

Introduction to Session Types

Lecture 2: Multiparty Session Types

Simon Fowler

University of Glasgow

SPLV 2023

Session types: types for protocols

Core formal models for session types

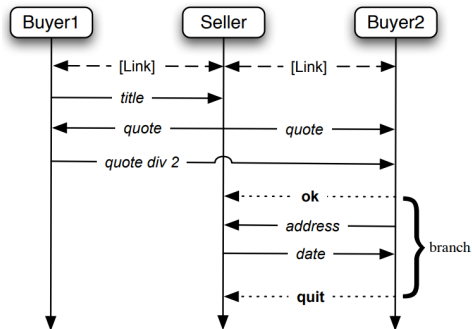
- Process calculus
- Lambda calculus (Hypersequent GV)

Implementation in Links

Multiparty session types: types with multiple participants

- Why do we need MPSTs?
- Global and local types
- Modern MPST calculi and their metatheory
- Implementations and extensions

What about protocols with multiple participants?



Classic (and somewhat maligned!) example of MPSTs

Two buyers co-ordinate to purchase an expensive item (traditionally a book):

- Buyer 1 requests title from Seller, seller responds
- Buyer 1 sends requested share to Buyer 2
- Buyer 2 either accepts, sending address to seller and receiving delivery date, or rejects

Can we implement this using binary session types?

Can we implement this using binary session types?

Buyer1(b2, s) \triangleq

```
let s = send "In your defence" s in  
let (price, s) = receive s in  
let b2 = send price/2 b2 in  
close b2; close s
```

Buyer2(b1, s) \triangleq

```
let (price, b1) = receive b1 in  
close b1;  
let s = select address s in  
let s = send "18 Lilybank Gardens" s in  
let (date, s) = receive s in  
close s
```

Seller(b1, b2) \triangleq

```
let (title, b1) = receive b1 in  
let b1 = send lookup(title) b1 in  
close b1;  
offer b2 {  
  case address  $\mapsto$   
    let b2 = send 2024-07-25 b2 in  
    close b2  
  case quit  $\mapsto$   
    close b2  
}
```

Yes, but...

Can we implement this using binary session types?

Buyer1(b2, s) \triangleq

let b2 = **send** 100 b2 **in**

let s = **send** "In your defence" s **in**

let (_, s) = **receive** s **in**

close b2; **close** s

Buyer2(b1, s) \triangleq

let s = **select** address s **in**

let (price, b1) = **receive** b1 **in**

close b1;

let s = **send** "18 Lilybank Gardens" s **in**

let (date, s) = **receive** s **in**

close s

Seller(b1, b2) \triangleq

let (title, b1) = **receive** b1 **in**

let b1 = **send** lookup(title) b1 **in**

close b1;

offer b2 {

case address \mapsto

let b2 = **send** 2024-07-25 b2 **in**

close b2

case quit \mapsto

close b2

}

Deadlock!

Key issue: no way of ordering
communication **on different**
channels

Multiparty session types

Multiparty Asynchronous Session Types

Kohei Honda

Queen Mary, University of London
kohai@dcs.qmul.ac.uk

Nobuko Yoshida

Imperial College London
yoshida@doc.ic.ac.uk

Marco Carbone

Queen Mary, University of London
carbonem@dcs.qmul.ac.uk

Abstract

Communication becoming one of the central elements in software development. As a potential typed foundation for structured communication-centred programming, session types have been studied over the last decade for a wide range of process calculi and programming languages, focussing on binary (two-party) sessions. This work extends the foregoing theories of binary session types to multiparty, asynchronous sessions. Which often arise in practical applications, such as distributed systems, multi-processor systems or mobile processes, the theory introduces a new notion of types in which interactions involving multiple peers are directly abstracted as a global scenario. Global types remain a friendly type syntax of binary session types while capturing complex causal chains of multiparty asynchronous interactions. A global type plays the role of a shared agreement among communication peers, and is used as a basis of efficiency type checking through its projection onto individual participants' functional types. The theory is applied to scenarios such as communication safety, progress and session fidelity are established for general *n*-party asynchronous interactions.

Categories and Subject Descriptors D.3.1 [Programming Lan-

vices (Carbone et al. 2006, 2007; WS-CDL; Sparkes 2006; Honda et al. 2007a). A basic observation underlying session types is that a communication-centred application often exhibits a highly structured sequence of interactions involving, for example, branching and recursion, which as a whole form a natural unit of conversation, or session. The structure of a conversation is abstracted as a type through an intuitive syntax, which is then used as a basis of validating programs through an associated type discipline.

As an example, the following session type describes a simple business protocol between Buyer and Seller from Buyer's viewpoint: Buyer sends the title of a book (*a string*), Seller sends a quote (*an integer*). If Buyer is satisfied by the quote, then sends his address (*a string*) and Seller sends back the delivery date (*a date*); otherwise it quits the conversation.

$$[\text{string}; ?\text{int}; \oplus [\text{ok}; ?\text{string}; ?\text{date}; \text{end}, \quad \text{quit}; \text{end}]] \quad (1)$$

Above t denotes an output of a value of type t , dually for $?t$; \oplus denotes a choice of the options; and end represents the termination of the conversation.

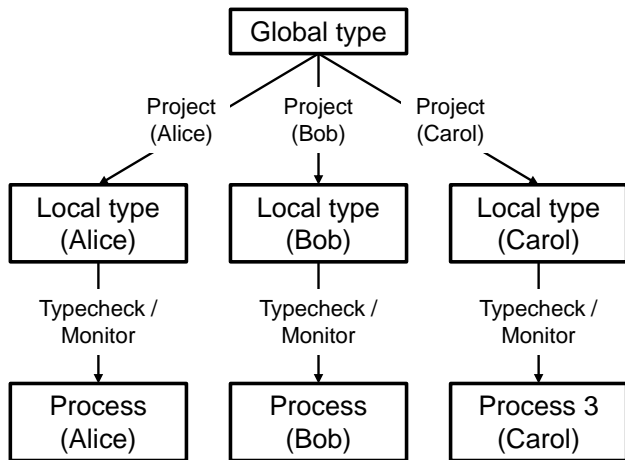
Such explicit representation of conversation structures helps us deal with one of the most common bugs in programming with com-

Idea: **global types** describe interactions between participants, projected to **local types** for typechecking

Key reference: K. Honda, N. Yoshida, M. Carbone. Multiparty Asynchronous Session Types. POPL'08.

Spawned a huge line of work; awarded
POPL most influential paper award in 2018

MPSTs: Overview



Two-Buyer Protocol: Types

Global type

```
Buyer1 → Seller : title(String) .  
Seller → Buyer1 : quote(Int) .  
Buyer1 → Buyer2 : share(Int) .  
Buyer2 → Seller : {  
  address(String) .  
  Seller → Buyer2 : date(Date) .  
  end,  
  quit(Unit) . end  
}
```

Two-Buyer Protocol: Types

Global type

```
Buyer1 → Seller : title(String) .  
Seller → Buyer1 : quote(Int) .  
Buyer1 → Buyer2 : share(Int) .  
Buyer2 → Seller : {  
  address(String) .  
  Seller → Buyer2 : date(Date) .  
  end,  
  quit(Unit) . end  
}
```

Buyer 1

```
Seller ⊕ title(String) .  
Seller & quote(Int) .  
Buyer2 ⊕ share(Int) . end
```

Buyer 2

```
Buyer1 & share(Int) . Seller ⊕ {  
  address(String) . Seller & date(Date) . end,  
  quit(Unit) . end }
```

Seller

```
Buyer1 & title(String) .  
Buyer1 ⊕ quote(Int) .  
Buyer2 & {  
  address(String) . Buyer2 ⊕ date(Date) . end,  
  quit(Unit) . end }
```

Global types

$G ::=$

- $\mathbf{p} \rightarrow \mathbf{q} : \{\ell_i(\mathbf{A}_i) . G_i\}_{i \in I}$
- $\mu \mathbf{t}.G$
- \mathbf{t}
- \mathbf{end}

communication

recursive type

recursive type variable

finished

Local types

$S, T ::=$

- $\mathbf{p} \& \{ \ell_i(A_i) . S_i \}_{i \in I}$
- $\mathbf{p} \oplus \{ \ell_i(A_i) . S_i \}_{i \in I}$
- $\mu \mathbf{t}. S$
- \mathbf{t}
- \mathbf{end}

offer choice to \mathbf{p}

send selection to \mathbf{p}

recursive type

recursive type variable

finished

Projection

$$\begin{aligned} q \rightarrow r : \{l_i(A_i) . G_i\}_{i \in I} \upharpoonright p &= \begin{cases} r \oplus \{l_i(A_i) . (G_i \upharpoonright p)\}_{i \in I} & \text{if } p = q \\ q \& \{l_i(A_i) . (G_i \upharpoonright p)\}_{i \in I} & \text{if } p = r \\ \prod_{i \in I} (G_i \upharpoonright p) & \text{otherwise} \end{cases} \\ (\mu t. G) \upharpoonright p &= \mu t. (G \upharpoonright p) \\ t \upharpoonright p &= t \\ \text{end} \upharpoonright p &= \text{end} \end{aligned}$$

Only interesting case is communication:

- Projecting at sender role gives local selection type
- Projecting at receiver role gives local branching type
- Projecting a selection at neither sender nor receiver requires **merge** \square

Plain Merge

Idea: projections at all branches must be the same.

$$S \sqcap S = S$$

(undefined otherwise)

Plain Merge

Idea: projections at all branches must be the same.

$$S \sqcap S = S$$

(undefined otherwise)

Full Merge

Idea: Number of branches in offer type don't need to match; merge overlapping ones

$$\begin{aligned} & p \&\{\ell_i(A_i) \cdot S_i\}_{i \in I} \sqcap p \&\{\ell_i(A_i) \cdot T_j\}_{j \in J} = \\ & p \&\{\ell_k(A_k) \cdot (S_k \sqcap T_k)\}_{k \in (I \cap J)} \\ & \cup \{\ell_i(A_i) \cdot S_i\}_{i \in (I/J)} \\ & \cup \{\ell_j(A_j) \cdot S_j\}_{j \in (J/I)} \end{aligned}$$

(remaining cases defined homomorphically)

Modern MPST Process Calculi



Less Is More: Multiparty Session Types Revisited

ALCESTE SCALAS, Imperial College London, UK

NOBUKO YOSHIDA, Imperial College London, UK

Multiparty Session Types (MPST) are a typing discipline ensuring that a message-passing process implements a given *multiparty session protocol*, without errors. In this paper, we propose a new, generalised MPST theory.

Our contribution is fourfold. (1) We demonstrate that a revision of the theoretical foundations of MPST is *necessary*: classic MPST have a limited *subject reduction* property, with inherent restrictions that are easily overlooked, and in previous work have led to flawed type safety proofs; our new theory removes such restrictions and fixes such flaws. (2) We contribute a new MPST theory that is *less* complicated, and yet *more* general, than the classic one: it does *not* require *global multiparty session types* nor *binary session type duality* – instead, it is grounded on general behavioural type-level properties, and proves type safety of many more protocols and processes. (3) We produce a detailed analysis of type-level properties, showing how, in our new theory, they allow to ensure decidability of type checking, and statically guarantee that processes enjoy, e.g., deadlock-freedom and liveness at run-time. (4) We show how our new theory can integrate type and model checking: type-level properties can be expressed in modal μ -calculus, and verified with well-established tools.

CCS Concepts: • **Theory of computation** → **Process calculi**; **Type structures**; *Verification by model checking*;

Additional Key Words and Phrases: session types, duality, deadlock-freedom, liveness

MPST **calculi** have traditionally been brittle, based on pointwise duality

Approach I will tell you about today based on “Less is More” by Scalas & Yoshida, POPL’19^a.

Key idea is based around **behavioural properties** of collections of local types

^aA. Scalas & N. Yoshida. Less is More: Multiparty Session Types Revisited. POPL 2019.

(Some) typing context properties

Safety

→ Values exchanged have matching types, selected labels are supported

(Some) typing context properties

Safety

→ Values exchanged have matching types, selected labels are supported

Deadlock-freedom

→ Either protocol is terminated or a communication can occur

(Some) typing context properties

Safety

→ Values exchanged have matching types, selected labels are supported

Deadlock-freedom

→ Either protocol is terminated or a communication can occur

Liveness

→ Each communication action can eventually be fired

(Some) typing context properties

Safety

→ Values exchanged have matching types, selected labels are supported

Deadlock-freedom

→ Either protocol is terminated or a communication can occur

Liveness

→ Each communication action can eventually be fired

Termination

→ Protocol always reaches final state after finite number of steps

Properties via projection

Determining whether a collection of local types satisfies a given property requires exploration of the state space

- Either by enumerating and checking state space directly, or using a model checker
- Can also check properties using a model checker

However, local types arising from projection have some properties **by construction**

- Safety, liveness, deadlock-freedom

Calculus

Variables x, y, z

Session names s

Names $c, d ::= x \mid s[p]$

Values $V, W ::= n \mid b \mid c$

Declarations $D ::= X(\vec{x}) = P$

Processes $P, Q ::=$

- 0
- $(\nu s)P$
- $P \parallel Q$
- $c[p] \oplus \ell(V) . P$
- $c[p] \& \{\ell_i(x_i) . P_i\}_{i \in I}$
- def** D **in** P
- $X\langle \vec{V} \rangle$

inactive process

session name restriction

parallel composition

send to p

receive from p

recursive process definition

recursive process call

Typing rules (1)

$$\frac{\text{un}(\Gamma)}{\Theta; \Gamma \vdash \mathbf{0}}$$

Inactive process typable under unrestricted environment

Typing rules (1)

$$\frac{\text{un}(\Gamma)}{\Theta; \Gamma \vdash \mathbf{0}}$$

Inactive process typable under unrestricted environment

$$\frac{\Theta; \Gamma_1 \vdash P \quad \Theta; \Gamma_2 \vdash Q}{\Theta; \Gamma_1 + \Gamma_2 \vdash P \parallel Q}$$

Parallel processes typable under disjoint environments (up to unrestricted types)

Typing rules (1)

$$\frac{\text{un}(\Gamma)}{\Theta; \Gamma \vdash \mathbf{0}}$$

Inactive process typable under unrestricted environment

$$\frac{\Theta; \Gamma_1 \vdash P \quad \Theta; \Gamma_2 \vdash Q}{\Theta; \Gamma_1 + \Gamma_2 \vdash P \parallel Q}$$

Parallel processes typable under disjoint environments (up to unrestricted types)

$$\frac{j \in I \quad \Gamma_1 \vdash c : \mathbf{p} \oplus \{\ell_i(A_i) . S_i\}_{i \in I} \quad \Gamma_2 \vdash V_j : A_j \quad \Theta; \Gamma_3, c : S_j \vdash P_j}{\Theta; \Gamma_1 + \Gamma_2 + \Gamma_3 \vdash c[\mathbf{p}] \oplus \ell_j(V_j) . P_j}$$

Label must be in session type, payload's type must match; bind x to S_j in continuation

Typing rules (2)

$$\frac{\Gamma_1 \vdash c : \mathbf{p} \&\{\ell_i(A_i) . S_i\}_{i \in I} \quad (\Theta; \Gamma_2, x_i : A_i, c : S_i \vdash P_i)_{i \in I}}{\Theta; \Gamma_1 + \Gamma_2 \vdash c[\mathbf{p}] \&\{\ell_i(x_i) . P_i\}_{i \in I}}$$

Receive branches must match session type;
bind payload type to x_i and S_i to c in continuation

Typing rules (2)

$$\frac{\Gamma_1 \vdash c : \mathbf{p} \&\{\ell_i(A_i) . S_i\}_{i \in I} \quad (\Theta; \Gamma_2, x_i : A_i, c : S_i \vdash P_i)_{i \in I}}{\Theta; \Gamma_1 + \Gamma_2 \vdash c[\mathbf{p}] \&\{\ell_i(x_i) . P_i\}_{i \in I}}$$

Receive branches must match session type; bind payload type to x_i and S_i to c in continuation

$$\frac{\Theta' = \Theta, X : \langle A_1, \dots, A_n \rangle \quad \Theta'; x_1 : A_1, \dots, x_n : A_n \vdash P \quad \Theta'; \Gamma \vdash Q}{\Theta; \Gamma \vdash \mathbf{def} X(x_1 : A_1, \dots, x_n : A_n) = P \mathbf{in} Q}$$

Bind parameters in type environment to type P , also bind definition in recursion environment

Typing rules (2)

$$\frac{\Gamma_1 \vdash c : \mathbf{p} \& \{ \ell_i(A_i) . S_i \}_{i \in I} \quad (\Theta; \Gamma_2, x_i : A_i, c : S_i \vdash P_i)_{i \in I}}{\Theta; \Gamma_1 + \Gamma_2 \vdash c[\mathbf{p}] \& \{ \ell_i(x_i) . P_i \}_{i \in I}}$$

Receive branches must match session type; bind payload type to x_i and S_i to c in continuation

$$\frac{\begin{array}{l} \Theta' = \Theta, X : \langle A_1, \dots, A_n \rangle \\ \Theta'; x_1 : A_1, \dots, x_n : A_n \vdash P \quad \Theta'; \Gamma \vdash Q \end{array}}{\Theta; \Gamma \vdash \mathbf{def} X(x_1 : A_1, \dots, x_n : A_n) = P \mathbf{in} Q}$$

Bind parameters in type environment to type P , also bind definition in recursion environment

$$\frac{\begin{array}{l} X : \langle A_1, \dots, A_n \rangle \in \Theta \\ (\Gamma_i \vdash c_i : A_i)_{i \in 1..n} \end{array}}{\Theta; \Gamma_i \vdash X \langle c_1, \dots, c_n \rangle}$$

Check that argument types match definition in recursion environment

Typing rules: name restriction

$$\frac{\begin{array}{l} \Gamma' = \{s[\mathbf{p}_i] : S_{\mathbf{p}_i}\}_{i \in I} \quad s \notin \Gamma \quad \varphi(\Gamma') \\ \varphi \text{ is a safety property} \quad \Theta; \Gamma, \Gamma' \vdash P \end{array}}{\Theta; \Gamma \vdash (\nu s)P}$$

Key idea: Collection of local types that satisfy some property φ

For type preservation, φ must be at least a safety property

→ The exact property can be tailored depending on the desired behaviour

Write $\text{safe}(\Gamma)$ if $\varphi(\Gamma)$ for some safety property

Reduction rules

(E-Comm)

$$s[\mathbf{p}][\mathbf{q}] \oplus \ell_j(V) . P \parallel s[\mathbf{q}][\mathbf{p}] \& \{ \ell_i(x_i) . Q_i \}_{i \in I} \longrightarrow P \parallel Q_j \{V/x_j\} \quad (\text{if } j \in I)$$

Reduction rules

(E-Comm)

$$s[\mathbf{p}][\mathbf{q}] \oplus \ell_j(V) . P \parallel s[\mathbf{q}][\mathbf{p}] \& \{ \ell_i(x_i) . Q_i \}_{i \in I} \longrightarrow P \parallel Q_j \{ V / x_j \} \quad (\text{if } j \in I)$$

(E-Call)

$$\mathbf{def} X(\vec{x}) = P \mathbf{in} (X\langle \vec{V} \rangle \parallel Q) \longrightarrow \mathbf{def} X(\vec{x}) = P \mathbf{in} (P\{ \vec{V} / \vec{x} \} \parallel Q)$$

Reduction rules

(E-Comm)

$$s[\mathbf{p}][\mathbf{q}] \oplus \ell_j(V) . P \parallel s[\mathbf{q}][\mathbf{p}] \& \{ \ell_i(x_i) . Q_i \}_{i \in I} \longrightarrow P \parallel Q_j \{ V / x_j \} \quad (\text{if } j \in I)$$

(E-Call)

$$\mathbf{def} X(\vec{x}) = P \mathbf{in} (X\langle \vec{V} \rangle \parallel Q) \longrightarrow \mathbf{def} X(\vec{x}) = P \mathbf{in} (P\{ \vec{V} / \vec{x} \} \parallel Q)$$

$$\frac{P \longrightarrow P'}{P \parallel Q \longrightarrow P' \parallel Q}$$

$$\frac{P \longrightarrow Q}{(\nu s)P \longrightarrow (\nu s)Q}$$

$$\frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

Metatheory: Preservation

Theorem (Preservation)

If $\Theta; \Gamma \vdash P$ where $\text{safe}(\Gamma)$, and $P \longrightarrow Q$, then there exists some Γ' such that $\Gamma \longrightarrow \Gamma'$ and $\Theta; \Gamma' \vdash Q$.

Metatheory: Using type-level properties to check process behaviour (1)

Preservation is an important, but basic property

We want to show that if a **protocol** satisfies a property (e.g., deadlock-freedom), then so does the process

Session fidelity theorem shows that (under some conditions), any reduction in the environment can be reflected by the process

Metatheory: Using type-level properties to check process behaviour (1)

Preservation is an important, but basic property

We want to show that if a **protocol** satisfies a property (e.g., deadlock-freedom), then so does the process

Session fidelity theorem shows that (under some conditions), any reduction in the environment can be reflected by the process

Theorem (Session Fidelity (Informally))

If:

- $\Theta; \Gamma \vdash P$ with $\text{safe}(\Gamma)$
- Each basic subprocess has guarded recursive calls, and acts only on a single role
- $\Gamma \longrightarrow \Gamma'$

then there exists some Q such that $P \longrightarrow Q$.

Metatheory: Using type-level properties to check process behaviour (2)

Deadlock-freedom on **processes**: $P \longrightarrow^* Q \not\longrightarrow$ implies that $Q \equiv 0$.

Deadlock-freedom on **typing contexts**: $\Gamma \longrightarrow^* \Gamma' \not\longrightarrow$ implies $\text{un}(\Gamma')$

Therefore:

- only way a process cannot reduce is if the environment cannot reduce
- since $\text{safe}(\Gamma')$, the only way the Γ' cannot reduce is if $\text{un}(\Gamma')$
- since recursion guarded, only typable basic subprocesses under Γ' are 0

A note on asynchrony

We have so far considered a **synchronous** reduction semantics

In the synchronous setting, the typing context properties are decidable

→ The state-space is finite

It is also possible and often desirable to consider an **asynchronous** semantics

However, many context properties (e.g., safety) are **undecidable** in the asynchronous setting

→ Global types can help here

Implementations

The Scribble Protocol Description Language

```
global protocol TwoBuyer(role B1,  
    role B2, role S) {  
  title(String) from B1 to S;  
  price(Int) from S to B1;  
  share(Int) from B1 to B2;  
  choice at B2 {  
    address(String) from B2 to S;  
    date(Date) from S to B2;  
  } or {  
    quit() from B2 to S;  
  }  
}
```

Scribble: language-agnostic **protocol description language**

Based on MPST theory: validation, projection, monitor generation

More liberal syntax than global types (e.g., joins after choice blocks)

Validation based on bounded model checking



Global protocol

```
module twobuyer.TwoBuyer;

type <java> "java.lang.Integer" from "rt.jar" as int;
type <java> "java.lang.String" from "rt.jar" as String;
type <java> "test.twobuyer.Address" from
"test/twobuyer/Address.java" as Address;
type <java> "test.twobuyer.Date" from "test/twobuyer/Date.java"
as Date;

global protocol TwoBuyer(role A, role B, role S)
{
  title(int) from A to S;
  //quote(int) from S to A, B; // EFSM building for multicast
  not currently supported
  quote(x: int) from S to A;
  quote(y: int) from S to B;
  quoteByTwo(z:int) from A to B;
  choice at B
  {
    ok(int) from B to S; @*z > y - 2*
    empty(int) from S to B;
  }
  or
  {
    quit() from B to S; @*z ≤ y - 2*
  }
}
```

examples/annot/TwoBuyer.nuscr

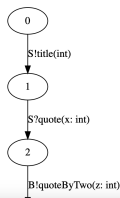
Analyse

Local types

- A@TwoBuyer[Project][FSM]
- B@TwoBuyer[Project][FSM]
- S@TwoBuyer[Project][FSM]

Projected on to A@TwoBuyer :

```
title(int) to S;
quote(x: int) from S;
quoteByTwo(z: int) to B;
end
```



nuScr: more stripped down version of Scribble, written in OCaml

Goal is to keep minimal and close to theory

Try it out online: <https://nuscr.dev>

The API Generation Approach

Hybrid Session Verification through Endpoint API Generation

Raymond Hu and Nobuko Yoshida

Imperial College London

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

Hybrid Session Verification through Endpoint API Generation. R. Hu & N. Yoshida. FASE'16.

Methodology for implementing MP-STs in mainstream languages

Compile local types to **endpoint FSMs**

“dot-driven” approach to chain together API calls

“Hybrid” approach: **runtime** checks to ensure no duplicate uses

SmtplibClient.java

src/test/scrub demo.smtplib SmtplibClient run(): void

```
68         s12 = s20.receive(SMTP._250);
69         break;
70     }
71     case _501:
72     {
73         s20.receive(SMTP._501).send(SMTP.S, new Quit());
74         System.exit(0);
75     }
76 }
77 s12.send(SMTP.S, new Rcpt("my.friend@imperial.ac.uk"))
78 // .async(SMTP._250)
79 .send(SMTP.S, new Data())
80 .async(SMTP._354)
81 .send(SMTP.S, new Subject("test"))
82 .send(SMTP.S, new DataLine("body"))
83 .send(SMTP.S, new EndOfData())
84 .receive(SMTP._250, new Buff<>())
85 .send(SMTP.S, new Quit());
86 }
```

The method send(S, Data) is undefined for the type SMTP_C. 2 quick fixes available:

- Create method 'send(S, Data)' in type 'SMTP_C 16'
- Add cast to method receiver

Problems Javadoc Declaration Search JUnit Console Console

1 error, 320 warnings, 0 others (Filter matched 101 of 321 items)

Description	Resource
Errors (1 item)	
The method send(S, Data) is undefined for the type SMTP_C_16	SmtplibClient.java

Implementation via typestate

Science of Computer Programming 155 (2018) 52–75



Typechecking protocols with Mungo and StMungo: A session type toolbox for Java

Dimitrios Kouzapas^a, Ornela Dardha^b, Roly Perera^{b,c}, Simon J. Gay^b

^a Department of Computer Science, University of Cyprus, Cyprus

^b School of Computing Science, University of Glasgow, UK

^c School of Informatics, University of Edinburgh, UK

ARTICLE INFO

Article history:

Received 4 January 2017

Received in revised form 13 October 2017

Accepted 16 October 2017

Available online 5 December 2017

Keywords:

Session types

Object-oriented programming

Typestate

Type inference

ABSTRACT

Static typechecking is an important feature of many standard programming languages. However, static typing focuses on data rather than communication, and therefore does not help programmers correctly implement communication protocols in distributed systems. The theory of session types provides a basis for tackling this problem; we use it to develop two tools that support static typechecking of communication protocols in Java. The first tool, Mungo, extends Java with typestate definitions, which allow classes to be associated with state machines defining permitted sequences of method calls; for example, communication methods. The second tool, StMungo, takes a session type describing a communication protocol, and generates a typestate specification of the permitted sequences of messages in the protocol. Protocol implementations can be validated by Mungo against their typestate definitions and then compiled with a standard Java compiler. The result is a toolbox for static typechecking of communication protocols in Java. We formalise and prove soundness of the typestate inference system used by Mungo, and show that our toolbox can be used to typecheck a client for the standard Simple Mail Transfer Protocol (SMTP).

Typestate: type discipline for OOP which governs object methods that can be invoked

Mungo: Typestate checker for Java

→ External specification of typestate FSMs, checked via tool

StMungo: Translation from Scribble into typestates

D. Kouzapas, O. Dardha, R. Perera, and S. J. Gay.

Typechecking Protocols with Mungo and StMungo.

PPDP'16 & SCP.

Implementations in Rust (1)

Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types

Nicolas Lagaillardie ✉

Department of Computing, Imperial College London, UK

Rumyana Neykova ✉

Department of Computer Science, Brunel University London, UK

Nobuko Yoshida ✉

Department of Computing, Imperial College London, UK

Abstract

Communicating systems comprise diverse software components across networks. To ensure their robustness, modern programming languages such as Rust provide both strongly typed channels, whose usage is guaranteed to be *affine* (*at most once*), and *cancellation* operations over *binary* channels. For coordinating components to correctly communicate and synchronise with each other, we use the structuring mechanism from *multiparty session types*, extending it with affine communication channels and implicit/explicit cancellation mechanisms. This new typing discipline, *affine multiparty session types* (AMPST), ensures *cancellation termination* of multiple, independently running components and guarantees that communication will not get stuck due to error or abrupt termination. Guided by AMPST, we implemented an automated generation tool (MultiCrusty) of Rust APIs associated with cancellation termination algorithms, by which the Rust compiler auto-detects unsafe programs. Our evaluation shows that MultiCrusty provides an efficient mechanism for communication, synchronisation and propagation of the notifications of cancellation for arbitrary processes. We have implemented several usecases, including popular application protocols (OAuth, SMTP), and protocols with exception handling patterns (circuit breaker, distributed logging).

2012 ACM Subject Classification Software and its engineering → Software usability; Software and its engineering → Concurrent programming languages; Theory of computation → Process calculi

Keywords and phrases Rust language, affine multiparty session types, failures, cancellation

Encode MPSTs using a mesh of binary channels

→ Built on Sesh (Kokke, 2019)

Tool: MultiCrusty

Adapts **affine sessions** approach to support cascading failure

N. Lagaillardie, R. Neykova & N. Yoshida. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. ECOOP'22.

Implementations in Rust (2)

Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types

Zak Cutner

Imperial College London
London, UK

Nobuko Yoshida

Imperial College London
London, UK

Martin Vassor

Imperial College London
London, UK

Abstract

Rust is a modern systems language focused on performance and reliability. Complementing Rust's promise to provide "fearless concurrency", developers frequently exploit asynchronous message passing. Unfortunately, sending and receiving messages in an arbitrary order to maximise computation overlap (a popular optimisation in message-passing applications) opens up a Pandora's box of subtle concurrency bugs.

To guarantee deadlock-freedom by construction, we present RUMPSTEAK: a new Rust framework based on *multiparty session types*. Previous session type implementations in Rust are either built upon synchronous and blocking communication and/or are limited to two-party interactions. Crucially, none support the arbitrary reordering of messages for efficiency.

RUMPSTEAK instead targets asynchronous `async/await`

CCS Concepts: • Software and its engineering → Development frameworks and environments; Source code generation; • Computer systems organization → Reliability.

Keywords: Rust, Asynchronous Message Passing, Message Reordering, Computation-Communication Overlap, Multiparty Session Types

1 Introduction

Rust is a statically-typed language designed for systems software development. It is rapidly growing in popularity and has been voted "most loved language" over five years of surveys by Stack Overflow [19]. Rust aims to offer the safety of a high-level language without compromising on the performance enjoyed by low-level languages. *Message passing over*

Encode MPSTs using Rust's type system, using `async/await`

Key feature: safe message reordering using **asynchronous subtyping**

- Allows message reordering where there is no causality between communications
- Async. subtyping is **undecidable** in general
- Sound & efficient approximation

Extensions

Explicit Connection Actions

```
aux global protocol Pay
  (role C, role A, role S) {
  choice at C {
    // C connects to S, sends pay
    info
    pay(Str) connect C to S;
    // S returns a payment
    reference
    confirm(Int) from S to C;
    // C forwards the payref to A
    accpt(Int) from C to A;
  } or {
    reject() from C to A;
  }
}
```

Often does not make sense for all participants to be instantiated at start of session

→ e.g., when selecting a particular chat room to join

Idea: Explicitly invite a participant mid-way through a session

Protocol safety checking through 1-bounded model checking

MPSTs for Adaptable Actors

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.

Actor languages: explicitly addressable processes

Runtime adaptation:

- **Discover** other actors
- **Communicate** safely
- **Replace** without breaking communication guarantees

First statically-checked session type system for actors, using explicit connection actions to support safe runtime adaptation

P. Harvey, S.F., O. Dardha, S. J. Gay. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. ECOOP 2021.

Refinement types



Statically Verified Refinements for Multiparty Protocols

FANGYI ZHOU, Imperial College London, United Kingdom

FRANCISCO FERREIRA, Imperial College London, United Kingdom

RAYMOND HU, University of Hertfordshire, United Kingdom

RUMYANA NEYKOVA, Brunel University London, United Kingdom

NOBUKO YOSHIDA, Imperial College London, United Kingdom

With distributed computing becoming ubiquitous in the modern era, safe distributed programming is an open challenge. To address this, multiparty session types (MPST) provide a typing discipline for message-passing concurrency, guaranteeing communication safety properties such as deadlock freedom.

While originally MPST focus on the communication aspects, and employ a simple typing system for communication payloads, communication protocols in the real world usually contain *constraints* on the payload. We introduce *refined multiparty session types* (RMPST), an extension of MPST, that express data dependent protocols via *refinement types* on the data types.

We provide an implementation of RMPST, in a toolchain called SESSION*, using SCRIBBLE, a toolchain for multiparty protocols, and targeting F*, a verification-oriented functional programming language. Users can describe a protocol in SCRIBBLE and implement the endpoints in F* using *refinement-typed APIs* generated from the protocol. The F* compiler can then statically verify the refinements. Moreover, we use a novel approach of callback-styled API generation, providing *static* linearity guarantees with the inversion of control. We evaluate our approach with real world examples and show that it has little overhead compared to a naive implementation, while guaranteeing safety properties from the underlying theory.

CCS Concepts: • Theory of computation → Automated reasoning; Distributed computing models; • Software and its engineering → Source code generation.

```
G = A → B : Count(count : int{count ≥ 0}).
  μt(curr : int{curr ≥ 0 ∧ curr ≤ count})(curr := 0).
  B → C {
    Hello(it : int{curr < count ∧ it = count}).t(curr := curr + 1)
    Finish(it : int{curr = count ∧ it = count}).end
  }
```

F. Zhou, F. Ferreira, R. Hu, R. Neykova, and N. Yoshida.

Statically Verified Refinements for Multiparty Protocols. OOPSLA'20.

Refinement types: Boolean predicates on exchanged values

Extend global types with refinements, implemented in F*

Inversion of control: user-specified callbacks rather than direct-style programming

Explicit fault-tolerance

Generalised Multiparty Session Types with Crash-Stop Failures

Adam D. Barwell ✉
Imperial College London, UK

Alceste Scalas ✉
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Nobuko Yoshida ✉
Imperial College London, UK

Fangyi Zhou ✉
Imperial College London, UK

Abstract

Session types enable the specification and verification of communicating systems. However, their theory often assumes that processes never fail. To address this limitation, we present a generalised multiparty session type (MPST) theory with *crash-stop failures*, where processes can crash arbitrarily.

Our new theory validates more protocols and processes w.r.t. previous work. We apply minimal syntactic changes to standard session π -calculus and types: we model crashes and their handling semantically, with a generalised MPST typing system parametric on a behavioural safety property. We cover the spectrum between fully reliable and fully unreliable sessions, via *optional reliability assumptions*, and prove type safety and protocol conformance in the presence of crash-stop failures.

Introducing crash-stop failures has non-trivial consequences: writing correct processes that handle all crash scenarios can be difficult. Yet, our generalised MPST theory allows us to tame this complexity, via model checking, to validate whether a multiparty session satisfies desired behavioural properties, e.g. deadlock-freedom or liveness, even in presence of crashes. We implement our approach using the mCRL2 model checker, and evaluate it with examples extended from the literature.

2023 ACM Subject Classification: Theory of computation → Distributed computing models; Theory

MPSTs typically assume lack of failure

Idea: define a set of reliable roles; assume remainder can fail nondeterministically

User guided to handle each failure point

A. Barwell, A. Scalas, N. Yoshida, F. Zhou. Generalised Multiparty Session Types with Crash-Stop Failures. CONCUR 2022.



Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom

JULES JACOBS, Radboud University Nijmegen, The Netherlands

STEPHANIE BALZER, Carnegie Mellon University, USA

ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands

Session types have recently been integrated with functional languages, bringing message-passing concurrency to functional programming. Channel endpoints then become first-class and can be stored in data structures, captured in closures, and sent along channels. Representatives of the GV (Wadler's "Good Variation") session type family are of particular appeal because they not only assert session fidelity but also deadlock freedom, inspired by a Curry-Howard correspondence to linear logic. A restriction of current versions of GV, however, is the focus on binary sessions, limiting concurrent interactions within a session to two participants. This paper introduces Multiparty GV (MPGV), a functional language with multiparty session types, allowing concurrent interactions among several participants. MPGV upholds the strong guarantees of its ancestor GV, including deadlock freedom, despite session interleaving and delegation. MPGV has a novel redirecting construct for modular programming with first-class endpoints, thanks to which we give a type-preserving translation from binary session types to MPGV to show that MPGV is strictly more general than binary GV. All results in this paper have been mechanized using the Coq proof assistant.

CCS Concepts • **Software and its engineering** → **Concurrent programming languages**.

Additional Key Words and Phrases: Session types, message-passing concurrency, deadlock freedom

ACM Reference Format:

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom. *Proc. ACM Program. Lang.* 6, ICFP, Article 107 (August 2022), 30 pages. <https://doi.org/10.1145/3547638>

J. Jacobs, S. Balzer, and R. Krebbers. Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom. ICFP 2022.

Multiparty session-typed λ -calculus
(like the one in the last lecture)

A multi-fork construct allows for
deadlock-freedom

Everything mechanised in Coq!

→ Uses **connectivity graph** approach

Wrapping up

Conclusion

Today

- Introduction to multiparty session types
- Global & local types, and projection
- Generalised MPST theory based on “Less is More”
- Practical implementations and implementations

There is **much** more to session types than I can cover in 3 hours!

- Dependent session types, subtyping, logical correspondence, sharing...

Happy to chat / answer questions through the week!

Conclusion

Today

- Introduction to multiparty session types
- Global & local types, and projection
- Generalised MPST theory based on “Less is More”
- Practical implementations and implementations

There is **much** more to session types than I can cover in 3 hours!

- Dependent session types, subtyping, logical correspondence, sharing...

Happy to chat / answer questions through the week!

Enjoy the rest of SPLV!