Fig.1 shows the classic **Bookstore** (also known as *Two-Buyer*) example, written in Scribble. We have three communicating participants (*roles*): two buyers **Buy1** and **Buy2**, and one seller **Sell**, where the buyers wish to buy a book from the seller. **Buy1** sends the title of the book of type **string** to **Sell** (line 3). Next, **Sell** sends the price of the book of type **int** to **Buy1** (line 4). At this stage, **Buy1** invites **Buy2** to share the cost of the book, by sending them a **quote** of type **int** that **Buy2** should pay (line 5). It is **Buy2**'s *internal choice* (line 6) to either **agree** (line 7), or **quit** the protocol (line 11). After agreement, both **Buy1** and **Buy2** transfer their **quote** to **Sell** (lines 8 and 9, respectively).

Projecting the **Bookstore** global protocol into each of the communicating participants returns their local protocols. Fig.1 shows the local protocol for **Sell**; we omit **Buy1** and **Buy2** as they are similar. Note that the local protocol only includes actions relevant to **Sell**.

Explicit connection actions. The **Bookstore** protocol assumes that all roles are connected at the start of the session. This is undesirable when a participant is only needed for *part* of a session, or the identity of a participant depends on data exchanged in the protocol.

Consider Figure 2, which details the protocol for an online shopping service, inspired by the travel agency protocol detailed by Hu & Yoshida [32]. The protocol is organised as three subprotocols: **OnlineStore**, the entry-point; **Browse**, where the customer repeatedly requests quotes for items; and **Deliver**, where the store requests delivery from a courier. In contrast to **Bookstore**, each connection must be established explicitly (note that **connect** replaces **from** when initiating a connection).

```

1 type Isnd is interface(out integer output)
2 type Ircv is interface(in integer input)
3
4 stage home {
5   actor sender presents Isnd {
6     value = 1;
7     constructor() {}
8     behaviour {
9       send value on output;
10      value := value + 1;
11    } }
12  actor receiver presents Ircv {
13    constructor() {}
14    behaviour{
15      receive data from input;
16      printString("\nreceived:");
17      printInt(data);
18    } }
19  boot {
20    s = new sender();
21    r = new receiver();
22    establish topology(s, r);
23  }
24 }

```

■ **Figure 3** A simple EnsembleS program

Note in particular that **Courier** is only involved in the **Deliver** subprotocol. The store can therefore *choose* which courier to use based on, for example, the weight of the item or the customer’s location. Furthermore, it is not necessary to involve the courier if the customer does not choose to make a purchase.

3 EnsembleS: An Actor Language for Runtime Adaptation

In this section, we present EnsembleS, a new session-typed actor-based language based on Ensemble [25, 26]. EnsembleS actors are addressable, single-threaded entities with share-nothing semantics, and communicate via message passing. However, differently from the classic definition of the actor model [28, 1], the communication model in EnsembleS is channel-based. EnsembleS supports both *static* and *dynamic* topologies:

Static Topologies All participants are present at the start of the session and remain involved for the duration of the session. This is based on traditional MPSTs [31].

Dynamic Topologies Participants can connect and disconnect during a session. This builds on the more recent idea of explicit connection actions [32].

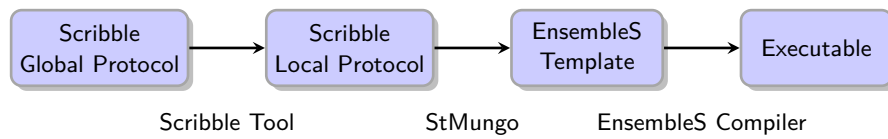
3.1 EnsembleS: basic language features

An EnsembleS actor has its own private state and a single thread of control expressed as a **behaviour** clause, which is repeated until explicitly told to stop. Every actor executes within a **stage**, which represents a memory space. Actors do not share state, but instead communicate via message passing along half-duplex, simply-typed channels.

Fig. 3 shows a simple EnsembleS program which defines, instantiates and connects two actors, one of which sends increasing values to the other. The program defines two interfaces **Isnd** and **Ircv**, declaring an output and input channel respectively. The **boot** clause (lines 19–23) is executed first and creates an instance of each actor (lines 20–21), using the appropriate **constructor** (lines 7 and 13, respectively). This creates and begins executing new threads for each actor, which follow the logic of the relevant **behaviour** clause. Next, the **boot** clause binds the actor’s channels together (line 22, discussed in §3.3). Once bound, the **sender** actor sends the contents of **value** on its channel, increments it, and goes back to the beginning of its behaviour loop (lines 8–11). The **receiver** actor waits for a message, binds the message to **data**, displays it, and returns to the top of its behaviour loop (lines 14–18). EnsembleS inherits Ensemble’s support for runtime software adaptation actions [26]:

Discover The ability to *locate* an arbitrary actor or stage reference at runtime, given an **interface** and **query**.

Install Given an actor type, the ability to **spawn** it at a specified stage.



■ **Figure 4** Automatic Actor Skeleton Generation Process

Migrate The ability for an executing actor to *move* to another stage.

Replace The ability to *replace* an executing actor A by a new instantiation of actor B, the latter continuing at the same stage as A, if A and B have the same **interface**.

Interact : Given an actor reference (either spawned, discovered, or communicated), the ability to *connect* to its channels at runtime and then communicate.

We focus on the underlined actions and apply session types to guarantee communication safety. The reason for this choice is that discover, replace and interact are actions that modify *how* actors operate, whereas the other actions, install and migrate, affect *where* actors operate, but not their behaviour.

3.2 Session types in EnsembleS

A **session** type in EnsembleS represents a communication protocol for an actor, i.e., a local protocol (or local session type) validated and projected from a global session type.

We extend the StMungo [40] tool to generate EnsembleS template code that supports session types. Fig. 4 shows an overview of the actor template code generation from a global session type, and Fig. 5 shows an example of the generated code.

First, a developer defines a global session type in Scribble [57] (Fig. 4, first stage). The Scribble tool checks that the protocol is well-formed and valid according to MPST theory and *projects* the global protocol into local protocols for each participant (Fig. 4, second stage). For each local protocol, the StMungo tool produces (Fig. 4, third stage) *i*) the **session** type, *ii*) the **interface** and type definitions, and *iii*) the **actor** template. The generated code is parsed by the EnsembleS compiler, producing executable code (Fig. 4, fourth stage).

Let us now look at the **Buy1** local protocol, given in Fig. 1. Following the code generation process in Fig. 4, the EnsembleS template items *i*), *ii*) and *iii*) for **Buy1** correspond respectively to the code blocks starting in lines 3, 14, and 24 in Fig. 5.

The **Buy1** local protocol is translated as an EnsembleS **session** type in Fig. 5 (lines 3–12). It shows a sequence of send and receive actions (lines 4–6), followed by a choice at **Buy2** (lines 7–12), which determines the next set of communication actions.

Following session type specifications, EnsembleS channels define both the payload type and the **session** that this channel expects to interact with (lines 14–21, Fig. 5). The EnsembleS compiler uses this information to ensure that the **session** of each channel matches the **session** associated with the actor it is connected to.

An **actor** may follow a **session** type (line 24, Fig. 5). This tells the EnsembleS compiler that the logic within the **behaviour** clause of that actor must follow the communication protocol defined in the **session**.

It is important to note that the code generation in Fig. 4 is optional and the EnsembleS typechecker is independent of this process.

```

1 // FILE AUTOMATICALLY GENERATED
2 //*****SESSIONS*****
3 type Buy1 is session(
4   book(string) to Sell;
5   book(int) from Sell;
6   quote(int) to Buy2;
7   choice at Buy2{
8     Choice0_agree(string) from Buy2;
9     transfer(int) to Sell;
10  } or {
11    Choice0_quit(string) from Buy2;
12  }
13 //*****INTERFACES*****
14 type Buy1I is interface(
15   out {Seller, string} toSell_string,
16   in {Seller, integer} fromSell_integer,
17   out {Buy2, integer} toBuy2_integer,
18   in {Buy2, Choice0} fromBuy2_agreequit,
19   in {Buy2, string} fromBuy2_string,
20   out {Sell, integer} toSell_integer,
21 )
22 //*****ACTORS*****
23 stage home{
24 actor Buy1A presents Buy1I follows Buy1 {
25   constructor() {}
26   behaviour {
27     payload1 = "";
28     send payload1 on toSell_string;
29     receive payload2 from fromSell_integer;
30     payload3 = 42;
31     send payload3 on toBuy2_integer;
32     // Receive choice from other actor
33     receive payload4 from fromBuy2_agreequit;
34     switch(payload4) {
35       case Choice0_agree:
36         receive payload5 from fromBuy2_string;
37         payload6 = 42;
38         send payload6 on toSell_integer;
39         break;
40       case Choice0_quit:
41         receive payload7 from fromBuy2_string;
42         break;
43     }
44   }
45 // Omitted: Buy2A and SellA actors
46 boot {
47   buyer1 = new Buy1A();
48   buyer2 = new Buy2A();
49   seller = new SellA();
50   // other actors...
51   establish topology(buyer1,buyer2,seller);
52 } }

```

■ Figure 5 EnsembleS static session template

3.3 Channel connections: static and dynamic

If an actor follows a `session` type, then its channel connections must be *1-1*. This is the standard linearity requirement for session types: if there are multiple senders on one channel, then their messages can interfere and it is not possible to statically check that the session is followed correctly. EnsembleS avoids this problem by using a single channel for each message type between each pair of participants. For example, in Fig. 1, each of the three actors communicates strings and integers with both of the other actors. Because channels are unidirectional, each actor therefore has 8 channels: 2 to send strings and 2 to send integers to both other actors, and similarly 4 channels for receiving.

Static connections. When using session types with *static* topologies, and all actors in the session are known from the beginning of the application, EnsembleS provides the `establish topology` statement to create the connections between the specified `session` actors (line 22, Fig. 3; line 51, Fig. 5). A compile-time error is generated if the topology is ill-defined (e.g., if the sessions do not compose or if the channels do not match).

Dynamic connections. EnsembleS supports reconfigurable channels and *dynamic* connections, via `link` and `unlink` statements. The `link` statement takes two references to actors which follow sessions (line 5, top of Fig. 6), and connects all of the channels of the two specified actors such that the actors' sessions match. A compile-time error is raised if the sessions are incompatible. Conversely, the `unlink` statement disconnects (line 8).

3.4 Adaptation via discovery and replacement

EnsembleS supports runtime discovery of *local* or *remote* actor instances. As an example, in a sensor network, it may be desirable to connect to a sensor which has a battery level above a certain threshold. The EnsembleS query language allows us to define a query on non-functional properties (such as battery level or signal strength), as well as the channels exposed by an actor's interface. This ensures that any discovered actor has the correct

Discovery and explicit connections

```

1 query alpha() { $serial==823 && $version<4; }
2 actor_s = discover(
3   Buyer1_interface, Buyer1_session, alpha());
4 if (actor_s[0].length > 1){
5   link me with actor_s[0];
6   msg = "book";
7   send msg on toB_string;
8   unlink Buyer1_session;
9 }

```

Replacement

```

1 // session and interface definitions
2 actor fastA presents accountingI
3   follows accountingSession{
4   constructor() {}
5   behaviour{
6     receive data on input;
7     quicksort(data);
8     send data on output;
9   }
10 }
11
12 actor slowA presents accountingI
13   follows accountingSession{
14   pS= new property[2] of property("",0);
15   constructor() {
16     pS[0]:= new property("serial",823);
17     pS[1]:= new property("version",2);
18     publish pS;
19   }
20   behaviour{
21     receive data on input;
22     bubblesort(data);
23     send data on output;
24   } }
25
26 actor main presents mainI {
27   constructor() { }
28   behaviour {
29     // Find the slow actors matching query
30     actor_s = discover(accountingI,
31       accountingSession, alpha());
32     // Replace them with efficient versions
33     if(actor_s[0].length > 1) {
34       replace actor_s[0] with fastA();
35     }
36   } }

```

■ Figure 6 Session type-based adaptation

number and type of channels, and satisfies user’s preferences. To ensure that the discovered actor also obeys a declared protocol, EnsembleS uses `session` types in the discovery process. The green box in Fig. 6 shows how a `session` is used in the actor discovery process, and the yellow box shows how such actors are connected together. Runtime discovery does not appear in the `session` because it does not affect the communication behaviour of an actor.

EnsembleS also supports the replacement of executing actors, much like the hot-code swapping in Erlang [12]. The new actor must present the same interface as it *takes over* the channels of the actor being replaced at the location it was executing. Replacement happens at the beginning of an actor’s `behaviour` loop. Replacement has many uses, such as updating, changing, or extending some of the functionalities of existing software, and is particularly useful in embedded systems [33, 34]. The existing and new actors must follow the same `session` type, guaranteeing that replacement will not break existing actor interactions.

Fig. 6 (bottom) shows an example of a `main` actor searching for actors of type `slowA` (line 30), and replacing them with new actors of type `fastA` (line 34). The `slowA` actors are located by defining a query (line 1, top) over user-defined properties, which are published (lines 16–18). The discovery process is the same as above, but now the discovered actors are used for replacement rather than just communication.

3.5 Implementation

EnsembleS is implemented in C, and supports reference-counted garbage collection and exceptions. Applications are compiled to Java source code, and then to custom Java class files for use with a custom VM [10]. These applications can be executed on the desktop,


```

1  type Client is session(
2    connect RootServer;
3    RootRequest(DomainName) to RootServer;
4    choice at RootServer{
5      TLDResponse(ZoneServerAddress)
6        from RootServer;
7      disconnect RootServer;
8    rec Lookup {
9      connect ZoneServer;
10     ResolutionRequest(DomainName) to ZoneServer;
11     choice at ZoneServer {
12       PartialResolution(ZoneServerAddress)
13         from ZoneServer;
14       disconnect ZoneServer;
15     }
16   } or {
17     InvalidDomain(String) from ZoneServer;
18     disconnect ZoneServer;
19   } or {
20     ResolutionComplete(IPAddress)
21       from ZoneServer;
22     disconnect ZoneServer;
23   }
24 } or {
25   InvalidTLD(String) from RootServer;
26   disconnect RootServer;
27 }
28 )
29 )

```

■ **Figure 7** EnsembleS DNS client `session` type

parallel accelerators (e.g. GPUs), Raspberry Pi, Lego NXT, and Tmote Sky hardware platforms, and use a range of networking technologies.

Compact representations of session types are retained at runtime in order to support discovery. EnsembleS skeleton generation code is based on the StMungo tool [40], which is implemented as an ANTLR listener, and session typechecking is supported by modifying the original Ensemble typechecker to ensure that each communication action is permitted by the actor’s declared session type.

Since EnsembleS builds directly on top of the original Ensemble implementation, it inherits Ensemble’s runtime system. Performance results can be found in [26].

4 Case study: DNS

To illustrate the use of session types for adaptive programming, we consider a real-world case study: the domain name system (DNS). DNS is a hierarchical, globally distributed translation system that converts an internet host name (domain name) into its corresponding numerical Internet Protocol (IP) address [43].

The process begins by transmitting a domain name to one of many well-known *root* servers. This server either rejects bad requests, or provides the information to contact a *zone* server. The zone server may know the IP address of the domain name; if not it refers the request to another zone server. This process continues until either the IP address is returned, or the name cannot be found.

To develop an adaptive DNS example, we assume no *a priori* information about server location, and instead use explicit discovery to find root and zone servers based on session types and server properties. We use an existing Scribble description of DNS as a starting point [21]. To illustrate adaptation we focus on the client who is querying DNS.

Fig. 7 shows the `session` type for the client actor which asks DNS to resolve a domain name. The client first asks for a root server (lines 2–3), and then either is informed that the request is invalid (lines 26–27) or recursively queries zone servers (lines 7–23) until the IP address is found (lines 20–22), or an error is reported (lines 17–18). Based on this `session`, StMungo generates EnsembleS types and interface definitions and a skeleton actor. Minimally completing the generated skeleton produces the code in Fig. 8.

In this example, discovery is used to locate the root server (lines 21–25, in Fig. 8) and the zone server (line 37). In each case, the `session` for the relevant server is provided to ensure that the discovered actor follows the expected protocol. When either server is located, the client links with it (lines 26 and 39), enabling communication. When communication with

```

1  type Iclient is interface(
2  out{RootServer,string} RootServer_stringOut,
3  in {RootServer,string} RootServer_stringIn,
4  out{ZoneServer,string} ZoneServer_stringOut,
5  in {ZoneServer,string} ZoneServer_stringIn,
6  in {ZoneServer,choice_enum} ZoneServer_choiceIn,
7  in {RootServer,choice_enum} RootServer_choiceIn)
8
9  type choice_enum is
10 enum(TLDResponse,PartialResolution,
11 InvalidDomain,ResolutionComplete,
12 InvalidTLD)
13
14 query find_name(string n){ $name == n; }
15
16 actor c presents Iclient
17 follows Client {
18 dom_name = "nii.ac.jp";
19 constructor() { }
20 behaviour{
21 rootQuery = find_name("jp");
22 // Find Root Server
23 root_s =
24   discover(IServer, RootServer, rootQuery);
25 // search until root_s non-empty
26 link me with root_s[0];
27 send domain_name on RootServer_stringOut;
28 receive c_msg from RootServer_choiceIn;
29 switch(c_msg){
30   case TLDResponse:
31     receive ZoneServerAddr_msg
32
33   from RootServer_stringIn;
34   unlink RootServer;
35   while(true) Lookup : {
36     // Find ZoneServer
37     zone_s =
38       discover(IServer, ZoneServer,
39         find_name(ZoneServerAddr_msg));
40     link me with zone_s[0];
41     // Ask ZoneServer
42     send dom_name on ZoneServer_stringOut;
43     receive c_msg2 from ZoneServer_choiceIn;
44     switch(c_msg2){
45       case PartialResolution:
46         receive str_msg from ZoneServer_stringIn;
47         ZoneServerAddr_msg := str_msg;
48         unlink ZoneServer;
49         continue Lookup;
50       case InvalidDomain:
51         receive str_msg from ZoneServer_stringIn;
52         unlink ZoneServer;
53         break;
54       case ResolutionComplete:
55         receive str_msg from ZoneServer_stringIn;
56         unlink ZoneServer;
57         break Lookup;
58     }
59     // keep looking
60   }
61   case InvalidTLD:
62     receive str_msg from RootServer_stringIn;
63     unlink RootServer;
64   } } }

```

■ Figure 8 EnsembleS DNS client

the server is no longer required, the client `unlinks` explicitly (lines 33, 47, 51, 55, 62).

Although explicit discovery is used at the language level, there is nothing to prevent the implementation of discovery from caching the addresses of the root and zone servers. This does not affect the use of sessions in discovery or the safety they provide, as the type-based guarantees are still enforced. However, this would potentially improve performance of the system. Additionally, if a cached entry becomes stale, the full discovery process can again be used without code modification or degradation in trust.

A version of DNS which uses discovery allows the system to become more flexible and resilient to changing operational conditions, such as topology changes in the servers and their data. Session types ensure compatibility with the discovered actors.

5 A Core Calculus for EnsembleS

In this section, we provide a formal characterisation of ENSEMBLES. In doing so, we show that our integration of adaptation with multiparty session types is safe, allowing adaptation while ruling out communication mismatches.

Relationship to implementation. Our core calculus aims to distil the essence of the interplay between adaptation and session-typed communication with explicit connection actions. Therefore, we concentrate on a functional core calculus rather than an imperative one: imperative variable binding serves only to clutter the formalism, and our fine-grain call-by-value representation can be thought of as an intermediate language.

Interfaces and unidirectional, simply-typed channels in ENSEMBLES are an implementation artifact: sending on a channel whose type changes is equivalent to sending on multiple

Syntax of Types and Terms

Actor class names	u	
Actor definitions	D	$::= \mathbf{actor} \ u \ \mathbf{follows} \ S \ \{M\}$
Roles	$\mathbf{p, q, s, t}$	
Recursion Labels	l	
Behaviours	κ	$::= M \mid \mathbf{stop}$
Types	A, B	$::= \mathbf{Pid}(S) \mid \mathbf{1}$
Values	V, W	$::= x \mid ()$
Actions	L	$::= \mathbf{return} \ V \mid \mathbf{continue} \ l \mid \mathbf{raise}$ $\mid \mathbf{new} \ u \mid \mathbf{self} \mid \mathbf{replace} \ V \ \mathbf{with} \ \kappa \mid \mathbf{discover} \ S$ $\mid \mathbf{connect} \ \ell(V) \ \mathbf{to} \ W \ \mathbf{as} \ \mathbf{p} \mid \mathbf{accept} \ \mathbf{from} \ \mathbf{p} \ \{\ell_i(x_i) \mapsto M_i\}_i$ $\mid \mathbf{send} \ \ell(V) \ \mathbf{to} \ \mathbf{p} \mid \mathbf{receive} \ \mathbf{from} \ \mathbf{p} \ \{\ell_i(x_i) \mapsto M_i\}_i$ $\mid \mathbf{wait} \ \mathbf{p} \mid \mathbf{disconnect} \ \mathbf{from} \ \mathbf{p}$
Computations	M, N	$::= \mathbf{let} \ x \leftarrow M \ \mathbf{in} \ N \mid \mathbf{try} \ L \ \mathbf{catch} \ M \mid l :: M \mid L$

Syntax of Session Types

Session Actions	α, β	$::= \mathbf{p}!\ell(A) \mid \mathbf{p}!\ell(A) \mid \mathbf{p}?\ell(A) \mid \mathbf{p}??\ell(A) \mid \#\uparrow\mathbf{p}$
Session Types	S, T, U	$::= \Sigma_{i \in I} (\alpha_i . S_i) \mid \mu X. S \mid X \mid \#\downarrow\mathbf{p} \mid \mathbf{end}$
Communication Actions	\dagger	$::= ! \mid ?$
Disconnection Actions	\ddagger	$::= \#\uparrow \mid \#\downarrow$

■ **Figure 9** Syntax

channels with different types. Moreover, following theoretical accounts of multiparty session types [31, 14, 32], instead of having send and receive (resp. connect and accept) operations followed by branching (as done in Mungo and StMungo), we have unified **send** and **receive** constructs which communicate a label along with the message payload.

Since session typing is the interesting part of discovery, we omit properties and queries from the formalism; their inclusion is routine. Finally, we concentrate on *dynamic* topologies with explicit connection actions rather than static topologies since they are important for adaptation and more interesting technically.

5.1 Syntax

Definitions. Figure 9 shows the syntax of Core ENSEMBLES terms and types. We let u range over actor class names and D range over definitions; each definition **actor** u **follows** $S \{M\}$ specifies the actor’s class name, session type, and behaviour. Like class tables in Featherweight Java [36], we assume a fixed mapping from class names to definitions.

Values. Since our calculus is inherently effectful, we work in the setting of *fine-grain call-by-value* [41], where we have an explicit static stratification of values and computations and an explicit evaluation order similar to A-normal form [20]. Values V, W describe data that has been computed, and for the sake of simplicity, consist of variables and the unit value. Other base values (such as integers or booleans) can be encoded or added straightforwardly.

Computations. The **let** $x \leftarrow M$ **in** N construct evaluates M , binding its result to x in N . The calculus supports exception handling over a single *action* L using **try** L **catch** M , where M is evaluated if L raises an exception, and labelled recursion using $l :: M$, stating that inside term M , a process can recurse to label l using **continue** l . *Actions* L denote the basic steps of a computation. The **return** V construct denotes a value.

12:12 Multiparty Session Types for Safe Runtime Adaptation in an Actor Language

Concurrency and adaptation constructs. The **new** u construct spawns a new actor of class u and returns its PID. The **self** construct returns the current actor's PID. An actor can replace the behaviour of itself or another actor V using **replace V with κ** . An actor can *discover* other actors following a session type S using the **discover S** construct, which returns the PID of the discovered actor.

Session communication constructs. An actor can connect to an actor W playing role \mathbf{p} using **connect $\ell(V)$ to W as \mathbf{p}** , sending a message with label ℓ and payload V . An actor can accept a connection from another actor playing role \mathbf{p} using **accept from \mathbf{p} $\{\ell_i(x_i) \mapsto M_i\}_i$** , which allows an actor to receive a choice of messages; given a message with label ℓ_j , the payload is bound to x_j in the continuation N_j . Once connected, an actor can communicate using the **send** and **receive** constructs. An actor can disconnect from \mathbf{p} using **disconnect from \mathbf{p}** , and await the disconnection of \mathbf{p} using **wait \mathbf{p}** .

Types. Types, ranged over by A, B , include the unit type $\mathbf{1}$ and process IDs $\text{Pid}(S)$; the parameter S refers to the statically-known initial session type of the actor (i.e., the session type declared in the **follows** clause of a definition). Unlike in channel-based session-typed systems, process IDs themselves need not be linear: any number of actors can have a *reference* to another actor, but each actor may only be in a single session at a time. PIDs can be passed as payloads in session communications.

Session types. Session types are ranged over by S, T, U and follow the formulation of Hu & Yoshida [32]. A session type can be a choice of actions, written $\Sigma_{i \in I}(\alpha_i . S_i)$, a recursive session type $\mu X.S$ binding recursion variable X in continuation S , a recursion variable X , a disconnection action $\#\downarrow\mathbf{p}$, or the finished session **end**. The syntax of session types is more liberal than traditional 'directed' presentations in order to allow output-directed choices to send or connect to different roles.

Session actions α involve sending (!), receiving (?), connecting (!!), or accepting (??) a message $\ell(A)$ with label ℓ and type A ; or awaiting another participant's disconnection ($\#\uparrow$). As well as disallowing self-communication, following Hu & Yoshida [32], we require the following syntactic restrictions on session types:

► **Definition 1** (Syntactic validity). *A choice type $S = \Sigma_{i \in I}(\alpha_i . S_i)$ is syntactically valid if:*

1. *it is an output choice, i.e., each α_i is a send or connection action; or*
2. *it is a directed input choice, i.e., $S = \Sigma_{i \in I}(\mathbf{p}?\ell_i(A_i).S_i)$ or $S = \Sigma_{i \in I}(\mathbf{p}??\ell_i(A_i).S_i)$; or*
3. *the choice consists of single wait action $\#\uparrow\mathbf{p}.S$.*

In the remainder of the paper, we assume that all session types are syntactically valid.

Session correlation. The most general form of explicit connection actions allows a participant to leave and re-join a session, or accept connections from multiple different participants. Such generality comes at a cost, since care must be taken to ensure that the *same* participant plays the role throughout the session.

To address this *session correlation* issue, Hu & Yoshida [32] propose two solutions: either augment global types with type-level assertions and check conformance dynamically, or adopt a lightweight syntactic restriction which requires that each local type must contain at most a single accept action as its top-level construct. We opt for the latter, enforcing the constraint as part of our safety property (§5.4.2), and by requiring that $\#\downarrow\mathbf{p}$ does not have a continuation. (Note that the behaviour will repeat, so \mathbf{p} will be able to accept again after

disconnecting). As Hu & Yoshida [32] show, this design still supports the most common use cases of explicit connection actions.

Global types. Traditional MPST works [31, 14] use *global types* to describe the interactions between participants at a global level, which are then projected into *local types*; projectability ensures safety and deadlock-freedom.

Since we are using explicit connection actions, traditional approaches are insufficiently flexible as they do not account for certain roles being present in certain branches but not others. Following [53] and subsequently non-classical MPSTs [54], we instead formulate our typing rules and safety properties using collections of local types.

It is, however, still convenient to write a global type and have local types computed programatically. Global types are defined as follows:

$$\begin{array}{lcl} \text{Global Actions } \pi & ::= & \mathbf{p} \rightarrow \mathbf{q} : \ell(A) \mid \mathbf{p} \rightarrow \mathbf{q} : \ell(A) \mid \mathbf{p}\#\mathbf{q} \\ \text{Global Types } G & ::= & \Sigma_{i \in I} (\pi_i . G_i) \mid \mu X . G \mid X \mid \text{end} \end{array}$$

Global actions π describe interactions between participants: $\mathbf{p} \rightarrow \mathbf{q} : \ell(A)$ states that role \mathbf{p} sends a message with label ℓ and payload type A to \mathbf{q} . Similarly, $\mathbf{p} \rightarrow \mathbf{q} : \ell(A)$ states that \mathbf{p} connects to \mathbf{q} by sending a message with label ℓ and payload type A . The disconnection action $\mathbf{p}\#\mathbf{q}$ states that role \mathbf{p} disconnects from role \mathbf{q} .

We can write the `OnlineStore` example from §2 as follows:

```

Customer → Store : login(String) . μBrowse .
Customer → Store : item(String) . Store → Customer : price(Int) . Browse
+
Customer → Store : address(String) . Store → Courier : deliver(String) .
Courier → Store : ref(Int) . Courier#Store . Store → Customer : ref(Int) .
Store#Customer . end
+
Customer → Store : quit(1) . Store#Customer . end

```

Although projectability in our setting does not necessarily guarantee safety and deadlock-freedom, we show a projection algorithm, adapted from that of Hu & Yoshida [32], in the extended version. The resulting local types can then be checked for safety (§5.4.2).

Protocols and Programs. Terms do not live in isolation; they refer to a set of *protocols*, and evaluate in the context of an actor. A *protocol* maps role names to local session types.

► **Definition 2** (Protocol). A protocol is a set $\{\mathbf{p}_i : S_i\}_i$ mapping role names to session types.

As an example, consider the protocol for the online shop example:

$$\left\{ \begin{array}{l} \mathbf{Customer} : \mathbf{Store}!!\text{login}(\text{String}) . \mu\text{Browse} . \\ \quad \mathbf{Store}!\text{item}(\text{String}) . \mathbf{Store}?\text{price}(\text{Int}) . \text{Browse} \\ + \mathbf{Store}!\text{address}(\text{String}) . \mathbf{Store}?\text{ref}(\text{Int}) . \#\uparrow\mathbf{Store} . \text{end} \\ + \mathbf{Store}!\text{quit}(1) . \#\uparrow\mathbf{Store} . \text{end}, \\ \mathbf{Store} : \mathbf{Customer}??\text{login}(\text{String}) . \mu\text{Browse} . \\ \quad \mathbf{Customer}?\text{item}(\text{String}) . \mathbf{Customer}!\text{price}(\text{Int}) . \text{Browse} \\ + \mathbf{Customer}?\text{address}(\text{String}) . \mathbf{Courier}!!\text{deliver}(\text{String}) . \mathbf{Courier}?\text{ref}(\text{Int}) . \\ \quad \#\uparrow\mathbf{Courier} . \mathbf{Customer}!\text{ref}(\text{Int}) . \#\downarrow\mathbf{Customer} \\ + \mathbf{Customer}?\text{quit}(1) . \#\downarrow\mathbf{Customer}, \\ \mathbf{Courier} : \mathbf{Store}??\text{deliver}(\text{String}) . \mathbf{Store}!\text{ref}(\text{Int}) . \#\downarrow\mathbf{Store} \end{array} \right.$$

We can now consider an implementation of a `Store` actor, which uses discovery to find a courier. We write **receive** $\ell(x)$ **from** \mathbf{p} ; M and **accept** $\ell(x)$ **from** \mathbf{p} ; M as syntactic sugar for **receive from** \mathbf{p} $\{\ell(x) \mapsto M\}$ and **accept from** \mathbf{p} $\{\ell(x) \mapsto M\}$ respectively, and write $M; N$ as

12:14 Multiparty Session Types for Safe Runtime Adaptation in an Actor Language

syntactic sugar for $\mathbf{let} x \leftarrow M \mathbf{in} N$ for a fresh variable x . We assume the existence of a function `lookupPrice`, and define `CourierType` as $\mathbf{Store}??\mathit{deliver}(\mathbf{String}) . \mathbf{Store}!\mathit{ref}(\mathbf{Int}) . \#\downarrow\mathbf{Store}$.

```

actor Store follows ty(Store) {
  accept login(credentials) from Customer;
  Browse ::
  receive from Customer {
    item(name)  $\mapsto$ 
      send price(lookupPrice(name)) to Customer;
      continue Browse
    address(addr)  $\mapsto$ 
      let pid  $\leftarrow$  discover CourierType in
      connect deliver(addr) to pid as Courier;
      receive ref(r) from Courier;
      wait Courier;
      send ref(r) to Customer;
      disconnect from Customer
    quit()  $\mapsto$  disconnect from Customer
  }
}

```

A *program* consists of actor definitions, protocol definitions, and the ‘boot’ clause to be run in order to set up initial actor communication.

► **Definition 3 (Program).** An *EnsembleS* program is a 3-tuple (\vec{D}, \vec{P}, M) of a set of definitions, protocols, and an initial term to be evaluated.

In the context of a program, we write $\text{ty}(\mathbf{p})$ to refer to the session type associated with role \mathbf{p} as defined by the set of protocols. Given an actor definition $\mathbf{actor} u \mathbf{follows} S \{M\}$, we define $\text{sessionType}(u) = S$ and $\text{behaviour}(u) = M$.

5.2 Typing rules

Figures 10 and 11 show the typing rules for EnsembleS. Value typing, with judgement $\Gamma \vdash V:A$, states that under environment Γ , value V has type A . Judgement $\vdash D$ states that an actor definition $\mathbf{actor} u \mathbf{follows} S \{M\}$ is well-typed if its body is typable under, and fully consumes, its statically-defined session type S . The behaviour typing judgement $\{S\} \Gamma \vdash \kappa$ states that given static session type S , behaviour κ is well-typed under Γ . Specifically, **stop** is always well-typed, and M is well-typed if it is typable under and fully consumes S .

5.2.1 Term typing.

The typing judgement for terms $\{T\} \Gamma \mid S \triangleright M:A \triangleleft S'$ reads “in an actor following T , under typing environment Γ and with current session type S , term M has type A and updates the session type to S' ”. Note that the term typing judgement, reminiscent of parameterised monads [3], contains a session precondition S and may perform some session communication actions to arrive at postcondition S' .

Functional rules. Rule T-LET is a sequencing operation: given a construct $\mathbf{let} x \leftarrow M \mathbf{in} N$ where M has pre-condition S and post-condition S' , and where N has pre-condition S' and post-condition S'' , the overall construct has pre-condition S and post-condition S'' .

Following Kouzapas *et al.* [40], we formalise recursion through annotated expressions: term $l::M$ states that M is an expression which can loop to l by evaluating **continue** l . We take an equi-recursive view of session types, identifying recursive sessions with their unfolding $(\mu X.S = S\{\mu X.S/X\})$, and assume that recursion is guarded. Rule T-REC extends

Definition typing $\boxed{\vdash D}$	Value typing $\boxed{\Gamma \vdash V:A}$	Behaviour typing $\boxed{\{S\} \Gamma \vdash \kappa}$
$\frac{\text{T-DEF}}{\{S\} \cdot S \triangleright M:A \triangleleft \text{end}} \vdash \mathbf{actor } u \text{ follows } S \{M\}$	$\frac{\text{T-VAR}}{x : A \in \Gamma} \Gamma \vdash x:A$	$\frac{\text{T-UNIT}}{\Gamma \vdash ():\mathbf{1}}$
	$\frac{\text{T-STOP}}{\{S\} \Gamma \vdash \mathbf{stop}}$	$\frac{\text{T-BODY}}{\{S\} \Gamma \vdash M}$
Typing rules for computations $\boxed{\{T\} \Gamma S \triangleright M:A \triangleleft S'}$		
<i>Functional Rules</i>		
$\frac{\text{T-LET}}{\{T\} \Gamma S \triangleright M:A \triangleleft S' \quad \{T\} \Gamma, x:A S' \triangleright N:B \triangleleft S''} \{T\} \Gamma S \triangleright \mathbf{let } x \leftarrow M \text{ in } N:B \triangleleft S''$	$\frac{\text{T-RETURN}}{\Gamma \vdash V:A} \{T\} \Gamma S \triangleright \mathbf{return } V:A \triangleleft S$	
$\frac{\text{T-REC}}{\{T\} \Gamma, l:S S \triangleright M:A \triangleleft S'} \{T\} \Gamma S \triangleright l::M:A \triangleleft S'$	$\frac{\text{T-CONTINUE}}{\{T\} \Gamma, l:S S \triangleright \mathbf{continue } l:A \triangleleft S'}$	
<i>Actor / Adaptation Rules</i>		
$\frac{\text{T-NEW}}{\text{sessionType}(u) = U} \{T\} \Gamma S \triangleright \mathbf{new } u:\text{Pid}(U) \triangleleft S$	$\frac{\text{T-SELF}}{\{T\} \Gamma S \triangleright \mathbf{self}:\text{Pid}(T) \triangleleft S}$	$\frac{\text{T-DISCOVER}}{\{T\} \Gamma S \triangleright \mathbf{discover } U:\text{Pid}(U) \triangleleft S}$
	$\frac{\text{T-REPLACE}}{\Gamma \vdash V:\text{Pid}(U) \quad \{U\} \Gamma \vdash \kappa} \{T\} \Gamma S \triangleright \mathbf{replace } V \text{ with } \kappa:\mathbf{1} \triangleleft S$	

■ **Figure 10** Typing rules (1)

the typing environment with a recursion label defined at the current session type. Rule T-CONTINUE ensures that the pre-condition must match the label stored in the environment, but has arbitrary type and any post-condition since the return type and post-condition depend on the enclosing loop's base case.

Actor and adaptation rules. Rule T-NEW states that creating an actor of class u returns a PID parameterised by the session type declared in the class of u . Rule T-SELF retrieves a PID for the current actor, parameterised by the statically-defined session type of the local actor (i.e., the T in the judgement $\{T\} \Gamma | S \triangleright M:A \triangleleft S'$). Rule T-DISCOVER states **discover** U returns a PID of type $\text{Pid}(U)$. Finally, given a behaviour κ typable under a static session type U , and a process ID with the matching static type $\text{Pid}(U)$, T-REPLACE allows replacement, and returns the unit type.

Exception handling rules. Figure 11 shows the rules for exception handling and session communication. T-RAISE denotes raising an exception; since it does not return, it can have an arbitrary return type and postcondition. Rule T-TRY types an exception handler **try** L **catch** M which acts over a single action L . If L raises an exception, then M is evaluated instead. Since L only scopes over a single action, the **try** and **catch** clauses have the same pre- and post-conditions to allow the action to be retried if necessary.

► **Remark 4.** Following Mostrous & Vasconcelos [45], our **try** L **catch** M construct scopes over a *single* action and is discarded afterwards. We opt for this simple approach since in our setting exceptions are a means to an end, but (at the coast of a more involved type system)

12:16 Multiparty Session Types for Safe Runtime Adaptation in an Actor Language

Exception handling rules

$$\frac{\text{T-RAISE}}{\{T\} \Gamma \mid S \triangleright \mathbf{raise}:A \triangleleft S'}$$

$$\frac{\text{T-TRY} \quad \{T\} \Gamma \mid S \triangleright L:A \triangleleft S' \quad \{T\} \Gamma \mid S \triangleright M:A \triangleleft S'}{\{T\} \Gamma \mid S \triangleright \mathbf{try} L \mathbf{catch} M:A \triangleleft S'}$$

Session communication rules

$$\frac{\text{T-CONN} \quad \mathbf{p}_j! \ell_j(A_j) \in \{\alpha_i\}_{i \in I} \quad \Gamma \vdash V:A_j \quad \Gamma \vdash W:\text{Pid}(T) \quad T = \mathbf{ty}(\mathbf{p}_j)}{\{T\} \Gamma \mid \Sigma_{i \in I}(\alpha_i . S_i) \triangleright \mathbf{connect} \ell_j(V) \mathbf{to} W \mathbf{as} \mathbf{p}_j:\mathbf{1} \triangleleft S'_j}$$

$$\frac{\text{T-SEND} \quad \mathbf{p}_j! \ell_j(A_j) \in \{\alpha_i\}_{i \in I} \quad \Gamma \vdash V:A_j}{\{T\} \Gamma \mid \Sigma_{i \in I}(\alpha_i . S_i) \triangleright \mathbf{send} \ell_j(V) \mathbf{to} \mathbf{p}_j:\mathbf{1} \triangleleft S'_j}$$

$$\frac{\text{T-ACCEPT} \quad (\{T\} \Gamma, x_i : B_i \mid S_i \triangleright M_i:A \triangleleft S)_{i \in I}}{\{T\} \Gamma \mid \Sigma_{i \in I}(\mathbf{q}??\ell_i(B_i) . S_i) \triangleright \mathbf{accept} \mathbf{from} \mathbf{q} \{\ell_i(x_i) \mapsto M_i\}_{i \in I}:A \triangleleft S}$$

$$\frac{\text{T-RECV} \quad (\{T\} \Gamma, x_i : B_i \mid S_i \triangleright M_i:A \triangleleft S)_{i \in I}}{\{T\} \Gamma \mid \Sigma_{i \in I}(\mathbf{q}?\ell_i(B_i) . S_i) \triangleright \mathbf{receive} \mathbf{from} \mathbf{q} \{\ell_i(x_i) \mapsto M_i\}_{i \in I}:A \triangleleft S}$$

$$\frac{\text{T-WAIT}}{\{T\} \Gamma \mid \#\uparrow \mathbf{p} . S \triangleright \mathbf{wait} \mathbf{q}:\mathbf{1} \triangleleft S}$$

$$\frac{\text{T-DISCONN}}{\{T\} \Gamma \mid \#\downarrow \mathbf{q} \triangleright \mathbf{disconnect} \mathbf{from} \mathbf{q}:\mathbf{1} \triangleleft \mathbf{end}}$$

■ **Figure 11** Typing rules (2)

we could potentially scope over multiple actions as long as the handler is compatible with all potential exit conditions [23]. We leave a thorough exploration to future work.

Session communication rules. Rule T-CONN types a term **connect** $\ell_j(V)$ **to** W **as** \mathbf{p}_j . Given the precondition is a choice type containing a branch $\mathbf{p}_j! \ell_j(A_j) . S'_j$, and the remote actor reference is W of type $\text{Pid}(S)$, the rule ensures that S is compatible with the type of \mathbf{p}_j , and ensures that the label and payload are compatible with the session type. The session type is then advanced to S'_j . Rule T-SEND follows the same pattern.

Given a session type $\Sigma_{i \in I}(\mathbf{p}??x_i(A_i)) . S_i$, rule T-ACCEPT types term **accept from** $\mathbf{p} \{\ell_i(x_i) \mapsto M_i\}_{i \in I}$, enabling an actor to accept connections with messages ℓ_i , binding the payload x_i in each continuation M_i . Like **case** expressions in functional languages, each continuation must be typable under an environment extended with $x_i : A_i$, under session type S_i , and each branch must have same result type and postcondition. Rule T-RECV is similar.

Rule T-WAIT handles waiting for a participant \mathbf{p} to disconnect from a session, requiring a pre-condition of $\#\uparrow \mathbf{p} . S$, returning the unit type and advancing the session type to S . Rule T-DISCONNECT is similar and advances the session type to \mathbf{end} .

5.3 Operational semantics

We describe the semantics of EnsembleS via a deterministic reduction relation on terms, and a nondeterministic reduction relation on configurations.

5.3.1 Runtime syntax

Figure 12 shows the runtime syntax and the first part of the reduction rules for EnsembleS.

Runtime syntax

Names	n	$::=$	$a \mid s$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E}$	$::=$	$(\nu n)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \langle a, M, \sigma, \kappa \rangle \mid \zeta s[\mathbf{p}] \mid \mathbf{0}$
Connection state	σ	$::=$	$\perp \mid s[\mathbf{p}]\langle \bar{\mathbf{q}} \rangle$
Runtime environments	Δ	$::=$	$\cdot \mid \Delta, a : S \mid \Delta, s[\mathbf{p}]\langle \bar{\mathbf{q}} \rangle : S$
Evaluation contexts	E	$::=$	$F \mid \mathbf{let } x \leftarrow E \mathbf{ in } M$
Top-level contexts	F	$::=$	$[] \mid \mathbf{try } [] \mathbf{ catch } M$
Pure contexts	$E_{\mathbf{p}}$	$::=$	$[] \mid \mathbf{let } x \leftarrow E_{\mathbf{p}} \mathbf{ in } M$

Term reduction

$$\boxed{M \longrightarrow_M N}$$

E-LET	$\mathbf{let } x \leftarrow \mathbf{return } V \mathbf{ in } M$	\longrightarrow_M	$M\{V/x\}$
E-TRYRETURN	$\mathbf{try return } V \mathbf{ catch } M$	\longrightarrow_M	$\mathbf{return } V$
E-TRYRAISE	$\mathbf{try raise catch } M$	\longrightarrow_M	M
E-REC	$l :: M$	\longrightarrow_M	$M\{l :: M/\mathbf{continue } l\}$
E-LIFTM	$E[M]$	\longrightarrow_M	$E[N]$ if $M \longrightarrow_M N$

Configuration reduction (1)

$$\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$$

Actor / adaptation rules

E-LOOP	$\frac{}{\langle a, \mathbf{return } V, \perp, M \rangle \longrightarrow \langle a, M, \perp, M \rangle}$	E-NEW	$\frac{b \text{ is fresh} \quad \mathbf{behaviour}(u) = M}{\langle a, E[\mathbf{new } u], \sigma, \kappa \rangle \longrightarrow (\nu b)(\langle a, E[\mathbf{return } b], \sigma, \kappa \rangle \parallel \langle b, M, \perp, M \rangle)}$
E-REPLACE	$\frac{}{\langle a, E[\mathbf{replace } b \text{ with } \kappa'], \sigma_1, \kappa_1 \rangle \parallel \langle b, N, \sigma_2, \kappa_2 \rangle \longrightarrow \langle a, E[\mathbf{return } ()], \sigma_1, \kappa_1 \rangle \parallel \langle b, N, \sigma_2, \kappa' \rangle}$	E-REPLACESSELF	$\frac{}{\langle a, E[\mathbf{replace } a \text{ with } \kappa'], \sigma, \kappa \rangle \longrightarrow \langle a, E[\mathbf{return } ()], \sigma, \kappa' \rangle}$
E-DISCOVER	$\frac{\mathbf{sessionType}(b) = S \quad \neg((N = \mathbf{return } V \vee N = \mathbf{raise}) \wedge \kappa_2 = \mathbf{stop})}{\langle a, E[\mathbf{discover } S], \sigma_1, \kappa_1 \rangle \parallel \langle b, E'[N], \sigma_2, \kappa_2 \rangle \longrightarrow \langle a, E[\mathbf{return } b], \sigma_1, \kappa_1 \rangle \parallel \langle b, E'[N], \sigma_2, \kappa_2 \rangle}$	E-SELF	$\frac{}{\langle a, E[\mathbf{self}], \sigma, \kappa \rangle \longrightarrow \langle a, E[\mathbf{return } a], \sigma, \kappa \rangle}$

■ **Figure 12** Operational semantics (1)

Whereas static syntax and typing rules describe code that a user would write, runtime syntax arises during evaluation. We introduce two types of runtime name: s ranges over *session names*, which are created when a process initiates a session, and a ranges over *actor names*, which uniquely identify each actor once it has been spawned by **new**.

Configurations. Configurations, ranged over by $\mathcal{C}, \mathcal{D}, \mathcal{E}$, represent the concurrent fragment of the language. Like in the π -calculus [42], name restrictions $(\nu n)\mathcal{C}$ bind name n in \mathcal{C} , $\mathcal{C} \parallel \mathcal{D}$ denotes \mathcal{C} and \mathcal{D} running in parallel, and the $\mathbf{0}$ configuration denotes the inactive process.

Actors are represented at runtime as a 4-tuple $\langle a, M, \sigma, \kappa \rangle$, where a is the actor's runtime name; M is the term currently evaluating; σ is the connection state; and κ is the actor's current behaviour. A connection state is either *disconnected*, written \perp , or playing role \mathbf{p} in session s and connected to roles $\bar{\mathbf{q}}$, written $s[\mathbf{p}]\langle \bar{\mathbf{q}} \rangle$.

Inspired by Mostrous & Vasconcelos [45] and Fowler *et al.* [22], a *zapper thread* $\zeta s[\mathbf{p}]$ indicates that participant \mathbf{p} in session s cannot be used for future communications, for example due to the actor playing the role crashing due to an unhandled exception.

To run a program, we place it in an *initial configuration*. of the form $(\nu a)(\langle a, M, \perp, \mathbf{stop} \rangle)$.

Configuration reduction (2)
 $\boxed{C \longrightarrow D}$
Session reduction rules

$$\begin{array}{c}
 \text{E-CONNINIT} \\
 \frac{j \in I}{\langle a, E[F[\mathbf{connect} \ell_j(V) \text{ to } b \text{ as } \mathbf{q}]], \perp, \kappa_1 \rangle \parallel \langle b, E'[F'[\mathbf{accept from } \mathbf{p} \{ \ell_i(x_i) \mapsto M_i \}_{i \in I}]], \perp, \kappa_2 \rangle \longrightarrow (\nu s)(\langle a, E[\mathbf{return} ()], s[\mathbf{p}]\langle \mathbf{q} \rangle, \kappa_1 \rangle \parallel \langle b, E'[M_j\{V/x_j\}], s[\mathbf{q}]\langle \mathbf{p} \rangle, \kappa_2 \rangle)} \\
 \\
 \text{E-CONN} \\
 \frac{\mathbf{q} \notin \tilde{r}}{\langle a, E[F[\mathbf{connect} \ell_j(V) \text{ to } b \text{ as } \mathbf{q}]], s[\mathbf{p}]\langle \tilde{r} \rangle, \kappa_1 \rangle \parallel \langle b, E'[F'[\mathbf{accept from } \mathbf{p} \{ \ell_i(x_i) \mapsto N_i \}_{i \in I}]], \perp, \kappa_2 \rangle \longrightarrow (\nu s)(\langle a, E[\mathbf{return} ()], s[\mathbf{p}]\langle \tilde{r}, \mathbf{q} \rangle, \kappa_1 \rangle \parallel \langle b, E'[N_j\{V/x_j\}], s[\mathbf{q}]\langle \mathbf{p} \rangle, \kappa_2 \rangle)} \\
 \\
 \text{E-CONNFAIL} \\
 \frac{((N = \mathbf{return} V \vee N = E_{\mathbf{p}}[\mathbf{raise}]) \wedge \kappa_2 = \mathbf{stop}) \vee \sigma_2 \neq \perp}{\langle a, E[\mathbf{connect} \ell_j(V) \text{ to } b \text{ as } \mathbf{q}]], \sigma_1, \kappa_1 \rangle \parallel \langle b, N, \sigma_2, \kappa_2 \rangle \longrightarrow \langle a, E[\mathbf{raise}], \sigma_1, \kappa_1 \rangle \parallel \langle b, N, \sigma_2, \kappa_2 \rangle} \\
 \\
 \text{E-DISCONN} \\
 \frac{\langle a, E[F[\mathbf{wait} \mathbf{q}]], s[\mathbf{p}]\langle \tilde{r}, \mathbf{q} \rangle, \kappa_1 \rangle \parallel \langle b, E'[F'[\mathbf{disconnect from } \mathbf{p}]], s[\mathbf{q}]\langle \mathbf{p} \rangle, \kappa_2 \rangle \longrightarrow (\nu s)(\langle a, E[\mathbf{return} ()], s[\mathbf{p}]\langle \tilde{r} \rangle, \kappa_1 \rangle \parallel \langle b, E'[\mathbf{return} ()], \perp, \kappa_2 \rangle)} \\
 \\
 \text{E-COMM} \\
 \frac{j \in I \quad \mathbf{q} \in \tilde{r} \quad \mathbf{p} \in \tilde{s}}{\langle a, E[F[\mathbf{send} \ell_j(V) \text{ to } \mathbf{q}]], s[\mathbf{p}]\langle \tilde{r} \rangle, \kappa_1 \rangle \parallel \langle b, E'[F'[\mathbf{receive from } \mathbf{p} \{ \ell_i(x_i) \mapsto M_i \}_{i \in I}]], s[\mathbf{q}]\langle \tilde{s} \rangle, \kappa_2 \rangle \longrightarrow (\nu s)(\langle a, E[\mathbf{return} ()], s[\mathbf{p}]\langle \tilde{r} \rangle, \kappa_1 \rangle \parallel \langle b, E'[M_j\{V/x_j\}], s[\mathbf{q}]\langle \tilde{s} \rangle, \kappa_2 \rangle)} \\
 \\
 \text{E-COMPLETE} \\
 \frac{(\nu s)(\langle a, \mathbf{return} V, s[\mathbf{p}]\langle \emptyset \rangle, \kappa \rangle)}{\langle a, \mathbf{return} V, \perp, \kappa \rangle}
 \end{array}$$

■ **Figure 13** Operational semantics (2)

Runtime typing environments. Whereas Γ is an unrestricted typing environment used for typing values and configurations, we introduce Δ as a linear runtime environment. Runtime environments can contain entries of type $a : S$, stating that actor a has *statically-defined* session type S , and entries of type $s[\mathbf{p}]\langle \tilde{\mathbf{q}} \rangle : S$, stating that in session s , role \mathbf{p} is connected to roles $\tilde{\mathbf{q}}$ and *currently* has session type S .

Evaluation contexts. Due to our fine-grain call-by-value presentation, evaluation contexts E allow nesting only in the immediate subterm of a **let** expression. The top-level frame F can either be a hole, or a single, top-level exception handler. Pure contexts $E_{\mathbf{p}}$ do not include exception handling frames.

5.3.2 Reduction rules

Term reduction $\longrightarrow_{\mathbf{M}}$ is standard β -reduction, save for E-TRYRAISE which evaluates the failure continuation in the case of an exception. We consider four subcategories of configuration reduction rules: actor and adaptation rules; session communication rules; exception handling rules; and administrative rules.

Actor / adaptation rules. Given a fully-evaluated actor, E-LOOP runs the term specified by the actor's behaviour. Rule E-NEW allows actor a to spawn a new actor of class u by creating a fresh runtime actor name b and a new actor process of the form $\langle b, M, \perp, M \rangle$ where M is the behaviour specified by u , returning the process ID b . Rules E-REPLACE and

Configuration reduction (3)

$$\boxed{C \longrightarrow D}$$

Exception handling rules

$$\frac{\text{E-COMMRAISE} \quad \text{subj}(M) = \mathbf{q}}{\langle a, E[M], s[\mathbf{p}]\langle \bar{r} \rangle, \kappa \rangle \parallel \not\downarrow s[\mathbf{q}] \longrightarrow \langle a, E[\mathbf{raise}], s[\mathbf{p}]\langle \bar{r} \rangle, \kappa \rangle \parallel \not\downarrow s[\mathbf{q}]}$$

$$\frac{\text{E-FAILS}}{\langle a, E_{\mathbf{P}}[\mathbf{raise}], s[\mathbf{p}]\langle \bar{r} \rangle, \kappa \rangle \longrightarrow \langle a, \mathbf{raise}, \perp, \kappa \rangle \parallel \not\downarrow s[\mathbf{p}]}$$

$$\frac{\text{E-FAILLOOP}}{\langle a, E_{\mathbf{P}}[\mathbf{raise}], \perp, M \rangle \longrightarrow \langle a, M, \perp, M \rangle}$$

Administrative rules

$$\frac{\text{E-LIFTM} \quad M \longrightarrow_M M'}{\langle a, E[M], \sigma, \kappa \rangle \longrightarrow \langle a, E[M'], \sigma, \kappa \rangle}$$

$$\frac{\text{E-EQUIV} \quad C \equiv C' \quad C' \longrightarrow D' \quad D' \equiv D}{C \longrightarrow D}$$

$$\frac{\text{E-PAR} \quad C \longrightarrow C'}{C \parallel D \longrightarrow C' \parallel D}$$

$$\frac{\text{E-NU} \quad C \longrightarrow D}{(\nu n)C \longrightarrow (\nu n)D}$$

Configuration equivalence

$$\boxed{C \equiv D}$$

$$C \parallel D \equiv D \parallel C \quad C \parallel (D \parallel E) \equiv (C \parallel D) \parallel E \quad (\nu n_1)(\nu n_2)C \equiv (\nu n_2)(\nu n_1)C$$

$$C \parallel (\nu n)D \equiv (\nu n)(C \parallel D) \quad \text{if } n \notin \text{fn}(C) \quad (\nu s)(\not\downarrow s[\mathbf{p}_1] \parallel \dots \parallel \not\downarrow s[\mathbf{p}_n]) \parallel C \equiv C \quad C \parallel \mathbf{0} \equiv C$$

■ **Figure 14** Operational semantics (3)

E-REPLACESSELF handle replacement by changing the behaviour of an actor, returning the unit value to the caller. Rule E-DISCOVER returns the process ID of an actor b if it has the desired static session type S . Rule E-SELF returns the PID of the local actor.

Session communication rules. An actor begins a session by connecting to another actor while disconnected; such a case is handled by rule E-CONNINIT. Suppose we have a disconnected actor a evaluating a connection statement **connect** $\ell_j(V)$ **to** b **as** \mathbf{p} , evaluating in parallel with a disconnected actor b evaluating an accept statement **accept from** \mathbf{p} $\{\ell_i(x_i) \mapsto M_i\}_{i \in I}$. Rule E-CONNINIT returns the unit value to actor a ; creates a fresh session name restriction s , sets the connection state of a to $s[\mathbf{p}]\langle \mathbf{q} \rangle$ and of b to $s[\mathbf{q}]\langle \mathbf{p} \rangle$; accepting actor b then evaluates continuation M_j with V substituted for x_j . Since exception handlers only scope over a single communication action, the top-level frames F, F' in each actor are discarded if the communication succeeds. Rule E-CONN handles the case where the connecting actor is already part of a session and behaves similarly to E-CONNINIT, without creating a new session name restriction. A connection can fail if an actor attempts to connect to another actor which is terminated or is already involved in a session; in these cases, E-CONNFAIL raises an exception in the connecting actor.

Rule E-DISCONN handles the case where an actor b leaves a session, synchronising with an actor a . In this case, the unit value is returned to both callers, and the connection state of b is set to \perp . Rule E-COMM handles session communication when two participants are already connected to the same session, and is similar to E-CONN. Rule E-COMPLETE garbage collects a session after it has completed and sets the initiator's connection state to \perp .

Exception handling rules. Exception handling rules allow safe session communication in the presence of exceptions. Rule E-COMMRAISE states that if an actor is attempting to communicate with a role no longer present due to an exception, then an exception should be raised. We write $\text{subj}(E[M]) = \mathbf{p}$ if $M \in \{\mathbf{send} \ell(V) \mathbf{to} \mathbf{p}, \mathbf{receive from} \mathbf{p} \{\ell_i(x_i) \mapsto N_i\}_i, \mathbf{wait} \mathbf{p}, \mathbf{disconnect from} \mathbf{p}\}$. Rule E-FAILS states that if a connected actor encounters an unhandled exception, then a zipper thread will be generated for the current role, the

Runtime Typing Rules

$$\begin{array}{c}
 \boxed{\Gamma \vdash V:A} \quad \boxed{\Gamma; \Delta \vdash \mathcal{C}} \\
 \\
 \text{T-PID} \quad \frac{\Gamma, a : \text{Pid}(S); \Delta, a : S \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}} \\
 \\
 \text{T-SESSION} \quad \frac{\Delta' = \{s[\mathbf{p}_i](\tilde{\mathbf{q}}_i):S_{\mathbf{p}_i}\}_{i \in I} \quad \varphi(\Delta') \quad s \notin \Delta \quad \Gamma; \Delta, \Delta' \vdash \mathcal{C} \quad \varphi \text{ is a safety property}}{\Gamma; \Delta \vdash (\nu s)\mathcal{C}} \\
 \\
 \text{T-PAR} \quad \frac{\Gamma; \Delta_1 \vdash \mathcal{C} \quad \Gamma; \Delta_2 \vdash \mathcal{D}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C} \parallel \mathcal{D}} \quad \text{T-ZAP} \quad \frac{}{\Gamma; s[\mathbf{p}](\tilde{\mathbf{q}}):S \vdash \not\leq s[\mathbf{p}]} \quad \text{T-ZERO} \quad \frac{}{\Gamma; \cdot \vdash \mathbf{0}} \\
 \\
 \text{T-DISCONNECTEDACTOR} \quad \frac{T = S \vee T = \text{end} \quad a : \text{Pid}(S) \in \Gamma \quad \{S\} \Gamma \mid T \triangleright M:A \triangleleft \text{end} \quad \{S\} \Gamma \vdash \kappa}{\Gamma; a : S \vdash \langle a, M, \perp, \kappa \rangle} \quad \text{T-CONNECTEDACTOR} \quad \frac{a : \text{Pid}(T) \in \Gamma \quad \{T\} \Gamma \mid S \triangleright M:A \triangleleft \text{end} \quad \{T\} \Gamma \vdash \kappa}{\Gamma; a : T, s[\mathbf{p}](\tilde{\mathbf{q}}):S \vdash \langle a, M, s[\mathbf{p}](\tilde{\mathbf{q}}), \kappa \rangle}
 \end{array}$$

■ **Figure 15** Runtime typing rules

actor will become disconnected, and the current evaluation context will be discarded. Rule E-FAILLOOP restarts an actor encountering an unhandled exception.

Administrative rules. The remaining rules are administrative: E-LIFTM allows term reduction inside an actor; E-EQUIV allows reduction modulo structural congruence; E-PAR allows reduction under parallel composition; and E-NU allows reduction under name restrictions.

Configuration equivalence. Reduction includes configuration equivalence \equiv , defined as the smallest congruence relation satisfying the axioms in Figure 14. The equivalence rules extend the usual π -calculus structural congruence rules with a ‘garbage collection’ equivalence, which allows us to discard a session where all participants have exited due to an error.

5.4 Metatheory

We now turn our attention to showing that session typing allows runtime adaptation and discovery while precluding communication mismatches and deadlocks.

5.4.1 Runtime typing

To reason about the metatheory, we introduce typing rules for configurations (Fig. 15): the judgement $\Gamma; \Delta \vdash \mathcal{C}$ states that configuration \mathcal{C} is well-typed under term typing environment Γ and runtime typing environment Δ .

Rule T-PID types actor name restriction $(\nu a)\mathcal{C}$ by adding a PID into the term environment, and extending the runtime typing environment $a : S$; the linearity of the runtime typing environment therefore means that the system must contain precisely one actor with name a .

Session name restrictions $(\nu s)\mathcal{C}$ are typed by T-SESSION. We follow the formulation of Scalas & Yoshida [54] which types multiparty sessions using a parametric safety property φ ; we discuss safety properties in more depth in Section 5.4.2. Let Δ' be a runtime typing environment containing only mappings of the form $s[\mathbf{p}_i](\tilde{\mathbf{q}}_i):S_i$. Assuming Δ does not contain any mappings involving session s and Δ' satisfies φ , the rule states that \mathcal{C} is typable under typing environment Γ and runtime typing environment Δ, Δ' . It is sometimes convenient to annotate session ν -binders with their environment, e.g., $(\nu s : \Delta')\mathcal{C}$.

Labels

Labels $\gamma ::= s:\mathbf{p}\dagger\mathbf{q}::\ell(A) \mid s:\mathbf{p} \rightarrow \mathbf{q}::\ell \mid s:\mathbf{p}\dagger\mathbf{q}$
 Synchronisation labels $\rho ::= s:\mathbf{p}, \mathbf{q}::\ell \mid s:\mathbf{p} \rightarrow \mathbf{q}::\ell \mid s:\mathbf{p}\#\mathbf{q}$

Reduction on runtime typing environments

Local Reduction

$$\boxed{\Delta \xrightarrow{\gamma} \Delta'}$$

ET-CONN

$$\frac{\exists j \in I. \alpha_j = \mathbf{q}!!\ell_j(A_j) \quad j \in K \quad A_j = B_j \quad \text{ty}(\mathbf{q}) = \Sigma_{k \in K} (\mathbf{p}??\ell_k(B_k) \cdot T_k)}{s[\mathbf{p}]\langle \tilde{\mathbf{r}} \rangle : \Sigma_{i \in I} (\alpha_i \cdot S_i) \xrightarrow{s:\mathbf{p} \rightarrow \mathbf{q}::\ell_j} s[\mathbf{p}]\langle \tilde{\mathbf{r}}, \mathbf{q} \rangle : S_j, s[\mathbf{q}]\langle \mathbf{p} \rangle : T_j}$$

ET-ACT

$$\frac{\exists j \in I. \alpha_j = \mathbf{q}\dagger\ell_j(A_j) \quad \mathbf{q} \in \tilde{\mathbf{r}}}{s[\mathbf{p}]\langle \tilde{\mathbf{r}} \rangle : \Sigma_{i \in I} (\alpha_i \cdot S_i) \xrightarrow{s:\mathbf{p}\dagger\mathbf{q}::\ell_j(A_j)} s[\mathbf{p}]\langle \tilde{\mathbf{r}} \rangle : S_j}$$

ET-WAIT

$$\frac{}{s[\mathbf{p}]\langle \tilde{\mathbf{r}}, \mathbf{q} \rangle : \#\uparrow\mathbf{q} \cdot S \xrightarrow{s:\mathbf{p}\#\uparrow\mathbf{q}} s[\mathbf{p}]\langle \tilde{\mathbf{r}} \rangle : S}$$

ET-DISCONN

$$\frac{}{s[\mathbf{p}]\langle \mathbf{q} \rangle : \#\downarrow\mathbf{q} \xrightarrow{s:\mathbf{p}\#\downarrow\mathbf{q}} .}$$

ET-REC

$$\frac{\Delta, s[\mathbf{p}]\langle \tilde{\mathbf{q}} \rangle : S\{\mu X. S/X\} \xrightarrow{\gamma} \Delta'}{\Delta, s[\mathbf{p}]\langle \tilde{\mathbf{q}} \rangle : \mu X. S \xrightarrow{\gamma} \Delta'}$$

ET-CONG1

$$\frac{\Delta \xrightarrow{\gamma} \Delta'}{\Delta, s[\mathbf{p}]\langle \tilde{\mathbf{q}} \rangle : S \xrightarrow{\gamma} \Delta', s[\mathbf{p}]\langle \tilde{\mathbf{q}} \rangle : S}$$

ET-CONG2

$$\frac{\Delta \xrightarrow{\gamma} \Delta'}{\Delta, a : S \xrightarrow{\gamma} \Delta', a : S}$$

Synchronisation

$$\boxed{\Delta \xRightarrow{\rho} \Delta'}$$

ET-CONNSYNC

$$\frac{\Delta \xrightarrow{s:\mathbf{p} \rightarrow \mathbf{q}::\ell} \Delta'}{\Delta \xRightarrow{s:\mathbf{p} \rightarrow \mathbf{q}::\ell} \Delta'}$$

ET-COMM

$$\frac{\Delta_1 \xrightarrow{s:\mathbf{p}\dagger\mathbf{q}::\ell(A)} \Delta'_1 \quad \Delta_2 \xrightarrow{s:\mathbf{q}??\mathbf{p}::\ell(A)} \Delta'_2}{\Delta_1, \Delta_2 \xRightarrow{s:\mathbf{p}, \mathbf{q}::\ell} \Delta'_1, \Delta'_2}$$

ET-DISCONN

$$\frac{\Delta_1 \xrightarrow{s:\mathbf{p}\#\uparrow\mathbf{q}} \Delta'_1 \quad \Delta_2 \xrightarrow{s:\mathbf{q}\#\downarrow\mathbf{p}} \Delta'_2}{\Delta_1, \Delta_2 \xRightarrow{s:\mathbf{p}\#\mathbf{q}} \Delta'_1, \Delta'_2}$$

■ **Figure 16** Labelled transition system for runtime typing environments

Rule T-PAR types each subconfiguration of a parallel composition by splitting the linear runtime environment. Rule T-ZAP types a zipper thread $\frac{1}{2}s[\mathbf{p}]$, assuming the runtime environment contains an entry $s[\mathbf{p}]\langle \tilde{\mathbf{q}} \rangle : S$ for any session type S .

Finally, rules T-DISCONNECTEDACTOR and T-CONNECTEDACTOR type disconnected and connected actor configurations respectively. Given an actor with name a and static session type T , both rules require that the typing environment contains $a : \text{Pid}(T)$ and runtime environment contains $a : T$. Both rules require that the current session type is fully consumed by the currently-evaluating term and that the actor's behaviour should be typable under T . Rule T-DISCONNECTEDACTOR requires that the currently-evaluating term must be typable under either T or end , whereas to type a connection state of $s[\mathbf{p}]\langle \tilde{\mathbf{q}} \rangle$ and current session type S , T-CONNECTEDACTOR requires an entry $s[\mathbf{p}]\langle \tilde{\mathbf{q}} \rangle : S$ in the runtime environment.

5.4.2 Preservation

We now prove that reduction preserves typability and thus that actors only perform communication actions specified in their session types. Due to our use of explicit connection actions, classical MPST approaches are too limited for our purposes. Our approach, following that of Scalas & Yoshida [54], is to introduce a labelled transition system (LTS) on local types, and specify a generic safety property based around local type reduction. The property can then be refined; in our case, we will later specialise the property in order to prove progress.

Reduction on runtime typing environments. Figure 16 shows the LTS on runtime typing environments. There are two judgements: $\Delta \xrightarrow{\gamma} \Delta'$, which handles reduction of individual local types, and a *synchronisation* judgement $\Delta \xRightarrow{\rho} \Delta'$.

Rule ET-CONN handles the reduction of role \mathbf{p} , where the choice session type contains a connection action $\mathbf{q}!!\ell_j(A_j) \cdot S'_j$. If \mathbf{q} has a statically-defined session type $\Sigma_{k \in K}(\mathbf{p}??\ell_k(B_k) \cdot T_k)$ which can accept ℓ_j from from \mathbf{p} , and the payload types match, reduction advances \mathbf{p} 's session type, adds \mathbf{q} to \mathbf{p} 's connected role set, and introduces an entry for \mathbf{q} into the environment. The reduction emits a label $s:\mathbf{p} \rightarrow \mathbf{q}::\ell_j$.

Given a role \mathbf{p} connected to \mathbf{q} with a session choice containing a send or receive action $\mathbf{q}\uparrow\ell_j(A) \cdot S'_j$, rule ET-ACT will emit a label $s:\mathbf{p}\uparrow\mathbf{q}::\ell_j(A_j)$ and advance the session type of \mathbf{p} .

Rule ET-WAIT handles the reduction of $\#\uparrow\mathbf{q} \cdot S$ actions, $s[\mathbf{p}]\langle\tilde{r}, \mathbf{q}\rangle:\#\uparrow\mathbf{q} \cdot S$, where \mathbf{p} waits for \mathbf{q} to disconnect: the reduction emits label $\mathbf{p}:\mathbf{q}\#\uparrow$ and removes \mathbf{q} from \mathbf{p} 's connected roles. Similarly, rule ET-DISCONN handles disconnection, by emitting label $\mathbf{p}:\mathbf{q}\#\downarrow$ and removing the entry from the environment. ET-REC handles recursive types, and the ET-CONG rules handle reduction of sub-environments.

Rule ET-CONNSYNC states that connection is a synchronisation action, and rules ET-COMM and ET-DISCONN handle synchronisation between dual actions in sub-environments, emitting synchronisation labels $s:\mathbf{p}, \mathbf{q}::\ell$ and $s:\mathbf{p}\#\mathbf{q}$ respectively. We omit the congruence rules for synchronisation actions. We say that a runtime environment *reduces*, written $\Delta \Longrightarrow$, if there exists some Δ' such that $\Delta \Longrightarrow \Delta'$.

Safety. A *safety property* describes a set of invariants on typing environments which allow us to prove preservation. Since the type system is parametric in the given safety property, we can tweak the property to permit or rule out different typing environments satisfying particular behavioural properties; however, we need only prove type preservation once, using the weakest safety property. Our safety property is different to the safety property described by Scalas & Yoshida [54] in order to account for explicit connection actions.

► **Definition 5 (Safety Property).** φ is a safety property of runtime typing contexts Δ if:

1. $\varphi(\Delta, s[\mathbf{p}]\langle\tilde{r}\rangle:\Sigma_{i \in I}(\alpha_i \cdot S_i), s[\mathbf{q}]\langle\tilde{s}\rangle:\Sigma_{j \in J}(\mathbf{p}??\ell_j(B_j) \cdot T_j))$ implies that if $\mathbf{q}!!\ell_k(A_k) \in \{\alpha_i\}_{i \in I}$, then $k \in J$, $\mathbf{q} \in \tilde{r}$, $\mathbf{p} \in \tilde{s}$, and $A_k = B_k$.
2. $\varphi(\Delta, s[\mathbf{p}]\langle\tilde{r}\rangle:\Sigma_{i \in I}(\alpha_i \cdot S_i))$ implies that if $\alpha_i = \mathbf{q}!!\ell_j(A_j) \in \{\alpha_i\}_{i \in I}$, then $\mathbf{q} \notin \tilde{r}$, $s[\mathbf{q}]\langle\tilde{s}\rangle \notin \text{dom}(\Delta)$, and $\text{ty}(\mathbf{q}) = \Sigma_{k \in K}(\mathbf{p}??\ell_k(B_k) \cdot T_k)$ with $j \in K$ and $A_j = B_j$.
3. $\varphi(\Delta, s[\mathbf{p}]\langle\tilde{q}\rangle:\mu X.S)$ implies $\varphi(\Delta, s[\mathbf{p}]\langle\tilde{q}\rangle:S\{\mu X.S/X\})$
4. $\varphi(\Delta)$ and $\Delta \Longrightarrow \Delta'$ implies $\varphi(\Delta')$

A runtime typing environment is safe, written $\text{safe}(\Delta)$, if $\varphi(\Delta)$ for a safety property φ .

Clause (1) ensures that communication actions between participants are compatible: if \mathbf{p} is sending a message with label ℓ and payload type A to \mathbf{q} , and \mathbf{q} is receiving from \mathbf{p} , then the two roles must be connected, and \mathbf{q} must be able to receive ℓ with a matching payload.

Clause (2) states that if \mathbf{p} is connecting to a role \mathbf{q} with label ℓ , then \mathbf{q} should not already be involved in the session, and should be able to accept from \mathbf{p} on message label ℓ with a compatible payload type. The requirement that \mathbf{q} is not already involved in the session rules out the *correlation* errors described in Section 5.2.1. Clause (3) handles recursion, and clause (4) requires that safety is preserved under environment reduction.

Concretising the safety property. In order to deduce that a runtime typing environment Δ is safe, we define $\varphi(\Delta) = \{\Delta' \mid \Delta \Longrightarrow^* \Delta'\}$ and verify that φ is a safety property by ensuring that it satisfies all clauses in Definition 5.

Properties on protocols and programs. It is useful to distinguish active and inactive session types, depending on whether their associated role is currently involved in a session, and identify the initiator of a session.

► **Definition 6** (Active and Inactive Session Types). *A session type S is inactive, written $\text{inactive}(S)$, if $S = \text{end}$ or $S = \Sigma_{i \in I} (\mathbf{p}??\ell_i(A_i) . S_i)$. Otherwise, S is active, written $\text{active}(S)$.*

► **Definition 7** (Initiator, unique initiator). *Given a protocol P , a role $\mathbf{p} : S_{\mathbf{p}} \in P$ is an initiator if $S_{\mathbf{p}} = \Sigma_{i \in I} (\alpha_i . S_i)$, and each α_i is a connection action $\mathbf{q}!!\ell_i(A_i)$. Role \mathbf{p} is a unique initiator of P if $\text{inactive}(S_{\mathbf{q}})$ for all $\mathbf{q} \in P \setminus \{\mathbf{p} : S_{\mathbf{p}}\}$.*

A protocol is *well-formed* if it is safe and has a unique initiator.

► **Definition 8** (Well-formed protocol). *A protocol $P = \{\mathbf{p}_i : S_i\}_{i \in I}$ is well-formed if it has a unique initiator \mathbf{q} of type S and $\text{safe}(s[\mathbf{q}](\emptyset):S)$ for any s .*

By way of example, the online shopping protocol is well-formed: **Customer** is the protocol's unique initiator, and it is straightforward to verify that $\text{safe}(s[\mathbf{Customer}](\emptyset):\text{ty}(\mathbf{Customer}))$.

► **Definition 9** (Well-formed program). *A program (\vec{D}, \vec{P}, M) is well-formed if:*

1. *For each actor definition $D = \mathbf{actor} \ u \ \text{follows} \ S \ \{N\} \in \vec{D}$, there exists some role $\mathbf{p} \in \vec{P}$ such that $\text{ty}(\mathbf{p}) = S$, and $\{S\} \cdot | \ S \triangleright N:A \triangleleft \text{end}$*
2. *Each protocol $P \in \vec{P}$ is well-formed and has a distinct set of roles*
3. *The 'boot clause' M is typable under the empty typing environment and does not perform any communication actions: $\{\text{end}\} \cdot | \ \text{end} \triangleright M:A \triangleleft \text{end}$*

When discussing the metatheory, we only consider configurations defined with respect to a well-formed program. Specifically, we henceforth assume that each actor definition in the system follows a session type matched by a role in a given protocol, assume each role belongs to a single protocol, and assume that all protocols are well-formed.

Given a safe runtime environment, configuration reduction preserves typability; details can be found in the extended version. We write $\mathcal{R}^?$ for the reflexive closure of a relation \mathcal{R} .

► **Theorem 10** (Preservation (Configurations)). *Suppose $\Gamma; \Delta \vdash \mathcal{C}$ with $\text{safe}(\Delta)$ and where \mathcal{C} is defined wrt. a well-formed program. If $\mathcal{C} \longrightarrow \mathcal{C}'$, then there exists some Δ' such that $\Delta \Longrightarrow^? \Delta'$ and $\Gamma; \Delta' \vdash \mathcal{C}'$.*

Preservation shows that each actor conforms to its session type, and that communication never introduces unsoundness due to mismatching payload types.

5.4.3 Progress

We now show a progress property, which shows that given *protocols* which satisfy a progress property, EnsembleS *configurations* do not get stuck due to deadlocks.

A *final* runtime typing environment contains a single, disconnected role of type end , reflecting the intuition that all roles will eventually disconnect from a protocol initiator.

► **Definition 11** (Final environment). *An environment Δ is final, written $\text{end}(\Delta)$, $\Delta = \{s[\mathbf{p}](\emptyset):\text{end}\}$ for some s and \mathbf{p} .*

So far, we have considered *safe* protocols, which ensure the absence of communication mismatches. We say that an environment *satisfies progress* if each active role can eventually perform an action, each potential send is eventually matched by a receive, and non-reducing environments are final. Let $\text{roles}(\rho)$ denote the roles referenced in a synchronisation label (i.e., $\text{roles}(\rho) = \{\mathbf{p}, \mathbf{q}\}$ for $\rho \in \{s:\mathbf{p} \Rightarrow \mathbf{q}::\ell, s:\mathbf{p}, \mathbf{q}::\ell, s:\mathbf{p}\#\mathbf{q}\}$).

► **Definition 12** (Progress (Runtime typing environments)). A runtime typing environment Δ satisfies progress, written $\text{prog}(\Delta)$, if:

- (Role progress) for each $s[\mathbf{p}_i](\tilde{\mathbf{q}}_i):S_i \in \Delta$ s.t. $\text{active}(S_i)$, $\Delta \Longrightarrow^* \Delta' \xrightarrow{\rho}$ with $\mathbf{p} \in \text{roles}(\rho)$
- (Eventual comm.) if $\Delta \Longrightarrow^* \Delta' \xrightarrow{s:\mathbf{p}!\mathbf{q}::\ell(A)}$, then $\Delta' \xrightarrow{\vec{\rho}} \Delta'' \xrightarrow{s:\mathbf{q}?\mathbf{p}::\ell(A)}$, with $\mathbf{p} \notin \text{roles}(\vec{\rho})$
- (Correct termination) $\Delta \Longrightarrow^* \Delta' \not\Rightarrow$ implies $\text{end}(\Delta)$

The online shopping example satisfies progress, since all roles will always eventually be able to fire an action once connected, and since all roles disconnect, the non-reducing final environment will be of the form $s[\mathbf{Customer}](\emptyset):\text{end}$.

► **Definition 13** (Progress (Programs)). A well-formed program (\vec{D}, \vec{P}, M) satisfies progress if each $P \in \vec{P}$ has a unique initiator \mathbf{q} of type S and $\text{prog}(s[\mathbf{q}](\emptyset):S)$ for any s .

A configuration context \mathcal{G} is the context $\mathcal{G} ::= [] \mid (\nu s)\mathcal{G} \mid \mathcal{G} \parallel \mathcal{C}$. A session consists of a session name restriction and all connected actors and zipper threads. A well-typed configuration can be written as a sequence of sessions, followed by all disconnected actors.

► **Definition 14** (Session). A configuration is a session \mathcal{S} if it can be written:

$$(\nu s)(\langle a_1, M_1, s[\mathbf{p}_1](\tilde{\mathbf{q}}_1), \kappa_1 \rangle \parallel \cdots \parallel \langle a_m, M_m, s[\mathbf{p}_m](\tilde{\mathbf{q}}_m), \kappa_m \rangle \parallel \downarrow s[\mathbf{p}_{m+1}] \parallel \cdots \parallel \downarrow s[\mathbf{p}_n])$$

An actor is *terminated* if it has reduced to a value or has an unhandled exception, and has the behaviour **stop**. An unmatched discover occurs when no other actors match a given session type. An actor is *accepting* if it is ready to accept a connection.

► **Definition 15** (Terminated actor, unmatched discover, accepting actor).

- An actor $\langle a, M, \sigma, \kappa \rangle$ is terminated if $M = \mathbf{return} V$ or $M = E_P[\mathbf{raise}]$, and $\kappa = \mathbf{stop}$.
- An actor $\langle a, E[\mathbf{discover} S], \sigma, \kappa \rangle$ which is a subconfiguration of \mathcal{C} has an unmatched discover if no other non-terminated actor in \mathcal{C} has session type S .
- An actor $\langle a, M, \sigma, \kappa \rangle$ is accepting if $M = E[\mathbf{accept from} \mathbf{p} \{ \ell_j(x_j) \mapsto N_j \}_j]$ for some evaluation context E and role \mathbf{p} .

Unhandled exceptions will propagate through a session, progressively cancelling all roles. A *failed session* consists of only zipper threads.

► **Definition 16** (Failed session). We say that a session \mathcal{S} is a failed session, written $\text{failed}(\mathcal{S})$, if $\mathcal{S} \equiv (\nu s)(\downarrow s[\mathbf{p}_1] \parallel \cdots \parallel \downarrow s[\mathbf{p}_n])$.

The key *session progress* lemma establishes the reducibility of each session which does not contain an unmatched discover and is typable under a reducible runtime typing environment.

► **Lemma 17** (Session Progress). If $\cdot; \cdot \vdash \mathcal{C}$ where \mathcal{C} does not contain an unmatched discover, $\mathcal{C} \equiv \mathcal{G}[S]$ and $\mathcal{S} = (\nu s : \Delta)\mathcal{D}$ with $\text{prog}(\Delta)$, and \mathcal{S} is not a failed session, then $\mathcal{C} \longrightarrow$.

There are several steps to proving Lemma 17 (see the extended version). First, we introduce *exception-aware* reduction on runtime typing environments, which explicitly accounts for zipper threads at the type level, and show that exception-aware environments threads retain safety and progress. Second, we introduce *flattenings*, which show that runtime typing environments containing only unary output choices can type configurations blocked on communication actions, and that flattenings preserve environment reducibility. Finally, we show that configurations typable under flat, reducible typing environments can reduce.

We can now show our second main result: in the absence of unmatched discovers, a configuration can either reduce, or it consists only of accepting and terminating actors.

► **Theorem 18 (Progress).** *Suppose $;\cdot \vdash \mathcal{C}$ where \mathcal{C} is defined wrt. a well-formed program which satisfies progress, and $\text{prog}(\Delta)$ for each $(\nu s : \Delta)\mathcal{C}'$ in \mathcal{C} . If \mathcal{C} does not contain an unmatched discover, either $\exists \mathcal{D}$ such that $\mathcal{C} \longrightarrow \mathcal{D}$, or $\mathcal{C} \equiv \mathbf{0}$, or $\mathcal{C} \equiv (\nu b_1 \cdots \nu b_n)((b_1, N_1, \perp, \kappa_1) \parallel \cdots \parallel (b_n, N_n, \perp, \kappa_n))$ where each b_i is terminated or accepting.*

The proof eliminates all failed sessions by the structural congruence rules; shows that the presence of sessions implies reducibility (Lem. 17); and reasons about disconnected actors.

In addition to each actor conforming to its session type (Thm. 10), Theorem 18 guarantees that the system does not deadlock. It follows that session types ensure safe communication.

Theorem 18 assumes the absence of unmatched discovers. This is not a significant limitation in practice, however, as unmatched discovers can be mitigated with timeouts, where a timeout would trigger an exception.

6 Related Work

Behavioural typing for actors. Mostrous & Vasconcelos [46] present the first theoretical account of session types in an actor language; their work effectively overlays a channel-based session typing discipline on mailboxes using Erlang’s reference generation capabilities.

Neykova & Yoshida [49] use MPSTs to specify communication in an actor system, implemented in Python. Fowler [21] implements similar ideas in Erlang, with extensions to allow subsessions [17] and failure handling. Neykova & Yoshida [48] later improve the recovery mechanism of Erlang by using MPSTs to calculate a minimal set of affected roles. The above works check multiparty session typing dynamically. We are first to both formalise and implement static multiparty session type checking for an actor language.

Active objects (AOs) [15] are actor-like concurrent objects where results of asynchronous method invocations are returned through futures. Bagherzadeh & Rajan [4] study *order types* for an AO calculus, which characterise causality and statically rule out data races. In contrast to MPSTs, order types work bottom-up through type inference. Kamburjan *et al.* [39] apply an MPST-like system to Core ABS [38], a core AO calculus; they establish soundness via a translation to register automata rather than via an operational semantics.

de’Liguoro & Padovani [16] introduce *mailbox types*, a type system for first-class, unordered mailboxes. Their calculus generalises the actor model, since each process can be associated with more than one mailbox. Their type discipline allows multiple writers and a single reader for each mailbox, and ensures conformance, deadlock-freedom, and for many programs, junk-freedom. Our approach is based on MPSTs and is more process-centric.

Non-classical multiparty session types. MPSTs were introduced by Honda *et al.* [31]. *Classical* MPST theories are grounded in binary duality: safety follows as a consequence of *consistency* (pointwise binary duality of interactions between participants), and deadlock-freedom follows from projectability from a global type.

Unfortunately, classical MPSTs are restrictive: there are many protocols which are intuitively safe but not consistent. Scalas & Yoshida [54] introduced the first *non-classical* multiparty session calculus. Instead of ensuring safety using binary duality, they define an LTS on local types and *safety property* suitable for proving type preservation; since the type system is *parametric* in the safety property (inspired by Igarashi & Kobayashi [35] in the π -calculus), the property can be customised in order to guarantee different properties such as deadlock-freedom or liveness. Hu & Yoshida [32] formalise MPSTs with explicit connection actions via an LTS on types rather than providing a concrete language design or calculus; in our setting, a calculus is vital in order to account for the impact of adaptation constructs. A

key contribution of our work is the use of non-classical MPSTs to prove preservation and progress properties for a calculus incorporating MPSTs with explicit connection actions.

Adaptation. None of the above work considers adaptation. The literature on formal studies of adaptation is mainly based on process calculi, without programming language design or implementation. Bravetti *et al.* [9] develop a process calculus that allows parts of a process to be dynamically replaced with new definitions. Their later work [8] uses temporal logic rather than types to verify adaptive processes. Di Giusto and Pérez [19] define a session type system for the same process calculus, and prove that adaptation does not disrupt active sessions. Later, Di Giusto and Pérez [18] use an event-based approach so that adaptation can depend on the state of a session protocol. Anderson and Rathke [2] develop an MPST-like calculus and study dynamic software update providing guarantees of communication safety and liveness. Differently from our work, they do not consider runtime discovery of software components and do not provide an implementation.

Coppo *et al.* [13] consider self-adaptation, in which a system reconfigures itself rather than receiving external updates. They define an MPST calculus with self-adaptation and prove type safety. Castellani *et al.* [11] extend [13] to also guarantee properties of secure information flow, but neither of these works have been implemented. Dalla Preda *et al.* [51] develop the AIOJ system based on choreographic programming for runtime updates. Their work is implemented in the Jolie language [44], but they do not consider runtime discovery.

In this work we focus on correct communication in the absence of adversaries, and do not consider security. The literature on security and behavioural types is surveyed by Bartoletti *et al.* [5] and could provide a basis for future work on security properties.

7 Conclusion and Future Work

Modern computing increasingly requires software components to *adapt* to their environment, by *discovering*, *replacing*, and *communicating* with other components which may not be part of the system’s original design. Unfortunately, up until now, existing programming languages have lacked the ability to support adaptation both *safely* and *statically*. We therefore asked:

Can a programming language support static (compile-time) verification of safe runtime dynamic self-adaptation, i.e., discovery, replacement and communication?

We have answered this question in the affirmative by introducing EnsembleS, an actor-based language supporting adaptation, which uses multiparty session types to guarantee communication safety, using explicit connection actions to invite discovered actors into a session. We have demonstrated the safety of our system by proving type soundness theorems which state that each actor follows its session type, and that communication does not introduce deadlocks. Our formalism makes essential use of *non-classical* MPSTs.

Future work. Each actor only takes part in a single session. Unlike dynamically-checked implementations of session typing for actors [47, 21], this means that a message received by an actor in one session cannot trigger an interaction in another (e.g., the Warehouse example in [47]). A key focus for future work will be to allow actors to partake in multiple sessions.

EnsembleS discovery and replacement requires type equality. We expect we could relax this constraint to subtyping [53] or perhaps bisimilarity on local types to increase expressiveness.

In order to avoid session correlation errors, we require that each role includes at most a single top-level **accept** construct (c.f. [32]). It would be interesting to investigate the more general setting, which would likely require dependent types.

References

- 1 Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.
- 2 Gabrielle Anderson and Julian Rathke. Dynamic software update for message passing programs. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, volume 7705 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2012. doi:10.1007/978-3-642-35182-2_15.
- 3 Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009.
- 4 Mehdi Bagherzadeh and Hridesh Rajan. Order types: static reasoning about message races in asynchronous message passing concurrency. In *AGERE!@SPLASH*, pages 21–30. ACM, 2017.
- 5 Massimo Bartoletti, Iaria Castellani, Pierre-Malo Deniérou, Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Jovanka Pantovic, Jorge A. Pérez, Peter Thiemann, Bernardo Toninho, and Hugo Torres Vieira. Combining behavioural types with security analysis. *Journal of Logical and Algebraic Methods in Programming*, 84(6):763–780, 2015. doi:10.1016/j.jlamp.2015.09.003.
- 6 J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. *MOLE — Concepts of Mobile Agent System*, page 535–554. ACM Press/Addison-Wesley Publishing Co., USA, 1999.
- 7 G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May. The autonomic network architecture (ANA). *IEEE Journal on Selected Areas in Communications*, 28(1):4–14, 2010. doi:10.1109/JSAC.2010.100102.
- 8 Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro. Adaptable processes. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:13)2012.
- 9 Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro. Towards the verification of adaptable processes. In *Proceedings (Part I) of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 7609 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2012. doi:10.1007/978-3-642-34026-0_20.
- 10 Callum Cameron, Paul Harvey, and Joseph Sventek. A virtual machine for the Insense language. In *Proceedings of the International Conference on Mobile Wireless Middleware, Operating Systems and Applications (Mobilware)*, pages 1–10. IEEE, 2013. doi:10.1109/Mobilware.2013.17.
- 11 Iaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. Self-adaptation and secure information flow in multiparty communications. *Formal Aspects of Computing*, 28(4):669–696, 2016. doi:10.1007/s00165-016-0381-3.
- 12 Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang/OTP: Implement Robust, Fault-Tolerant Systems*. O’Reilly Media, Inc., 1st edition, 2016.
- 13 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Self-adaptive multiparty sessions. *Service Oriented Computing and Applications*, 9(3-4):249–268, 2015. doi:10.1007/s11761-014-0171-9.
- 14 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.
- 15 Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.
- 16 Ugo de’Liguoro and Luca Padovani. Mailbox types for unordered interactions. In *ECOOP*, volume 109 of *LIPICs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
- 17 Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012.

- 18 Cinzia Di Giusto and Jorge A. Pérez. Disciplined structured communications with disciplined runtime adaptation. *Science of Computer Programming*, 97:235–265, 2015. doi:10.1016/j.scico.2014.04.017.
- 19 Cinzia Di Giusto and Jorge A. Pérez. An event-based approach to runtime adaptation in communication-centric systems. In *Proceedings of the 11th and 12th International Workshops on Web Services, Formal Methods and Behavioural Types (WS-FM 2014, WS-FM/BEAT 2015)*, Lecture Notes in Computer Science, pages 67–85. Springer, 2015. doi:10.1007/978-3-319-33612-1_5.
- 20 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247. ACM, 1993.
- 21 Simon Fowler. An Erlang implementation of multiparty session actors. In *Proceedings of the 9th Interaction and Concurrency Experience (ICE)*, volume 223 of *Electronic Proceedings in Theoretical Computer Science*, pages 36–50. Open Publishing Association, 2016. doi:10.4204/EPTCS.223.3.
- 22 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019.
- 23 Colin S. Gordon. Lifting sequential effects to control operators. In *ECOOP*, volume 166 of *LIPICs*, pages 23:1–23:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 24 T. Gu, H. K. Pung, and D. Q. Zhang. Toward an OSGi-based infrastructure for context-aware applications. *IEEE Pervasive Computing*, 3(4):66–74, 2004. doi:10.1109/MPRV.2004.19.
- 25 Paul Harvey. *A linguistic approach to concurrent, distributed, and adaptive programming across heterogeneous platforms*. PhD thesis, School of Computing Science, University of Glasgow, 2015. URL: <http://theses.gla.ac.uk/6749/>.
- 26 Paul Harvey and Joseph Sventek. Adaptable actors: just what the world needs. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS)*, pages 22–28. ACM, 2017. URL: <http://doi.acm.org/10.1145/3144555.3144559>, doi:10.1145/3144555.3144559.
- 27 Richard Hayton, Michael Bursell, Douglas I. Donaldson, W. Harwood, and Andrew Herbert. Mobile Java objects. *Distributed Syst. Eng.*, 6(1):51, 1999. doi:10.1088/0967-1846/6/1/306.
- 28 Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL: <http://dl.acm.org/citation.cfm?id=1624775.1624804>.
- 29 Kohei Honda. Types for dyadic interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.
- 30 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.
- 31 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, volume 43, pages 273–284. ACM, 2008. URL: <http://doi.acm.org/10.1145/1328897.1328472>, doi:10.1145/1328897.1328472.
- 32 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science, pages 116–133. Springer, 2017. doi:10.1007/978-3-662-54494-5_7.
- 33 Danny Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Javier Del Cid, Sam Michiels, Christophe Huygens, and Wouter Joosen. LooCI: a loosely-coupled component infrastructure for networked embedded systems. In *Proceedings of the 7th International Conference on Advances*

- in *Mobile Computing and Multimedia (MoMM)*, pages 195–203. ACM, 2009. URL: <http://doi.acm.org/10.1145/1821748.1821787>, doi:<http://doi.acm.org/10.1145/1821748.1821787>.
- 34 Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 81–94. ACM, 2004. URL: <http://doi.acm.org/10.1145/1031495.1031506>, doi:[10.1145/1031495.1031506](http://doi.acm.org/10.1145/1031495.1031506).
 - 35 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.
 - 36 Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
 - 37 Antonio J Jara, Pedro Martinez-Julia, and Antonio Skarmeta. Light-weight multicast DNS and DNS-SD (lmDNS-SD): IPv6-based resource and service discovery for the web of things. In *2012 Sixth international conference on innovative mobile and internet services in ubiquitous computing*, pages 731–738. IEEE, 2012.
 - 38 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010.
 - 39 Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. Session-based compositional analysis for actor-based languages using futures. In *ICFEM*, volume 10009 of *Lecture Notes in Computer Science*, pages 296–312, 2016.
 - 40 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo: a session type toolchain for Java. *Science of Computer Programming*, 155:52–75, 2018. URL: <http://www.sciencedirect.com/science/article/pii/S0167642317302186>, doi:<https://doi.org/10.1016/j.scico.2017.10.006>.
 - 41 Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.
 - 42 Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.
 - 43 Paul V Mockapetris. RFC 1035: Domain names - implementation and specification, 1987.
 - 44 Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Composing services with JOLIE. In *ECOWS*, pages 13–22. IEEE Computer Society, 2007.
 - 45 Dimitris Mostrous and Vasco T. Vasconcelos. Affine sessions. *Log. Methods Comput. Sci.*, 14(4), 2018.
 - 46 Dimitris Mostrous and Vasco Thudichum Vasconcelos. Session typing for a featherweight Erlang. In *Proceedings of the 13th International Conference on Coordination Models and Languages (COORDINATION)*, volume 6721 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2011. doi:[10.1007/978-3-642-21464-6_7](https://doi.org/10.1007/978-3-642-21464-6_7).
 - 47 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *Proceedings of the 16th IFIP WG 6.1 Conference on Coordination Models and Languages (COORDINATION)*, volume 8459 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2014. doi:[10.1007/978-3-662-43376-8_9](https://doi.org/10.1007/978-3-662-43376-8_9).
 - 48 Rumyana Neykova and Nobuko Yoshida. Let it recover: multiparty protocol-induced recovery. In *Proceedings of the 26th International Conference on Compiler Construction (CC)*, pages 98–108. ACM, 2017. URL: <http://dl.acm.org/citation.cfm?id=3033031>.
 - 49 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. *Logical Methods in Computer Science*, 13(1:17):1–30, 2017. doi:[10.23638/LMCS-13\(1:17\)2017](https://doi.org/10.23638/LMCS-13(1:17)2017).
 - 50 Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. REX: A development platform and online learning approach for runtime emergent software systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 333–348. USENIX Association, 2016. URL: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/porter>.

- 51 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017. doi:10.23638/LMCS-13(2:1)2017.
- 52 Jan S Rellermeier, Gustavo Alonso, and Timothy Roscoe. R-OSGi: distributed applications through software modularization. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 1–20. Springer, 2007.
- 53 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:31, 2017. doi:10.4230/LIPIcs.ECOOP.2017.24.
- 54 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019.
- 55 Filippo Visintainer, Leandro D’Orazio, Marco Darin, and Luciano Altomare. Cooperative systems in motorway environment: The example of Trento test site in Italy. In Jan Fischer-Wolfarth and Gereon Meyer, editors, *Advanced Microsystems for Automotive Applications 2013*, pages 147–158, Heidelberg, 2013. Springer International Publishing.
- 56 Feng Xia, Laurence T. Yang, Lizhe Wang, and Alexey V. Vinel. Internet of things. *International Journal of Communication Systems*, 25(9):1101–1102, 2012. doi:10.1002/dac.2417.
- 57 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *Proceedings of the 8th International Symposium on Trustworthy Global Computing (TGC)*, volume 8358, pages 22–41. Springer, 2014. doi:10.1007/978-3-319-05119-2_3.