Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Type-safe Web Programming Using Routed Multiparty Session Types in TypeScript

Author:
Anson Miu

Supervisor:
Prof. Nobuko Yoshida

Second Marker:
Dr. Iain Phillips

June 18, 2020

Abstract

Modern web programming involves coordinating interactions between web browser clients and a web server. Typically, the interactions are informally described, making it difficult to verify communication correctness in web-based distributed systems.

Multiparty Session Types (MPST) are a typing discipline for concurrent processes that communicate via message passing. MPST theory can ensure communication safety properties and protocol conformance. Existing work in session-typed web development over WebSocket transport is incompatible with modern web programming practices and is limited to supporting communication protocols that implement the server-centric network topology.

We address limitations in the current state of the art by: **(1)** implementing SESSIONTS, a code generation toolchain for session-typed web development over WebSocket transport using TypeScript; and **(2)** presenting ROUTEDSESSIONS, a new multiparty session type theory that supports routed communications.

SESSIONTS provides developers with TypeScript APIs generated from a communication protocol specification based on multiparty session type theory. Our work is compatible with modern web programming industrial practices. The generated APIs build upon TypeScript concurrency practices, complement the event-driven style of programming in full-stack web development, and are compatible with the Node.js runtime for server-side endpoints and the React.js framework for browser-side endpoints. We evaluate the expressiveness of SESSIONTS for modern web programming using case studies of protocols found in web services, and analyse the performance overhead through running benchmarks against a baseline implementation.

ROUTEDSESSIONS can express interactions to be routed via an intermediate participant. Using ROUTEDSESSIONS, we propose an approach for supporting peer-to-peer communication between browser-side endpoints through routing communication via the server in a way that avoids excessive serialisation and preserves communication safety. We evaluate the correctness of ROUTEDSESSIONS by proving communication safety properties, such as deadlock freedom.

Acknowledgements

I would like to thank Prof. Nobuko Yoshida, Fangyi Zhou and Dr. Francisco Ferreira for their continuous support, encouragement and motivation during the project. I extend my gratitude to the members of the Mobility Reading Group. Their regular meetings have been a source of inspiration for me, and helped me spark new ideas to incorporate into the project. I would also like to thank Prof. Peter Pietzuch for supporting me as my personal tutor throughout the degree.

I would like to thank all the friends I have made throughout my time at Imperial. In particular, to the friends under the namespace labelled *Reply 404* at the time of writing: I am forever grateful for every bit of shared memory over the past four years.

Finally, I would like to thank my family for their unconditional love and support throughout my life, and encouraging me to pursue a degree abroad.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Modern interactive web applications aim to provide a highly responsive user experience by minimising the communication latency between clients and servers. Whilst the *HTTP* request-response model is sufficient for retrieving static assets, applying the same stateless communication approach for interactive use cases introduces undesirable performance overhead from having to frequently set up new connections for client-server interactions. Developers have since adopted other communication transport abstractions over HTTP connections such as the WebSockets protocol [17] to enjoy low-latency full-duplex client-server communication in their applications over a single persistent connection. Enabling more complex communication patterns caters for more interactive use cases, but introduces additional concerns to the developer with respect to implementation correctness.

Communication Safety in Interactive Web Applications Consider a *flight booking service* between a `Traveller` and a `FlightServer`. The `Traveller` queries the `FlightServer` about prices to a particular destination. If there is still availability, the `FlightServer` will *reserve a seat* and respond with the price, to which the `Traveller` confirms or rejects the purchase; the latter choice will notify the `FlightServer` to *release the seat*. Otherwise, the `FlightServer` notifies the `Traveller` that all flights are full. The `Traveller` can proceed to query for another destination.

We can implement the flight booking service as a web application. `FlightServer` can be a web server running on the *Node.js* JavaScript runtime [43]. `Traveller` can run on the web browser as a *single-page application* (SPA) written in a popular framework like *React.js* [10]. SPAs feature a single HTML page and dynamically renders content via JavaScript in the browser. The `Traveller` connects to the `FlightServer` using a WebSocket connection and they enjoy bi-directional communication to execute the flight booking operations outlined above. We visualise the interactions in Figure 1.1

Whilst WebSockets make this web-based implementation possible, it introduces the developer to a new family of communication errors (in addition to the usual testing for application logic correctness), even for this simple example:

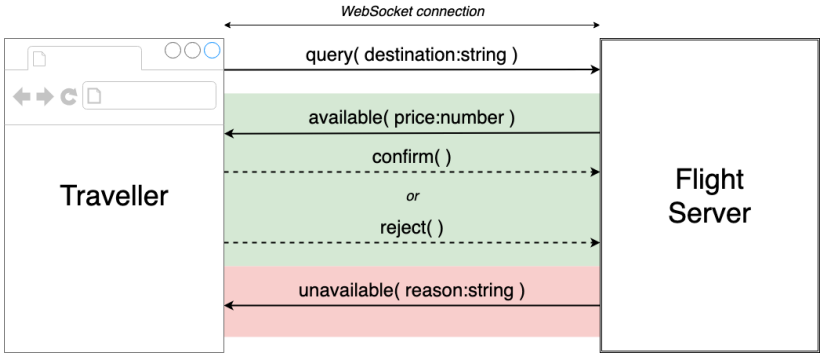


Figure 1.1: Message-Passing Architecture of Web-Based Flight Booking Service

- **Deadlocks:** how can we prevent both sides waiting for each other to respond at the same time?
- **Communication mismatches:** what if the `FlightServer` sends a string to the `Traveller` who is expecting the price to be a numeric value?
- **Channel linearity:** if the `FlightServer` takes time to respond to a query from the `Traveller`, what if the `Traveller` sends the same query twice? Given that the `FlightServer` reserves a seat on a "successful" query, violating channel linearity would hold up seats unnecessarily.

The complexity of these errors, which correlate to the complexity of tests required against these errors, scale with the complexity of the communication patterns involved. Over-reliance on integration testing to attempt to expose communication-related bugs will also slow the development process, not to mention that the time taken for these integration tests would scale with the number of roles involved. A localised, static way for verifying communication correctness is highly desirable.

Formalising Interactions with Multiparty Session Types *Multiparty Session Types* (MPST) [58] provide a framework for formally specifying a structured communication pattern between concurrent processes and verifying implementations for correctness with respect to the communications aspect.

By specifying the client-server interactions of our flight booking service as a protocol and verifying the implementations against the protocol for well-formedness, MPST theory guarantees well-formed implementations to be free from communication errors.

Limitations of State-of-the-Art Session-Typed Web Development Existing work [19, 32] that adapt the MPST framework for web development acknowledge the limitations of *JavaScript* – the language of the browser – in providing static type-level guarantees for communication safety, and proceed to use languages equipped with stronger type systems that compile to JavaScript instead.

With respect to the end goal of offering developers a development workflow that provides communication safety guarantees in modern web programming through multiparty session types, we observe two main limitations from the state-of-the-art:

L1: Incompatibility with modern web programming practices The current state-of-the-art succeeds in statically providing communication safety guarantees, but requires the developer to compromise by adapting to programming paradigms and practices that are not conventional for modern web programming. Fowler [19] requires developers to implement their web applications in a *functional* paradigm using the *Links* web programming language [5]. King et al. [32] also endorses functional web programming by requiring developers to write PureScript [47] applications, but also enforces user interfaces to be written using sequential UI frameworks.

PureScript and Links are not interoperable with the ever-growing universe of JavaScript libraries for web development. It is also unclear whether these tools support idiomatic practices in event-driven programming, such as callbacks and asynchronous implementations. Whilst communication safety guarantees provided by Fowler [19] and King et al. [32] are based on canonical MPST theory, they are not contextualised for the dynamic web-based environment. Going back to the flight booking service as an example, [19, 32] do not handle premature disconnections; if the `Traveller` receives a quote and closes their browser prematurely and the `FlightServer` is not notified of this event, the reserved seat will not be released. These compromises limit the usability of the MPST framework in modern web programming.

L2: Limited expressiveness of communication protocols The current state of the art leverage the WebSocket protocol as the communication transport between roles, which is simple to reason about and straightforward to relate to session type theory. However, WebSocket transport enforces a **server-centric** network topology, so [19, 32] enforces the same constraint on the range of communication protocols supported: server-centric protocols feature exactly one server role that is involved in all interactions.

This limits the types set of communication topologies that can be implemented by session-typed web development proposals. Suppose we extend the flight booking service as follow: we introduce a `TravelAgent` role, which runs on the browser; the `Traveller` first asks for a quote from the `TravelAgent` before carrying out the usual interactions with the `FlightServer`, so the `Traveller` can compare between both options. The current state of the art cannot implement this extended flight booking service, as the described communication interactions are no longer server-centric.

1.2 Objectives

The objective of this project is to provide developers with a development workflow that *provides communication safety guarantees in modern web programming* through multiparty session types.

We tackle the limitations of the current state of the art with two solutions:

S1: SESSIONTS We implement an end-to-end framework for writing full-stack *TypeScript* applications that **statically** conform to a communication protocol.

Developers specify their communication protocol using the Scribble protocol description language [57] which is based on multiparty session type (MPST) theory [58]. The Scribble protocol is used as a basis for generating TypeScript APIs targeted for the Node.js server-side runtime and React.js browser-side framework. By implementing the generated APIs in their full-stack applications, developers can enjoy static guarantees with respect to communication safety: there will *not* be any **deadlocks** or **communication mismatches**, and **channel linearity** will be respected.

S2: ROUTEDSESSIONS We present an extension of the canonical MPST theory that supports *routed communication*. We prove that ROUTEDSESSIONS also extends the communication safety properties of the canonical MPST theory, so processes that implement ROUTEDSESSIONS also enjoy guarantees of **deadlock-freedom** and the absence of **communication mismatches**.

Through ROUTEDSESSIONS, we can express interactions to be routed via an intermediate endpoint. By defining an encoding from canonical MPST theory to ROUTEDSESSIONS, we show that we do not lose expressiveness using routed communication. We also extend SESSIONTS to implement ROUTEDSESSIONS. SESSIONTS generates APIs that provide **channel linearity** by construction, and the extended implementation to support ROUTEDSESSIONS enjoys the same benefits.

By solution **S1**, we address the limitation **L1**. Our work provides the same communication safety guarantees as the current state of the art, but is compatible with modern web programming tools and idioms. Among the languages that compile to JavaScript, we select TypeScript as it is arguably the most intuitive to use, given it is a *superset* of JavaScript [2]. It provides developers with type-safety through its gradual, structural type system, and is also interoperable with existing JavaScript libraries that the developer may incorporate into their application. Whilst some [19] point out that this limits its usability for encoding multiparty session types, we believe that the language offers sufficient features that we can use to provide developers with communication safety guarantees whilst preserving a flexible, natural and idiomatic workflow. The TypeScript APIs we generate also support idioms commonly found in modern web development (such as callbacks and asynchronous operations), and provide an error handling mechanism tailored to the dynamic web-based environment (where roles can disconnect prematurely).

In solution **S2**, we show that routing communication interactions through an intermediate role, such as a web server, preserves the communication safety properties and the semantics from the original interactions. By extending SESSIONTS to implement ROUTEDSESSIONS, our workflow can implement protocols of arbitrary communication structures over a server-centric network topology using WebSockets; this tackles limitation **L2**.

1.3 Contributions

We present an implementation of the MPST framework for developing interactive web applications over WebSocket transport. We also present a theory of routed multiparty session types, which forms the basis for supporting arbitrary communication structures over a server-centric WebSocket-based network topology.

Related work [5, 19, 20, 32] requires the developer to implement their application using tools that are unconventional for web development, and only support communication protocols that adhere to the WebSocket network structure. To the best of our knowledge, this is the first work on multiparty session type-safe web development that provides the same communication safety guarantees using familiar industry tools in TypeScript and React, and supports peer-to-peer browser interactions over WebSockets.

In **Chapter 2**, we introduce the core session type theory and provide an overview of related work on integrating session types into application development, with a focus on the state-of-the-art proposals for session-typed web development. We also introduce TypeScript and highlight the defining features of the language.

The remainder of the report is organised in three parts as such:

Part I We present an end-to-end workflow for generating TypeScript APIs for server-centric communication protocols over WebSockets.

In **Chapter 3**, we introduce the system architecture for SESSIONTS. The implementation integrates with the Scribble toolchain to generate handler style APIs that the developer can use to write their web applications to guarantee protocol conformance by construction.

We present two code generation strategies for back-end and front-end TypeScript web development as NODEMPST and REACTMPST in **Chapter 4** and **Chapter 5** respectively.

In **Chapter 6**, we extend SESSIONTS to support advanced practices commonly found in full-stack web development, in a way that preserves communication safety. We modify the API generation strategies to support *asynchronous implementations* and introduce how to gracefully handle errors and *session cancellation* in the context of interactive web applications.

Part II We present a method to relax the server-centric constraint over WebSocket transport in order to support *peer-to-peer interactions* amongst client roles, through the concept of **routing**.

In **Chapter 7**, we introduce the TWO BUYER multiparty protocol to motivate our proposal of providing the server role with routing responsibilities, in addition to its existing interactions specified in the protocol.

In **Chapter 8**, we introduce a variant of multiparty session type theory with routing. We prove that our extensions preserve properties required for communication safety, session fidelity and deadlock freedom. We provide an encoding for global

types into routed multiparty session type theory, and prove the preservation of well-formedness and communication. Detailed lemmas and proofs in this chapter can be found in Appendix A.1.

In **Chapter 9**, we demonstrate how our API generation strategies implement our routed session type theory to support arbitrary communication topologies.

Part III In **Chapter 10**, we evaluate our work through case studies of communication protocols implemented as webservices. Through examples of multiparty sessions and routed multiparty sessions, we demonstrate how the developer may implement the generated APIs and consider the compatibility of our work with respect to modern web programming practices. We perform benchmarks on a web application built using our generated APIs against a baseline implementation to evaluate the overhead of our implementation.

Finally, we summarise our work in **Chapter 11** and propose potential areas of improvement and future work.

Remark. A part of this project was published as **Generating Interactive WebSocket Applications in TypeScript** in Volume 314 of the Electronic Proceedings in Theoretical Computer Science (EPTCS) [40], in which I am the first and lead author.

The materials in Chapters 3 to 5 are expanded from [40]. The materials in Chapters 6 to 10 are developed exclusively in this project by myself.

Chapter 2

Background

In this chapter, we introduce the theory of session types (Section 2.1), discuss related work in the area of applying session types for application development (Section 2.2), and provide background on the TypeScript language (Section 2.3).

2.1 Session Types

Web applications are one of many examples of distributed systems in practice. Distributed systems are built upon the interaction between concurrent processes, which can be implemented using the two main communication abstractions in *shared memory* and *message passing*.

Shared memory provides processes with the impression of a logical single large monolithic memory but requires programmers to understand consistency models in order to correctly reason about the consistency of shared state.

Message passing interprets the interaction between processes as the exchange of messages. This best describes the communication transports found in web applications, ranging from the stateless request-response client-server interactions via HTTP to full-duplex communication channels via the WebSocket protocol [17].

The process algebra π-calculus introduced by Milner [38] formalises the message passing abstraction in terms of the basic building blocks of sending and receiving processes. The composition of these primitives allow us to describe more complex communication sessions. *Session types* define the typing discipline for the π-calculus and provide reliability guarantees for communication sessions; the latter addresses a key challenge when reasoning about the correctness of distributed systems.

Practical application of session types in software engineering range from developing languages providing native session type support [55] to implementing session types in existing programming languages across different paradigms. Implementations of the latter approach differ by how they leverage the design philosophy and features provided by the programming language. For example, King et al. [32] leveraged the expressive type system of PureScript to perform *static* session type checking, whilst Neykova [42] introduced dynamic approaches to check the conformance of Python programs with respect to session types. We discuss these related work, among others, in Section 2.2.

2.1.1 Process Calculus

The π-calculus models concurrent computation, where processes can execute in parallel and communicate via shared names. We first consider the *asynchronous* π-calculus introduced by Honda and Tokoro [24]. Among the many flavours of the calculus which vary depending on the application domain, we outline the variant as presented in [56].

We define the syntax of processes in Figure 2.1; the asynchrony comes from the lack of continuation in the output process.

- $\mathbf{0}$ is the nil process and represents inactivity.
- $u\langle v\rangle$ is the output process that will send value v on u.
- $u(x).P$ is the input process that, upon receiving a message on u, will bind the message to x and carry on executing P under this binding.
- $P \mid Q$ represents the parallel composition of processes executing simultaneously.
- $!P$ represents the parallel composition of *infinite* instances of P; more specifically, $!P \equiv P \mid !P$.

  $\equiv$ represents the *structural congruence* equivalence relation on processes, meaning that $!P$ is indistinguishable from $P \mid !P$. We inherit the definition of structural congruence from [56].
- $(\nu a)\ P$ represents a name restriction where any occurrence of a in P is local and will not interfere with other names outside the scope of P.

The operational semantics model the interaction between parallel processes. We inherit the full operational semantics presented by Yoshida [56], and highlight the [COMM] reduction rule which specifically models message passing: if the parallel composition of an input process and output process share the same name, the composition reduces to the continuation of the input process, substituting the variable x with the message received v. We omit the definitions of substitution, free variables and free names, α-equivalence and structural congruence; the interested reader may refer to [56].

$$\frac{}{a\langle v\rangle \mid a(x).P \longrightarrow P[v/x]}\ [\text{COMM}]$$

We additionally define a process P to be stuck if P is not the nil process and P cannot be reduced any further. For example, the process $P = a(x).\mathbf{0} \mid b\langle v\rangle$ is stuck as the parallel composition of an input process and an output process that do not share the same name cannot be reduced using [COMM] . In practice, a stuck process contains communications that will never be executed.

2.1.2 Binary Session Types

A *session* represents the sequence of send and receive actions of a single participant. A *binary session* describes a session with two distinct participants. In the context of web applications, a binary session may describe the interactions between client and server.

We introduce a synchronous session calculus in Figure 2.2, inspired by [58]. We briefly discuss the main components and how it differs from the variant introduced in [56]:

- **Synchronous communication**: Output processes have a continuation that will be executed upon a successful send.
- **Polyadic communication**: More than one value can be communicated at once. We refer to these as a *vector* of values, $\vec{v}$.
- **Branching and selection**: A branching process can offer a set of branches, each defined by its own label identifier and continuation process. A selection process can select a branch by sending the corresponding label identifier alongside the payload to the branching process.
- **Labelled messages**: A label identifier is attached to all messages; the input process in Figure 2.1 is generalised as a branching process offering one branch.

The [COMM] rule in the operational semantics for this calculus exemplifies these new additions: given a binary session between distinct participants p and q where q

$P, Q ::=$		Processes
	$\mathbf{0}$	Nil Process
$\mid$	$u\langle v \rangle$	Output
$\mid$	$u(x).P$	Input
$\mid$	$P \mid Q$	Parallel Composition
$\mid$	$!P$	Replication
$\mid$	$(\nu a)\ P$	Restriction
$u, v ::=$		Identifiers
	$a, b, c, \ldots$	Names
$\mid$	$x, y, z, \ldots$	Variables

Figure 2.1: Syntax of Asynchronous π-calculus

offers a set of labelled branches, if p selects a label offered by q and sends a vector of expressions $e_1,\ldots,e_n$ that evaluate[1] to the corresponding vector of values $v_1,\ldots,v_n$, the session reduces to a session with the continuation from the selection process composed with the continuation from the selected branch of the branching process. The branching process binds the received values $v_1,\ldots,v_n$ to the variables $x_1,\ldots,x_n$.

$$\frac{\exists j \in I.\ \left(l_j = l \qquad \vec{x_j} = x_1,\ldots,x_n \qquad e_1 \downarrow v_1 \ldots e_n \downarrow v_n\right) \qquad \mathrm{p} \neq \mathrm{q}}{\mathrm{p} :: \mathrm{q} \lhd l\langle e_1 \ldots e_n\rangle.\ P \mid \mathrm{q} :: \mathrm{p} \rhd \{l_i(\vec{x_i}) : Q_i\}_{i\in I} \longrightarrow \mathrm{p} :: P \mid \mathrm{q} :: Q_j[v_k/x_k]_{k=1}^{n}} \text{ [Comm]}$$

$v ::=$	$\underline{n} \mid \texttt{true} \mid \texttt{false}$	Values
$e, e' ::=$		Expressions
	v	Values
$\mid$	x	Variables
$\mid$	$e + e' \mid e - e'$	Arithmetic Operators
$\mid$	$e = e' \mid e < e' \mid e > e'$	Relational Operators
$\mid$	$e \wedge e' \mid e \vee e' \mid \neg e$	Logical Operators
$\mid$	$e \oplus e'$	Non-Determinism
$\mathrm{p} ::=$	$\texttt{Client}, \texttt{Server}$	Participant
$P, Q ::=$		Processes
	$\mathbf{0}$	Nil Process
$\mid$	$\mathrm{p} \lhd l\langle \vec{e} \rangle.\ P$	Selection
$\mid$	$\mathrm{p} \rhd \{l_i(\vec{x_i}) : P_i\}_{i\in I}$	Branching
$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	Conditional
$\mid$	$\mu X.P$	Recursive Process
$\mid$	X	Process Variable
$l, l' ::=$	$\texttt{``str''}$	Label Identifiers
$\mathcal{M} ::=$	$\mathrm{p} :: P \mid \mathrm{q} :: Q$	Binary Session

Figure 2.2: Syntax of Session Calculus with Branching, Selection and Recursion

Additionally, the calculus introduces:

[1] We adopt the operational semantics for expression evaluation $e \downarrow v$ as defined in [56].

- **Conditionals**: If $e \downarrow \texttt{true}$, the process $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$ reduces to P; if $e \downarrow \texttt{false}$, the process $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$ reduces to Q.
- **Recursion**: Following the *equirecursive* approach, the occurrence of the process variable X in the recursive process can be expanded into the process transparently; more specifically, $\mu X.P \equiv P[(\mu X.P)/X]$.

Session types represent the type theory for our session calculus. We define the syntax of session types for binary sessions in Figure 2.3. $\vec{U}$ denotes a vector of sorts.

$U ::=$	$\texttt{int} \mid \texttt{bool}$	Sorts
$T ::=$		Session Types
	$\texttt{end}$	Termination
$\mid$	$\mathrm{p} \,\&\, \left\{ l_i(\vec{U_i}) : T_i \right\}_{i \in I}$	Branching
$\mid$	$\mathrm{p} \oplus \left\{ l_i(\vec{U_i}) : T_i \right\}_{i \in I}$	Selection
$\mid$	$\mu \mathtt{t}.T$	Recursive Type
$\mid$	$\mathtt{t}$	Type Variable

Figure 2.3: Syntax of Session Types

We derive the type of a process with a typing judgement of the form $\Gamma \vdash P : S$, which reads, *under the typing context* Γ*, process* P *has session type* S. The *typing context* records typing assumptions used as part of the derivation: in the case of binary session types, the context maps expressions to sorts, and process variables to session types. A typing judgement is constructed in terms of inference rules defined inductively on the structure of processes and expressions.

We present the rules for [Ty-Sel] and [Ty-Bra] ; the remaining rules follow from [56] and can be trivially defined as they leverage the syntactic similarities between session types and our session calculus.

$$\frac{\Gamma \vdash e_1 : U_1 \ \ldots\ \Gamma \vdash e_n : U_n \qquad \Gamma \vdash P : T}{\Gamma \vdash \mathrm{p} \lhd l\langle e_1 \ldots e_n \rangle.\ P : \mathrm{p} \oplus l(U_1, \ldots, U_n) : T} \text{ [Ty-Sel]}$$

$$\frac{\forall i \in I. \left(\vec{x_i} = x_1, \ldots, x_n \qquad \vec{U_i} = U_1, \ldots, U_n \qquad \Gamma, x_1 : U_1, \ldots, x_n : U_n \vdash P_i : T_i \right)}{\Gamma \vdash \mathrm{p} \rhd \{ l_i(\vec{x_i}) : P_i \}_{i \in I} : \mathrm{p} \,\&\, \left\{ l_i(\vec{U_i}) : S_i \right\}_{i \in I}} \text{ [Ty-Bra]}$$

The definition of stuck processes from Section 2.1.1 motivate the discussion of communication errors that may occur during interactions among participants. We outline two of the main classes of errors:

- **Deadlock**: Progress cannot be made when the two participants expect to be receiving a message from each other at the same time.
- **Communication mismatch**: Progress cannot be made when the selection process sends a message with a label identifier not offered by the branching process; likewise, the payload sent must be compatible with the sort expected by the branching process for the selected branch.

Session types ensure that well-typed binary sessions are guaranteed to be free from these communication errors through the concept of *duality*. Duality defines a notion of *compatibility* between processes: two session types are dual with respect to each other if the communication between them (i.e. pairs of sending and receiving actions) always match (i.e. with respect to the selected label and message payload type). We define $\overline{S}$ as the dual type of S in Table 2.1; message payload types are omitted for brevity.

$$
\begin{aligned}
\overline{\texttt{end}} &= \texttt{end} \\
\overline{\mathtt{p} \,\&\, \{l_i : T_i\}_{i \in I}} &= \mathtt{q} \oplus \left\{ l_i : \overline{T_i} \right\}_{i \in I} \\
\overline{\mathtt{p} \oplus \{l_i : T_i\}_{i \in I}} &= \mathtt{q} \,\&\, \left\{ l_i : \overline{T_i} \right\}_{i \in I} \\
\overline{\mu \mathtt{t}.T} &= \mu \mathtt{t}.\overline{T} \\
\overline{\mathtt{t}} &= \mathtt{t}
\end{aligned}
$$

Table 2.1: Duality of Binary Session Types Involving Participants p and q

Consequently, a binary session is well-typed if the participating processes are typed to be dual with respect to each other: we illustrate this in [MTY] .

$$
\frac{\cdot \vdash P : T \qquad \cdot \vdash Q : \overline{T}}{\vdash \mathtt{p} :: P \mid \mathtt{q} :: Q} \text{[MTY]}
$$

The definition of duality alone restricts the definition of well-typed binary sessions to those where the two processes are derived to be *exactly* dual types of one another. Consider the pair of session types below:

$$
\begin{aligned}
T_{\text{Client}} &= \texttt{Server} \oplus \texttt{Succ(nat).}\ \texttt{Server} \,\&\, \texttt{Succ(int).}\ \texttt{end} \\
T_{\text{Server}} &= \texttt{Client} \,\&\, \begin{cases} \texttt{Succ(int)} & : \texttt{Client} \oplus \texttt{Succ(int).}\ \texttt{end} \\ \texttt{Quit()} & : \texttt{end} \end{cases}
\end{aligned}
$$

Whilst $\overline{T_{\text{Client}}} \neq T_{\text{Server}}$, this pair of session types is intuitively compatible as the client is selecting a branch offered by the server, where the session types for the continuations of this branch for both participants are indeed dual. Regarding payload, the server is expecting `int`, but the client sends `nat`, which is a *subtype* of `int`.

This motivates the concept of subtyping in session types, which allows a process to be typed by its "supertype" when required. $\prec$[2] defines the subtyping relation: $T \prec T'$ reads *T is a subtype of T'*, and is defined coinductively on the structure of T.

We present the inference rules for [Sub-Sel] and [Sub-Bra] inspired by [58] but adapted for polyadic communication; the intuition behind subtyping and subsorting is outlined below:

- **Selection**: The supertype of a selection process offers a superset of the internal choices and can send more generic types of payload; intuitively, if a process sends a `nat`, the payload is compatible with receivers expecting a more generic `int` payload.
- **Branching**: The supertype of a branching process offers a subset of the branches and expects more specific types of payload; intuitively, if a process expects to receive an `int`, it can handle a `nat` payload.

$$\frac{\forall i \in I.\left(\vec{U}_i = U_1,\ldots,U_n \quad \vec{U}'_i = U'_1,\ldots,U'_n \quad U_1 \prec U'_1 \ \ldots \ U_n \prec U'_n \quad T_i \prec T'_i\right)}{\mathrm{p} \oplus \left\{l_i(\vec{U}_i) : T_i\right\}_{i\in I} \prec \mathrm{p} \oplus \left\{l_i(\vec{U}'_i) : T'_i\right\}_{i\in I\cup J}} \text{[Sub-Sel]}$$

$$\frac{\forall i \in I.\left(\vec{U}_i = U_1,\ldots,U_n \quad \vec{U}'_i = U'_1,\ldots,U'_n \quad U'_1 \prec U_1 \ \ldots \ U'_n \prec U_n \quad T_i \prec T'_i\right)}{\mathrm{p} \,\&\, \left\{l_i(\vec{U}_i) : T_i\right\}_{i\in I\cup J} \prec \mathrm{p} \,\&\, \left\{l_i(\vec{U}'_i) : T'_i\right\}_{i\in I}} \text{[Sub-Bra]}$$

We also introduce *subsumption* in [Ty-Sub] to incorporate the subtyping relation into the typing judgement.

$$\frac{\Gamma \vdash P : T \qquad T \prec T'}{\Gamma \vdash P : T'} \text{[Ty-Sub]}$$

This allows us to construct a derivation to show that the binary session

$$\mathcal{M} = \texttt{Client} :: P_{\text{Client}} \mid \texttt{Server} :: P_{\text{Server}}$$

is well-typed, assuming P_{Client} and P_{Server} are typed T_{Client} and T_{Server} respectively.

$$\dfrac{\dfrac{\vdots}{\cdot \vdash P_{\text{Client}} : T_{\text{Client}}} \qquad \dfrac{\dfrac{\vdots}{\cdot \vdash P_{\text{Server}} : T_{\text{Server}}} \qquad \dfrac{\vdots}{T_{\text{Server}} \prec \overline{T_{\text{Client}}}}}{\cdot \vdash P_{\text{Server}} : \overline{T_{\text{Client}}}}\text{[Ty-Sub]}}{\vdash \texttt{Client} :: P_{\text{Client}} \mid \texttt{Server} :: P_{\text{Server}}}\text{[MTy]}$$

[2] The $\prec$ operator is also an overloaded relation on sorts to express subsorting, i.e. `nat` $\prec$ `int`.

2.1.3 Multiparty Session Types

Whilst binary session types provide communication guarantees between exactly 2 participants, distributed systems generally involve more parties in practice. This is equally relevant in interactive web applications, as motivated by the *Battleships* game example in [32] where the server coordinates interactions between two players.

Whilst there is a natural syntactical extension to our session calculus for describing multiparty sessions[3] as

$$\mathcal{M} ::= \quad \mathtt{p}_1 :: P_1 \mid \mathtt{p}_2 :: P_1 \mid \ldots \mid \mathtt{p}_\mathtt{n} :: P_n$$

the same cannot be said for the binary session typing discipline, particularly with respect to duality. The same notion of duality does not extend to the decomposition of multiparty interactions into multiple binary sessions: Yoshida and Lorenzo [58] presents counterexamples of well-typed binary sessions that, when composed to represent a multiparty session, results in communication errors thus violating guarantees of well-typed sessions.

Honda et al. [25] presents multiparty session types to extend the binary session typing discipline for sessions involving more than 2 participants, whilst redefining the notion of compatibility in this multiparty context. *Multiparty session types* (MPST) are defined in terms a *global type*, which provides a bird's eye view of the communication protocol describing the interactions between pairs of participants. Figure 2.4 defines the syntax of global types inspired by [58]; we omit message payload types for brevity.

$G ::=$		Global Types
	$\mathtt{end}$	Termination
$\mid$	$\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$	Message Exchange
$\mid$	$\mu \mathtt{t}.G$	Recursive Type
$\mid$	$\mathtt{t}$	Type Variable

Figure 2.4: Syntax of Global Types

To check the conformance of a participant's process against the protocol specification described by the global type, we *project* the global type onto the participant to obtain a session type that only preserves the interactions described by the global type that pertain to the participant. Projection is defined by the $\upharpoonright$ operator, more commonly seen in literature in its infix form as $G \upharpoonright \mathtt{p}$ describing the projection of global type G for participant $\mathtt{p}$. Intuitively, the projected local type of a participant describes the protocol from the viewpoint of the participant.

[3] We also adopt the shorthand $\mathcal{M} ::= \prod_{i=1}^{n} \mathtt{p}_\mathtt{i} :: P_i$ as used in the literature.

More formally, projection can be interpreted as a *partial function* $\upharpoonright :: G \times \mathtt{p} \rightharpoonup S$, as the projection for a participant may be undefined for an ill-formed global type; Yoshida [56] presents examples of where this is the case, and Yoshida and Lorenzo [58] presents the formal definition of projection. We focus on the definition of projection for the message exchange global type given below:

$$\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \upharpoonright \mathtt{r} = \begin{cases} \mathtt{q} \oplus \{l_i : G_i \upharpoonright \mathtt{p}\}_{i \in I} & \text{if } \mathtt{r} = \mathtt{p} \\ \mathtt{p} \,\&\, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i \in I} & \text{if } \mathtt{r} = \mathtt{q} \\ \bigsqcap_{i \in I} G_i \upharpoonright \mathtt{r} & \text{otherwise} \end{cases}$$

The third case describes that, for a participant not involved in the communication, the projections of the continuations $G_i \upharpoonright \mathtt{r}$ do not necessarily have to be the same, but they need to be "compatible". The $\sqcap$ syntax denotes the *merging operator*. It is a partial function on local types allows us to construct a projection that is compatible with the projections on the continuations. We use the *inductive* definition presented by Scalas and Yoshida in [50] Definition 3.3, shown in Figure 2.5.

$$\begin{aligned}
\mathtt{end} \sqcap \mathtt{end} &= \mathtt{end} && [\textsc{Merge-End}] \\
\mathtt{t} \sqcap \mathtt{t} &= \mathtt{t} && [\textsc{Merge-RecVar}] \\
\mu \mathtt{t}.T \sqcap \mu \mathtt{t}.T' &= \mu \mathtt{t}.T \sqcap T' && [\textsc{Merge-Rec}] \\
\mathtt{p} \oplus \{l_i : T_i\}_{i \in I} \sqcap \mathtt{p} \oplus \{l_i : T_i\}_{i \in I} &= \mathtt{p} \oplus \{l_i : T_i\}_{i \in I} && [\textsc{Merge-Sel}] \\
\mathtt{p} \,\&\, \{l_i : T_i\}_{i \in I} \sqcap \mathtt{p} \,\&\, \left\{l_j : T'_j\right\}_{j \in J} &= \mathtt{p} \,\&\, \left\{l_k : T''_k\right\}_{k \in I \cup J} && [\textsc{Merge-Bra}]
\end{aligned}$$

$$\text{where } T''_k = \begin{cases} T_k & \text{if } k \in I \setminus J \\ T'_k & \text{if } k \in J \setminus I \\ T_k \sqcap T'_k & \text{if } k \in I \cap J \end{cases}$$

otherwise undefined

Figure 2.5: Definition of Merging Operator

The notion of compatibility in multiparty session types is still captured by [MTY] , but adapted to consider the local projections for all participants as supposed to dual types in the binary case.

$$\frac{\forall i \in I.\ (\cdot \vdash P_i : G \upharpoonright \mathtt{p_i}) \qquad \mathtt{pt}(G) \subseteq \{\mathtt{p_i} \mid i \in I\}}{\vdash \prod_{i \in I} \mathtt{p_i} :: P_i \ : G} \ [\text{MTY}]$$

For a multiparty session $\mathcal{M} = \prod_{i \in I} \mathtt{p}_i :: P_i$ to be well-typed by a global type G:

1. All participant processes $\mathtt{p}_i P_i$ are well-typed with respect to their corresponding well-defined projection $G \upharpoonright \mathtt{p}_i$, and
2. G does not describe interactions with participants not defined in $\mathcal{M}$. $\mathtt{pt}(G)$ denotes the set of participants in the global type G, defined inductively on G as shown on Table 2.2.

$$
\begin{aligned}
\mathtt{pt}(\mathtt{end}) &= \emptyset \\
\mathtt{pt}(\mathtt{t}) &= \emptyset \\
\mathtt{pt}(\mu \mathtt{t}.G) &= \mathtt{pt}(G) \\
\mathtt{pt}(\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}) &= \{\mathtt{p}, \mathtt{q}\} \cup \bigcup_{i \in I} \mathtt{pt}(G_i)
\end{aligned}
$$

Table 2.2: Definition of Participants in Global Types

Well-typed multiparty sessions enjoy the following communication guarantees as outlined in [6]:

- **Communication safety**: The types of sent and expected messages will always match.
- **Protocol fidelity**: The exchange of messages between processes will abide by the global protocol.
- **Progress**: Messages sent by a process will be eventually received, and a process waiting for a message will eventually receive one; this also means there will not be any sent but unreceived messages.

This motivates an elegant, decentralised solution for checking protocol conformance in practice: once the global type for the protocol is defined, local processes can verify their implementation against their corresponding projection in isolation, independent of each other. We illustrate this in Figure 2.6.

2.2 Related Work

Whilst session type theory represents the type language for concurrent processes, it also forms the theoretical basis for proposals introduced to implement session types for real-world application development: the *Scribble* project is one such proposal. We discuss related work that implement session types for software engineering using the Scribble project in Section 2.2.1, and focus on existing work for session-typed web development in Section 2.2.2.

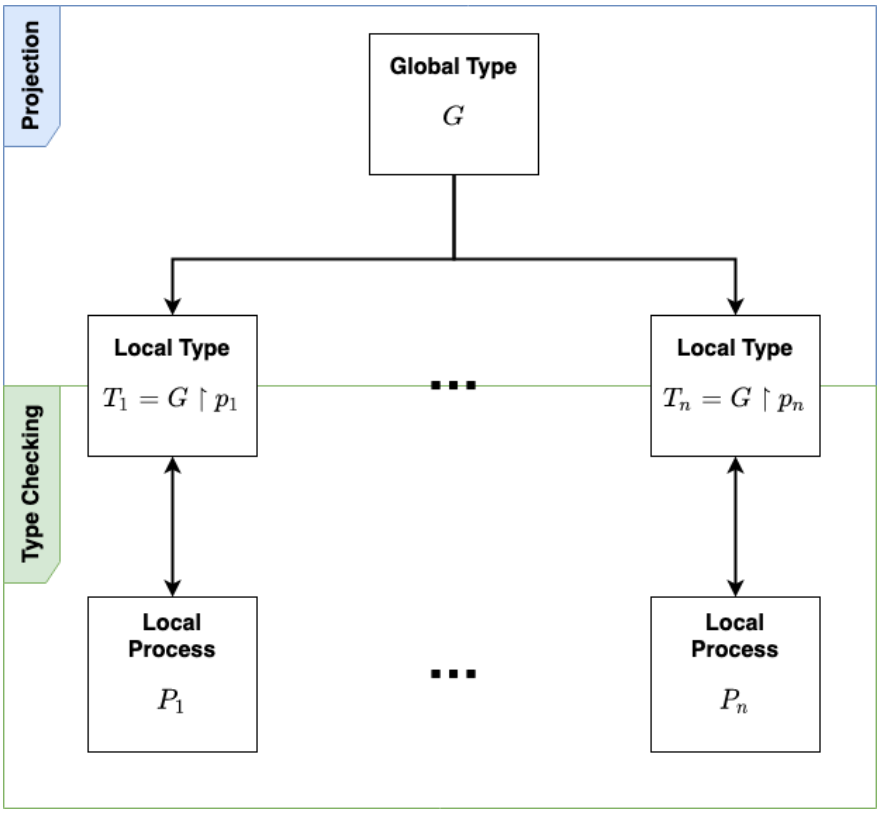


Figure 2.6: Type Checking with Multiparty Session Types

2.2.1 Scribble and Endpoint API Generation

Scribble [57] is a platform-independent description language for the specification of message-passing protocols. The language describes the behaviour of communicating processes at a high level of abstraction: more importantly, the description is independent from implementation details in the same way that the type signature of a function declaration is decoupled from the corresponding function definition.

A Scribble *protocol specification* describes an agreement of how participating systems, referred to as roles, interact. The protocol stipulates the sequence of structured messages exchanged between roles; each message is labelled with a name and the type of payload carried by the message.

We present an example of a Scribble protocol in Figure 2.7 adapted from [28]. The protocol specifies an arithmetic web service offered by a `Server` to a `Client`. The `Client` is permitted to either:

- Send two `int`s attached to an `Add` message, where the server will respond with an `int` in a message labelled `Res`, and the protocol restarts; or,
- Send a `Quit` message, where the server will respond with a `Terminate` message and the protocol ends.

The platform-independent nature of Scribble can be observed from the `type` declaration on Line 1: the developer has the freedom to specify message payload formats

```
type <java> "java.lang.Integer" from "rt.jar" as int;

global protocol Adder(role C, role S) {
    choice at C {
        Add(int, int)   from C to S;
        Res(int)        from S to C;
        do Adder(C, S);
    } or {
        Quit()      from C to S;
        Terminate() from S to C;
    }
}
```

Figure 2.7: Adder Protocol in Scribble

and data types from the target language of the implementation – in this case, aliasing the built-in Java integer as `int` throughout the protocol.

To observe the parallels between MPST theory and the Scribble language, we present the corresponding global type for the ADDER protocol below.

$$G_{\text{Adder}} = \mu \mathtt{t}.\mathtt{C} \to \mathtt{S} : \begin{cases} \mathtt{Add(int,\ int)} & : \mathtt{S} \to \mathtt{C} : \mathtt{Res(int)}\ .\ \mathtt{t} \\ \mathtt{Quit()} & : \mathtt{S} \to \mathtt{C} : \mathtt{Terminate()}\ .\ \mathtt{end} \end{cases}$$

The protocol specification language is a component of the broader *Scribble toolchain* initiated by Honda et al. [57], through which the toolchain also facilitates the development of *endpoint applications* that conform to user-specified protocols.

A Scribble global protocol can be projected to a role, or *endpoint*, to obtain a *local protocol* which represents the global protocol viewed from the perspective of the endpoint. This allows the endpoint to *verify* their implementation against their local protocol for conformance, independent of other endpoints. The communication safety guarantees from MPST theory also apply here: if the implementation for each endpoint is verified against their local protocol, the distributed system as a whole will conform to the global protocol.

The Scribble toolchain can convert the local protocol into an *endpoint finite state machine* (EFSM). An EFSM encodes the control flow of the local protocol into a communication automaton: there is a well-defined initial *state*, and each *transition* from some state to a successor state corresponds to a valid communication action (i.e. sending or receiving a message) permitted at that endpoint at that state.

We show the EFSM of the `Server` for the ADDER protocol in Figure 2.8. We use this as an example to walk through the components that build up the EFSM.

- **Role Identifiers:** The set of roles that the endpoint interacts with. For Figure 2.8, this is the singleton set { `Client` }.
- **Label Identifiers:** The set of message labels that the endpoint sends or expects to receive. For Figure 2.8, this is set { `Add`, `Res`, `Quit`, `Terminate` }.
- **Payload Types:** The set of message payload data types that the endpoint sends or expects to receive. For Figure 2.8, this is singleton set { `int` }.

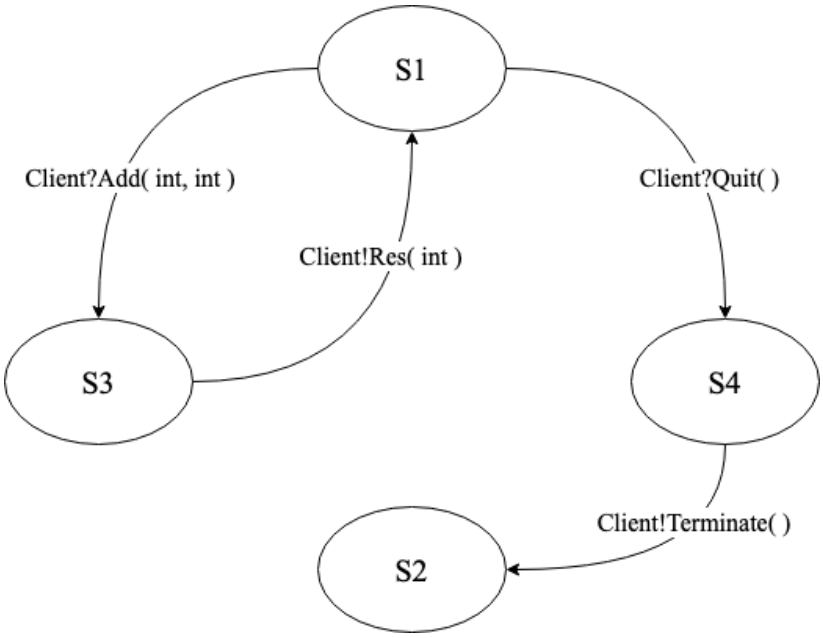


Figure 2.8: Server EFSM for Adder protocol

- **Actions:** The set of send and receive actions permitted for the endpoint. An action is defined in terms of a role identifier, label identifier and a vector of payload types. For Figure 2.8, this is the set of visible transitions as illustrated.

  `Client!Terminate()` denotes a *send action* that sends the `Terminate` message with no payload to `Client`.

  `Client?Add(int,int)` denotes a *receive action* that receives the `Add` message with two `int` values as payload from `Client`.

- **State Identifiers:** For Figure 2.8, this is the set { `S1,S2,S3,S4` }.

- **State Transition Function:** The *partial function*, denoted δ in the literature, which maps state identifiers and actions to successor state identifiers. It is a partial function because not all tuples of state identifiers and actions constitute valid transitions permitted in the state machine.

  For Figure 2.8, (`S1, Client?Add(int,int)`) is in the domain of δ; the successor state is `S3`. Conversely, δ(`S1, Client!Res(int)`) is undefined.

We also inherit the characteristics of EFSMs derived from well-formed protocols, as presented in [28]:

- There is *exactly one* initial state (`S1` in Figure 2.8).

  An initial state has no incoming transitions. Formally, the state identifier of the initial state is not in the range of δ.

- There is *at most one* terminal state (`S2` in Figure 2.8).

  A terminal state has no outgoing transitions. Formally, the state identifier of the terminal state is not in the domain of δ.

- Every non-terminal state is either a *send* state or a *receive* state.

  A send state only has send actions in its outgoing transitions. `S3` and `S4` are the send states in Figure 2.8.

  A receive state only has receive actions in its outgoing transitions. `S1` is the only receive state in Figure 2.8.

Developers can use the EFSM to verify their implementation for protocol conformance with respect to an endpoint in the global protocol. Endpoint API generation is a common approach [27, 28, 32, 42, 49]: these proposals generate APIs in the target language which the developer can use to implement their endpoint application and enjoy the following guarantees:

- **Behavioural typing:** The execution trace of messages sent and received by the application is accepted by the EFSM, which means it conforms to the communication structure of the global protocol;
- **Channel linearity:** Each transition in the EFSM represents one channel resource. The application transitions from some state S to some successor state S' once *and only once*, after which it must no longer be able to access a reference (e.g. have an alias) to S.

We highlight two distinct proposals for endpoint API generation that differ by how they leverage features in the target language to guarantee the above properties.

Hybrid Session Verification in Java [28]

Hu and Yoshida implemented a workflow that performs *hybrid* verification of communication protocol conformance for Java applications. The "hybrid" concept is composed of the two main components below:

Static session typing The EFSM is interpreted under the object-oriented paradigm as follow: **(1)** states are represented as classes; **(2)** supported transitions on each state are represented as instance methods, parameterised by the role, label and payload involved in the message exchange. A send method takes the payload to send as a parameter, whilst a receive method is a blocking call that requires the caller to allocate a `Buf<T>` wrapper on the stack (where `T` is the expected payload type), then the receive method populates the payload into the wrapper and returns upon receiving from the channel. These instance methods return a new instance of the successor state class.

Runtime channel linearity checks By exposing send and receive actions as instance methods, the developer can keep aliases of EFSM state instances and invoke the IO action more than once. Java does not have a linear type system, and hence cannot statically verify linear usage; this means linearity checks need to be done at runtime. Each state keeps track of its usage in a private boolean flag and throws an

exception when the instance method is called twice. Similarly, the `SessionEndpoint` class keeps track of whether the connection is open or close, and throws an exception when program execution exits the scope of the session endpoint and a terminal state has yet to be reached.

We find that this proposal strikes a good balance between maximising static communication safety guarantees whilst providing an intuitive set of APIs for developers to efficiently write their applications. For example, the EFSM encoding takes advantage of the type system to statically enforce valid transitions; whilst this exposes channel resources and linear usage needs to be monitored dynamically to compensate for the lack of linear types, the generated APIs complement the imperative style of application code written in Java

Runtime Monitors in Python [42]

Neykova targeted the MPST methodology for Python programs and proposed to generate *runtime monitors* from the EFSM. These monitors expose APIs for sending and receiving messages, which is used by the developer in their implementation. The runtime monitor is an abstraction between the developer's implementation and the actual communication channel, and "executes" the EFSM internally to ensure protocol conformance. When the developer sends a message (with some label and payload) using the API, the runtime monitor checks whether this send action conforms to the current EFSM state, and if so, performs the send and advances to the successor state. Likewise, when the developer invokes a receive, the runtime monitor verifies that this is permitted at the current EFSM state before returning the received payload.

We observe that this approach complements the dynamic typing nature of the Python language, which makes it sensible to perform behavioural typing at runtime. As the send and receive IO primitives are made available to the developer, there are no "instances" of channel resources created, so the developer cannot explicitly hold a reference to some state in the EFSM (let alone keep aliases), so channel linearity is trivially guaranteed here.

2.2.2 Session Types in Web Development

We discuss the current state-of-the-art proposals for session-typed web development.

API Generation in PureScript [32]

King et al. presented an approach for integrating *multiparty* session types into web development using the PureScript language [47]. PureScript is a strongly typed functional language that compiles to JavaScript. This proposal takes advantage of its expressive type system to provide *static* guarantees for protocol conformance. We outline the main components of their EFSM encoding:

Actions as type classes The semantics of the state transition function in the EFSM express that a `(state, action)` tuple *uniquely defines* a successor `state`. These semantics can be expressed by *multi-parameter type classes (MPTC) with functional dependencies*: `class Send r s t a | s -> t r a` defines `Send` as a MPTC parameterised by recipient `r`, current state `s`, successor state `t` and payload type `a`; `s -> t r a` expresses the functional dependency that, for an instance of this type class, the current state uniquely determines the successor state, the recipient and payload type. These type classes are independent of the EFSM.

Transitions as instances of type classes By encoding states as data types, valid EFSM transitions are encoded as *instances* of the type classes. If `S1` is an output state, sending an `Int` to `Svr` with successor state `S2`, the PureScript encoding would be `instance SendS1 :: Send Svr S1 S2 Int`. Because of the functional dependency, the developer cannot instantiate an invalid transition (e.g. `Send Svr S1 S3 Bool`, since `S1` uniquely determines the other type parameters. PureScript statically resolves these type constraints at compile time.

Channel linearity by construction Whilst PureScript does not support linear types, this proposal carefully designed the session runtime using a collection of communication combinators to conceal the channel resource, which guarantees channel linearity by construction.

This proposal is relevant to the problem we are tackling. In particular, the final point about guaranteeing channel linearity by construction is something we can build upon in our solution, since linear types is a feature left to be desired in "mainstream" programming languages. We also observe the challenges of session-typed GUI programming, as shown by the careful choice made in [32] to use the *Concur UI* framework [30] which builds UI elements sequentially to model sequential sessions. Not doing so would make channel linearity violation possible (e.g. by binding a send transition to a button click event, the button may still be active after the transition is complete; clicking on it again will reuse the channel resource). Whilst King et al. [32] supports multiparty session types, the communication protocol must describe all interactions to go through a centralised server role to respect the network topology enforced by WebSocket transport.

As motivated in Section 1.1, the functional paradigm and sequential UI framework are necessary features for the proposal to statically provide communication safety guarantees, but this type of workflow is not compatible with modern web programming practices and compromises on usability.

Session-Typed GUI Programming using Links [19]

Fowler presented the first formal integration of session typing and GUI programming. His work formalised the *Model-View-Update* (MVU) architectural pattern for GUI development. An MVU application features a *model* encapsulating application state, a *view function* rendering the state on the user interface, and an *update function*

handling *messages* produced by the rendered model to produce a new model. Fowler introduces the concept of model types to express type dependencies between these components: a **model type** uniquely defines a *view* function, set of *messages* and *update function* – rather than producing a new model, the update function defines valid transitions to other model types. This approach guarantees channel linearity for GUI programming: the update function transitions to a successor model, which only renders an interface with the set of messages unique to that model, so channel actions from the predecessor model are not rendered, which makes reuse impossible. The core MVU calculus is also extended with *commands* and *linearity* required to express communication actions in session type theory.

Fowler's proposal [19] is implemented using the *Links* web programming language [5]. Links is a strict, typed, functional language that can be used to write full-stack web applications; and similar to PureScript, it is compiled to JavaScript. As it has a linear type system, session types can be implemented using native language features to monitor linear channel usage. Links also supports error handling for session types, as introduced in [20]; given side-effects must be made explicit in functional programming, Fowler et al. [20] introduces a *session cancellation* operator to ensure that channel resources are closed after handling side-effects, such as errors.

This work succeeds in retaining the declarative approach for writing web interfaces, as the view function incorporates standard *Hyper-Text Mark-up Language* (HTML). The novel approach of guaranteeing channel linearity in GUI programming through the model type abstraction is something we can build upon. However, Links does not support multiparty session types, and similar to [32], it can only express server-centric communication structures over WebSocket transport.

2.3 TypeScript

We introduce the TypeScript language [2] as our choice of target language for session type API generation. Developed by *Microsoft Research*, TypeScript is an extension to JavaScript to address the deficiencies of the latter in *developing* and *maintaining* large-scale complex applications. Syntactically, TypeScript is a *superset* of JavaScript, so every JavaScript program is a TypeScript program. The TypeScript Compiler is used to compile a TypeScript program into JavaScript source code, with full type erasure.

We introduce specific language features used to implement our API generation solution throughout the report as needed; here, we highlight the key properties of the type system implemented in the language.

Structural Typing In a structural type system, type equivalence is determined by *shape* rather than by name (which is the case in a *nominal* type system).

Consider the following TypeScript code:

```
1 class ThisSquare {
2     constructor(public side: number) { }
3 };
4
```

```
5  class ThatSquare {
6      constructor(public side: number) { }
7  };
8
9  const area = (sq: ThisSquare) => sq.side * sq.side;
10
11 area(new ThisSquare(2));     // ok
12 area(new ThatSquare(2));     // ok
13 area({ side: 2 });           // ok
```

The `area` function takes a `ThisSquare` as parameter. Under a structural type system, Lines 12 and 13 will type-check, because `ThatSquare` and the object literal created from scratch matches the *shape* of `ThisSquare` – all of them have a `side` property typed `number`.

In languages (e.g. Java) that use a nominal type system, Line 12 will not type-check because `ThatSquare` is not named `ThisSquare`.

Gradual Typing In a gradual type system, a program can have parts that are statically typed and other parts are dynamically typed [51]. TypeScript distinguishes dynamically typed code using the `any` type.

```
// Invoke remote API
fetch('https://jsonplaceholder.typicode.com/todos/1')
    // Convert to JavaScript Object Notation
    .then((response: Response) => response.json())
    .then((json: any) => {
        // Up to the developer to correctly deserialise;
        // incorrect implementations will cause
        // runtime type error.
    });
```

The rationale for this decision in [2] is that, JavaScript programs tend to interact with data of unspecified types (such as fetching data from API calls over the network); these parts need to be dynamically typed in order to give developers a smooth transition into TypeScript, and for TypeScript to be usable in a distributed system setting.

Based on the compatibility of TypeScript with JavaScript, we believe that TypeScript API generation for session-typed web development best achieves our objective of providing developers with a workflow that provides communication safety guarantees in *modern web programming* through multiparty session types. We argue that the type system of TypeScript, along with other language features we introduce throughout the course of the report, is sufficient for implementing session type theory in a manner that complements idiomatic web development practices.

Part I

Implementing Server-Centric Topologies over WebSockets

Chapter 3

SESSIONTS: Session Type API Generation for TypeScript

In this chapter, we present SESSIONTS, a toolchain for generating TypeScript APIs that developers can use to write web applications that conform to their specified communication protocol. The toolchain is publicly available at [39].

SESSIONTS supports communication protocols that define **server-centric** topologies, meaning that: **(1)** there is exactly one participant executed on the Node.js runtime; **(2)** all other participants run on the web browser; and **(3)** all non-server participants only communicate with the server. An example would be the NOUGHTS AND CROSSES multiplayer game (Listing 10.1). We relax this assumption in Part II.

3.1 Development Workflow

We motivate our development workflow from previous work [28, 32, 49] by extending the Scribble toolchain and generating APIs that integrate the developer's application logic into the execution of the communication automata.

We visualise the workflow in Figure 3.1 and provide a brief overview:

1. The developer supplies the communication protocol written in Scribble (Section 3.1.1), stating the role (hereafter *endpoint*) to generate APIs for, and the code generation *target* (i.e. whether the role runs on the back-end web server or the front-end web browser).

2. SESSIONTS delegates to the Scribble toolchain for verifying the well-formedness of the protocol and expects to receive a DOT graph representation of the endpoint FSM (Section 3.1.2). SESSIONTS parses the endpoint's interactions from the DOT graph and generates TypeScript APIs for the developer (Section 3.1.3) tailored to the specified target.

3. The developer implements their web application using the generated APIs. Implementations that pass the type-checking phase of the TypeScript Compiler are guaranteed to be free from communication errors by session type theory.

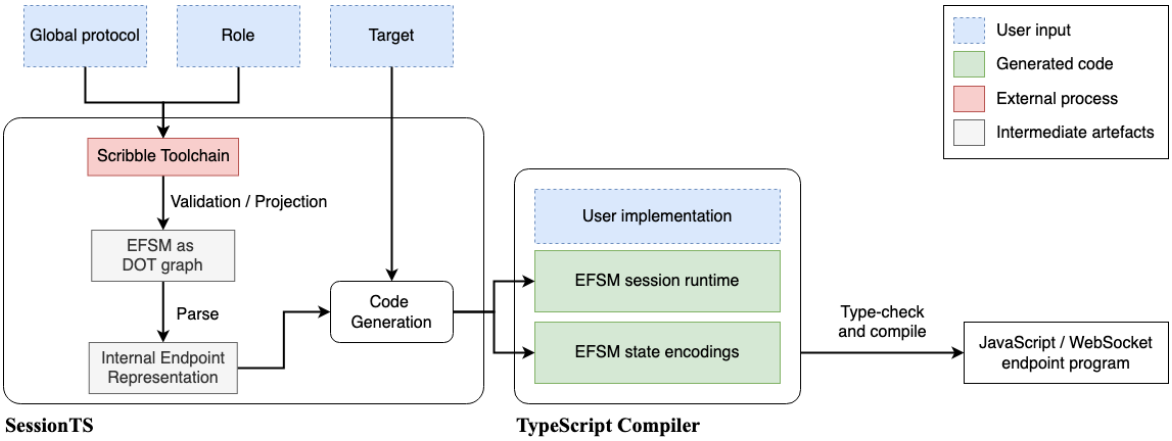


Figure 3.1: Overview of SESSIONTS Development Workflow

3.1.1 Protocol Specification with Scribble

We use the Scribble protocol description language, as presented in [57], for formalising the communication structure. This is inspired by existing work on implementing session type theory in mainstream programming languages [28, 32, 42, 49]. We use the variant of the Scribble language discussed in Section 2.2.1.

Type declaration statements Specific to SESSIONTS, the developer is *not* required to explicitly add type declaration statements for built-in types. Listing 3.1 is a *syntactically correct* Scribble protocol as far as SESSIONTS is concerned. Internally, SESSIONTS inspects the protocol file and parses existing type declarations using regular expressions (or *regex*) – this is necessary to extract any custom data types that will appear in the communication (for example, Listing 10.1), and allows SESSIONTS to inject "boilerplate" type declarations for built-in TypeScript types before calling Scribble.

```
1  global protocol Adder(role Client, role Svr) {
2      choice at Client {
3          ADD(number, number) from Client to Svr;
4          RES(number)         from Svr to Client;
5          do Adder(Client, Svr);
6      } or {
7          QUIT(string)        from Client to Svr;
8          choice at Svr {
9              THANKS()        from Svr to Client;
10         } or {
11             TERMINATE()     from Svr to Client;
12         }
13     }
14 }
```

Listing 3.1: The ADDER Protocol

We will use the ADDER protocol as a running example to demonstrate how our

work performs TypeScript API generation. This describes a binary session between `Client` and `Svr`: if `Client` asks `Svr` to `ADD` two numbers, `Svr` responds with the `RESult`; this loop continues until `Client` kindly asks to `QUIT` with a message, where `Svr` either responds kindly with `THANKS`, or just says `TERMINATE`, but both choices will close the session.

3.1.2 From Scribble to EFSM

Given the protocol and endpoint, we use Scribble to validate the well-formedness of the protocol and extract information from the protocol relevant for the endpoint. The latter is expressed as a finite state machine where each state restricts the possible transitions, and transitions between states are represented by communication actions, i.e. the sending or receiving of a message.

Scribble expresses the EFSM using the *DOT* graph description language [23], with each communication action encoded as the label of the corresponding state transition. SESSIONTS uses the *pydot* library [48] to parse the graph into an internal representation of the EFSM. We define an `EfsmBuilder` class with APIs designed for constructing the EFSM representation by iterating over the state transitions from the DOT representation.

```
1 @dataclass
2 class Endpoint:
3     protocol: str
4     role    : str
5     server  : str
6     efsm    : EFSM
7     types   : typing.Iterable[DataType]
```

Listing 3.2: The Endpoint API

As the code generation process requires additional information, we define an `Endpoint` dataclass[1] (Listing 3.2) to contain the `EFSM` representation, along with the information passed in from the command line (`protocol`, `role`, `server`) and the custom type declarations (`types`) parsed from the protocol specification.

3.1.3 API Generation

Formally, API generation is a function of the constructed `Endpoint` instance *and* the target specified in the command line. We use a different code generation strategy for implementations running on the server (Chapter 4) versus the browser (Chapter 5), hereafter referred to as *server-side endpoints* and *browser-side endpoints* respectively. In this subsection, we explain how we perform API generation at a higher level of abstraction.

[1] A Python dataclass uses the `@dataclass` decorator to generate "boilerplate" methods, such as the constructor, based on the properties listed in the annotations.

Traditional methods of code generation involve applying the Visitor pattern on the internal representation. In the context of the MPST framework, this may involve defining a Visitor class that implements a `generate()` operation to be performed on the EFSM states, such that the `generate()` implementation specialises to the type of EFSM state, i.e. send, receive or terminal. This is not straightforward in Python, as method overloading is not supported, so the "visit" methods would need different names. More importantly, it is less straightforward to visualise the structure of the generated code, as the string interpolation aspect is likely to be interleaved with source code implementing additional logic for code generation.

For SESSIONTS, we leverage the *Jinja* [31] template engine library for code generation. We first construct templates for the TypeScript files we wish to generate, specifying placeholders for dynamic content (to be extracted from the `Endpoint` object); we then provide Jinja with the template path and the `Endpoint` object, and the template engine renders the TypeScript code by filling in the dynamic placeholders. We show an example in Listing 3.3.

efsm.ts.j2

```
export namespace Message {
{% for state in endpoint.efsm.nonterminal_states %}
  {% for action in state.actions -%}
  export type S{{ state ~ action.label }} = {
    label: Labels.S{{ state }}.{{ action.label }},
    payload: [{{ action.payloads|join(', ') }}],
  };
  {% endfor %}
  export type S{{ state }} = {% for action in state.actions -%}
  | S{{ state ~ action.label }}{% endfor %};
{% endfor %}
```

Listing 3.3: Example Jinja Template for SESSIONTS API Generation

Jinja provides lightweight syntax for injecting content and markup for simple control structure: {{ `state` }} denotes a placeholder for Jinja to render the `state` variable, and the {% %} syntax is used for conditionals and control structures (such as for loops, to dynamically render the enclosing "sub-template" by iterating over a collection). The main advantage that Jinja brings is that it decouples the "presentation" from the "content" and makes it quick to prototype and extend the generated code, *usually* without modifications to the code generator.

As we generate a different set of TypeScript artefacts depending on the specified target, we structure the different code generators using the Strategy design pattern. Each target extends the abstract base class `CodeGenerationStrategy` and implements its own `generate()` method to return a list of `(path, content)` tuples. Each tuple specifies the `content` of the generated code, and the file `path` where which to save the TypeScript code. We define a `CodeGenerator` class that is parameterised by `target`: when instantiated, it will select and perform the specialised `generate()` method based on `target`, before formatting and committing the generated code to the file system. We implement a subclass hook in `CodeGenerationStrategy` (List-

```
1  class CodeGenerationStrategy(ABC):
2
3      target_to_strategy = {}
4
5      def __init__(self):
6          super().__init__()
7
8      @classmethod
9      def __init_subclass__(cls, *, target):
10         CodeGenerationStrategy.target_to_strategy[target] = cls
11         return super().__init_subclass__()
12
13     @abstractmethod
14     def generate(self, endpoint: Endpoint):
15         pass
16
17 # Register the code generator to the
18 # command line interface using the 'node' identifier.
19 class NodeCodeGenerationStrategy(target='node'):
20     ...
21
22 class BrowserCodeGenerationStrategy(target='browser'):
23     ...
```

Listing 3.4: Implementing `CodeGenerationStrategy`

ing 3.4), such that each derived class must provide the target name to "register" with the base class (Line 10), and the base class keeps an internal mapping of the concrete strategies (Line 3); `CodeGenerator` accesses this mapping to select the appropriate strategy.

3.2 Implementation

SESSIONTS is written in Python. It offers flexible syntax, a rich standard library, and a healthy ecosystem of DOT graph parsers and template engines, making it a suitable choice for implementing our code generator. Its rich standard library also simplifies many tasks: we use the `argparse` package to generate an informative command line interface (CLI) for developers to supply the correct information (Listing 3.5) to use SESSIONTS and the `subprocess` package to invoke external tools, such as Scribble and the TypeScript Compiler.

We use Docker [37] to encapsulate our code generator and its dependencies – we found the canonical Python virtual environment solution to be insufficient, as the Scribble toolchain is a standalone Java executable with non-trivial setup procedures. The Dockerfile builds a Docker image with Scribble and the Python dependencies all pre-configured, and the provided `build.sh` script instantiates the image as a container for development. The `start.sh` script enters the Docker container development environment and mounts the local project directory onto the container as a volume to synchronise changes between the two environments.

```
root@mpst_ts:/home# python3.7 -m mpst_ts --help
usage: __main__.py [-h] [-s SERVER] [-o OUTPUT]
                   filename protocol role {browser,node}

positional arguments:
  filename        Path to Scribble protocol
  protocol        Name of protocol
  role            Role to project
  {browser,node} Code generation target

optional arguments:
  -h, --help      show this help message and exit
  -s SERVER, --server SERVER
                  Server role (only applicable for browser targets)
  -o OUTPUT, --output OUTPUT
                  Output directory for generation
```

Listing 3.5: Entry Point for SESSIONTS

3.3 Testing

The challenge for testing SESSIONTS is to verify that the generated code is valid TypeScript code. Here, we detail our methodology for *system testing* – executing SESSIONTS end-to-end and testing the generated code.

We implement a test suite on top of `unittest` APIs for verifying that SESSIONTS generates *valid TypeScript code*. This is especially useful as our templates contain both TypeScript syntax and Jinja markup, so we cannot easily make sure that each template generates valid code, let alone checking that the collection of templates generate a valid TypeScript project altogether.

```
def test_code_generation(self):
    flags = [scr, protocol, role, target]
    if svr is not None:
        flags.append('-s')
        flags.append(svr)

    rc = mpst_ts.main(flags)
    self.assertEqual(rc, 0)

    completion = subprocess.run(self.npm_test_cmd, shell=True)
    self.assertEqual(completion.returncode, 0)

    shutil.rmtree(self.output_dir)
```

Listing 3.6: Main logic in SESSIONTS system testing test case

We provide a collection of Scribble protocols under `protocols/` to generate test cases, one per protocol participant. For each test case, the test suite (Listing 3.6) will:

1. Invoke SESSIONTS to generate the TypeScript project (Line 7), expecting a zero exit code (Line 8);
2. Run the TypeScript Compiler on the generated directory (Line 10), passing the `noEmit` flag to purely perform type-checking, and expecting a zero exit code (Line 11).

As the generated TypeScript code makes assumptions about the environment in which it is used (for example, having the `ws` WebSocket package installed on server-side endpoints), we require a *sandbox environment* to type-check the generated code. The sandbox contains the minimal boilerplate required for testing – this involves having the WebSocket package installed for server-side endpoints, the React.js framework (Chapter 5) instantiated for browser-side endpoints, and corresponding `tsconfig.json` files for both targets to be picked up by the TypeScript Compiler.

For convenience, we extend the Dockerfile and `build.sh` script to set up the sandbox environments. We also make use of the optional `--output` flag exposed by the SESSIONTS CLI to redirect the generated code to the correct sandbox environment to simply the testing process.

Chapter 4

NODEMPST: Back-End Session Type Web Development

In this chapter we present NODEMPST, our session type API generation strategy for server-side endpoints implemented on the Node.js runtime [43]. We continue to use the running example of the ADDER protocol (Listing 3.1), and refer to the FSM of the `Svr` endpoint (Figure 4.1) throughout this chapter.

We discuss the challenges of implementing session types on the Node.js runtime (Section 4.1) and motivate our approach (Section 4.2). We explain our design choices for encoding the EFSM for server-side endpoints (Section 4.3), and present a session runtime designed to execute the EFSM on Node.js (Section 4.4). We conclude by analysing some alternative designs (Section 4.5) and evaluating the limitations of our approach (Section 4.6).

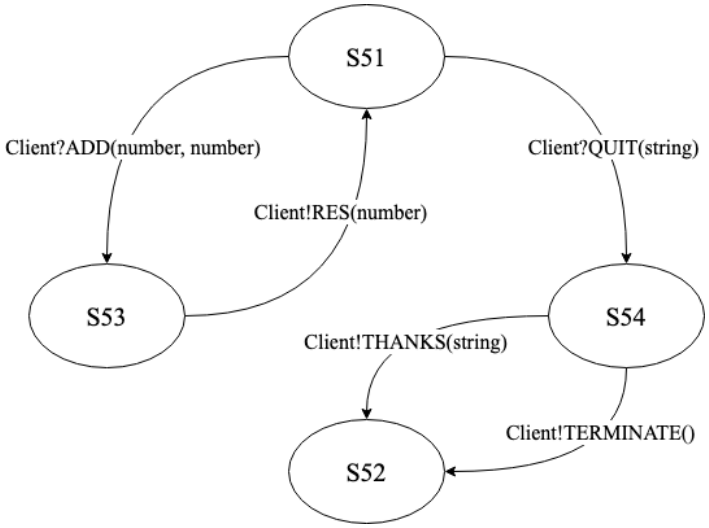


Figure 4.1: `Svr` Endpoint FSM in ADDER Protocol

4.1 Challenges

Our goal with NODEMPST is to generate session type implementations that provide as much **static** communication safety guarantees as possible, in a manner suitable for both the TypeScript language and the single-threaded event loop model in Node.js TypeScript's type system is neither affine nor linear, so we need to work around the language's limitations to enforce linear channel usage. We also aim to provide the developer with idiomatic event-driven APIs commonly found in server-side TypeScript development.

4.2 Approach

We motivate our approach from [27, 32] to generate **handler-style APIs** to be implemented by developers. This aligns with idiomatic TypeScript practices of defining *callbacks* in the application logic. We type the parameters and return values of the handlers to reflect the message types specified in the protocol. By strictly specifying handlers for send and receive actions, we do not expose send and receive APIs to the developer, contrary to [27], making it impossible for the developer to reuse channels, hence saving the need for linearity checks.

The responsibility of guaranteeing linear channel usage now falls under the runtime that executes the EFSM. As we generate this for the developer, we can provide **static** linearity guarantees by construction.

When executing NODEMPST to generate code for the `Svr` endpoint specifying the `node` target, the developer obtains the following files:

- **EFSM.ts:** TypeScript types constructed for the EFSM encoding (Section 4.3);
- **Svr.ts:** Session runtime for executing the EFSM (Section 4.4).

4.3 EFSM Encoding

We show the structure of the generated `EFSM.ts` file in Listing 4.1. Note that the formal definition of the EFSM in Section 2.2.1 contains more than just states and the state transition function, so we encode the additional information as well. Each type of information is grouped into their own *namespace*, and are collectively exported in the EFSM *module* for the developer to use.

4.3.1 Roles, Labels, Messages

We generate TypeScript constructs for these pieces of information so they can be reused throughout the generated code, and in particular, the runtime.

EFSM.ts

```
// Section 4.3.1
export namespace Roles {...};      // for roles in protocol
export namespace Labels {...};     // for message labels
export namespace Message {...};    // for message structure

// Section 4.3.2
export namespace Handler {...};    // for handler APIs

// Section 4.3.3
abstract class ISend {...};
abstract class IReceive {...};
abstract class ITerminal {...};
export namespace Implementation {...};
```

Listing 4.1: Structure of Generated EFSM Encoding for Server Endpoint

Roles The runtime needs to know the identifiers of participants involved in the session, and who to send/receive from depending on the EFSM state. We generate string enumerations, or *enums*, for each participant in the protocol, *excluding* the first person endpoint. The enum appropriately groups the collection of participants involved and scales for multiparty sessions, whilst making it simple to derive other types, e.g. a mapping from participants (indexed by the enum) to WebSockets.

Labels The runtime needs to decide which handler to invoke, based on the label of the received message. Similarly, the developer needs to provide handlers specifying their internal choice (e.g. which message label to send) and how to handle external choice (e.g. how to handle received message with particular label). For the same reason, we also generate string enums for message labels, one enum per state. Enums are compatible with switch statements, which can be used to dispatch messages to the correct handlers in the runtime based on the message label. We give an example in Listing 4.2.

```
// Inside the Labels namespace...
export enum S51 { ADD = "ADD", QUIT = "QUIT", };
export enum S53 { RES = "RES", };
export enum S54 { THANKS = "THANKS", TERMINATE = "TERMINATE", };
```

Listing 4.2: Generated Label Enums for `Svr` endpoint

Messages The handler APIs that we generate for developers need to refer to the label identifier and payload type: we refer to this as the message structure. Each message structure is expressed as an interface with properties for the label and payload. These interfaces are grouped based on the EFSM state they belong using

union types. We illustrate this in Listing 4.3. By expressing the payload type as a *tuple*[1], we easily generalise our type definition to polyadic payloads.

```
// Inside the Message namespace...
export interface S54THANKS {
    label: Labels.S54.THANKS,
    payload: [string],
};
export interface S54TERMINATE {
    label: Labels.S54.TERMINATE,
    payload: [],
};

export type S54 = | S54THANKS | S54TERMINATE;
```

Listing 4.3: Generated Message Type Definition for State 54

4.3.2 Handler APIs

Handler APIs act as the *seams* of the EFSM. A seam is "a place where one can alter the behaviour in the program without editing in that place" [15]. The developer implements handlers to define what message to send and how to process a received message, without accessing the send and receive APIs that are used to execute the EFSM. We collect these APIs under the `Handler` namespace. We introduce the generated handlers for sending and receiving states. These are non-terminal states that will involve the encoding of its successor state. The reader will notice that, in the listings below, the successor state is stated to be under the `Implementation` namespace: we explain in Section 4.3.3, but for now, it is sufficient to acknowledge that those refer to the encoding of the successor state. As a design choice, we *do not* generate handlers for terminal states, because the semantics of inactivity mean there is nothing to handle.

Send We model selections using a union type to encapsulate the possible send actions, as shown in Listing 4.4. Each send action is encoded as a tuple of the label, the payload, and the successor state encoding. We see some benefits from defining Message Types as interfaces: TypeScript supports *index type queries* to extract named property types, so `Message.S54THANKS['payload']` would resolve to `[string]`, based on the interface definition in Listing 4.3.

We generalise deterministic send actions as a trivial *selection*, as motivated from the theory (Figure 2.4), so the encoding for State 53 in the `Svr` FSM would be the union of a single tuple.

Receive We model branching using an interface to enumerate the possible branches, as shown in Listing 4.5. As with send states, we generalise deterministic receive actions as a trivial *branch*, which would be an interface with one property.

[1] In TypeScript, a tuple is an array with fixed size and known types for elements at each position.

```
// Inside the Handler namespace...
export type S54 =
  | [Labels.S54.THANKS, Message.S54THANKS['payload'],
      Implementation.S52]
  | [Labels.S54.TERMINATE, Message.S54TERMINATE['payload'],
      Implementation.S52];
```

Listing 4.4: Generated Type for Svr Send State in ADDER protocol

```
// Inside the Handler namespace...
export interface S51 {
  [Labels.S51.ADD]: (...payload: Message.S51ADD['payload']) =>
    Implementation.S53,
  [Labels.S51.QUIT]: (...payload: Message.S51QUIT['payload']) =>
    Implementation.S54,
}
```

Listing 4.5: Generated Type for Svr Receive State in ADDER protocol

The interface properties are defined by the labels of the permitted receive actions: the square-bracket notation means that the property name is derived from the value of the enclosing variable, so `[Labels.S51.ADD]` resolves to the `'ADD'` string.

The interface values are functions parameterised by the message payload, and must return the successor state encoding. We see another benefit of defining the payload in the message structure interface as a tuple: we can define the receive handler parameter using the *spread syntax*, which allows the tuple expression to be expanded into a list of function arguments. More concretely, as shown in Listing 4.6, it allows the developer to pattern match on the individual payload values (Line 2) rather than defining their function to expect a tuple and manually destructing it (Line 5), so the former is more intuitive.

```
1 // More intuitive
2 const withSpread = (x: number, y: number) => {...}
3
4 // Needs manual destructuring
5 const withoutSpread = (payload: [number, number]) => {...}
6
7 const handler1: Handler.S51 = { ADD: withSpread   , ... };  // OK
8 const handler2: Handler.S51 = { ADD: withoutSpread, ... };  // OK
```

Listing 4.6: Example Handler Signature Compatible with Spread Syntax

4.3.3 Wrapping Handlers in "Implementations"

The behaviour of the runtime is dependent on the current state, so the runtime needs a way to distinguish between all the different states – one can think of this as implementing the state transition function from the theory, which is analogue to

overloading a `next()` method for each state. TypeScript does not support method overloading, so the `next()` method would need to be defined on some base type assignable to all states, and the implementation of the `next()` method can use a switch statement to distinguish between the different states. Currently, the state is only determined by the handler to be implemented by the developer, so the switch statement and base type would have to be defined on the handler APIs.

Unfortunately, this is not practical. Handlers for send states are union types and handlers for receive states are interfaces, both of which are not supported by the `instanceof` operator.

Distinguishing Handlers using Conditional Types

We *attempt* to address this by defining an enum of state identifiers for each type of state (i.e. an enum for send states, an enum for receive states) upon which to execute the EFSM. This can solve the switch statement problem. Now, we are left with defining a mapping between the state identifier enum to the handler type. This construct would be analogue to *dependent types*, which again, is not a feature of the TypeScript type system.

We try to define type dependencies using *conditional types* in TypeScript. A conditional types is a type-level expression

```
T extends U ? X : Y;
```

which reads, *if `T` is assignable to `U`, then the type is `X`; otherwise, the type is `Y`.*

Combined with *generic constraints*[2], we can approximate the dependency between the state identifier enum and the generated handler API using something similar to Listing 4.7.

```
1  enum SendState { S53, S54 };
2  enum ReceiveState { S51 };
3  type State = SendState | ReceiveState;
4
5  type SendHandler<S extends SendState> =
6    S extends SendState.S53 ? Handler.S53 :
7    S extends SendState.S54 ? Handler.S54 : never ;
8
9  type ReceiveHandler<S extends ReceiveState> =
10   S extends ReceiveState.S51 ? Handler.S51 : never;
```

Listing 4.7: Approximating Type Dependency using *Conditional Types*

We intend to use this construct when defining the EFSM transition function for the runtime, for each type of state, so the method signature for transitioning to send states would resemble Listing 4.8

Unfortunately, this approach **does not work** for the simple fact that conditional types were not designed to be exploited in this manner. The main limitation of

[2] `<T extends U>` defines a generic type `T` and enforces that it must be a type assignable to `U`.

```
declare function transitionToSend<S extends SendState>(
    stateId: S, handler: SendHandler<S>
);
```

Listing 4.8: EFSM Transition Function using Conditional Types

conditional types is its *distributivity* when the type parameter is an union type (which is the case for enums, as `S = SendState.S53 | SendState.S54 | ...`), where

```
(T1 | T2) extends U ? X : Y
```

results in the conditional type being *distributed* among each constituent,

```
(T1 extends U ? X : Y) | (T2 extends U ? X : Y)
```

so the type expression returns to an union type,

```
X | Y
```

Returning to Listing 4.8, the type of `handler` will end up being a union type, rather than the "dependent type" construct we were hoping for.

Distinguishing Handlers using Discriminated Unions

Instead, we leverage *discriminated unions*: all members of the union type share a common property (the *discriminant*) of which they each define an unique value for, so that the TypeScript Compiler can refine the union to the specific constituent upon checking the value of the discriminant (e.g. applying a switch statement).

We illustrate this with an example below. A `Shape` is either a `Circle` or `Square`. Both define a property name `type`, which is the discriminant property, along with properties unique to their shape (i.e. radius for circles, side lengths for squares). The `area()` function uses the discriminant property (here, in a switch statement) to narrow the type of `s` and perform operations specific to that shape.

```
1  // The 'type' property is the discriminant
2  interface Circle { type: 'Circle', radius: number };
3  interface Square { type: 'Square', side: number };
4  type Shape = Circle | Square;
5
6  function area(s: Shape) {
7      switch (s.type) {
8          case 'Circle': return 3.14 * s.radius;  // s: Circle
9          case 'Square': return s.side * s.side;  // s: Square
10     }
11 }
```

For the time being, it is sufficient to understand that for each EFSM state, in addition to the API defined under the `Handler` namespace, it also has a wrapper API defined under the `Implementation` namespace (Listing 4.9), which defines the `type` discriminant property internally (via inheritance). This explains why the successor state encodings in Listings 4.4 and 4.5 were defined as such. The type `Implementation.Type` is the discriminated union, which is used by the runtime.

```
// The ‘type‘ property is the discriminant
abstract class ISend { type: 'Send' = 'Send'; ... }
abstract class IReceive { type: 'Receive' = 'Receive'; ... }
abstract class ITerminal { type: 'Terminal' = 'Terminal'; ... }

export namespace Implementation {
  export type Type = ISend | IReceive | ITerminal;

  export class S51 extends IReceive {

    // Stores the handler defined by the developer
    // as a private property
    constructor(private handler: Handler.S51) { super(); }
    ...
  }
  ...
};
```

Listing 4.9: Discriminated Unions in EFSM for Server-Side Endpoints

We discuss our runtime implementation shortly (Section 4.4), where we disclose more details regarding the role of the `Implementation` wrapper API in the runtime.

4.4 Runtime

We define the session runtime for the `Svr` endpoint of the ADDER protocol in `Svr.ts`, named after the endpoint. It exposes a **public API** with seams for the developer to pass in the WebSocket server and application logic (i.e. implementations of the handler APIs). It is developer's responsibility to construct the WebSocket server and set it up to listen for incoming connections. Internally, it keeps a *private API* for executing the EFSM, when all participants have joined the session.

```
// Exported to developer
export class Svr {
    constructor(wss: WebSocket.Server,
                initialState: Implementation.S51) { ... }
    ...
}

// Not exported to developer
class Session {
    private wss: WebSocket.Server;
    private initialState: Implementation.S51;
    private roleToSocket: RoleToSocket;
    ...
}
```

The role of the public API is to manage incoming connections and wait for all participants to join the session (Section 4.4.1), before handing off to the private API to execute the EFSM (Section 4.4.2).

4.4.1 Managing Connections

The constructor of the public API class sets up the framework for mapping incoming WebSocket connections to participants. The main challenge is to wait for all participants to connect to the server endpoint before EFSM execution begins. We address this by defining an internal protocol for managing session joining – since we generate the runtime for both server and browser endpoints, we can implement this in a way that is transparent to the developer.

```
1  const waiting: Set<Roles.Peers> = new Set([Roles.Peers.Client]);
2
3  // Mapping of roles to WebSocket connections
4  const roleToSocket: Partial<RoleToSocket> = {
5    [Roles.Peers.Client]: undefined,
6  };
7
8  // Invoked when a connection request is received
9  const onSubscribe = ({ data, target: socket }) => {
10
11   // Deserialise connection request message
12   const { connect: role } = JSON.parse(data)
13     as Message.ConnectRequest;
14
15   // Ignore if role is already taken
16   if (!waiting.has(role)) { return socket.close(); }
17
18   // Map the role in the connection request to this WebSocket
19   roleToSocket[role] = socket;
20   waiting.delete(role);
21
22   // Start executing EFSM when all roles have joined
23   if (waiting.size === 0) {
24     new Session(wss, roleToSocket as RoleToSocket, initialState);
25   }
26 };
27
28 // For every new connection, process message with `onSubscribe`
29 wss.addEventListener('connection', ws => {
30   ws.onmessage = onSubscribe;
31 });
```

Listing 4.10: Handling Connections in Server Endpoint

We show this in Listing 4.10 and walk through the main parts:

1. The server keeps track of the participants that have yet to join the session – this is initialised to the complete set of non-server endpoints at the start (Line 1).

2. Browser endpoints request to join the session by sending a *connection request* with the role identifier as payload (Line 13). We generate role enums for browser targets in the same way, so the server can correctly interpret the message. We listen to connection requests by overriding the `onmessage` event listener for every new connection (Line 30).

3. If the role is already occupied, then the server responds by closing the connection (Line 16).

4. Otherwise, the role is not occupied, so the server binds the WebSocket (which the message was received from) to the role (Line 19). This is accumulated in an interface type, mapping each role to an *optional*[3] WebSocket property – for roles who have not yet connected, we do not know the WebSocket binding, so the WebSocket for these roles are `undefined`, as initialised at the start (Line 4).

5. When the server is no longer waiting for any participants, it notifies all other roles through the bounded WebSocket connections that the session will start, and delegates EFSM execution to the private API by constructing an instance of the `Session` class (Line 24). The notification process is managed by the `Session` class.

```
Object.values(roleToSocket).forEach(socket => {
  socket.send(JSON.stringify(Message.ConnectConfirm));
});
```

6. We define the interfaces and factories for connection messages in the generated EFSM.ts file under the `Message` namespace.

```
// Inside the Message namespace...
export interface ConnectRequest { connect: Role.Peers };
export const ConnectConfirm = { connected: true };
```

4.4.2 Executing the EFSM

The `Session` class executes the EFSM. We define a transition function, `next()`, parameterised by the current state, The constructor of the `Session` class explicitly calls `next()` with the initial state implementation provided by the developer to start EFSM execution. `next()` invokes the handler defined by the developer and performs the required channel actions for non-terminal states.

- For **send** states, the handler will return the label and payload to be sent, along with the successor state implementation. The transition function should construct and send the message, and transition to the successor state.

- For **receive** states, we change the message event listener on the WebSocket to pass the incoming message to the handler. The handler will return the successor state, which the runtime can transition to.

We conceptualise this in Listing 4.11. The discriminated union lets the runtime figure out the type of the current state.

However, we still face problems with resolving types, as highlighted. Just because we know that the current state is a send state, we do not know *which* particular

[3]TypeScript provides utility types for common type transformations: `Partial<T>` constructs a type with all properties of `T` set to optional.

```
1  next(state: Implementation.Type) {
2    // Distinguish between states using discriminant property
3    switch (state.type) {
4      case 'Send': {
5        const [label, payload, succ]: ??? = state.handler;
6        this.send(???, label, payload);  // Who to send to?
7        return this.next(succ);
8      }
9      case 'Receive': {
10       // Handle incoming messages using this anonymous function
11       this.wss.onmessage = ({ data }) => {
12
13         // Which message structure to use to deserialise?
14         const { label, payload } = JSON.parse(data) as ???;
15         const succ: ??? = state.handler[label](...payload);
16         return this.next(succ);
17       }
18     }
19     case 'Terminate': { return; }
20   }
21 }
```

Listing 4.11: Conceptual EFSM Transition Function for Server-Side Endpoint

state it is, so we cannot accurately type the handler (Line 5). The same problem is amplified for the receive state: we need to know the specific receive state in order to correctly serialise the message (Line 14) and interpret the successor state (Line 15). We see another problem with handling send states: because we do not know the specific send state, we do not know which participant to send the message to (Line 6).

```
1  abstract class ISend {
2    type: 'Send' = 'Send';
3    abstract performSend(
4      next: EfsmTransitionHandler,
5      send: (role: Roles.Peers, label: string, payload: any[])
6        => void,
7    ): void;
8  };
9
10 abstract class IReceive {
11   type: 'Receive' = 'Receive';
12   abstract prepareReceive(
13     next: EfsmTransitionHandler,
14     register: (from: Roles.Peers, messageHandler: MessageHandler)
15       => void,
16   ): void;
17 };
```

Listing 4.12: Class Definitions for `Implementation` API

These all reduce to the same core problem: the runtime needs to know the specific state at compile-time[4]. We solve this through *runtime polymorphism* instead, since the specific type of `state` is known at runtime. For each type of state, we define a common API that can be invoked by the EFSM transition function. To achieve runtime polymorphism, each concrete state must provide a specific implementation: this motivates our design for defining the discriminated union using abstract classes with abstract methods (Listing 4.12).

```
1  next(state: Implementation.Type) {
2    switch (state.type) {
3      case 'Send':         // recall Listing 4.9 Line 2
4        return state.performSend(this.next, this.send);
5      case 'Receive':      // recall Listing 4.9 Line 3
6        return state.prepareReceive(this.next, this.register);
7      case 'Terminal':     // recall Listing 4.9 Line 4
8        return;
9    }
10 }
```

Listing 4.13: Final EFSM Transition Function for Server-Side Endpoint

Using this approach, we can pass the transition function and channel actions from the `Session` runtime class to the individual state `Implementation` classes: these are also generated by NODETS, so we guarantee linear usage of channel resources by construction as well. This *significantly* simplifies the design of the runtime (Listing 4.13), because the transition function no longer needs to know the type of the specific state at compile-time.

By passing the specific functions defined in the runtime to the individual EFSM states, we can visualise our runtime as a *message passing* abstraction (Figure 4.2): the runtime uses the common `performSend()` or `prepareReceive()` API to delegate to the specialised implementation, which will in turn ask the runtime to perform specialised channel actions using the parameterised methods, and finally delegate back to the runtime to transition to the specific successor state.

Remark. We need to be careful when passing *instance* methods as *function* arguments – namely, the semantics of `this` is different. In short, we have to *explicitly* `bind()` the `Session` object to the instance methods that we pass as function arguments:

```
1 // Inside the Session class constructor...
2 this.next = this.next.bind(this);
3 this.send = this.send.bind(this);
4 this.register = this.register.bind(this);
```

Not doing so will result in `this` taking a different value (either the global object or `undefined`).

We elaborate on how this mechanism handles the sending and receiving of messages in Sections 4.4.3 and 4.4.4 respectively.

[4]Whether TypeScript "compiles" or "transpiles" (or even *"transcompiles"* [16]) to JavaScript is not relevant to our work; we stick with compilation and keep our terminology consistent.

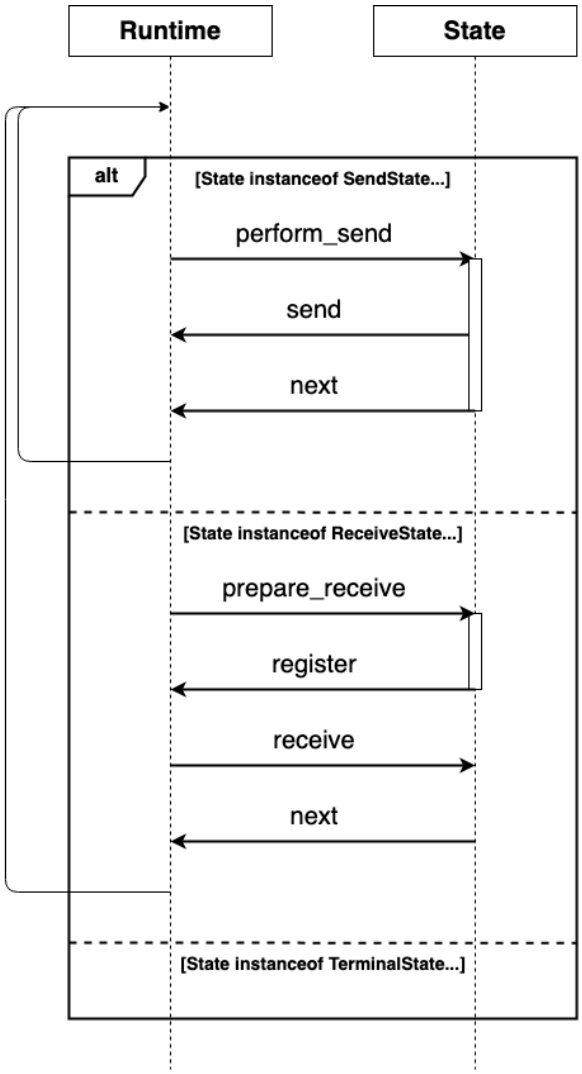


Figure 4.2: "Message Passing" Abstraction of EFSM Execution for Server Endpoints

4.4.3 Sending Messages

This is rather straightforward: we show the generated code in Listing 4.14.

We get the label, payload and successor implementation directly from the handler implemented by the developer, accurately typed by how we define the handler API in `EFSM.ts`. The developer does not need to specify which role to send to: this is a sensible design choice, as we know this from the Scribble protocol, so we do not need the developer to specify separately. As a result, we generate the code to send the message to the correct role (Line 10). We use the `send()` method passed down by the runtime to commit our communication action: the runtime will handle how to serialise the message and perform the send. We guarantee that `send()` is called **exactly once** by construction, thus channel linearity is never violated. Finally, we use the parameterised EFSM transition handler to notify the runtime which specific state to transition to.

```
export class S54 extends ISend {
  constructor(private handler: Handler.S54) { super(); }

  performSend(
    next: EfsmTransitionHandler,
    send: (role: Roles.Peers, label: string, payload: any[])
      => void
  ) {
    const [label, payload, successor] = this.handler;
    send(Roles.Peers.Client, label, payload);
    return next(successor);
  }
}
```

Listing 4.14: Generated Code for `Implementation` API for Send State

Sending through WebSockets We define message structures as interfaces, which are represented by objects. By convention in SESSIONTS, we serialise messages into *JavaScript Object Notation* (or JSON) [46] using the built-in `JSON.stringify()` method.

```
send(role: Roles.Peers, label: string, payload: any[]) {
    this.roleToSocket[role].send(JSON.stringify({
        label, payload
    });
}
```

They are decoded using the same interface schema on the receiving end using `JSON.parse()`. Whilst the method return type is the dynamic `any` type, we guarantee type safety by construction as we performed the serialisation in the first place, so we can safely interpret the deserialised content using a concrete type.

4.4.4 Receiving Messages

We need to update the message event listener on the WebSocket to use the developer's handler – specific to the *current state* – to process the message. Our approach is to keep the WebSocket message event listener untouched, but define it in a way that allows *dynamic* behaviour. We walk through the concept implemented in (Listing 4.15):

1. `Session` keeps track of the *current* message receive handler (Line 5). The `?` syntax denotes it is an *optional* type: not every state is a receive state, so there does not *have* to be an active message handler.

2. The receive handler does *not* need a specialised type (Line 1). The receive handler is defined in the `Implementation` class of the concrete receive state, so it will deserialise the message to the correct form.

3. The `register()` method (Line 19) is passed to the `Implementation` class of the concrete receive state, which will construct the message handler around

the developer's handler implementation and register it with the runtime.

4. When a message is received from the channel, we dynamically process it with the current registered handler (Line 30). We encapsulate this dynamic behaviour in an instance method and bind it as an event listener (Line 12) for the WebSocket connection of each non-server endpoint.

```
1  type MessageHandler = (message: any) => void;
2
3  class Session {
4    // Optional type, same as `MessageHandler | undefined`
5    private handler?: MessageHandler;
6
7    constructor(...) {
8      ...
9      // Process incoming messages from each WebSocket
10     // using `this.receive()`
11     Object.values(this.roleToSocket).
12       .forEach(ws => ws.onmessage = this.receive.bind(this));
13
14     // Initialise handler as undefined
15     this.handler = undefined;
16   }
17
18   // Set the handler to be used to handle the next receive event
19   register(handler: MessageHandler) { this.handler = handler; }
20
21   receive({ data }: WebSocketMessage) {
22     const handler = this.handler;
23
24     // `Unregister` the current receive handler
25     this.handler = undefined;
26
27     // `handler` has an optional type, so we first
28     // check if there is a value set, then invoke the handler.
29     // Same as `if (handler !== undefined) { handler(data); }`
30     handler?.(data);
31   }
32 }
```

Listing 4.15: Attempt to Dynamic WebSocket Message Event Listener

We also show the generated code for the `Implementation` class of the receive state in Listing 4.16 – this should appear consistent with the explanation above.

Ideally, a more succinct (and direct) representation would be

```
1 (message: any) => {
2     const decoded = JSON.parse(message) as Message.S51;
3     const successor =
4         this.handler[decoded.label](...decoded.payload);
5     return next(successor);
6 }
```

```
export class S51 extends IReceive {

  constructor(private handler: Handler.S51) { super(); }

  prepareReceive(
    next: EfsmTransitionHandler,
    register: (from: Roles.Peers, messageHandler: MessageHandler)
      => void
  ) {
    // Define dynamic WebSocket message event listener
    const messageHandler = (message: any) => {

      // Deserialise message
      const decoded = JSON.parse(message) as Message.S51;

      // Discriminate between messages using label
      switch (decoded.label) {
        case Labels.S51.ADD: {
          // Invoke handler to get successor state
          const successor =
            this.handler[decoded.label](...decoded.payload);

          // Invoke callback to advance EFSM
          return next(successor);
        }
        case Labels.S51.QUIT: {
          const successor =
            this.handler[decoded.label](...decoded.payload);
          return next(successor);
        }
      }
    }

    // Register the message handler under the
    // WebSocket bound to the 'Client' role
    register(Roles.Peers.Client, messageHandler);
  }
}
```

Listing 4.16: Generated Code for `Implementation` API for Receive State

But this expresses a type dependency between `label` and `payload` which, as discussed (Section 4.3.3), cannot be implemented. However message structures *precisely* define a discriminated union (the label acts as the discriminant to distinguish between payload types), so we handle this with a switch statement, at the cost of having the same code in each case – TypeScript does infer the correct specific type in each case statement, so code duplication here does serve a functional purpose.

Returning to Listing 4.15, note that the type of the `handler` property is *optional* – this hints at a problem: *how do we know that `this.handler` is set when a message is received?* The types imply that we *do not*, and this is indeed the case. In fact, when we consider a *multiparty* context, our approach with receive handler registration

using an optional value actually *fails* to guarantee correctness. We motivate the problem with a worked example.

Example: "Out-of-order" message receives
Recall that Node.js is a single-threaded event loop runtime, so when a message arrives, the `onmessage` event is *queued*, and current execution is *not pre-empted*.

Now consider a multiparty session specified by the global type

$$A \to S : \texttt{M1(string)}.\ B \to S : \texttt{M2(number)}.\ \texttt{end}$$

Suppose `S` is the server endpoint. We describe a possible execution flow for the protocol that breaks our implementation:

1. `S` transitions to its initial state, "receive `M1(string)` from `A`". The receive handler for `M1` is registered.
2. `M2` arrives at `S`, so the `onmessage` handler is queued. This is *perfectly plausible*: there is no causal relation between `M1` and `M2`.
3. `M1` arrives at `S`, so the `onmessage` handler is queued.
4. The `onmessage` handler for `M2` is executed. The registered handler expects `M1(string)`, but it is called with `M2(number)`, which raises a runtime type error.

This exposes a problem: the order of message *arrivals* may not correspond with the order of *receiving* messages as specified in the protocol, *so* the message may arrive before its corresponding handler is registered.

We observe that message arrivals do not have to be causally related. However, if we consider a similar *binary* example

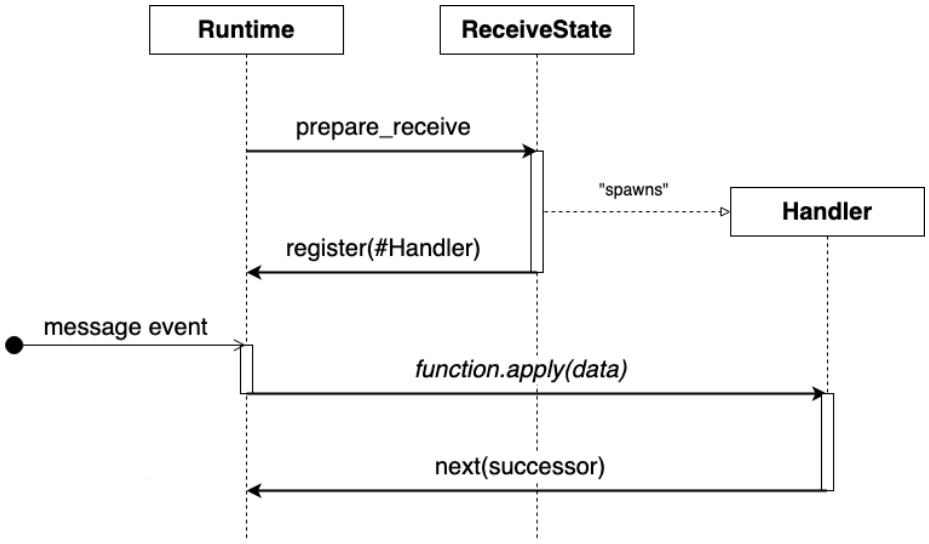
$$A \to S : \texttt{M1(string)}.\ A \to S : \texttt{M2(number)}.\ \texttt{end}$$

then `M1` *must* arrive before *M2*, since this is sent through the WebSocket connection between `A` and `S`, and FIFO guarantees are respected for each individual WebSocket connection. We visualise the possible orders of message receive events in Figure 4.3.

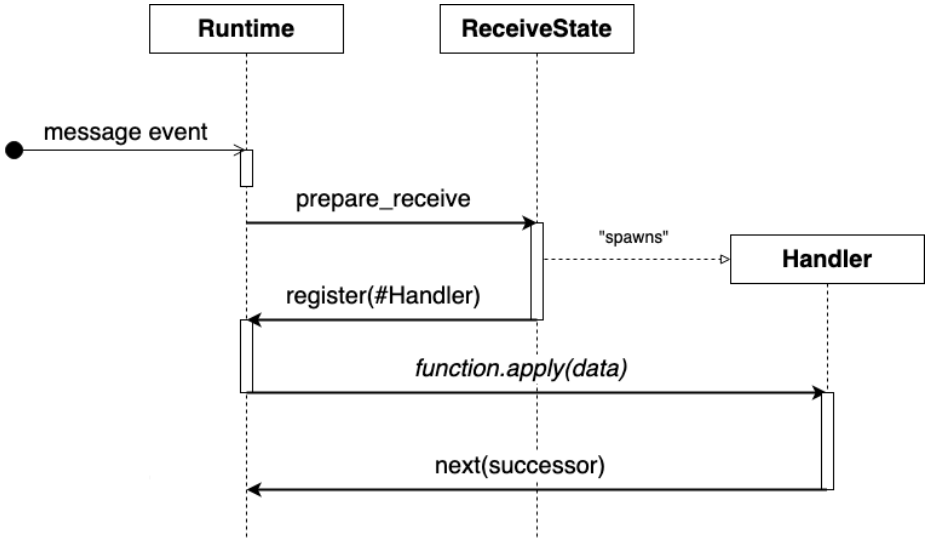
We observe that defining the handler using an optional type is insufficient. We need a similar mechanism for handling messages waiting for handlers, and we cannot assume an ordering on the arrival of messages that are not causally related.

We proceed to generalise our approach from one optional-type handler to two mappings: **(1)** a mapping from endpoint to message queues[5], and **(2)** a mapping from endpoint to handler queues. To simply put, if an incoming message is waiting for its handler, it gets enqueued in the message queue labelled by the sender of the message; when the handler is created, it pops the message off the queue and directly

[5]TypeScript arrays have built-in $O(1)$ time complexity `shift()` and `push()` operations, which can be used as a queue.



(a) Message processed before transitioning to receive state



(b) Message processed after transitioning to receive state

Figure 4.3: Possible Orderings for Receiving Message and Registering Handler

processes it; the same logic applies for a handler waiting for its message. We could have used a mapping from endpoint to optional type, but queue operations elegantly hides the mechanics of Lines 22, 25 and 30 in Listing 4.15.

We outline the changes made to the `Session` class, as shown in Listing 4.17:

1. We construct types for these two mappings (Lines 1 and 2), using a generic mapped typed defined below.

```
1 // Inside the Roles namespace...
2 export type PeersToMapped<Value> = { [Role in Peers]: Value };
```

2. Empty queues are initialised for both mappings (Lines 19 and 20).

3. Each endpoint has a different `onmessage` event listener, which will interact with the message queue and handler queue corresponding to that endpoint. We achieve this by changing the `receive()` method to be parameterised on the role instead (Line 25), so it *generates* an event listener (Line 31) tailored for receiving messages from that particular role.

4. The `register()` method now also takes the role as a parameter (Line 42) in order to check the corresponding pair of queues.

Each endpoint must have consistency between its message queue and handler queue. We get the consistency from the simple fact that Node.js is a single-threaded runtime and execution is never pre-empted, so there is no need to worry about atomic queue operations or locking data structures.

We walk through how this design addresses the problems in the previous example.

Example: Revisiting "out-of-order" message receives
Consider the same multiparty session specified by the global type

$$A \to S : \texttt{M1(string)}.B \to S : \texttt{M2(number)}.\texttt{end}$$

We show that our modified implementation addresses the problem:

1. `S` transitions to its initial state, "receive `M1(string)` from `A`". *The message queue for `A` is empty,* so the receive handler for `M1` is registered *under the handler queue for `A`.*

2. `M2` arrives at `S`, so the `onmessage` handler is queued.

3. `M1` arrives at `S`, so the `onmessage` handler is queued.

4. The `onmessage` handler for `M2` is executed. *The handler queue for `B` is empty, so `M2` is added to the message queue for `B`.*

5. The `onmessage` handler for `M1` is executed. *The handler queue for `A` is non-empty, so the handler is popped off the front of the queue and processes `M1`.*

6. `S` transitions to the successor state, "receive `M2(number)` from `B`". *The message for `B` is non-empty, so `M2` is popped off the queue and processed by the handler.*

This execution is free of communication mismatches.

4.4.5 Handling Termination

WebSocket connections should be closed when the session terminates, and both the browser endpoint and the server endpoint are capable of closing connection. As we generate code for both, we define a convention that the browser endpoint will close the WebSocket connection, so we do nothing for this state at Listing 4.13.

```
type RoleToMessageQueue = Roles.PeersToMapped<any[]>;
type RoleToHandlerQueue = Roles.PeersToMapped<MessageHandler[]>;
type RoleToSocket = Roles.PeersToMapped<WebSocket>;

class Session {
  ...
  private roleToSocket: RoleToSocket;
  private messageQueue: RoleToMessageQueue;
  private handlerQueue: RoleToHandlerQueue;

  constructor(...) {
    ...
    Object.values(Roles.Peers).forEach(role => {
      const socket = this.roleToSocket[role];
      socket.onmessage = this.receive(role).bind(this);
    });

    // Set up empty queues
    this.messageQueue = { [Roles.Peers.Client]: [], };
    this.handlerQueue = { [Roles.Peers.Client]: [], };

    this.next(initialState);    // Advance to initial state
  }

  receive(from: Roles) {

    // Return a WebSocket message event listener
    // that looks at the message/handler queue
    // specific to the `from` role

    return ({ data }) => {
      // Array.shift() can return undefined if empty
      const handler = this.handlerQueue[from].shift();
      if (handler !== undefined) {
        handler(data);
      } else {
        this.messageQueue[from].push(data);
      }
    }
  }

  register(from: Roles.Peers, handler: MessageHandler) {
    const message = this.messageQueue[from].shift();
    if (message !== undefined) {
      handler(message);
    } else {
      this.handlerQueue[from].push(data);
    }
  }
}
```

Listing 4.17: Modified `Session` class to correctly handle message receive events

4.5 Alternative Designs

The main alternative EFSM encoding would be similar to those presented in [28], which encodes each EFSM state in its own class and expose communication APIs (e.g. send and/or receive) for permitted state transitions. We would still a session runtime to provide abstractions over the WebSocket APIs and handle the "out-of-order" arrival of non-causally-related messages.

We conceptualise this alternative design (Figure 4.4) using the ONE ADDER protocol (Figure 4.4a), a simplified version of the ADDER protocol but with *curried* addition and without recursion. An implementation using the alternative API design (Figure 4.4b) could have the state transition API return a tuple of payload and continuation, and we could express the event-driven nature of a receive action using built-in async/await concurrency primitives. But ultimately, exposing channel resources without a linear type system means that the developer could violate channel linearity (i.e. by un-commenting Line 7), so we would need runtime checks as part of the session runtime.

```
1 global protocol OneAdder(role Client, role Svr) {
2     NUM1(number) from Client to Svr;
3     NUM2(number) from Client to Svr;
4     SUM(number)  from Svr to Client;
5 }
```

(a) The ONEADDER Protocol

```
1 const logic = async (init) => {
2   const [x, num2] =
3     await init.receive();
4   const [y, sum] =
5     await num2.receive();
6
7   // init.receive();
8   return sum.send(x + y);
9 }
```

(b) Alternative API Design

```
1 const logic = new S4({
2   NUM1: (x) => new S6({
3     NUM2: (y) => ([
4       Labels.S7.SUM,
5       [x + y],
6       new S5(),
7     ]),
8   }),
9 });
```

(c) Current API Design

Figure 4.4: Comparing Alternative NODEMPST Design using ONE ADDER Protocol

Looking at alternative designs for the runtime, we could adopt the approach presented in [22], which eliminates the need for our `Implementation` wrapper API by pushing the workload back to the runtime in the form of a dense switch statement enumerating all EFSM states. This may simply the developer API (by removing the need for the `Implementation` wrapper), but results in a session runtime that increases in complexity as the size of the EFSM increases.

4.6 Limitations

Providing **static** communication safety guarantees have come at a cost of providing relatively verbose APIs to the developer, which introduces a learning curve. In particular, the distinction between the `Implementation` and `Handler` API should be an internal detail, but the developer still needs to use the `Implementation` API in their logic.

Our APIs also rely on state identifiers, which also does not help code readability, as it is neither apparent nor self-documenting what `S51` refers to – the work of Hu and Yoshida [28] augment the state identifiers with the channel action and labels involved (e.g. `S51_Receive__Add_Number__Quit_String`).

Using handler-style APIs also mean that we require nested scoping to propagate values between states, which does not scale elegantly as the levels of nesting increase. For example, the following implementation of the ONE ADDER protocol would be invalid if we define the handlers for the continuations separately, because the received payload are no longer in scope (Line 3).

```
1 const first = new S4({ NUM1: (x) => secondOperand, });
2 const second = new S6({ NUM2: (y) => sum, });
3 const sum = [Labels.S7.SUM, ???, new S5()];
```

However, the developer may choose to propagate values between states through persistent storage APIs (such as storing values in a database and retrieving them for handlers of continuation states), so our generated APIs do not *strictly* enforce scoping to be the only way to propagate values in the application logic.

4.7 Summary

In this chapter, we have presented our session type API generation strategy which targets the Node.js runtime for server-side endpoints.

We motivated our approach of generating handler-style APIs that act as seams in the session runtime which executes the EFSM. By doing so, the mechanism for sending and receiving messages are not exposed to the developer, which makes channel reuse impossible by construction.

We discussed our session runtime implementation in great depth and iterated upon the design to arrive at a minimal runtime implementation that executes the EFSM using a message-passing abstraction internally as well. In particular, we motivated how to handle message receives in a way that preserves the message ordering specified in the communication protocol.

Developers that implement their Node.js endpoint application using the generated APIs enjoy protocol conformance by construction: the TypeScript type system will prevent type mismatches in the handlers defined by the developer (e.g. sending a `string` instead of a `number`, or using a message label that is not defined on the EFSM state), and the handler-style APIs themselves conceal channel resources to prevent channel reuse.

Chapter 5

REACTMPST: Front-End Session Type Web Development

In this chapter, we present REACTMPST, our session type API generation strategy for browser-side endpoints implemented using React [10]. We will refer to the FSM of the `Client` endpoint (Figure 5.1) from the ADDER protocol throughout this chapter.

We highlight the challenges of session-typed GUI programming (Section 5.1) and motivate our approach (Section 5.2) for adapting the proposal from [19] for the React.js framework, along with a brief introduction to the framework. We proceed to explain how we leverage React to encode the EFSM (Section 5.3) and implement the session runtime (Section 5.4), focusing on our novel approach for preventing channel reuse in GUI programming using React. We conclude by analysing alternative designs (Section 5.5) and evaluating the limitations of our approach (Section 5.6).

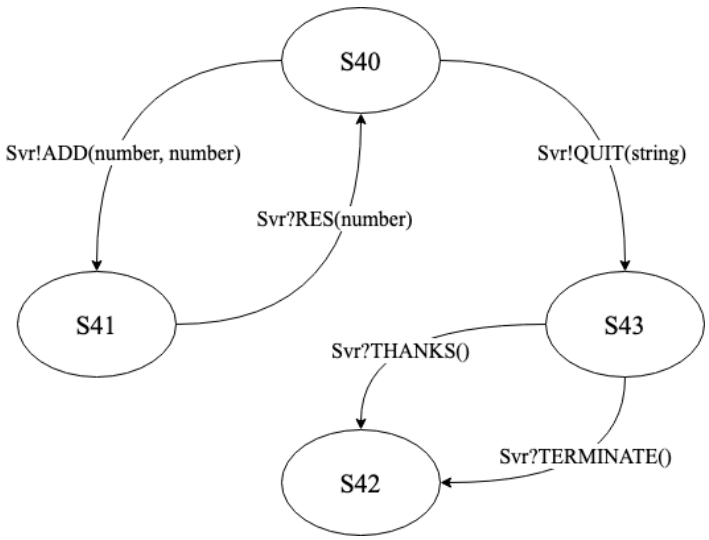


Figure 5.1: `Client` Endpoint FSM in ADDER Protocol

5.1 Challenges

Our goal with ReactMPST is to generate session type implementations for the web browser. Integrating session types into user interfaces is inherently difficult – assuming channel actions are bound to user interface events (i.e. clicking a button sends a message), how does one formalise channel linearity? How do we guarantee that, if the button triggers a send at some EFSM state, that it triggers not more than one channel action, *and* the user cannot trigger the action at another EFSM state?

We recap how existing work (as discussed in Section 2.2.2) tackle browser-side session typing. Fowler [19] introduced the concept of model types to prevent channel linearity violation in his proposal for integrating session types with GUI programming, but it uses the Links web programming language [5] which lacks compatibility with the ecosystem of JavaScript libraries that one might use in front-end development as well. The session type-safe web development framework presented by King et al. [32] generates APIs for a functional target language in PureScript, and relies on the *Concur UI* framework that constructs UIs sequentially.

As motivated in Section 1.1, we find these proposals to come at the cost of limiting developer productivity by adopting unconventional practices that may require a learning curve. Through our work, we aim to distil the key concepts from [5, 19, 32] that provide session type safety for web-based GUI programming, and implement them using mainstream front-end web development tools (specifically TypeScript and React), to provide developers with an intuitive way to implement browser-side endpoints that guarantee communication safety.

5.2 Approach

We motivate our approach from [19] by extending their work on *multiple model types* motivated by the *Model-View-Update* architecture (MVU), introduced in Section 2.2. The concept of model types express type dependencies between these components: a **model type** uniquely defines a *view function*, set of *messages* and *update function* – rather than producing a new model, the update function defines valid transitions to other model types.

We leverage the correspondence between model types and states in the EFSM: each state in the EFSM is a model type, the set of messages represent the possible channel actions available at that state, and the update function defines which successor state to transition to, given the supported channel actions at this state.

We implement model types for the EFSM on top of the **React.js** (React) framework developed by Facebook [10]. React is widely used in industry to create scalable single-page web applications, so this makes our workflow beneficial in an industrial context. The framework defines a way for data to flow between UI elements, and empowers the UI to subscribe and "react" to data changes; we introduce the framework in Section 5.2.1. We aim to implement similar behaviour with respect to the EFSM: the UI should react to EFSM state transitions, so we can **statically** ensure that the channel actions "present" on the browser at any given time are those permitted by the current EFSM state.

When executing ReactMPST to generate code for the `Client` endpoint specifying the `browser` target, the developer obtains the following groups of files:

- **S[40-43].tsx**[1]**:** Developer APIs for implementing EFSM states (Section 5.3);
  - **EFSM.ts**, **Message.ts**, **Roles.ts:** Utility types for EFSM encoding;
- **Client.tsx:** Session runtime for executing the EFSM (Section 5.4);
  - **Session.ts**, **Types.ts:** Utility types for session runtime.

5.2.1 The React Framework

We introduce the key features of the framework through illustrating a web-based counter in Listing 5.1. The browser shows a counter (initialised to zero) and an "Increment" button: when the user clicks on the "Increment" button, the count is incremented and the UI shows the updated count.

```
type Props = { count: number };
class Count extends React.Component<Props>{
  render() {
    return <strong>{this.props.count}</strong>;
  }
}

type State = { count: number };
class App extends React.Component<{}, State>{
  constructor(props: {}) {
    super(props);
    this.state = { count: 0 };
  }

  increment() { this.setState({ count: this.state.count + 1 });

  render() {
    return (<div>
      <button onClick={this.increment.bind(this)}>
        Increment
      </button>
      <Count count={this.state.count} />
    </div>);
  }
}
```

Listing 5.1: Simple Counter in React

[1]The `.tsx` file extension allows for embedding *JSX* [9] elements inside the file. JSX is a XML-like syntax extension to JavaScript for elements and components.

Components A *component* is a reusable UI element which contains its own mark-up and logic. Components implement a `render()` method which returns a React element, the smallest building blocks of a React application. This is analogue to the *view* function in the MVU architecture. React uses the *JSX* syntax extension [9] to interpolate TypeScript logic (enclosed in curly braces) within HTML mark-up: in Line 3, the `Count` component evaluates the TypeScript expression `this.props.count` and renders it in bold on the web page.

Components can render other components, which give rise to a tree of UI elements. Line 17 shows that our `Count` component is rendered by the `App` component.

Uni-directional Data Flow User-defined components derive from the abstract base class `React.Component<P, S>`, which is an abstract base class with generic type parameters `<P, S>` for *props* (short for properties) and *state* respectively.

The `App` component maintains `count` in its state (Line 12). Clicking on the increment button updates the state (Line 15), which invokes a re-render, so the UI "reacts" to state change.

Data flows from parent components down to their children, in the form of props. The `App` component passes the `count` from its state to the `Count` component (Line 22), which accesses it via `this.props`. Because `App` is re-rendered when the count is incremented, the `Count` child component will also be re-rendered with updated props.

Virtual DOM (VDOM) and Reconciliation The `render()` methods give the developer a declarative API to specify what should be rendered. React uses a *virtual DOM* abstraction, where the tree of React elements are rendered on the virtual DOM, and React internally runs a *reconciliation* algorithm to update the browser DOM accordingly using minimal operations.

For example, the `<button>` will not be re-rendered on the browser DOM on every counter increment as it does not depend on the updated state.

5.3 EFSM Encoding

We encode each EFSM state as a React component. This encoding is consistent with the semantics of model types, as outlined in Table 5.1. ReactMPST generates an abstract React component class for each EFSM state. The developer implements the generated API by extending the abstract class to define their own view function and implement any abstract methods required by the EFSM state.

Our EFSM encoding for browser-side endpoints remain consistent with those for server-side endpoints, in the sense that channel resources are abstracted away from the APIs generated for the developer, so it is impossible to trigger channel actions *more than once* at any given EFSM state, by construction.

Model Type Property	Equivalent Abstraction for EFSM
View Function	Each EFSM state is a React component with its own `render()` method.
Set of Messages	The permitted channel actions are defined either as component props (for send states) or abstract instance methods (for receive states).
Update Function	EFSM state components are rendered by a session runtime, so they notify the runtime to trigger a transition.

Table 5.1: Implementing Model Types as React Components

5.3.1 Send States

For React components implementing send states, we need to guarantee that the channel actions on the DOM precisely correspond to the permitted transitions. We require that send actions can only be invoked by user-triggered events on UI elements, such as clicking a button or typing in a textbox – this generally aligns with idiomatic front-end web development practices.

Without ReactMPST, an implementation of state S40 from Figure 5.1 may resemble the following, where the developer attaches event listeners on UI elements to trigger channel actions:

```
<button onClick={ev => Add(this.state.num1, this.state.num2)}>
    Add
</button>
<button onMouseOver={ev => Quit(this.state.message)}>
    Quit
</button>
```

If we directly provide the highlighted channel actions (i.e. via props), we cannot guarantee linear usage in the code without a linear type system.

What we want to work towards is something similar to the following, where the highlighted component encapsulates the required information: **(1)** what event to react to, and **(2)** what payload to send when the event is triggered. `<AddOnClick>` should guarantee, by construction, that clicking on any child element will send the `ADD` message with the specified payload.

```
<AddOnClick payload={[this.state.num1, this.state.num2]}>
    <button>Add</button>
</AddOnClick>;
```

This elegantly hides channel actions, but at the same time, we cannot make excessive assumptions about the event (`onClick`) and UI element (`<button>`) that the send action will be attached to – these should be left as implementation details that the developer can customise for their application.

Instead, we present a novel approach for binding channel actions to UI elements in a way that *never exposes channel resources*:

1. The abstract send state React component defines **factory properties** that allow the developer to *generate React components* for performing send actions, such that the API lets the developer specify which UI event should trigger the channel action, and the payload to send. It defines one factory property per permitted send action at that state.

2. The runtime provides a *factory method* which allows the abstract send state component to generate the aforementioned factory properties for the channel actions supported at that state.

We walk through the implementation details by working backwards: showing how the developer would use the API first, before deriving the type signatures that we need to generate for the abstract send state component inherited by the developer.

Developer API

We show the developer's implementation of a send state for the `Client` endpoint from the ADDER protocol in Listing 5.2. Component state is an implementation detail that is not related to EFSM execution, so we allow the developer to customise this through specifying an interface (Line 2) as the generic type.

```
1  // UI component keeps track of data entered by user
2  type State = { num1: number, num2: number, message: string };
3
4  class InputWindow extends S40<State> {
5    render() {
6      const { num1, num2, message } = this.state;
7
8      // Generate React component that, when clicked,
9      // sends ADD message with `num1` and `num2`.
10     const Add = this.ADD('onClick',
11       (ev: React.MouseEvent) => [num1, num2]);
12
13     // Generate React component that, when hovered,
14     // sends QUIT message with `message` string.
15     const Quit = this.QUIT('onMouseOver',
16       (ev: React.MouseEvent) => [message]);
17     return (<div>
18       // Omitting <input>s for entering numbers, message
19       <Add><button>Sum</button></Add>
20       <Quit><button>Quit</button></Quit>
21     </div>);
22   }
23 }
```

Listing 5.2: Developer Implementation for `Client` Send State in ADDER protocol

The factory API for binding send actions to UI elements appear on Lines 10 and 15. For example, Line 10 reads: *build a React component that sends the `ADD` message with `[num1, num2]` as payload, when the user clicks on it*. `ADD` is the factory API we

generate: it takes an event identifier and an event handler which must return the payload – *appropriately typed with respect to the Scribble protocol* – for sending the `ADD` message. Swapping Lines 11 and 16 will result in a compile time type error due to invalid payload types, which is the intended behaviour.

We will define the factory API implementation in a way that calls the event handler to obtain the payload, sends the payload message, and transitions to the successor state, such that the send action is never performed more than once.

Typing the Abstract Send Component

Referring back to Listing 5.2, a first attempt for typing `ADD` could be

```
(ev: string, handler: (e: any) => [number, number])
    => Constructor<React.Component>
```

However, the type of `ev` depends on the value of `event` – for example, an `onClick` event is handled by a `React.MouseEvent` handler. React defines the supported event handlers under the `React.DOMAttributes` interface (Listing 5.3). We can use an index type query to express the dependency, and type inference in conditional types[2] to extract the function argument types:

```
<K extends keyof React.DOMAttributes>
(ev: K, handler: (e: FunctionArguments<React.DOMAttributes[K]>)
        => [number, number])
    => Constructor<React.Component>)
```

Contrary to our discussions about modelling dependent types in Section 4.3.3, we can do this here because: **(1)** we are using index type queries instead of conditional types, and **(2)** we know the type of `ev` at compile time from the developer's implementation, so we are not resolving index type queries with union types.

```
interface DOMAttributes<T> {
    children?: ReactNode;
    onChange?: FormEventHandler<T>;
    onClick?: MouseEventHandler<T>;
    ...
};
```

Listing 5.3: Snippet of the `React.DOMAttributes` Interface

However, `React.DOMAttributes` also contains non-function properties – on Listing 5.3 Line 2, "children" is not a valid event. We define utility types under Types.ts to extract function properties from an interface (Listing 5.4): we first extract the *names* of function properties into an union type by mapping property names of non-function properties to `never` (Line 2) and performing an index type query (Line 3)

[2]We can `infer` type variables inside the `extends` clause of a conditional type. Specifically, `FunctionArguments<T> = T extends (...args: infer R) ? R : never`.

to obtain an union type of function property names[3] which removes all `never` constituents; then extract the properties from the interface that are indexed by the filtered names (Line 4)

```
1 type FunctionPropertyNames<T> = {
2   [K in keyof T]: T[K] extends Function | undefined ? K : never;
3 }[keyof T];
4 type FunctionProperties<T> = Pick<T, FunctionPropertyNames<T>>;
5 type DOMEvents = FunctionProperties<React.DOMAttributes<any>>;
```

Listing 5.4: Extracting Function Properties from TypeScript Interface

We construct a generic type parameterised on the payload type, which lets us define the `ADD` and `QUIT` factory property easily.

```
type EventHandler<Payload, K extends keyof DOMEvents> =
    (event: FunctionArguments<DOMEvents[K]>) => Payload;

type SendComponentFactory<Payload> = <K extends keyof DOMEvents>
    (event: K, handler: EventHandler<Payload, K>) =>
        Constructor<React.Component>;
```

The factories are instantiated by the abstract class as `protected` properties, in order to allow access by the developer's subclass implementation. Channel resources are managed by the runtime, so we need the runtime to pass a "higher-order factory" (a factory method that generates a factory) which lets the send state component generate the required component factories (i.e. `ADD` and `QUIT`) with channel actions pre-injected into the component. We discuss the runtime in Section 5.4, but for now, it is sufficient to understand that the generated send component abstract class receives the higher-order factory as a prop, and that this prop binds the channel action into the factory that it returns.

Now we are in a position to accurately type the abstract send component `S40` in Listing 5.5. We hint at the usage of the higher-order factory on Lines 16 and 19: just using Line 16 as an example, the send component asks the runtime to generate a component factory for sending a message labelled `'ADD'` with payload typed `[number, number]`, then transition to the `ReceiveState.S42` successor state.

5.3.2 Receive States

We also encode receive states as abstract React components, and leverage props to ensure channel resources are not exposed to the developer.

According to the EFSM, a receive state means the endpoint is "waiting for the message", and the actual receive event represents a transition to a successor state. This means that the developer's implementation will define what to render on the DOM when the endpoint is waiting for the message, rather than updating the DOM with the received message. Referring to Figure 5.1 again, S42 (preceding state) will

[3]The union type technically contains `never` as the names for non-function properties, but TypeScript removes `never` from union types in the same way that $A \vee \bot \equiv A$.

S40.tsx

```
// Expect to receive ‘factory‘ function passed in by runtime
type Props = { factory: SendComponentFactoryFactory };

abstract class S40<State> extends React.Component<Props, State> {

  // Allow developer implementation subclass to use factories
  protected ADD: SendComponentFactory<[number, number]>;
  protected QUIT: SendComponentFactory<[string]>;

  constructor(props: Props) {
    super(props);

    // Generate a factory that generates React components
    // which sends an [ADD] message along with two numbers,
    // then transitions to State S42.
    this.ADD = props.factory<[number, number]>(
      'ADD', ReceiveState.S42);

    this.QUIT = props.factory<[string]>(
      'QUIT', ReceiveState.S43);
  }
}
```

Listing 5.5: Generated Type for `Client` Send State in ADDER Protocol

be rendered when the endpoint is waiting for RES, but once received, S40 (successor state) will be immediately rendered. The developer will need to manage the flow of stateful application data between UI components separately – REACTMPST only generates code for managing the communication automaton.

We generate an *abstract method* for each permitted receive transition, which is used as the receive handler. This allows the developer to intercept the message and perform operations – such as updating application state – before the successor state is rendered. Specifying receive handlers as abstract methods mandate the developer to provide a concrete implementation.

We demonstrate below how the developer could implement receive state S42 – here, the developer manages stateful application data using the *sessionStorage* API [36] available on the browser.

```
class WaitScreen extends S42 {
    render() { return <h1>Waiting</h1>; }
    RES(n: number) {
        sessionStorage.setItem('result', n);
    }
}
```

Receive states in REACTMPST work in a similar way as with NODEMPST. The runtime will invoke the RES method implemented by the developer upon receiving the message through the WebSocket. In order to do so, the S42 component needs to *register* the handler with the runtime.

Recall that React supports an uni-directional data flow, so there is no direct way to pass the handler back up to the parent component. We overcome this limitation by allowing the runtime to pass a `register` function to the receive state component as a prop; once the receive state component is mounted on the DOM, it proactively registers the handlers with the runtime.

This raises questions about the type signature for the `register` function, as the receive handler identifiers and type signatures differ between receive states. We adopt the same strategy as with NODEMPST: the receive handler registered by the receive state to the runtime also performs the message parsing, so all receive states register a "general handle function" typed

```
type MessageHandler = (message: any) => State;
```

and define their own `MessageHandler` to handle state-specific message parsing and processing. We construct similar types for the state-specific labels and message structures. This is illustrated in Listing 5.6) – we use state S43 from Figure 5.1 hereafter to show how our approach scales with multiple branches.

```
1  enum Labels { THANKS = 'THANKS', TERMINATE = 'TERMINATE' };
2  interface THANKSMessage {
3      label: Labels.THANKS,
4      payload: [string],
5  };
6  interface TERMINATEMessage {
7      label: Labels.TERMINATE,
8      payload: [],
9  };
10 type Message = | THANKSMessage | TERMINATEMessage;
```

Listing 5.6: Generated Label and Message Types in REACTMPST

We generate the `handle()` method for each abstract receive state component for state-specific message parsing and processing. We show the full generated code for state S42 in Listing 5.7. Unlike NODEMPST, the `handle()` method (Line 20) returns the successor state identifier to the runtime to make the transition. We leverage the discriminated union from our generated message types (Listing 5.6) to invoke the correct receive handler (Line 22).

Again, due to language limitations, we are unable to simplify the message processing step as

```
this[parsedMessage.label](...parsedMessage.payload);
```

because this is implicitly expressing a type dependency over union types, and `parsedMessage.label` is unknown at compile-time.

5.3.3 Terminal States

To be consistent with the conventions defined for NODEMPST the browser endpoint proactively closes the connection at its terminal state. We generate an abstract React

```
// Expect to receive `register` function passed in by runtime
type Props = {
  register: (handle: ReceiveHandler) => void
};

abstract class S42<S> extends React.Component<Props, S> {

  // Developer must implement receive handlers
  // for possible messages received at this state.
  abstract THANKS(payload1: string): void;
  abstract TERMINATE(): void;

  // When component is mounted on DOM,
  // and the register function is available,
  // register the message handler with the runtime.
  componentDidMount() {
    this.props.register(this.handle.bind(this));
  }

  handle(message: any): State {
    const parsedMessage = JSON.parse(message) as Message;
    switch (parsedMessage.label) {
      case Labels.THANKS:
        this.THANKS(...parsedMessage.payload);
        return TerminalState.S41;
      case Labels.TERMINATE:
        this.TERMINATE(...parsedMessage.payload);
        return TerminalState.S41;
    }
  }
}
```

Listing 5.7: Generated Code for `Client` Receive State S42 in ADDER Protocol

component for terminal states to allow the developer to customise what to `render` upon protocol termination.

```
type Props = { terminate: () => void };
abstract class S41<State> extends React.Component<Props, State> {
    componentDidMount() { this.props.terminate(); }
}
```

When the component is mounted on the DOM, it notifies the runtime (via the received `terminate` prop) to terminate the session.

5.4 Runtime

There are many parallels between the runtime we generate for REACTMPST compared to that of NODEMPST. We focus on the details unique to React endpoints.

We define the public API for the session runtime as a React component. The developer provides implementation details via props. We show the public API for

the `Client` endpoint of the ADDER protocol below.

```
type Props = {
    endpoint: string,                 // WebSocket URL to connect to
    states: {                         // mapping of state identifier
        S40: Constructor<S40>,        // to EFSM state components
        S42: Constructor<S42>,        // implemented by developer
        S43: Constructor<S43>,
        S41: Constructor<S41>,
    },
    waiting: React.ReactNode,         // render when waiting to start
    connectFailed: React.ReactNode, // render if connection failed
};

export class Session extends React.Component<Props> { ... }
```

The developer provides the WebSocket URL for the runtime to instantiate the connection. As for the EFSM state implementations derived from the abstract classes, the developer constructs an object to map state identifiers to concrete implementations. The developer only needs to pass the *constructor function*, as the runtime will instantiate the React components separately and pass the required props based on the type of state. Additionally, the developer needs to define what to render whilst `waiting` for the session to begin, and possibly an error screen if the connection has failed. We show an example below.

```
// Developer implementations
class InputWindow extends S40 { ... };
class QuitReceived extends S41 { ... };
class WaitScreen extends S42 { ... };
class PendingQuit extends S43 { ... };

// Main application component
class MainApp extends React.Component {
  render() {
    return <div>
      <h1>Adder Client</h1>
      <Session
        endpoint='ws://localhost:8080'
        states={{
          S40: InputWindow,
          S41: QuitReceived,
          S42: WaitScreen,
          S43: PendingQuit,
        }}>
        waiting={<h1>Pending Connection</h1>}
        connectFailed={<p>Connection Failed</p>}
      />
    </div>;
  }
}
```

When the session starts, the private API takes over and executes the EFSM by rendering the React component corresponding to the current EFSM state, as well as performing channel actions that adhere to the protocol.

5.4.1 Connecting to the Session

We connect to the session by creating a new `WebSocket`. Traditionally, this would be done in the constructor, but for React components, the constructor may be invoked more than once depending on how reconciliation works. For this reason, we create the `WebSocket` in the `componentDidMount`[4] method as it is guaranteed to be only called once. In fact, under React's *Strict Mode*, constructors are *explicitly invoked twice* to prevent impure constructors [13].

This means the WebSocket must have optional type as it is strictly undefined on construction. To avoid having to deal with an optional WebSocket value in the EFSM execution (even though we know for sure that it has been instantiated when we use it), we define the public API (the React component named `Session`) to manage the (possibly `undefined`) WebSocket in its state, and the private API (the React component named after the role – in this case, `Client`) after as a separate non-exported class with the WebSocket available via props. Once the WebSocket is instantiated in the `Session` component, it renders the `Client` component, passing the non-optional WebSocket value via props.

```
type Transport = { ws: WebSocket };
class Session extends React.Component<Props, Partial<Transport>> {
  componentDidMount() {
    this.setState({ ws: new WebSocket(this.props.endpoint) });
  }

  render() {
    const { ws } =  this.state;
    return ws === undefined ? this.props.waiting
      : <Client ws={ws} {...this.props} />;
  }
}
export default Session;
```

The connection phase is managed in the same way as NODEMPST: when the `Client` component is mounted, it sends a connection request to the server and overrides the `onmessage` event handler to listen for the connection confirmation, before advancing the EFSM to the initial state to begin executing the protocol.

5.4.2 Executing the EFSM

Unlike NODEMPST, our EFSM state encoding does not define continuation states using the actual implementation. This is because state encodings are React components that need to be instantiated by the runtime. Instead, as previewed in Listings 5.5 and 5.7, we define an enum-based abstraction for specifying successor state, and this is also used for executing the EFSM. Our enum-based abstraction defines a string enum for each type of EFSM state, and collects all state identifiers in an union type.

[4]The `componentDidMount` method is part of the "commit" phase in the lifecycle of a React component. This is when the component is mounted on the browser DOM and methods in the phase are guaranteed to be called exactly once.

```
enum SendState { S40 = 'S40' };
enum ReceiveState { S42 = 'S42', S43 = 'S43' };
enum TerminalState { S41 = 'S41' };
type State = ReceiveState | SendState | TerminalState;
```

The React runtime component also defines a transition function parameterised by the current state's enum. The runtime renders the current EFSM state component with different props depending on the type of the current state.

- For **send** states, the runtime provides the higher-order factory method to allow the send state to construct its own send component factory properties for each permitted send transition.
- For **receive** states, the runtime provides the `register` function to let the developer pass back the receive handlers for each permitted receive transition, which is used to process incoming messages received via the WebSocket.
- For the **terminal** state, the runtime provides the `terminate` function to close the WebSocket connection.

Enum unions cannot be used in the same manner as discriminated unions to distinguish between `SendState` and `ReceiveState`, so we provide utility functions under EFSM.ts that define *type guards* to narrow the type of `State`.

```
function isReceiveState(state: State): state is ReceiveState {
    return Object.values(ReceiveState).includes(state)
}
```

We use these type guards in the transition function, as shown in Listing 5.8. The choice of a string enum for state identifiers allow us to use it to index into the EFSM state component mapping provided by the developer, as seen on Lines 3, 8 and 13. Note that the value of `state` can be an union type (depending on the possible values in the enum), so the type of `View` may also be union types. We can still deterministically pass props and render the component, as all states of each particular type agree on what props to receive.

We keep track of the current EFSM state in the state of the private `Client` component. This enables the UI to be re-rendered, and therefore "react", to EFSM state transitions, which is precisely the goal of ReactMPST.

5.4.3 Sending Messages

Messages are sent through the WebSocket in the same way as NodeMPST (see Section 4.4.3) – objects marked with label and payload are serialised into JSON. The process of sending is always followed by a transition to the successor state, so we define the `sendMessage()` method on the runtime to also take the successor state as a function argument, so it can make that transition immediately after.

```
1  private advance(state: State) {
2      if (isSendState(state)) {
3          const View = this.props.states[state];
4          this.setState({
5              elem: <View factory={this.buildFactory} />
6          });
7      } else if (isReceiveState(state)) {
8          const View = this.props.states[state];
9          this.setState({
10             elem: <View register={this.registerReceiveHandler} />
11         });
12     } else if (isTerminalState(state)) {
13         const View = this.props.states[state];
14         this.setState({
15             elem: <View terminate={this.terminate} />
16         });
17     }
18 }
```

Listing 5.8: EFSM Transition Function for Browser-Side Endpoint

```
private sendMessage(label: string, payload: any, next: State) {
    this.props.ws.send(JSON.stringify({ label, payload }));
    this.advance(next);
}
```

We focus on the higher-order factory that the runtime passes to the send component as a prop. The implementation shown in Listing 5.9 is less straightforward, so we will walk through the steps.

The prop allows the send component to build factory functions for sending messages with a particular `label` followed by a transition to some `successor` state (Line 1). The higher-order factory relies on *closures* to bind the channel action into the generated component. Here, `buildFactory` returns a (factory) function that builds an anonymous React component which wraps the component's children into a `<div>` with special props (Line 14). The props for this anonymous React component comprises of a new event listener bound to the UI event (`ev`) specified by the developer (such as `'onClick'`). This new event listener is defined in terms of the `handler` from the developer's implementation: **(1)** it invokes the handler to obtain the payload to send (Line 10), which is constrained by the generic type parameter to ensure it matches the protocol specification; then, **(2)** it sends the message, which triggers the EFSM to advance to the successor state. We cannot directly call `this.sendMessage` inside the generated React component because `this` refers to the anonymous class; instead, we define a partially applied version of the `sendMessage` method as a local variable (Line 4).

```
1  private buildFactory<T>(label: string, successor: State) {
2    return <K extends keyof DOMEvents>
3           (ev: K, handler: EventHandler<E, K>) => {
4      const send = (payload: T) =>
5        this.sendMessage(label, payload, successor);
6      return class extends React.Component {
7        render() {
8          const props = {
9            [ev as string]: (e: FunctionArguments<DOMEvents[K]>) => {
10             const payload = handler(e);
11             send(payload);
12           }
13         };
14         return <div {...props}>{this.props.children}</div>
15       }
16     }
17   }
18 }
```

Listing 5.9: Higher-Order Factory Function in Browser-Side Runtime

5.4.4 Receiving Messages

We take the same approach as NODEMPST – the runtime keeps track of the "active" receive handler, and the `onmessage` event listener for the WebSocket is defined to dynamically process the incoming message using the active receive handler. We also need to address the same consistency problem, since a message arrival event can be handled *before* the receive handler is registered. The receive handler is registered when the receive state component is rendered on the DOM, which occurs when `this.setState` is invoked in the transition function. `this.setState` may be *asynchronous* as React will internally batch multiple calls into a single update for performance [12], so it is possible for the WebSocket message event to be queued (and hence, processed) before the receive handler is registered.

We address this problem using a pair of queues to keep track of messages waiting for handlers, and handlers waiting for messages. Unlike NODEMPST, we do *not* need a mapping of queues because browser-side endpoints only receive from the server.

```
private messageQueue: any[];
private handlerQueue: MessageHandler[];
```

This lets us define the WebSocket `onmessage` event listener and the `register` function (which we pass as a prop to the receive state component) in a similar way compared with NODEMPST – we show this in Listing 5.10. Recall that the message handler registered by the receive state component returns the successor state, so we invoke the transition function on the result directly (Lines 3 and 9).

```
private register(handler: MessageHandler) {
    const message = this.messageQueue.shift();
    if (message !== undefined) { this.advance(handler(message)); }
    else { this.handlerQueue.push(handler); }
}

private receive({ data }: MessageEvent) {
    const handler = this.handlerQueue.shift();
    if (handler !== undefined) { this.advance(handler(data); }
    else { this.messageQueue.push(data); }
}
```

Listing 5.10: Receive Handler Registration and Message Handler in REACTMPST

5.4.5 Handling Termination

The terminal state component notifies the runtime (through the `terminate` function passed down as a prop) to close the connection. The implementation is straightforward – we simply invoke the `close()` method on the WebSocket.

```
private terminate() {
    this.props.ws.close();
}
```

5.5 Alternative Designs

For send states, a simpler approach would be to provide the developer with a `send()` function for each permitted selection. The factory approach would still apply – the runtime could pass something of the form

```
declare function buildSend<T>(label: string): (payload: T) => void
```

as a prop to the send state component. This allows the send state component to build a handler, bound to the payload type and label identifier, that performs a send action when called with the payload value. This provides more flexibility for the developer's implementation, but this clearly exposes channel resources and comes at the cost of not being able to provide guarantees on affine channel usage.

An alternative design for receive states would be to define message receive handlers on the *successor* state. Using Figure 5.1 as an example, the receive handler for `RES(number)` would be defined on state S51 instead – in fact, it wouldn't be defined as a callback, but the runtime would simply pass the received payload to S40 via props. This allows the developer to render UI changes based on received messages more easily, but comes at the cost of less robust typing: the `RES` prop would have to be an optional value, as an EFSM state can be preceded by more than one receive state, so the developer has to implement logic (i.e. checking for `undefined` props) to determine whether the predecessor state was a receive state (and if so, *which* receive state) and render accordingly.

5.6 Limitations

Our EFSM encoding provides **affine** channel usage guarantees for send states as supposed to *linear* usage guarantees. Suppose we bind a send action to a button click event – strictly speaking, even if we guarantee that the button can be clicked exactly once, we cannot guarantee that the button will ever be clicked by the user. However, existing work [19, 32] share the same limitation, and we view this as an inherent limitation in front-end development, since there is a limit to the types of assumptions we can make about how the user interacts with the browser. Redefining the definition of channel linearity for GUI programming use cases would be an interesting discussion.

Our runtime implementation also limits the developer from passing data into their EFSM state component through props. This is because the runtime takes full control of which state component to render, and by definition, what props are passed to the EFSM state components. However, we equally view this as a way to encourage developers to decouple communication state management from business logic state management, in the spirit of separating concerns. In Chapter 10, we present examples of developer implementations that use separate abstractions (e.g. the *React Context API*) for propagating stateful application data in Chapter 10.

5.7 Summary

In this chapter, we have presented our session type API generation strategy which targets the React framework for browser-side endpoints.

We motivated our approach of implementing the concept of model types introduced in [19] using a popular front-end framework in React. We define an interpretation of model types using the EFSM obtained from a Scribble protocol.

We discussed how to prevent channel reuse in GUI programming in React. We build upon the design choices implemented in NODEMPST by concealing channel resources in the session runtime and exposing seams for the developer to customise what interface to render at each EFSM state, what user interface events are allowed to trigger a send action, and how to handle messages received from the channel.

Chapter 6

Extensions

In this chapter, we extend SESSIONTS to address concerns specific to modern web programming practices. We focus on two key areas: supporting asynchronous implementations (Section 6.1), and handling session cancellation in a web-based context (Section 6.2).

6.1 Supporting Asynchronous Implementations

Asynchronous APIs are commonplace in modern web programming; typical uses range from database operations to third-party API calls. We motivate the need for supporting asynchronous operations through an authentication protocol in Section 6.1.1, demonstrate how we support asynchronous APIs through extending the generated APIs (in Section 6.1.2) and runtime (in Section 6.1.3), and highlight the limitations of our approach in Section 6.1.4.

6.1.1 Motivation

Consider a basic example of two-factor authentication (2FA) in Listing 6.1 motivated from [20]: `Client` tries to log in to `Svr`, which either authorises or denies the login attempt. If `Client` is logging in from a new device, the `Svr` sends a challenge key and waits for a response code before proceeding.

The developer's implementation on Lines 3 and 9 is likely to query a database for credential verification. Database operations are implemented using *asynchronous*, *non-blocking* APIs; it is recommended that IO-bound operations do not block the main execution, so a typical JavaScript database API may require a *callback* to propagate the return value when it is available. A server endpoint implementation of Listing 6.1 may resemble the following:

```
1 new Implementation.S40({
2     Login: (username: string, password: string) => {
3         db.lookup(username, password, (result) => {
4             // returns Implementation.S42 here
5         });
6     }
7 });
```

```
global protocol 2FA(role Svr, role Client) {
    Login(string, string) from Client to Svr;
    choice at Svr {
        Authorised() from Svr to Client;
        do Main(Svr, Client);       // Details are not important
    } or {
        Challenge(string)   from Svr to Client;
        Response(number)    from Client to Svr;
        choice at Svr {
            Authorised() from Svr to Client;
            do Main(Svr, Client);
        } or {
            AccessDenied() from Svr to Client;
        }
    } or {
        AccessDenied() from Svr to Client;
    }
}
```

Listing 6.1: The TWO FACTOR AUTHENTICATION Protocol

Callbacks are commonplace in dealing with asynchronous APIs in JavaScript [44]; we will go over other widely-used concurrency primitives shortly. Whatever the case may be, asynchronous developer implementations using our generated APIs will not type-check as the function returns immediately (hence, has return type `void`), even though the callback function has the expected return type. The developer will encounter the following type error message:

```
Type '(username: string, password: string) => void' is not
assignable to type '(payload_0: string, payload_1: string) => S42'.
```

The same scenario is also possible for browser-side implementations: when the user invokes a UI event, the browser-side logic may first consult a third-party API for additional information (which is not part of the communication protocol) before performing a send action; the *Fetch* API [34] for making API calls is asynchronous.

In order to make SESSIONTS more relevant and compatible with modern web programming practices, it is important to support asynchronous developer implementations as part of our generated APIs and runtime.

This means we should support the concurrency patterns built into TypeScript:

Callbacks Callback-based APIs are higher-order functions – they take functions as parameters and invoke them when the asynchronous operation is complete. For example, the `setTimeout(fn, delay)` function available on the browser waits for `delay` milliseconds before invoking `fn()` – or more precisely, before queuing the execution of `fn()`. As a result, the code listing below prints `'Second!'` before `'First!'`, even though `setTimeout` was called with 0ms delay.

```
setTimeout(() => console.log('First!', 0));
console.log('Second!');
```

Promises A Promise formalises the "completion" callback and its error handling construct. The developer passes a "success" callback and "error" callback, conventionally named "then" and "catch" respectively, to construct a Promise. This is analogue to the `Future` interface in Java [45]. The Promise will invoke the correct handler on completion. The benefits of Promises in our context is that TypeScript allows us to specify the type of payload that the Promise resolves to.

```
declare function promise(n: number): Promise<number>
const promise = (n: number) =>
    new Promise((resolve, reject) => {
        if (n < 10)  resolve(n));
        else reject('Number too big!');
    });

declare function handleNumber(n: number) { ... }
declare function handleError(err: string) { ... }

promise(10).then(handleNumber).catch(handleError)
```

Async/Await The `async`/`await` construct is syntactic sugar for Promises. An `async` function always returns a Promise, and marking `await` for the identifier of the return value for a function returning a Promise "unwraps" the resolved payload. A downside is that error handling reverts to the `try ... catch` construct.

```
const asyncSum = async (x: number, y: number) => {
    return (await promise(x)) + (await promise(y));
}

asyncSum(1, 2);  // returns Promise<number> that resolves to 3
```

6.1.2 API Extension

To support asynchronous implementations in the generated APIs, we use a type signature that allows the developer to use `Promises`. We construct a generic union type, `MaybePromise<T>`, to specify that the type parameter might be wrapped in a Promise, and leverage type inference in conditional types to define a generic type, `FromPromise<T>`, to extract the wrapped type.

```
type MaybePromise<T> = T | Promise<T>;
type FromPromise<T> = T extends MaybePromise<infer R> ? R : never;
```

Now we change the type signatures of our generated APIs to use `MaybePromise<T>`. For example, if the handler of a receive state generated by NODEMPST must return `Implementation.S42`, it is now permitted to return a `Promise` that resolves to a value of type `Implementation.S42`. The same logic applies to other types of EFSM states in both NODEMPST and REACTMPST.

Hence, the asynchronous `Svr` implementation for the 2FA protocol shown in Listing 6.2 now type-checks. Note that, because `async` functions return `Promises`, we also support this newer piece of syntactic sugar. Moreover, our generated APIs work

with both synchronous and asynchronous developer implementations transparently, as is the convention one would expect from modern JavaScript APIs.

```
new Implementation.S40({
  Login: async (username: string, password: string) => {
    const result = await db.lookup(username, password);
    if (result.valid)
      return new Implementation.S42('Authorised', ...);
    else if (result.unauthorised)
      return new Implementation.S42('AccessDenied', ...);
    else
      return new Implementation.S42('Challenge', ..., challenger);
  }
});

const challenger = (key: number) => {
  return new Promise((resolve, reject) => {
    keychain.check(key, ok => {
      if (ok)
        resolve(new Implementation.S44('Authorised', ...);
      else
        resolve(new Implementation.S44('AccessDenied', ...);
    });
  });
}
```

Listing 6.2: Implementing `Svr` in 2FA using Callbacks, `Promises` and `async`/`await`

6.1.3 Runtime Extension

Generally, minimal changes are required for the runtime. The key observation is that the "concrete" types for the any handlers are the same; the payload may simply be wrapped in a Promise construct. To handle potential Promises, we wrap existing logic into a delayed computation, colloquially known as a *thunk* [52]. We show an example in Figure 6.1 of how we adapt the `performSend()` function generated by NODEMPST to support optional `Promises`:

1. We wrap the original logic in Figure 6.1a into a function (Figure 6.1b Line 1) to delay the computation.
2. We distinguish asynchronous implementations from synchronous ones using the `instanceof` operator.
   - If the implementation is a `Promise`, we pass the thunk as the resolve function invoked on completion.
   - Otherwise, we directly invoke the computation.

Similar changes apply to other types of states in both NODEMPST and REACTMPST. For REACTMPST, channel-related functions in the runtime return the successor state

```
const [label, payload, succ] = this.handler;
send(label, payload);
next(succ);
```

(a) Original `performSend()` Implementation

```
const thunk = (label, payload, succ) => {
  send(label, payload);
  next(succ);
};

if (this.handler instanceof Promise) {
  this.handler.then(thunk);
} else {
  thunk(this.handler);
}
```

(b) New `performSend()` for Optional `Promises`

Figure 6.1: Adapting Send Operations in NODEMPST using a Thunk

identifier. Hence, aside from wrapping the functions to delay computation using thunks, the functions themselves need to return `MaybePromise<State>` instead to propagate the asynchrony to the execution of the EFSM – the transition occurs *after* the developer's handler finishes execution.

6.1.4 Limitations

Supporting asynchronous logic on top of our existing implementation in REACTMPST comes at a cost of losing affine linearity guarantees. If the send action handler is asynchronous, then the UI event handler returns before the runtime transitions, so the UI event that can trigger the send action is still active, which means the user can trigger a channel linearity violation. This means we need a boolean flag to keep track of channel usage, as done in [28]. We illustrate this in Listing 6.3, with the linearity checks highlighted.

Referring back to the `sendMessage()` method defined in Section 5.4.3, if we decouple the `send()` from the `advance()` function, we could advance to the successor state immediately *before* handling the asynchronous return value. This restores affine channel linearity guarantees, but results in the UI being inconsistent with the channel actions.

6.2 Error Handling

A robust error handling framework is critical in a distributed setting. It is naive to assume that endpoint implementations are entirely free from exceptions. As Fowler pointed out in [20], session type implementations that do not account for failure are of limited use in distributed programming.

```
let used = false;
...
return class extends React.Component {
    render() {
        const props = {
            [eventLabel as string]: (event) => {
                if (used) return;
                used = true;

                const result = handler(event);
                if (result instanceof Promise) {
                    result.then(send);
                } else {
                    send(result);
                }
            }
        }
    }
}
```

Listing 6.3: Generating Runtime Linearity Checks for React Send States

We motivate the need for handling a variety exceptions in modern web programming in Section 6.2.1 and demonstrate how we implement a structured error handling framework through extending the generated APIs (in Section 6.2.2) and runtime (in Section 6.2.3). We highlight the caveats of our error handling framework when dealing with asynchronous operations (in Section 6.2.4), and conclude by explaining the limitations of our framework in Section 6.2.5.

6.2.1 Motivation

Using the 2FA example from Listing 6.1, the database lookup function might rely on a cloud service, and will throw an exception if the cloud service cannot be reached. This may cause `Svr` to crash. From the perspective of session types, we need to ensure that `Client` isn't blindly waiting for the session to continue if `Svr` threw an exception.

We classify this as an example where `Svr` terminates the session because of a *logical error* in its implementation. Existing work [20, 41] on error handling for session types address this through extending the process calculus with an *explicit cancellation* operator to ensure that channel resources are closed during exception handling.

However, the context of modern web programming introduces new possibilities for session cancellation that are not addressed by existing work. Browser-side endpoints can disconnect from the session due to network connectivity issues or simply by closing the browser. Similarly, the server may also face connectivity issues. In order to deliver a robust error handling framework tailored for integrating session types in modern web programming, we must also take these variants of session cancellation into account.

We also need a novel approach for defining cancellation handlers. The work by Fowler et al. [20] integrates exception handling for session types in a *functional language*, where their design of exception handlers build upon existing work on effect handlers. Our work targets a non-functional language in TypeScript, which means any piece of code can be "effectful", so the exception handler API we generate for developers need to respect this property. Exception handlers should also be well parameterised to empower the developer to handle cancellations more effectively – for instance, if the developer knows which endpoint emitted the cancellation, appropriate clean-up operations (such as reverting database changes or application-specific logging) can be performed.

6.2.2 API Extension

We extend the generated APIs to allow developers to provide a **session cancellation handler**. The runtime will invoke this to handle *local* exceptions (e.g. exception thrown by application logic) and *global* session cancellations (e.g. other endpoint disconnecting).

The session cancellation handler is uniformly parameterised by the `role` which emitted the cancellation, and some arbitrary `reason`. For exceptions thrown by local application logic, the `reason` argument is populated with the thrown exception. Because the `try` / `catch` block is not typed, we give the `reason` parameter the dynamic type of `any` to give developers the autonomy for handling the exceptions they throw locally.

Consider a binary session protocol specifying interactions between an `ATMSvr` and `ATMClient`. We demonstrate how the developer may choose to define the session cancellation handlers in the context of this protocol.

Server (Listing 6.4): The `ATMSvr` may need to handle internal server errors (Line 3) as a result of exceptions thrown by its implementation. If the `ATMClient` disconnects prematurely, `ATMSvr` can also notify the database to revert the transaction (Line 7). Note that session cancellation handlers can be *asynchronous*, as we appreciate that any required "clean-up" operations may need to use asynchronous APIs provided by databases or other persistent storage solutions. The cancellation handler is passed into the runtime via the constructor of the public API.

```
1 new ATMSvr(wss, logic, async (role: Roles.All, reason: any) => {
2     if (role === Roles.Self) {
3         // Handle internal server error
4     } else {
5         // Handle client error:
6         // revert incomplete transactions from database
7         await db.revertTransactions(role);
8     }
9 });
```

Listing 6.4: Example Cancellation Handler for Server Endpoint

Client (Listing 6.5) Session cancellation handlers for REACTMPST are also parameterised by the `role` and `reason` for cancellation, but they must return a `ReactNode` for the runtime to render upon session cancellation, which guarantees that no channel actions will be rendered when a session is cancelled. Here, the developer renders a header depending on which role emitted the cancellation, and directly shows the reason in a paragraph. The cancellation handler is passed into the runtime via a prop defined on the public API.

```
<ATMClient
    ...
    cancellation={(role: Roles.All, reason: any) => (
        <div>
            // Customise heading to show which role cancelled
            {role === Roles.Self
                ? <h1>Internal Error</h1>
                : <h1>Server Error</h1>}
            <p>{reason}</p>     // Render reason in paragraph
        </div>
    )}
/>
```

Listing 6.5: Example Cancellation Handler for Browser Endpoint

6.2.3 Runtime Extension

The generated session runtime is extended with the capability of *emitting* session cancellation and *handling* session cancellation. We define session cancellation over WebSocket transport by invoking the `close()` method to emit cancellations, and overriding the `onclose` event listener to handle cancellations.

The WebSocket `close()` method accepts a *close code*, which is analogue to exit codes for executables. We extend this mechanism to distinguish between the different variants of session cancellation, as motivated in Section 6.2.1. As shown in Figure 6.2, we define unique close codes for different types of session cancellation under `Cancellation.ts` for both server-side (Figure 6.2a) and browser-side endpoints (Figure 6.2b) respectively. To respect the convention specified in the WebSocket Protocol [17] that the 1xxx close codes are reserved for internal use, we define our custom close codes from 4000 onwards. The enum values are named in a self-explanatory way.

The session runtime features two phases: **(1)** the *connection* phase, when the server accepts incoming connections from session participants; and **(2)** the *execution phase*, when all participants execute their EFSM. Cancellation can occur in both phases, and we outline how the runtime handles cancellation in each phase.

Cancellations during *Connection* Phase

As the global protocol has not began at this phase, any cancellations that occur here do *not* terminate the connection phase; the endpoints that have already joined just

```
export enum Receive {
  NORMAL = 1000,
  CLIENT_BROWSER_CLOSED = 1001,
  LOGICAL_ERROR = 4001,
};


export enum Emit {
  CLIENT_BROWSER_CLOSED = 4000,
  LOGICAL_ERROR = 4001,
  ROLE_OCCUPIED = 4002,
};
```

(a) Close Codes for NODEMPST

```
export enum Receive {
  NORMAL = 1000,
  SERVER_DISCONNECT = 1006,
  CLIENT_DISCONNECT = 4000,
  LOGICAL_ERROR = 4001,
  ROLE_OCCUPIED = 4002,
};

export enum Emit {
  NORMAL = 1000,
  LOGICAL_ERROR = 4001,
};
```

(b) Close Codes for REACTMPST

Figure 6.2: WebSocket Close Codes for Session Cancellation

simply continue waiting for the remaining participants to connect.

The server endpoint can emit a `ROLE_OCCUPIED` close event to the browser endpoint if the role is already taken. We know if a role is occupied if the payload of the incoming connection request message is not one of the roles that the server endpoint is waiting on.

It is possible for a browser endpoint to have already joined the session but is still in the connection phase, because the server endpoint is waiting on other participants. If this browser endpoint disconnects prematurely (e.g. the user closes the browser), the server endpoint detects this and adds the role back to the set of roles it is waiting on. To do this, we keep a mapping of socket to role identifier; this is populated when the endpoint sends the connection request message, so if the endpoint disconnects prematurely, the server knows which role to wait for again. We show the changes made to the connection management logic generated by NODEMPST in Listing 6.6.

Cancellations during *Execution* Phase

Any cancellation event that occurs during the execution of the EFSM will terminate the protocol for all roles. We see the concept of recovery to be an implementation detail unique to each application, so we do not build recovery mechanisms in our session cancellation framework. There are four possible ways for a cancellation event to be emitted during this phase.

Server endpoint experiences logical error: This is when the developer's implementation for the server endpoint throws an uncaught exception. We wrap the runtime EFSM execution methods in `try`/`catch` blocks, and handle the error with a `cancel()` method we generate for the runtime in NODEMPST. The `cancel()` method invokes the `close()` method on all browser endpoints, then executes the user-defined cancellation handler.

Browser endpoints detect this through the `onclose` WebSocket event listener. We extend the runtime generated by REACTMPST to handle this event in the corre-

```
// Inside constructor of session runtime for server-side endpoint

const socketToRole = new Map<WebSocket, Roles.Peers>();

// Invoked when socket is closed
const onClose = ({ target: socket }) => {
    // Wait for the role again
    waiting.add(socketToRole.get(socket));
}

// Invoked when server receives connection request.
const onSubscribe = (event) => {
    ...
    // Update role-WebSocket mapping.
    roleToSocket[role] = socket;
    socketToRole.set(socket, role);
    waiting.delete(role);
    ...
}
```

Listing 6.6: Modified Connection Management in Server Endpoint with Cancellation

sponding case of a switch statement defined on the close code, as shown in Listing 6.7.

```
private onClose({ code, reason }: CloseEvent) {
  switch (code) {
    case Cancellation.Receive.NORMAL: { ... }
    case Cancellation.Receive.SERVER_DISCONNECT: {
      // Server role disconnected
      this.processCancellation(Roles.Server, 'server␣disconnected')
      return;
    }
    case Cancellation.Receive.CLIENT_DISCONNECT: { ... }
    case Cancellation.Receive.LOGICAL_ERROR: { ... }
    default: { ... }
  }
}
```

Listing 6.7: WebSocket Close Event Listener in Browser Endpoints

Browser endpoints process the session cancellation by simply rendering the UI element generated by the user-supplied session cancellation handler.

```
private processCancellation(role: Roles.All, reason?: any) {
  this.setState({
    // Calls the user-specified cancellation handler,
    // which returns a React node to render on DOM
    elem: this.props.cancellation(role, reason);
  });
}
```

Server endpoint disconnects: This follows the same logic as the above, except for the fact that the server does not *emit* the cancellation through code, since it simply disconnects.

Browser endpoint experiences logical error: This is when the developer's implementation for the browser endpoint throws an uncaught exception. The runtime for the browser endpoint will catch the exception and emit the cancellation to the server using the corresponding close code.

The server endpoint will detect the cancellation event using an `onclose` WebSocket event listener similar to that defined in Listing 6.7, Then, the server propagates the cancellation to the other endpoints. We present the propagation implementation in Listing 6.8. The "payload" of the propagated cancellation is also structured as an object with the role and reason as properties, so the browser endpoint can parse and handle using the same framework presented in Listing 6.7. After propagating the cancellation to other endpoints, runtime invokes the user-defined session cancellation handler, then *restarts* the runtime to accept new connections once again. Any exceptions thrown by the developer's session cancellation handler will be ignored.

```
private propagateCancellation(cancelled: Roles.Peers, reason?: any)
{
  // Construct cancellation message as object
  // with role and reason as properties
  const message = Cancellation.toChannel(cancelled, reason);

  // Emit cancellation to other roles
  Object.entries(this.roleToSocket)
    .filter(([role, _]) => role !== cancelled)
    .forEach(([_, socket]) => {
      socket.removeAllListeners();
      socket.close(Cancellation.Emit.LOGICAL_ERROR,
        JSON.stringify(message));
    });

  try {
    // Execute user-defined cancellation handler
    const doCancel = this.cancellation(cancelled, reason);
    if (doCancel instanceof Promise) {
      doCancel.then(this.restart).catch(this.restart);
    } else {
      this.restart();
    }
  } catch {
    this.restart();
  }
}
```

Listing 6.8: Propagating Session Cancellation Events in Server Endpoint

Browser endpoint disconnects: This follows the same logic as the above, except for the fact that the browser does not *emit* the cancellation through code, since it simply disconnects.

6.2.4 Caveats with Asynchronous Operations

Implementing session cancellation whilst preserving support for asynchronous operations introduces subtle caveats.

Consider the case where the developer implements some asynchronous logic for a send state on the server endpoint. Whilst the `Promise` is being resolved, a cancellation event occurs concurrently. The session will be cancelled, but the asynchronous logic will *not* be pre-empted; it will run to completion, and proceed to try perform the send action on a WebSocket that has already been closed.

Traditionally, this will throw an error. However, the semantics of cancellation should dominate any asynchronous operations from the developer, so we need to work around the lack of pre-emption. We do this by defining a custom error handler for the WebSocket send action: in short, if a send action fails due to the session being cancelled prior, then there is nothing to worry about.

6.2.5 Limitations

The expressiveness of our error handling framework is limited by the expressiveness of the `try` / `catch` mechanism in TypeScript. Unlike languages like Java, we cannot enforce the type of exceptions being thrown, so we have to resort to typing the `reason` of the cancellation as `any` to be compatible with exception handling constructs defined by the developer.

We could change the session cancellation handler to an object with specific handlers for each of the custom WebSocket close codes we defined, but it clutters the API which increases the learning curve for the developer, and also makes it more difficult to share cancellation handling logic across different WebSocket close codes.

Part II

Implementing Arbitrary Topologies over WebSockets

Chapter 7

Motivation: Supporting Peer-to-Peer Interactions

In this chapter, we demonstrate how to support peer-to-peer interactions through routing communications between web browser endpoints over WebSocket transport, thus relaxing the server-centric topology assumption from Part I.

We motivate the problem using the TWO BUYER protocol (Section 7.1), outline our approach (Section 7.2), and discuss the challenges we need to address (Section 7.3) in order to validate the correctness of our approach.

7.1 TWO BUYER Protocol

We present the TWO BUYER protocol in Listing 7.1 – the canonical example for peer-to-peer interactions in literature [25, 26] – which describes a protocol between two buyers `A`, `B` and a Seller. We assume that the Seller runs on the server, and the buyers run on the browser.

```
1  global protocol TwoBuyer(role A, role B, role S) {
2      title(string)   from A to S;
3      quote(number)   from S to A;
4      quote(number)   from S to B;
5      split(number)   from A to B;
6      choice at B {
7          accept()    from B to A;
8          buy()       from A to S;
9      } or {
10         reject()    from B to A;
11         cancel()    from A to S;
12     }
13 }
```

Listing 7.1: The TWO BUYER Protocol

In the protocol, `A` asks the Seller for the quote of a book `title`. Seller sends the `quote` to both buyers `A` and `B`. `A` privately asks proposes a `split` with `B`. Now, `B` must

make a choice: either B can accept the split, where A confirms to buy from the Seller; otherwise, B will reject the split, which cancels the purchase.

Lines 5, 7 and 10 illustrate that the TWO BUYER protocol does not implement a server-centric communication topology, since these lines reference communication actions between the two client browsers. As a result, SESSIONTS cannot correctly generate APIs that support the specified interactions for buyers A and B.

7.2 Proposal: Server as a Router

We acknowledge that peer-to-peer browser interactions could be achieved via different transport abstractions, such as the WebRTC framework [53]. However, WebRTC has its own "protocol" for establishing connections and exchanging network information, which isn't formalised in session type theory. This setup phase generally requires a *signalling server* [35], which brings our problem full circle, as it can be implemented over WebSocket transport. Extending API generation to also include the boilerplate for initialising WebRTC connections could be an option, but would introduce complications in the session runtime and error handling (e.g. how to handle cancellations during the connection phase?). We believe that adopting WebRTC deviates from the focus of project in implementing type-safe web services.

Hence, for the purpose of our work in this part, we establish that WebSocket transport is an invariant, and proceed to build up our approach. We start by breaking down the concept of "peer-to-peer communication". Focusing on Listing 7.1 Line 5, we can say that the interaction satisfies three properties:

1. As far as A is concerned, A is sending a message that will be received by B.
2. As far as B is concerned, B is receiving a message that was sent by A.
3. The Seller is not involved in this interaction.

We proceed to relax point 3: for the Seller to be not involved in the interaction, it suffices to guarantee that the Seller cannot intervene or hijack the communication. This allows Seller to "oversee" the communication.

On this basis, we motivate our approach to empower the server endpoint to act as a *router*. That is to say, if the server endpoint receives a message of which it isn't the intended recipient for, it routes the message to the intended recipient. Based on the WebSocket transport invariant, all browser endpoints must join the session through the server. This means that the server has information about all browser endpoints, and is capable to perform the routing.

Returning to Listing 7.1 Line 5, A sends the split message through its existing WebSocket connection (with Seller on the other side), noting that B is the intended recipient; Seller receives this through the WebSocket bound to A, and routes the message to B, who receives it through its existing connection with Seller. It is trivial that the three properties[1] listed previously are still satisfied:

[1]Here, we use the relaxed version of point 3 mentioned previously.

1. *As far as* `A` *is concerned,* `A` *is sending a message that will be received by* `B` – the intended recipient is specified in the message.

2. *As far as* `B` *is concerned,* `B` *is receiving a message that was sent by* `A` – the original sender is specified in the message.

3. *Seller cannot intervene or hijack the communication* – this remains true, as Seller simply receives the message from the WebSocket bound to `A` and sends it through the WebSocket bound to `B`.

7.3 Challenges

We need to formalise our concept of routing by extending session type theory, such that, given a communication protocol implementing an arbitrary topology, we can encode it into our extended theory and prove that safety properties (e.g. well-formedness, deadlock freedom) are preserved, and that communication traces are preserved. This must be done *very carefully*, as naive definitions of routing risk over-serialising the original communication, thereby not preserving all communication traces.

Example: Naive definition of routing in global types
Consider the global type

$$A \to B : \texttt{M1}.S \to B : \texttt{M2}.\texttt{end}$$

By existing LTS semantics over global types introduced in Definition 3.2 of [7], then SB!M2 is a prefix of a valid execution trace. If we define our routing construct simply by replacing every single $A \to B : \texttt{M}$ interaction with $A \to S : \texttt{M}.S \to B : \texttt{M}$, we transform the global type into

$$A \to S : \texttt{M1}.S \to B : \texttt{M1}.S \to B : \texttt{M2}.\texttt{end}$$

our naive definition of routing has over-serialised the protocol since SB!M2 can no longer occur first because S is forced to send M1 first, which it cannot do so until receiving from A. This implies that we need to handle routing interactions differently from "normal" send/receive actions.

There is existing work [3, 4, 49] that resembles our routing proposal through *decomposing* multiparty sessions into a collection of binary sessions, but these approaches specify an endpoint to exclusively carry out routing responsibilities and act as a centralised orchestrator for the multiparty session. Our proposal recognises that, in the context of web applications, *there is a need for server-side endpoints to implement business logic*, so enforcing the server to be an exclusive orchestrator makes our work incompatible with web service protocols; we also argue that the server can still participate in the multiparty session and concurrently perform its routing duties

in a way that is transparent to non-server endpoints, and crucially, does not overserialise the permitted interactions. We introduce our extension as RoutedSessions, and proceed to prove these claims in Chapter 8.

Likewise, the implementation of routed session types in SessionTS needs to accurately reflect the theory. It also needs to be as transparent to the developer as possible to minimise the learning curve overhead demanded by complicating the generated APIs. We document our extensions to SessionTS in Chapter 9.

Chapter 8

RoutedSessions: A Theory of Routed Multiparty Session Types

In this chapter, we present RoutedSessions, an extension of the canonical multiparty session type (MPST) theory with a message routing mechanism. We build upon the variant of MPST theory introduced in [50] (which we have also discussed in Section 2.1.3), and extend it with constructs for communication actions that are routed through a participant.

RoutedSessions provide the theoretical basis for allowing developers to implement protocols with browser-to-browser interactions (such as the Two Buyer protocol introduced in Section 7.1) over a server-centric network topology which uses WebSocket transport. We introduce the extended syntax for types in Section 8.1, and give semantics of our syntax extensions in Section 8.2. Regarding the extended semantics on routed session types, we prove the soundness and completeness of the extended semantics in Section 8.3. In Section 8.4, we define a router-parameterised encoding for canonical MPST theory into RoutedSessions and prove the preservation of well-formedness and communication. We proceed to extend SessionTS to implement RoutedSessions in Chapter 9.

8.1 Syntax for Global and Local Types

We extend the syntax for global types in Section 8.1.1 and local types in Section 8.1.2 to express routed communication. The same conventions introduced in Section 2.1.3 apply here: we omit message payload types, and deterministic send and receive actions are represented by single selection and single branching constructs respectively. With these syntax extensions, we appropriately extend the definition of *projection* and *well-formedness* in Sections 8.1.3 and 8.1.4 respectively.

8.1.1 Global Types

Global types range over $G, G', G_i, \ldots$ As motivated by Section 7.3, we need to handle *routed communication* separately from direct send and receive actions. We define

the syntax of global types for RoutedSessions in Figure 8.1, and explain the new construct for expressing routed communication.

$G ::=$		Global Types
	$\mid$ `end`	Termination
	$\mid$ `t`	Type Variable
	$\mid \mu \mathtt{t}.G$	Recursive Type
	$\mid \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$	Direct Communication
	$\mid \mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \{l_i : G_i\}_{i \in I}$	Routed Communication

Figure 8.1: Global Types in RoutedSessions

The new construct $\mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \{l_i : G_i\}_{i \in I}$ reads, *"the communication interaction from* p *to* q *is routed by* s*"* . To be more verbose, p offers q a set of choices $\{l_i\}_{i \in I}$, q sends their choice to s, so p receives the selection l_i made by q via s, and the communication system proceeds with continuation G_i. We refer to s hereafter as the *router*. s ranges over the set of roles p, q, r, ..., but we use s by convention as the server endpoint is usually the router. Just as it is assumed that $\mathtt{p} \neq \mathtt{q}$ for direct communication, we also assume that $\mathtt{p} \neq \mathtt{q} \neq \mathtt{s}$ for routed communication.

We inherit the definitions of the participants function, $\mathtt{pt}(G)$, introduced in Table 2.2; we extend it in Definition 8.1 to formalise that the router participates in the routed communication.

Definition 8.1 (Participants)**.** The set of roles involved in the communication interactions specified by G, written $\mathtt{pt}(G)$, is:

$$\mathtt{pt}\left(\mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \{l_i : G_i\}_{i \in I}\right) \quad = \quad \{\mathtt{p}, \mathtt{q}, \mathtt{s}\} \cup \bigcup_{i \in I} \mathtt{pt}(G_i)$$

8.1.2 Local Types

Local types range over $T, T', T_i, \ldots$ Local types are obtained from global types via *projection*, so we also need new constructs to express routed communication from the *local* perspective of individual endpoints. We explain the extensions to projection in Section 8.1.3.

We define the syntax of local types for RoutedSessions in Figure 8.2, and walk through the new constructs below from the perspective of some arbitrary role r.

$T ::=$		Local Types
	$\mathtt{end}$	Termination
	$\mathtt{t}$	Type Variable
	$\mu \mathtt{t}.T$	Recursive Type
	$\mathtt{p} \oplus \{l_i : T_i\}_{i \in I}$	Selection
	$\mathtt{p} \,\&\, \{l_i : T_i\}_{i \in I}$	Branching
	$\mathtt{p} \hookrightarrow \mathtt{q} : \{l_i : T_i\}_{i \in I}$	Routing Communication
	$\mathtt{p}_\mathtt{q} \oplus \{l_i : T_i\}_{i \in I}$	Routed Selection
	$\mathtt{p}_\mathtt{q} \,\&\, \{l_i : T_i\}_{i \in I}$	Routed Branching

Figure 8.2: Global Types in RoutedSessions

- **Routed Communication:**

  If r has local type $\mathtt{p} \hookrightarrow \mathtt{q} : \{l_i : T_i\}_{i \in I}$, r is routing the communication between p and q. r routes the message l_i sent by p to the intended recipient q and proceeds with continuation T_i.

- **Routed Selection:**

  If r has local type $\mathtt{p}_\mathtt{q} \oplus \{l_i : T_i\}_{i \in I}$, r is making a selection from the *intended recipient* p, but the selection is sent to the intermediate router q., instead of to p directly. r sends their internal choice l_i to q and proceeds with continuation T_i.

- **Routed Branching:**

  If r has local type $\mathtt{p}_\mathtt{q} \,\&\, \{l_i : T_i\}_{i \in I}$, r is offering a choice to the *intended sender* p, but the message is received from the intermediate router q, instead of from p directly. r receives the external choice l_i from q and proceeds with continuation T_i.

As a general comment, the new local types of routed selection and routed branching should "behave" as normal selection and receive types with respect to their intended recipient and sender respectively. We keep track of the router role in the syntax to allow us to distinguish between routing communications from normal send and receive interactions.

8.1.3 Projection

In Definition 8.2, we extend the projection operator to be defined on routed communication.

Definition 8.2 (Projection)**.** The projection of G onto $\mathtt{r}$, written $G \upharpoonright \mathtt{r}$, is extended as:

$$\left(\mathtt{p} \underset{\mathtt{s}}{\rightarrow} \mathtt{q} : \{l_i : G_i\}_{i \in I}\right) \upharpoonright \mathtt{r} \quad = \quad \begin{cases} \mathtt{q}_{\mathtt{s}} \oplus \{l_i : (G_i \upharpoonright \mathtt{r})\}_{i \in I} & \text{if } \mathtt{r} = \mathtt{p} \\ \mathtt{p}_{\mathtt{s}} \,\&\, \{l_i : (G_i \upharpoonright \mathtt{r})\}_{i \in I} & \text{if } \mathtt{r} = \mathtt{q} \\ \mathtt{p} \hookrightarrow \mathtt{q} : \{l_i : (G_i \upharpoonright \mathtt{r})\}_{i \in I} & \text{if } \mathtt{r} = \mathtt{s} \\ \bigsqcap_{i \in I} G_i \upharpoonright \mathtt{r} & \text{otherwise} \end{cases}$$

As shown in the last (4th) case, the same concept of *merging* applies for routed communication: when projecting a routed communication onto a non-participant, the projections of all continuations must be "compatible", namely they can be merged using the the *merging operator*, $\sqcap$. As the merging operator is defined on local types, we extend the merging operator (introduced in Figure 2.5) to be defined on the extended syntax in Definition 8.3.

Definition 8.3 (Merging Operator)**.** The merging operator $\sqcap$ on local types is extended as:

$$\begin{aligned} \mathtt{p}_{\mathtt{q}} \oplus \{l_i : T_i\}_{i \in I} \sqcap \mathtt{p}_{\mathtt{q}} \oplus \{l_i : T_i\}_{i \in I} &= \mathtt{p}_{\mathtt{q}} \oplus \{l_i : T_i\}_{i \in I} \\ \mathtt{p}_{\mathtt{q}} \,\&\, \{l_i : T_i\}_{i \in I} \sqcap \mathtt{p}_{\mathtt{q}} \,\&\, \left\{l_j : T'_j\right\}_{j \in J} &= \mathtt{p}_{\mathtt{q}} \,\&\, \left\{l_k : T''_k\right\}_{k \in I \cup J} \\ \text{where} \quad T''_k = & \begin{cases} T_k & \text{if } k \in I \setminus J \\ T'_k & \text{if } k \in J \setminus I \\ T_k \sqcap T'_k & \text{if } k \in I \cap J \end{cases} \\ \text{otherwise} \quad & \text{undefined} \end{aligned}$$

Recall that routed selection and routed branching behave in the same way as their "non-routed" counterparts – the merging operator reflects this similarity.

8.1.4 Well-formedness

Recall that well-formedness is a predicate defined *solely* on the global type in canonical MPST theory: a global type G is well-formed if a projection is defined for all its participants.

$$\texttt{wellFormed}(G) \iff \forall \mathtt{p} \in \mathtt{pt}(G).\ G \upharpoonright \mathtt{p} \text{ exists}$$

In RoutedSessions, we need to express that a global type is well-formed *with respect to the role* $\mathtt{s}$ *acting as the router*.

We need to define the characteristics that $\mathtt{s}$ must display in G to prove that it is a router. We formalise this as an *inductively* defined relation, $G \circledast \mathtt{s}$, which reads *"*$\mathtt{s}$ *is a centroid in* G*"*. The intuition is that $\mathtt{s}$ is at the centre of all communication interactions. We define what it means to be a *centroid* in Definition 8.4.

Definition 8.4 (Centroid)**.** Let $G \circledast \mathtt{s}$ denote that $\mathtt{s}$ is the centroid of G.

$$\frac{}{\mathtt{end} \circledast \mathtt{s}}\ [\circledast\text{-End}]$$

$$\frac{}{\mathtt{t} \circledast \mathtt{s}}\ [\circledast\text{-RecVar}]$$

$$\frac{G \circledast \mathtt{s}}{\mu \mathtt{t}.G \circledast \mathtt{s}}\ [\circledast\text{-Rec}]$$

$$\frac{\mathtt{s} \in \{\mathtt{p}, \mathtt{q}\} \qquad \forall i \in I.\ G_i \circledast \mathtt{s}}{\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \circledast \mathtt{s}}\ [\circledast\text{-Comm}]$$

$$\frac{\mathtt{r} = \mathtt{s} \qquad \forall i \in I.\ G_i \circledast \mathtt{s}}{\mathtt{p} \underset{\mathtt{r}}{\to} \mathtt{q} : \{l_i : G_i\}_{i \in I} \circledast \mathtt{s}}\ [\circledast\text{-RouteComm}]$$

- [$\circledast$-Comm] : For direct communication, $\mathtt{s}$ must be a participant and a centroid of all continuations.
- [$\circledast$-RoutedComm] : For routed communication, $\mathtt{s}$ must be the router and be a centroid of all continuations.

Now we are in a position to formalise the definition of well-formedness in RoutedSessions. We present this in Definition 8.5.

Definition 8.5 (Well-formedness)**.** Let $\texttt{wellFormed}(G, \mathtt{s})$ denote that the global type G is well-formed with respect to the router $\mathtt{s}$.

$$\texttt{wellFormed}(G, \mathtt{s}) \iff (\forall \mathtt{p} \in \mathtt{pt}(G).\ G \upharpoonright \mathtt{p} \text{ exists}) \wedge G \circledast \mathtt{s}$$

We also assume that the syntax of G is *contractive*, i.e. that type variables are guarded.

8.2 Labelled Transition System (LTS) Semantics

We define the labelled transition system (LTS) semantics over global types (Section 8.2.1) and local types (Section 8.2.2) for RoutedSessions, building upon the work of Deniélou and Yoshida [7]. We show the soundness and completeness of projection with respect to the LTSs through proving the *trace equivalence* of a global type and the collection of local types projected from the global type (Section 8.3.3). We then use this result to conclude that RoutedSessions provide the same communication safety guarantees from canonical MPST theory for well-formed global types, namely deadlock freedom (Section 8.3.4).

First, we extend the label in the LTS, as shown in Figure 8.3, to distinguish the *direct* sending (and reception) of a message from the sending (and reception) of a message *via* an intermediate routing endpoint. Labels range over $l, l', \ldots$ We highlight and explain the new labels.

$l ::=$		Labels
$\mid$	$\texttt{pq}!j$	Direct Send
$\mid$	$\texttt{pq}?j$	Direct Receive
$\mid$	$\texttt{via}_{\texttt{s}}(\texttt{pq}!j)$	Routed Send
$\mid$	$\texttt{via}_{\texttt{s}}(\texttt{pq}?j)$	Routed Receive

Figure 8.3: LTS Labels in RoutedSessions

- **Routed Send:**

  The label $\texttt{via}_{\texttt{s}}(\texttt{pq}!j)$ represents the *sending* (performed by p) of a message labelled j to q through the intermediate router s.

- **Routed Receive:**

  The label $\texttt{via}_{\texttt{s}}(\texttt{pq}?j)$ represents the *reception* (initiated by q) of a message labelled j send from p through the intermediate router s.

Labels represent communication actions, so we refer to l as labels and actions interchangeably, as is the case in the literature.

Building upon [7], the *subject* of a label is the role that initiates the action. Intuitively, the actions for routed send and routed receive are still initiated by the original sender and recipient respectively; we extend the definition of subjects in Definition 8.6 to reflect this.

Definition 8.6 (Subject)**.** The subject of a LTS label, or subj(l), is defined as:

$$\mathrm{subj}(\mathtt{via}_{\mathrm{s}}(\mathrm{pq}!j)) = \mathrm{subj}(\mathrm{pq}!j) = \mathrm{p}$$

$$\mathrm{subj}(\mathtt{via}_{\mathrm{s}}(\mathrm{pq}?j)) = \mathrm{subj}(\mathrm{pq}?j) = \mathrm{q}$$

8.2.1 LTS Semantics over Global Types

The LTS semantics presented in [7] models *asynchronous communication*, which is consistent with our proposal. In order to define LTS over global types for asynchronous communication, we need to represent intermediate states (i.e. messages in transit) within the grammar of global types. Deniélou and Yoshida [7] added the construct $\mathrm{p} \rightsquigarrow \mathrm{q}.\ j : \{l_i : G_i\}_{i \in I}$ to represent that the message l_j has been sent by **p** but not yet received by **q**.

We add a similar construct $\mathrm{p} \underset{\mathrm{s}}{\rightsquigarrow} \mathrm{q}.\ j : \{l_i : G_i\}_{i \in I}$ to represent that the message l_j has been sent from **p** to the router **s** *but not yet routed to* **q**. We extend the projection operator to support this new construct.

$$\mathrm{p} \underset{\mathrm{s}}{\rightsquigarrow} \mathrm{q}.\ j : \{l_i : G_i\}_{i \in I} \upharpoonright \mathrm{r} = \begin{cases} \mathrm{p}_{\mathrm{s}}\ \&\ \{l_i : G_i \upharpoonright \mathrm{r}\}_{i \in I} & \text{if } \mathrm{r} = \mathrm{q} \\ \mathrm{p} \rightsquigarrow \mathrm{q}.\ j : \{l_i : G_i \upharpoonright \mathrm{r}\}_{i \in I} & \text{if } \mathrm{r} = \mathrm{s} \\ G_j \upharpoonright \mathrm{r} & \text{otherwise} \end{cases}$$

Because the router **s** holds Because the router holds a "global perspective" on the routed communication, we also need to represent the intermediate state (at which the message is being routed by the router) within the grammar of *local types*. We extend the grammar of local types with the construct $\mathrm{p} \rightsquigarrow \mathrm{q}.\ j : \{l_i : T_i\}_{i \in I}$ to represent that, from the local perspective of the router, the message l_j has been received from **p** *but not yet routed to* **q**.

Because routed communication is treated differently from direct send and receive actions, the notion of asynchrony differs between the two types of communication too. This definition allows us to extend the LTS semantics from [7] more naturally.

We define the LTS semantics over global types, denoted by $G \xrightarrow{l} G'$, in Figure 8.4. We highlight and explain the new rules.

- [GR6] and [GR7] are analogue to [GR1] and [GR2] for describing the emission and reception of messages in routed communication, but uses the "routed in-transit" construct instead.
- [GR8] and [GR9] are analogue to [GR4] and [GR5] in the sense that we only enforce the syntactic order of messages for the participants involved in the action l.

An important observation from [GR8] and [GR9] is that, for the router, the syntactic order of routed communication can be freely interleaved between the syn-

$$\frac{}{\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \xrightarrow{\mathtt{pq}!j} \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}}\ [\textsc{Gr}1]$$

$$\frac{}{\mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I} \xrightarrow{\mathtt{pq}?j} G_j}\ [\textsc{Gr}2]$$

$$\frac{G[\mu \mathtt{t}.G/\mathtt{t}] \xrightarrow{l} G'}{\mu \mathtt{t}.G \xrightarrow{l} G'}\ [\textsc{Gr}3]$$

$$\frac{\forall i \in I.\ G_i \xrightarrow{l} G_i' \qquad \mathrm{subj}(l) \notin \{\mathtt{p}, \mathtt{q}\}}{\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \xrightarrow{l} \mathtt{p} \to \mathtt{q} : \left\{l_i : G_i'\right\}_{i \in I}}\ [\textsc{Gr}4]$$

$$\frac{G_j \xrightarrow{l} G_j' \qquad \mathrm{subj}(l) \neq \mathtt{q} \qquad \forall i \in I \setminus \{j\}.\ G_i' = G_i}{\mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I} \xrightarrow{l} \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \left\{l_i : G_i'\right\}_{i \in I}}\ [\textsc{Gr}5]$$

$$\frac{}{\mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \{l_i : G_i\}_{i \in I} \xrightarrow{\mathtt{via}_{\mathtt{s}}(\mathtt{pq}!j)} \mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}}\ [\textsc{Gr}6]$$

$$\frac{}{\mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I} \xrightarrow{\mathtt{via}_{\mathtt{s}}(\mathtt{pq}?j)} G_j}\ [\textsc{Gr}7]$$

$$\frac{\forall i \in I.\ G_i \xrightarrow{l} G_i' \qquad \mathrm{subj}(l) \notin \{\mathtt{p}, \mathtt{q}\}}{\mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \{l_i : G_i\}_{i \in I} \xrightarrow{l} \mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \left\{l_i : G_i'\right\}_{i \in I}}\ [\textsc{Gr}8]$$

$$\frac{G_j \xrightarrow{l} G_j' \qquad \mathrm{subj}(l) \neq \mathtt{q} \qquad \forall i \in I \setminus \{j\}.\ G_i' = G_i}{\mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I} \xrightarrow{l} \mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{q}.\ j : \left\{l_i : G_i'\right\}_{i \in I}}\ [\textsc{Gr}9]$$

Figure 8.4: LTS Semantics over Global Types in RoutedSessions

tactic order of direct communication. This is a crucial result for proving that the router does not over-serialise communication – we show this in Section 8.4.3.

8.2.2 LTS Semantics over Local Types

We define the LTS semantics over local types, denoted by $T \xrightarrow{l} T'$, in Figure 8.5. We highlight and explain the new rules.

$$\frac{}{\mathtt{q} \oplus \{l_i : T_i\}_{i \in I} \xrightarrow{\mathtt{pq}!j} T_j} \text{[LR1]}$$

$$\frac{}{\mathtt{q} \,\&\, \{l_i : T_i\}_{i \in I} \xrightarrow{\mathtt{qp}?j} T_j} \text{[LR2]}$$

$$\frac{T[\mu \mathtt{t}.T/\mathtt{t}] \xrightarrow{l} T'}{\mu \mathtt{t}.T \xrightarrow{l} T'} \text{[LR3]}$$

$$\frac{}{\mathtt{q_s} \oplus \{l_i : T_i\}_{i \in I} \xrightarrow{\mathtt{via_s}(\mathtt{pq}!j)} T_j} \text{[LR4]}$$

$$\frac{}{\mathtt{q_s} \,\&\, \{l_i : T_i\}_{i \in I} \xrightarrow{\mathtt{via_s}(\mathtt{qp}?j)} T_j} \text{[LR5]}$$

$$\frac{}{\mathtt{p} \hookrightarrow \mathtt{q} : \{l_i : T_i\}_{i \in I} \xrightarrow{\mathtt{via_s}(\mathtt{pq}!j)} \mathtt{p} \,\mathrm{q}.\, j : \{l_i : T_i\}_{i \in I}} \text{[LR6]}$$

$$\frac{}{\mathtt{p} \,\mathrm{q}.\, j : \{l_i : T_i\}_{i \in I} \xrightarrow{\mathtt{via_s}(\mathtt{pq}?j)} T_j} \text{[LR7]}$$

$$\frac{\forall i \in I.\ T_i \xrightarrow{l} T_i' \qquad \mathrm{subj}(l) \notin \{\mathtt{p}, \mathtt{q}\}}{\mathtt{p} \hookrightarrow \mathtt{q} : \{l_i : T_i\}_{i \in I} \xrightarrow{l} \mathtt{p} \hookrightarrow \mathtt{q} : \left\{l_i : T_i'\right\}_{i \in I}} \text{[LR8]}$$

$$\frac{T_j \xrightarrow{l} T_j' \qquad \mathrm{subj}(l) \neq \mathtt{q} \qquad \forall i \in I \setminus \{j\}.\ T_i' = T_i}{\mathtt{p} \,\mathrm{q}.\, j : \{l_i : T_i\}_{i \in I} \xrightarrow{l} \mathtt{p} \,\mathrm{q}.\, j : \left\{l_i : T_i'\right\}_{i \in I}} \text{[LR9]}$$

$$\frac{l = \mathtt{via_s}(\cdot) \qquad \mathrm{subj}(l) \neq \mathtt{q} \qquad \forall i \in I.\ T_i \xrightarrow{l} T_i'}{\mathtt{q} \oplus \{l_i : T_i\}_{i \in I} \xrightarrow{l} \mathtt{q} \oplus \left\{l_i : T_i'\right\}_{i \in I}} \text{[LR10]}$$

$$\frac{l = \mathtt{via_s}(\cdot) \qquad \mathrm{subj}(l) \neq \mathtt{q} \qquad \forall i \in I.\ T_i \xrightarrow{l} T_i'}{\mathtt{q} \,\&\, \{l_i : T_i\}_{i \in I} \xrightarrow{l} \mathtt{q} \,\&\, \left\{l_i : T_i'\right\}_{i \in I}} \text{[LR11]}$$

Figure 8.5: LTS over Local Types in RoutedSessions

We walk through rules [LR4] and [LR5] from the perspective of role p.

- [LR4] and [LR5] are analogue to [LR1] and [LR2] for sending and

receiving messages respectively. The exception is that the new rules pattern-match on the router role s on the local type and the routed label.

We walk through rules [LR6] , [LR7] , [LR10] and [LR11] from the perspective of role s.

- [LR6] and [LR7] are analogue to [GR1] and [GR2] . Intuitively, the router s holds a "global" perspective on the interaction between p and q, which explains the correspondence with the LTS semantics over global types.
- [LR10] and [LR11] allow the router to perform routing actions before handling their own direct communication. The syntax $l = \texttt{via}_{\mathsf{s}}(\cdot)$ means that the label l is "of the form" of a routing action, i.e. there exists some p, q, j such that $l = \texttt{via}_{\mathsf{s}}(\mathsf{pq}!j)$ or $l = \texttt{via}_{\mathsf{s}}(\mathsf{pq}?j)$. The constraint of subj(l) $\neq$ q prevents the violation of the syntactic order of messages sent and received by q.

  Curious readers can consider the examples $G_1 \upharpoonright \mathsf{s}$ and $G_2 \upharpoonright \mathsf{s}$ to see why this constraint is needed.

$$G_1 = \mathsf{s} \to \mathsf{r} : \texttt{M1} \,.\, \mathsf{r} \underset{\mathsf{s}}{\to} \mathsf{q} : \texttt{M2} \,.\, \texttt{end}$$
$$G_1 \upharpoonright \mathsf{s} = \mathsf{r} \oplus \texttt{M1}.\ \mathsf{r} \hookrightarrow \mathsf{q} : \texttt{M2} \,.\, \texttt{end}$$

$$G_2 = \mathsf{s} \to \mathsf{r} : \texttt{M1} \,.\, \mathsf{p} \underset{\mathsf{s}}{\to} \mathsf{r} : \texttt{M2} \,.\, \texttt{end}$$
$$G_2 \upharpoonright \mathsf{s} = \mathsf{r} \oplus \texttt{M1}.\ \mathsf{p} \hookrightarrow \mathsf{r} : \texttt{M2} \,.\, \texttt{end}$$

As for the remaining rules, [LR8] and [LR9] are analogue to [GR4] and [GR5] because the router holds a "global" perspective on the communication, so transitions that do not violate the syntactic order of messages between roles p and q are allowed.

8.3 LTS Soundness and Completeness with respect to Projection

We work towards proving the soundness and completeness of our LTS semantics with respect to projection. Our approach is motivated from [7]:

1. We first extend the LTS semantics to a collection of local types (hereafter referred to as a *configuration* to be consistent with the literature) in Section 8.3.1;
2. Then, we extend the definition of projection to obtain the configuration of a global type (hereafter referred to as the *projected configuration*) in Section 8.3.2;
3. Finally, we prove the trace equivalence between the global type and its projected configuration in Section 8.3.3.

We use the trace equivalence result to prove deadlock freedom in Section 8.3.4.

8.3.1 LTS Semantics over Configurations

Let $\mathcal{P}$ denote the set of participants in the communication automaton. Also let T_p denote the local type of a participant $\mathrm{p} \in \mathcal{P}$.

A *configuration* describes the state of the communication automaton with respect to each participant $\mathrm{p} \in \mathcal{P}$. By definition of our LTS semantics, this includes *intermediate* states, so a configuration would also need to express the state of messages in transit.

We inherit the definition from [7], restated in Definition 8.7.

Definition 8.7 (Configuration)**.** A configuration $s = (\vec{T};\ \vec{w})$ of a system of local types $\{T_\mathrm{p}\}_{\mathrm{p}\in\mathcal{P}}$ is defined as a pair of:

- $\vec{T} = (T_\mathrm{p})_{\mathrm{p}\in\mathcal{P}}$ is the collection of local types. T_p describes the communication structure from the local perspective of participant $\mathrm{p} \in \mathcal{P}$.
- $\vec{w} = (w_{\mathrm{pq}})_{\mathrm{p}\neq\mathrm{q}\in\mathcal{P}}$ is the collection of *unbounded buffers*. The unbounded buffer w_{pq} represents a (FIFO) queue of messages sent by p but not yet received by q.

Remark. The *subtyping* relation defined on local types (see Section 2.1.2) can be extended to configurations:

$$\frac{\vec{w} = \vec{w}' \qquad \forall \mathrm{p} \in \mathcal{P}.\ T_\mathrm{p} \prec T'_\mathrm{p}}{(\vec{T};\vec{w}) \prec (\vec{T}';\vec{w}')}$$

We proceed to define the LTS over configurations in Definition 8.8, highlighting the extensions required for RoutedSessions.

Definition 8.8 (LTS Semantics over Configurations)**.** The LTS semantics over configurations is defined by the relation $s_T \xrightarrow{l} s'_T$.

Let $s_T = (\vec{T};\ \vec{w})$ and $s'_T = (\vec{T}';\ \vec{w}')$. We define the specific transitions on $\vec{T}$ and $\vec{w}$ by case analysis on the label l.

- $l = \mathrm{pq}!j$

  Then $T_\mathrm{p} \xrightarrow{l} T'_\mathrm{p}$ because p initiates the action, so $T'_{\mathrm{p}'} = T_{\mathrm{p}'}$ for all $\mathrm{p}' \neq \mathrm{p}$.

  The message j is in transit from p to q, so $w'_{\mathrm{pq}} = w_{\mathrm{pq}} \cdot j$ (j is appended to the queue of in-transit messages sent from p to q), and unrelated buffers $w'_{\mathrm{p}'\mathrm{q}'} = w_{\mathrm{pq}}$ are untouched for all $\mathrm{p}'\mathrm{q}' \neq \mathrm{pq}$.

- $l = \mathrm{pq}?j$

  Then $T_\mathrm{q} \xrightarrow{l} T'_\mathrm{q}$ because q initiates the action, so $T'_{\mathrm{p}'} = T_{\mathrm{p}'}$ for all $\mathrm{p}' \neq \mathrm{q}$.

  The message j is no longer in transit from p to q as it is received by q, so $w_{\mathrm{pq}} = j \cdot w'_{\mathrm{pq}}$ (j is removed from the front of the queue of in-transit messages sent from p to q), and unrelated buffers $w'_{\mathrm{p}'\mathrm{q}'} = w_{\mathrm{pq}}$ are untouched for all $\mathrm{p}'\mathrm{q}' \neq \mathrm{pq}$.

- $l = \mathtt{via}_{\mathtt{s}}(\mathtt{pq}!j)$

  Then $T_{\mathtt{p}} \xrightarrow{l} T'_{\mathtt{p}}$ because $\mathtt{p}$ initiates the action. Because the send action is routed, we also need $T_{\mathtt{s}} \xrightarrow{l} T'_{\mathtt{s}}$. This means $T'_{\mathtt{p}'} = T_{\mathtt{p}'}$ for all $\mathtt{p}' \notin \{\mathtt{p}, \mathtt{s}\}$.
  The message j is in transit from $\mathtt{p}$ to $\mathtt{q}$, so $w'_{\mathtt{pq}} = w_{\mathtt{pq}} \cdot j$ and unrelated buffers $w'_{\mathtt{p}'\mathtt{q}'} = w_{\mathtt{pq}}$ are untouched for all $\mathtt{p}'\mathtt{q}' \neq \mathtt{pq}$.

- $l = \mathtt{via}_{\mathtt{s}}(\mathtt{pq}?j)$

  Then $T_{\mathtt{q}} \xrightarrow{l} T'_{\mathtt{q}}$ because $\mathtt{q}$ initiates the action. Because the receive action is routed, we also need $T_{\mathtt{s}} \xrightarrow{l} T'_{\mathtt{s}}$. This means $T'_{\mathtt{p}'} = T_{\mathtt{p}'}$ for all $\mathtt{p}' \notin \{\mathtt{q}, \mathtt{s}\}$.
  The message j is no longer in transit from $\mathtt{p}$ to $\mathtt{q}$ as it is received by $\mathtt{q}$, so $w_{\mathtt{pq}} = j \cdot w'_{\mathtt{pq}}$, and unrelated buffers $w'_{\mathtt{p}'\mathtt{q}'} = w_{\mathtt{pq}}$ are untouched for all $\mathtt{p}'\mathtt{q}' \neq \mathtt{pq}$.

Routed actions are carried out by the router, so it makes sense for the local type of the router to also makes a step. The semantics of the message buffers for routed actions are the same as their non-routed counterparts; the only difference is that these message buffers are "managed" by the router, but this is a change of interpretation which isn't reflected in the theory.

8.3.2 Extending Projection for Configurations

When considering the grammar of global types G extended to include intermediate states, we can obtain the *projected configuration* from a global type G with participants $\mathcal{P}$:

$$\langle G \rangle = \left(\{ G \upharpoonright \mathtt{p} \}_{\mathtt{p} \in \mathcal{P}} \; ; \; \langle G \rangle_{\{\epsilon\}_{\mathtt{qq}' \in \mathcal{P}}} \right)$$

The collection of local types is obtained by projecting G onto each participant $\mathtt{p} \in \mathcal{P}$. The contents of the buffers is defined as $\langle G \rangle_{\{w_{\mathtt{qq}'}\}_{\mathtt{qq}' \in \mathcal{P}}}$. We inherit the definitions presented in [7], and introduce additional rules in Figure 8.6.

$$\begin{aligned}
\left\langle \mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{p}'.\ j : \{l_i : G_i\}_{i \in I} \right\rangle_{\{w_{\mathtt{qq}'}\}_{\mathtt{qq}' \in \mathcal{P}}} &= \left\langle G_j \right\rangle_{\{w_{\mathtt{qq}'}\}_{\mathtt{qq}' \in \mathcal{P}}[w_{\mathtt{pp}'} \mapsto w_{\mathtt{pp}'} \cdot j]} \\
\left\langle \mathtt{p} \underset{\mathtt{s}}{\rightarrow} \mathtt{p}' : \{l_i : G_i\}_{i \in I} \right\rangle_{\{w_{\mathtt{qq}'}\}_{\mathtt{qq}' \in \mathcal{P}}} &= \langle G_i \rangle_{\{w_{\mathtt{qq}'}\}_{\mathtt{qq}' \in \mathcal{P}}} \text{ for any } i \in I \\
\text{since} \qquad & \forall i, j \in I.\ \langle G_i \rangle_{\{w_{\mathtt{qq}'}\}_{\mathtt{qq}' \in \mathcal{P}}} = \left\langle G_j \right\rangle_{\{w_{\mathtt{qq}'}\}_{\mathtt{qq}' \in \mathcal{P}}}
\end{aligned}$$

Figure 8.6: Projection of Buffer Contents from Global Type in RoutedSessions

As explained in Section 8.3.1, the semantics of the message buffers for routed actions are the same as their non-routed counterparts, so the projected contents of the buffers for routed communication are the same as those under non-routed communication.

8.3.3 Trace Equivalence

A sequence of transitions is known as a *trace*. We want to prove that the set of traces that can be obtained from reducing a global type G is the same as those that can be obtained from reducing its projected configuration $\langle G \rangle$.

Our approach is based on [7] – namely, proving that this is the case for a single transition (i.e. *step equivalence*) is sufficient, as we can obtain trace equivalence as a direct consequence.

Lemma 8.1 (Step Equivalence)**.** *For all global types G and configurations s, if $\langle G \rangle \prec s$, then $G \xrightarrow{l} G' \iff s \xrightarrow{l} s'$ such that $\langle G' \rangle \prec s'$.*

Proof. By induction on the possible transitions in the LTSs over global types (to prove $\Longrightarrow$, i.e. *soundness*) and configurations (to prove $\Longleftarrow$, i.e. *completeness*).

Notation conventions We use the following notation for decomposing configurations and projected configurations.

$$\begin{aligned} s &= \{T_{\mathtt{q}}\}_{\mathtt{q} \in \mathcal{P}},\ \{w_{\mathtt{q}\mathtt{q}'}\}_{\mathtt{q}\mathtt{q}' \in \mathcal{P}} \\ s' &= \{T'_{\mathtt{q}}\}_{\mathtt{q} \in \mathcal{P}},\ \{w'_{\mathtt{q}\mathtt{q}'}\}_{\mathtt{q}\mathtt{q}' \in \mathcal{P}} \\ \langle G \rangle &= \{\hat{T}_{\mathtt{q}}\}_{\mathtt{q} \in \mathcal{P}},\ \{\hat{w}_{\mathtt{q}\mathtt{q}'}\}_{\mathtt{q}\mathtt{q}' \in \mathcal{P}} \\ \langle G' \rangle &= \{\hat{T}'_{\mathtt{q}}\}_{\mathtt{q} \in \mathcal{P}},\ \{\hat{w}'_{\mathtt{q}\mathtt{q}'}\}_{\mathtt{q}\mathtt{q}' \in \mathcal{P}} \end{aligned}$$

Soundness

By induction on the structure of LTS semantics over global types.

For each transition $G \xrightarrow{l} G'$, we take the configuration $s = \langle G \rangle$, derive $G \xrightarrow{l} G'$ and $s \xrightarrow{l} s'$ under the respective LTSs, and show that $s' \prec \langle G' \rangle$.

The proofs for rules [GR1-5] are the same as in [7]. We focus on the new rules introduced for routing.

- [GR6] , where $G = \mathtt{p} \underset{\mathtt{s}}{\rightarrow} \mathtt{p}' : \{l_i : G_i\}_{i \in I}, G' = \mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{p}'.\ j : \{l_i : G_i\}_{i \in I}, l = \mathtt{via}_{\mathtt{s}}(\mathtt{p}\mathtt{p}'!j)$

  Then $s = \langle G \rangle$ where

$$\begin{aligned} T_{\mathtt{p}} &= \mathtt{p}'_{\mathtt{s}} \oplus \{l_i : G_i \upharpoonright \mathtt{p}\}_{i \in I} \\ T_{\mathtt{p}'} &= \mathtt{p}_{\mathtt{s}} \,\&\, \{l_i : G_i \upharpoonright \mathtt{p}'\}_{i \in I} \\ T_{\mathtt{s}} &= \mathtt{p} \hookrightarrow \mathtt{p}' : \{l_i : G_i \upharpoonright \mathtt{s}\}_{i \in I} \\ T_{\mathtt{r}} &= \bigsqcap_{i \in I} G_i \upharpoonright \mathtt{r} \text{ for } \mathtt{r} \notin \{\mathtt{p}, \mathtt{p}', \mathtt{s}\} \\ \{w_{\mathtt{q}\mathtt{q}'}\}_{\mathtt{q}\mathtt{q}' \in \mathcal{P}} &= \langle G_i \rangle_{\{\vec{\epsilon}\}} \text{ for some } i \in I \end{aligned}$$

Global transition: We have

$$\begin{aligned}
\hat{T}'_{\mathrm{p}'} &= \mathrm{p}_{\mathrm{s}} \,\&\, \{l_i : G_i \upharpoonright \mathrm{p}'\}_{i\in I} \\
\hat{T}'_{\mathrm{s}} &= \mathrm{p} \twoheadrightarrow \mathrm{p}'.\ j : \{l_i : G_i \upharpoonright \mathrm{s}\}_{i\in I} \\
\hat{T}'_{\mathrm{r}} &= G_j \upharpoonright \mathrm{r} \text{ for } \mathrm{r} \notin \{\mathrm{p}', \mathrm{s}\} \\
\{\hat{w}'_{\mathrm{qq}'}\}_{\mathrm{qq}'\in\mathcal{P}} &= \langle G_i \rangle_{\{\vec{\epsilon}\}[w_{\mathrm{pp}'}\mapsto w_{\mathrm{pp}'}\cdot j]} \text{ for some } i \in I
\end{aligned}$$

So, $\hat{w}'_{\mathrm{qq}'} = w_{\mathrm{qq}'}$ for $\mathrm{qq}' \neq \mathrm{pp}'$ and $\hat{w}'_{\mathrm{pp}'} = w_{\mathrm{pp}'} \cdot j$.

Configuration transition: Take $T'_{\mathrm{r}} = T_{\mathrm{r}}$ for $\mathrm{r} \notin \{\mathrm{p}, \mathrm{s}\}$.

By [LR4] , $T_{\mathrm{p}} \xrightarrow{l} T'_{\mathrm{p}}$ where $T'_{\mathrm{p}} = G_j \upharpoonright \mathrm{p}$.

By [LR6] , $T_{\mathrm{s}} \xrightarrow{l} T'_{\mathrm{s}}$ where $T'_{\mathrm{s}} = \mathrm{p} \twoheadrightarrow \mathrm{p}'.\ j : \{l_i : G_i \upharpoonright \mathrm{s}\}_{i\in I}$.
Also, $w'_{\mathrm{qq}'} = w_{\mathrm{qq}'}$ for $\mathrm{qq}' \neq \mathrm{pp}'$ and $w'_{\mathrm{pp}'} = w_{\mathrm{pp}'} \cdot j$.

Correspondence: We have $w'_{\mathrm{qq}'} = \hat{w}_{\mathrm{qq}'}$ for $\mathrm{qq}' \in \mathcal{P}$ and $T'_{\mathrm{q}} = \hat{T}_{\mathrm{q}}$ for $\mathrm{q} \in \{\mathrm{p}, \mathrm{p}', \mathrm{s}\}$.
For $\mathrm{q} \notin \{\mathrm{p}, \mathrm{p}', \mathrm{s}\}$, we have

$$T'_{\mathrm{q}} = \bigsqcap_{i\in I} G_i \upharpoonright \mathrm{q} \prec G_j \upharpoonright \mathrm{q} = \hat{T}_{\mathrm{q}}$$

So, $s' \prec \langle G' \rangle$.

- [GR7] where $G = \mathrm{p} \underset{\mathrm{s}}{\rightsquigarrow} \mathrm{p}'.\ j : \{l_i : G_i\}_{i\in I}, G' = G_j, l = \mathtt{via}_{\mathrm{s}}(\mathrm{pp}'?j)$
  Then $s = \langle G \rangle$ where

$$\begin{aligned}
T_{\mathrm{p}'} &= \mathrm{p}_{\mathrm{s}} \,\&\, \{l_i : G_i \upharpoonright \mathrm{p}'\}_{i\in I} \\
T_{\mathrm{s}} &= \mathrm{p} \twoheadrightarrow \mathrm{p}'.\ j : \{l_i : G_i \upharpoonright \mathrm{s}\}_{i\in I} \\
T_{\mathrm{r}} &= G_j \upharpoonright \mathrm{r} \text{ for } \mathrm{r} \notin \{\mathrm{p}', \mathrm{s}\} \\
\{w_{\mathrm{qq}'}\}_{\mathrm{qq}'\in\mathcal{P}} &= \left\langle G_j \right\rangle_{\{\vec{\epsilon}\}[w_{\mathrm{pp}'}\mapsto w_{\mathrm{pp}'}\cdot j]}
\end{aligned}$$

Global transition: We have

$$\begin{aligned}
\hat{T}'_{\mathrm{r}} &= G_j \upharpoonright \mathrm{r} \text{ for } \mathrm{r} \in \mathcal{P} \\
\{\hat{w}'_{\mathrm{qq}'}\}_{\mathrm{qq}'\in\mathcal{P}} &= \left\langle G_j \right\rangle_{\{\vec{\epsilon}\}}
\end{aligned}$$

So, $\hat{w}'_{\mathrm{qq}'} = w_{\mathrm{qq}'}$ for $\mathrm{qq}' \neq \mathrm{pp}'$ and $w_{\mathrm{pp}'} = j \cdot \hat{w}'_{\mathrm{pp}'}$.

Configuration transition: Take $T'_{\mathrm{r}} = T_{\mathrm{r}}$ for $\mathrm{r} \notin \{\mathrm{p}', \mathrm{s}\}$.

By [LR5] , $T_{\mathrm{p}} \xrightarrow{l} T'_{\mathrm{p}}$ where $T'_{\mathrm{p}} = G_j \upharpoonright \mathrm{p}$.

By [LR7] , $T_{\mathrm{s}} \xrightarrow{l} T'_{\mathrm{s}}$ where $T'_{\mathrm{s}} = G_j \upharpoonright \mathrm{s}$.

Also, $w'_{\mathrm{qq}'} = w_{\mathrm{qq}'}$ for $\mathrm{qq}' \neq \mathrm{pp}'$ and $w_{\mathrm{pp}'} = j \cdot w'_{\mathrm{pp}'}$.

Correspondence: We have $w'_{\mathrm{qq}'} = \hat{w}_{\mathrm{qq}'}$ for $\mathrm{qq}' \in \mathcal{P}$ and $T'_{\mathrm{q}} = \hat{T}_{\mathrm{q}}$ for $\mathrm{q} \in \mathcal{P}$.

So, $s' = \langle G' \rangle$.

- [GR8] where $G = \mathrm{p} \underset{\mathrm{s}}{\to} \mathrm{p}' : \{l_i : G_i\}_{i \in I}$, $G' = \mathrm{p} \underset{\mathrm{s}}{\to} \mathrm{p}' : \left\{l_i : G'_i\right\}_{i \in I}$

  By hypothesis, $\forall i \in I.\ G_i \xrightarrow{l} G'_i$ and $\mathrm{subj}(l) \notin \{\mathrm{p}, \mathrm{p}'\}$.

  By induction, $\forall i \in I.\ \langle G_i \rangle \xrightarrow{l} \left\langle G'_i \right\rangle$.

  To show that $\langle G \rangle \xrightarrow{l} \langle G' \rangle$, it is sufficient to show that $G \upharpoonright \mathrm{q} \xrightarrow{l} G' \upharpoonright \mathrm{q}$ for $\mathrm{q} = \mathrm{subj}(l)$, since the projections for $\mathrm{q}' \neq \mathrm{subj}(l)$ remain the same.

  We know $G \upharpoonright \mathrm{q} = \bigsqcap_{i \in I} G_i \upharpoonright \mathrm{q}$ and $G' \upharpoonright \mathrm{q} = \bigsqcap_{i \in I} G'_i \upharpoonright \mathrm{q}$.

  By induction, $\bigsqcap_{i \in I} G_i \upharpoonright \mathrm{q} \xrightarrow{l} \bigsqcap_{i \in I} G'_i \upharpoonright \mathrm{q}$, so $G \upharpoonright \mathrm{q} \xrightarrow{l} G' \upharpoonright \mathrm{q}$.

- [GR9] where $G = \mathrm{p} \underset{\mathrm{s}}{\rightsquigarrow} \mathrm{p}'.\ j : \{l_i : G_i\}_{i \in I}$, $G' = \mathrm{p} \underset{\mathrm{s}}{\rightsquigarrow} \mathrm{p}'.\ j : \left\{l_i : G'_i\right\}_{i \in I}$

  By hypothesis, $G_j \xrightarrow{l} G'_j$, $\mathrm{p}' \neq \mathrm{subj}(l)$, and $\forall i \in I \setminus \{j\}.\ G'_i = G_i$.

  By induction, $\left\langle G_j \right\rangle \xrightarrow{l} \left\langle G'_j \right\rangle$.

  To show that $\langle G \rangle \xrightarrow{l} \langle G' \rangle$, it is sufficient to show that $G \upharpoonright \mathrm{q} \xrightarrow{l} G' \upharpoonright \mathrm{q}$ for $\mathrm{q} = \mathrm{subj}(l)$, since the projections for $\mathrm{q}' \neq \mathrm{subj}(l)$ remain the same.

  We know $G \upharpoonright \mathrm{q} = G_j \upharpoonright \mathrm{q}$ and $G' \upharpoonright \mathrm{q} = G'_j \upharpoonright \mathrm{q}$.

  By induction, $G_j \upharpoonright \mathrm{q} \xrightarrow{l} G'_j \upharpoonright \mathrm{q}$, so $G \upharpoonright \mathrm{q} \xrightarrow{l} G' \upharpoonright \mathrm{q}$.

Completeness

By considering the possible transitions in the LTS over configurations, which is defined by case analysis on the possible labels l.

For each transition $s \xrightarrow{l} s'$, we take the configuration s from the reduction rule, infer the structure of the global type G such that $s = \langle G \rangle$, derive $s \xrightarrow{l} s'$ and $G \xrightarrow{l} G'$ under the respective LTSs, and show that $s' \prec \langle G' \rangle$.

The proofs for $l = \mathrm{pq}!j$ and $l = \mathrm{pq}?j$ are the same as in Appendix A.1 of [7]. We focus on the new labels introduced for routing.

- $l = \mathtt{via}_{\mathtt{s}}(\mathtt{pq}!j)$:
  Then $T_{\mathtt{p}} = \mathtt{q}_{\mathtt{s}} \oplus \{l_i : G_i \upharpoonright \mathtt{p}\}_{i \in I}$.
  Also, $T_{\mathtt{s}}$ *contains* $\mathtt{p} \hookrightarrow \mathtt{q} : \{l_i : G_i \upharpoonright \mathtt{s}\}_{i \in I}$ as subterm. We denote this subterm $\tilde{T}_{\mathtt{s}}$.
  By definition of projection, G has $\mathtt{p} \underset{\mathtt{s}}{\rightarrow} \mathtt{q} : \{l_i : G_i\}_{i \in I}$ as subterm. We denote this subterm $\tilde{G}$.
  Also by definition of projection, no action in G will involve $\mathtt{p}$ before $\tilde{G}$.

  Configuration transition: By [LR4] , $T_{\mathtt{p}} \xrightarrow{l} T'_{\mathtt{p}}$, where $T'_{\mathtt{p}} = G_j \upharpoonright \mathtt{p}$.
  By [LR6] , $\tilde{T}_{\mathtt{s}} \xrightarrow{l} \tilde{T}'_{\mathtt{s}}$, where $\tilde{T}'_{\mathtt{s}} = \mathtt{p} \twoheadrightarrow \mathtt{q}.\ j : \{l_i : G_i \upharpoonright \mathtt{s}\}_{i \in I}$.
  We get $T_{\mathtt{s}} \xrightarrow{l} T'_{\mathtt{s}}$ by inversion lemma, as illustrated below.

$$\dfrac{\dfrac{\dfrac{}{\tilde{T}_{\mathtt{s}} \xrightarrow{l} \tilde{T}'_{\mathtt{s}}}\,[\text{LR6}]}{\vdots}}{T_{\mathtt{s}} \xrightarrow{l} T'_{\mathtt{s}}}\,[\text{LR8,9,10,11}]\ \text{as needed}$$

  Global transition: By [GR6] , $\tilde{G} \xrightarrow{l} \tilde{G}'$, where $\tilde{G}' = \mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}$.
  We get $G \xrightarrow{l} G'$ by inversion lemma, as illustrated below.

$$\dfrac{\dfrac{\dfrac{}{\tilde{G} \xrightarrow{l} \tilde{G}'}\,[\text{GR6}]}{\vdots}}{G \xrightarrow{l} G'}\,[\text{GR4,5,8,9}]\ \text{as needed}$$

  Correspondence: Since the projections for $\mathtt{p}' \notin \{\mathtt{p}, \mathtt{s}\}$ are unchanged, it is sufficient to show that $T'_{\mathtt{p}} \prec (\tilde{G}' \upharpoonright \mathtt{p})$ and $\tilde{T}'_{\mathtt{s}} \prec (\tilde{G}' \upharpoonright \mathtt{s})$.

$$\tilde{G}' \upharpoonright \mathtt{p} = G_j \upharpoonright \mathtt{p} = T'_{\mathtt{p}}$$
$$\tilde{G}' \upharpoonright \mathtt{s} = \mathtt{p} \twoheadrightarrow \mathtt{q}.\ j : \{l_i : G_i \upharpoonright \mathtt{s}\}_{i \in I} = \tilde{T}'_{\mathtt{s}}$$

- $l = \mathtt{via}_{\mathtt{s}}(\mathtt{pq}?j)$:
  Then $T_{\mathtt{q}} = \mathtt{p}_{\mathtt{s}} \,\&\, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i \in I}$.
  Also, $T_{\mathtt{s}}$ *contains* $\mathtt{p} \twoheadrightarrow \mathtt{q}.\ j : \{l_i : G_i \upharpoonright \mathtt{s}\}_{i \in I}$ as subterm. We denote this subterm $\tilde{T}_{\mathtt{s}}$.
  By definition of projection, G has $\mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}$ as subterm. We denote this subterm $\tilde{G}$.
  Also by definition of projection, no action in G will involve $\mathtt{q}$ before $\tilde{G}$.

Configuration transition: By [LR5] , $T_\mathtt{q} \xrightarrow{l} T'_\mathtt{q}$, where $T'_\mathtt{q} = G_j \upharpoonright \mathtt{q}$.

By [LR7] , $\tilde{T}_\mathtt{s} \xrightarrow{l} \tilde{T}'_\mathtt{s}$, where $\tilde{T}'_\mathtt{s} = G_j \upharpoonright \mathtt{s}$.

We get $T_\mathtt{s} \xrightarrow{l} T'_\mathtt{s}$ by inversion lemma, as illustrated below.

$$\dfrac{\dfrac{\dfrac{}{\tilde{T}_\mathtt{s} \xrightarrow{l} \tilde{T}'_\mathtt{s}}\,[\text{LR7}]}{\vdots}}{T_\mathtt{s} \xrightarrow{l} T'_\mathtt{s}}\,[\text{LR8,9,10,11}]\ \text{as needed}$$

Global transition: By [GR7] , $\tilde{G} \xrightarrow{l} \tilde{G}'$, where $\tilde{G}' = G_j$.

We get $G \xrightarrow{l} G'$ by inversion lemma, as illustrated below.

$$\dfrac{\dfrac{\dfrac{}{\tilde{G} \xrightarrow{l} \tilde{G}'}\,[\text{GR7}]}{\vdots}}{G \xrightarrow{l} G'}\,[\text{GR4,5,8,9}]\ \text{as needed}$$

Correspondence: Since the projections for $\mathtt{p}' \notin \{\mathtt{q}, \mathtt{s}\}$ are unchanged, it is sufficient to show that $T'_\mathtt{q} \prec (\tilde{G}' \upharpoonright \mathtt{q})$ and $\tilde{T}'_\mathtt{s} \prec (\tilde{G}' \upharpoonright \mathtt{s})$.

$$\tilde{G}' \upharpoonright \mathtt{q} = G_j \upharpoonright \mathtt{q} = T'_\mathtt{q}$$
$$\tilde{G}' \upharpoonright \mathtt{s} = G_j \upharpoonright \mathtt{s} = \tilde{T}'_\mathtt{s}$$

□

Theorem 8.1 (Trace Equivalence)**.** *Let G be a global type with participants $\mathcal{P} = \mathtt{pt}(G)$, and let $\vec{T} = \{G \upharpoonright \mathtt{p}\}_{\mathtt{p} \in \mathcal{P}}$ be the local types projected from G. Then $G \approx (\vec{T}, \vec{\epsilon})$.*

Proof. Direct consequence of Lemma 8.1. □

8.3.4 Deadlock Freedom

Lemma 8.2 (Preservation of Well-formedness)**.** *Let G be a global type. Suppose G is well-formed with respect to some router* s*, i.e.* $\mathtt{wellFormed}(G, \mathtt{s})$.

$$\forall G', l.\ \left(G \xrightarrow{l} G' \Longrightarrow \mathtt{wellFormed}(G', \mathtt{s})\right)$$

Proof. By induction on the structure of $G \xrightarrow{l} G'$.

For each transition, we show the two conjuncts for $\mathtt{wellFormed}(G', \mathtt{s})$: **(1)** $G' \upharpoonright \mathtt{r}$ exists for $\mathtt{r}$ such that $G \upharpoonright \mathtt{r}$ exists; and, **(2)** $G' \circledast \mathtt{s}$.

- [GR1] , where $G = \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$, $G' = \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}$, $l = \mathtt{pq}!j$.

(1) We know $G \upharpoonright \mathrm{r}$ by assumption. To show $G' \upharpoonright \mathrm{r}$, consider r by case:

- $\mathrm{r} = \mathrm{p}$: Then $G \upharpoonright \mathrm{p} = \mathrm{q} \oplus \{l_i : G_i \upharpoonright \mathrm{p}\}_{i \in I}$, so $\forall i \in I.\ G_i \upharpoonright \mathrm{p}$ exists.
  $G' \upharpoonright \mathrm{p} = G_j \upharpoonright \mathrm{p}$, which exists as $j \in I$.
- $\mathrm{r} = \mathrm{q}$: Then $G' \upharpoonright \mathrm{q} = \mathrm{p} \,\&\, \{l_i : G_i \upharpoonright \mathrm{q}\}_{i \in I}\ G \upharpoonright \mathrm{q}$, which exists.
- $\mathrm{r} \notin \{\mathrm{p}, \mathrm{q}\}$: Then $G \upharpoonright \mathrm{r} = \bigsqcap_{i \in I} G_i \upharpoonright \mathrm{r}$, so $\forall i \in I.\ G_i \upharpoonright \mathrm{r}$ exists.
  $G' \upharpoonright \mathrm{r} = G_j \upharpoonright \mathrm{r}$, which exists as $j \in I$.

(2) We know $G \circledast \mathrm{s}$ by assumption. We deduce $G' \circledast \mathrm{s}$ by consequence.

$$G \circledast \mathrm{s} \Longrightarrow \mathrm{s} \in \{\mathrm{p}, \mathrm{q}\} \wedge \bigwedge_{i \in I} G_i \circledast \mathrm{s} \Longrightarrow \mathrm{s} \in \{\mathrm{p}, \mathrm{q}\} \wedge G_j \circledast \mathrm{s} \Longrightarrow G' \circledast \mathrm{s}$$

- [GR2] , where $G = \mathrm{p} \rightsquigarrow \mathrm{q}.\ j : \{l_i : G_i\}_{i \in I}$, $G' = G_j$, $l = \mathrm{pq}?j$.

(1) We know $G \upharpoonright \mathrm{r}$ by assumption. To show $G' \upharpoonright \mathrm{r}$, consider r by case:

- $\mathrm{r} = \mathrm{p}$: Then $G' \upharpoonright \mathrm{p} = G_j \upharpoonright \mathrm{p} = G \upharpoonright \mathrm{p}$, which exists.
- $\mathrm{r} = \mathrm{q}$: Then $G \upharpoonright \mathrm{q} = \mathrm{p} \,\&\, \{l_i : G_i \upharpoonright \mathrm{q}\}_{i \in I}$, so $\forall i \in I.\ G_i \upharpoonright \mathrm{q}$ exists.
  $G' \upharpoonright \mathrm{q} = G_j \upharpoonright \mathrm{q}$, which exists as $j \in I$.
- $\mathrm{r} \notin \{\mathrm{p}, \mathrm{q}\}$: Then $G' \upharpoonright \mathrm{r} = G_j \upharpoonright \mathrm{r} = G \upharpoonright \mathrm{r}$, which exists.

(2) We know $G \circledast \mathrm{s}$ by assumption. We deduce $G' \circledast \mathrm{s}$ by consequence.

$$G \circledast \mathrm{s} \Longrightarrow \mathrm{s} \in \{\mathrm{p}, \mathrm{q}\} \wedge G_j \circledast \mathrm{s} \Longrightarrow G' \circledast \mathrm{s}$$

- [GR3] , where $\mu \mathtt{t}.G \xrightarrow{l} G'$. By hypothesis, $G[\mu \mathtt{t}.G/\mathtt{t}] \xrightarrow{l} G'$.
  We first show that $\mathtt{wellFormed}(G[\mu \mathtt{t}.G/\mathtt{t}], \mathrm{s})$.

(1) $\mu \mathtt{t}.G \upharpoonright \mathrm{r}$ exists for some r.

Note that $G \upharpoonright \mathrm{r}$ exists regardless of r's participation in G.

- If $\mathrm{r} \in \mathtt{pt}(G)$, then $\mu \mathtt{t}.G \upharpoonright \mathrm{r} = \mu \mathtt{t}.G \upharpoonright \mathrm{r}$, so $G \upharpoonright \mathrm{r}$ exists.
- Otherwise, $G \upharpoonright \mathrm{r} = \mathtt{end}$, which exists.

Projection is homomorphic under recursion, so $G[\mu \mathtt{t}.G/\mathtt{t}] \upharpoonright \mathrm{r}$ exists.

(2) By assumption, $(\mu \mathtt{t}.G) \circledast \mathtt{s}$, so $G \circledast \mathtt{s}$.

The $\circledast$ relation is also homomorphic under recursion, so we get $G[\mu \mathtt{t}.G/\mathtt{t}] \circledast \mathtt{s}$.

We conclude by induction to obtain $\mathtt{wellFormed}(G', \mathtt{s})$.

- [GR4] , where $G = \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$, $G' = \mathtt{p} \to \mathtt{q} : \left\{l_i : G_i'\right\}_{i \in I}$.

  By hypothesis, $\forall i \in I.\ (G_i \xrightarrow{l} G_i')$ and $\mathtt{p} \neq \mathtt{q} \neq \mathrm{subj}(l)$.

  If $G \upharpoonright \mathtt{r}$ exists, so does $G_i \upharpoonright \mathtt{r}$ for $i \in I$.

  By assumption, $G \circledast \mathtt{s}$, so $\mathtt{s} \in \{\mathtt{p}, \mathtt{q}\} \wedge \bigwedge_{i \in I} G_i \circledast \mathtt{s}$.

  By induction, $\forall i \in I.\ (G_i' \upharpoonright \mathtt{r}$ exists $\wedge\ G_i' \circledast \mathtt{s})$.

  (1) To show $G' \upharpoonright \mathtt{r}$, consider $\mathtt{r}$ by case:

  - $\mathtt{r} = \mathtt{p}$: Then $G' \upharpoonright \mathtt{p} = \mathtt{q} \oplus \{l_i : G' \upharpoonright \mathtt{p}\}_{i \in I}$.
  - $\mathtt{r} = \mathtt{q}$: Then $G' \upharpoonright \mathtt{q} = \mathtt{p} \,\&\, \left\{l_i : G_i' \upharpoonright \mathtt{q}\right\}_{i \in I}$.
  - $\mathtt{r} \notin \{\mathtt{p}, \mathtt{q}\}$: Then $G' \upharpoonright \mathtt{r} = \bigsqcap_{i \in I} G_i' \upharpoonright \mathtt{r}$. We know that $G \upharpoonright \mathtt{r} = \bigsqcap_{i \in I} G_i \upharpoonright \mathtt{r}$ exists. By Lemma A.1, $\bigsqcap_{i \in I} G_i' \upharpoonright \mathtt{r}$ exists too.

  (2) We have $\mathtt{s} \in \{\mathtt{p}, \mathtt{q}\}$ from assumption and $\bigwedge_{i \in I} G_i' \circledast \mathtt{s}$ from induction, so $G' \circledast \mathtt{s}$.

- [GR5] , where $G = \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}$, $G' = \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \left\{l_i : G_i'\right\}_{i \in I}$.

  By hypothesis, $G_j \xrightarrow{l} G_j'$, $\forall i \in I \setminus \{j\}.\ G_i' = G_i$, and $\mathtt{q} \neq \mathrm{subj}(l)$.

  If $G \upharpoonright \mathtt{r}$ exists, so does $G_i \upharpoonright \mathtt{r}$ for $i \in I$.

  By assumption, $G \circledast \mathtt{s}$, so $\mathtt{s} \in \{\mathtt{p}, \mathtt{q}\} \wedge G_j \circledast \mathtt{s}$.

  By induction on $G_j \xrightarrow{l} G_j'$ and hypothesis $\forall i \in I \setminus \{j\}.\ G_i' = G_i$, we get $\forall i \in I.\ (G_i' \upharpoonright \mathtt{r}$ exists $\wedge\ G_i' \circledast \mathtt{s})$.

  (1) To show $G' \upharpoonright \mathtt{r}$, consider $\mathtt{r}$ by case:

  - $\mathtt{r} = \mathtt{p}$: Then $G' \upharpoonright \mathtt{p} = G_j' \upharpoonright \mathtt{p}$.
  - $\mathtt{r} = \mathtt{q}$: Then $G' \upharpoonright \mathtt{q} = \mathtt{p} \,\&\, \left\{l_i : G_i' \upharpoonright \mathtt{q}\right\}_{i \in I}$.
  - $\mathtt{r} \notin \{\mathtt{p}, \mathtt{q}\}$: Then $G' \upharpoonright \mathtt{r} = G_j' \upharpoonright \mathtt{r}$.

(2) We have $\mathtt{s} \in \{\mathtt{p}, \mathtt{q}\}$ from assumption and $G'_j \circledast \mathtt{s}$ from induction, so $G' \circledast \mathtt{s}$.

- [GR6] , where $G = \mathtt{p} \underset{\mathtt{t}}{\to} \mathtt{q} : \{l_i : G_i\}_{i \in I}$, $G' = \mathtt{p} \underset{\mathtt{t}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}$, $l = \mathtt{via}_{\mathtt{s}}(\mathtt{pq}!j)$.

  By assumption, $\mathtt{wellFormed}(G, \mathtt{s})$, so $\mathtt{t} = \mathtt{s}$.

  (1) We know $G \upharpoonright \mathtt{r}$ by assumption. To show $G' \upharpoonright \mathtt{r}$, consider $\mathtt{r}$ by case:

  - $\mathtt{r} = \mathtt{p}$: Then $G \upharpoonright \mathtt{p} = \mathtt{q}_{\mathtt{s}} \oplus \{l_i : G_i \upharpoonright \mathtt{p}\}_{i \in I}$, so $\forall i \in I.\ G_i \upharpoonright \mathtt{p}$ exists.
    $G' \upharpoonright \mathtt{p} = G_j \upharpoonright \mathtt{p}$, which exists as $j \in I$.
  - $\mathtt{r} = \mathtt{q}$: Then $G' \upharpoonright \mathtt{q} = \mathtt{p}_{\mathtt{s}} \,\&\, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i \in I} = G \upharpoonright \mathtt{q}$, which exists.
  - $\mathtt{r} = \mathtt{s}$: Then $G \upharpoonright \mathtt{s} = \mathtt{p} \hookrightarrow \mathtt{q} : \{l_i : G_i \upharpoonright \mathtt{s}\}_{i \in I}$, so $\forall i \in I.\ G_i \upharpoonright \mathtt{s}$ exists.
    $G' \upharpoonright \mathtt{s} = \mathtt{p} \hookrightarrow \mathtt{q}.\ j : \{l_i : G_i \upharpoonright \mathtt{s}\}_{i \in I}$, which exists.
  - $\mathtt{r} \notin \{\mathtt{p}, \mathtt{q}, \mathtt{s}\}$: Then $G \upharpoonright \mathtt{r} = \bigsqcap_{i \in I} G_i \upharpoonright \mathtt{r}$, so $\forall i \in I.\ G_i \upharpoonright \mathtt{r}$ exists.
    $G' \upharpoonright \mathtt{r} = G_j \upharpoonright \mathtt{r}$, which exists as $j \in I$.

  (2) We know $G \circledast \mathtt{s}$ by assumption. We deduce $G' \circledast \mathtt{s}$ by consequence.

$$G \circledast \mathtt{s} \Longrightarrow \mathtt{t} = \mathtt{s} \wedge \bigwedge_{i \in I} G_i \circledast \mathtt{s} \Longrightarrow \mathtt{t} = \mathtt{s} \wedge G_j \circledast \mathtt{s} \Longrightarrow G' \circledast \mathtt{s}$$

- [GR7] , where $G = \mathtt{p} \underset{\mathtt{t}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}$, $G' = G_j$, $l = \mathtt{via}_{\mathtt{s}}(\mathtt{pq}?j)$.

  (1) We know $G \upharpoonright \mathtt{r}$ by assumption. To show $G' \upharpoonright \mathtt{r}$, consider $\mathtt{r}$ by case:

  By assumption, $\mathtt{wellFormed}(G, \mathtt{s})$, so $\mathtt{t} = \mathtt{s}$.

  - $\mathtt{r} = \mathtt{p}$: Then $G' \upharpoonright \mathtt{p} = G_j \upharpoonright \mathtt{p} = G \upharpoonright \mathtt{p}$, which exists.
  - $\mathtt{r} = \mathtt{q}$: Then $G \upharpoonright \mathtt{q} = \mathtt{p}_{\mathtt{s}} \,\&\, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i \in I}$, so $\forall i \in I.\ G_i \upharpoonright \mathtt{q}$ exists.
    $G' \upharpoonright \mathtt{q} = G_j \upharpoonright \mathtt{q}$, which exists as $j \in I$.
  - $\mathtt{r} = \mathtt{s}$: Then $G \upharpoonright \mathtt{s} = \mathtt{p} \hookrightarrow \mathtt{q}.\ j : \{l_i : G_i \upharpoonright \mathtt{s}\}_{i \in I}$, so $\forall i \in I.\ G_i \upharpoonright \mathtt{s}$ exists.
    $G' \upharpoonright \mathtt{s} = G_j \upharpoonright \mathtt{s}$, which exists as $j \in I$.
  - $\mathtt{r} \notin \{\mathtt{p}, \mathtt{q}, \mathtt{s}\}$: Then $G \upharpoonright \mathtt{r} = \bigsqcap_{i \in I} G_i \upharpoonright \mathtt{r}$, so $\forall i \in I.\ G_i \upharpoonright \mathtt{r}$ exists.
    $G' \upharpoonright \mathtt{r} = G_j \upharpoonright \mathtt{r}$, which exists as $j \in I$.

(2) We know $G \circledast \mathtt{s}$ by assumption. We deduce $G' \circledast \mathtt{s}$ by consequence.

$$G \circledast \mathtt{s} \Longrightarrow \mathtt{s} \in \{\mathtt{p}, \mathtt{q}\} \wedge G_j \circledast \mathtt{s} \Longrightarrow G' \circledast \mathtt{s}$$

- [GR8] , where $G = \mathtt{p} \underset{\mathtt{t}}{\rightarrow} \mathtt{q} : \{l_i : G_i\}_{i \in I}$, $G' = \mathtt{p} \underset{\mathtt{t}}{\rightarrow} \mathtt{q} : \left\{l_i : G'_i\right\}_{i \in I}$.

  By hypothesis, $\forall i \in I.\ (G_i \xrightarrow{l} G'_i)$ and $\mathtt{p} \neq \mathtt{q} \neq \mathrm{subj}(l)$.
  If $G \upharpoonright \mathtt{r}$ exists, so does $G_i \upharpoonright \mathtt{r}$ for $i \in I$.
  By assumption, $G \circledast \mathtt{s}$, so $\mathtt{t} = \mathtt{s} \wedge \bigwedge_{i \in I} G_i \circledast \mathtt{s}$.
  By induction, $\forall i \in I.\ (G'_i \upharpoonright \mathtt{r}$ exists $\wedge\ G'_i \circledast \mathtt{s})$.

  (1) To show $G' \upharpoonright \mathtt{r}$, consider $\mathtt{r}$ by case:

  - $\mathtt{r} = \mathtt{p}$: Then $G' \upharpoonright \mathtt{p} = \mathtt{q}_{\mathtt{s}} \oplus \{l_i : G' \upharpoonright \mathtt{p}\}_{i \in I}$.
  - $\mathtt{r} = \mathtt{q}$: Then $G' \upharpoonright \mathtt{q} = \mathtt{p}_{\mathtt{s}} \,\&\, \left\{l_i : G'_i \upharpoonright \mathtt{q}\right\}_{i \in I}$.
  - $\mathtt{r} = \mathtt{s}$: Then $G' \upharpoonright \mathtt{s} = \mathtt{p} \hookrightarrow \mathtt{q} : \left\{l_i : G'_i \upharpoonright \mathtt{s}\right\}_{i \in I}$.
  - $\mathtt{r} \notin \{\mathtt{p}, \mathtt{q}, \mathtt{s}\}$: Then $G' \upharpoonright \mathtt{r} = \bigsqcap_{i \in I} G'_i \upharpoonright \mathtt{r}$. We know that $G \upharpoonright \mathtt{r} = \bigsqcap_{i \in I} G_i \upharpoonright \mathtt{r}$ exists. By Lemma A.1, $\bigsqcap_{i \in I} G'_i \upharpoonright \mathtt{r}$ exists too.

  (2) We have $\mathtt{t} = \mathtt{s}$ from assumption and $\bigwedge_{i \in I} G'_i \circledast \mathtt{s}$ from induction, so $G' \circledast \mathtt{s}$.

- [GR9] , where $G = \mathtt{p} \underset{\mathtt{t}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}$, $G' = \mathtt{p} \underset{\mathtt{t}}{\rightsquigarrow} \mathtt{q}.\ j : \left\{l_i : G'_i\right\}_{i \in I}$.

  By hypothesis, $G_j \xrightarrow{l} G'_j$, $\forall i \in I \setminus \{j\}.\ G'_i = G_i$, and $\mathtt{q} \neq \mathrm{subj}(l)$.
  If $G \upharpoonright \mathtt{r}$ exists, so does $G_i \upharpoonright \mathtt{r}$ for $i \in I$.
  By assumption, $G \circledast \mathtt{s}$, so $\mathtt{t} = \mathtt{s} \wedge G_j \circledast \mathtt{s}$.
  By induction on $G_j \xrightarrow{l} G'_j$ and hypothesis $\forall i \in I \setminus \{j\}.\ G'_i = G_i$, we get $\forall i \in I.\ (G'_i \upharpoonright \mathtt{r}$ exists $\wedge\ G'_i \circledast \mathtt{s})$.

  (1) To show $G' \upharpoonright \mathtt{r}$, consider $\mathtt{r}$ by case:

  - $\mathtt{r} = \mathtt{p}$: Then $G' \upharpoonright \mathtt{p} = G'_j \upharpoonright \mathtt{p}$.
  - $\mathtt{r} = \mathtt{q}$: Then $G' \upharpoonright \mathtt{q} = \mathtt{p}_{\mathtt{s}} \,\&\, \left\{l_i : G'_i \upharpoonright \mathtt{q}\right\}_{i \in I}$.
  - $\mathtt{r} = \mathtt{s}$: Then $G' \upharpoonright \mathtt{s} = \mathtt{p} \rightarrowtail \mathtt{q}.\ j : \left\{l_i : G'_i \upharpoonright \mathtt{s}\right\}_{i \in I}$.
  - $\mathtt{r} \notin \{\mathtt{p}, \mathtt{q}, \mathtt{s}\}$: Then $G' \upharpoonright \mathtt{r} = G'_j \upharpoonright \mathtt{r}$.

(2) We have $\mathtt{t} = \mathtt{s}$ from assumption and $G'_j \circledast \mathtt{s}$ from induction, so $G' \circledast \mathtt{s}$.

□

Lemma 8.3 (Progress for Well-formed Global Types)**.** *Let G be a global type. Suppose G is well-formed with respect to some router* s*, i.e.* $\mathtt{wellFormed}(G, \mathtt{s})$.

$$(G = \mathtt{end}) \vee \exists G', l.\ (G \xrightarrow{l} G')$$

Proof. The following is logically equivalent.

$$(G \neq \mathtt{end}) \implies \exists G', l.\ (G \xrightarrow{l} G')$$

We prove this by induction on the structure of G.
We do not consider $G = \mathtt{end}$ by assumption.
We also do not consider $G = \mathtt{t}$ as the type variable is not guarded.

1. $G = \mu \mathtt{t}.G''$

   $\mathtt{t}$ must occur in G, so $G[\mu \mathtt{t}.G/\mathtt{t}] \neq \mathtt{end}$.

   By induction, $\exists G', l.\ (G[\mu \mathtt{t}.G/\mathtt{t}] \xrightarrow{l} G')$.

   Apply [GR3] to get $\exists G', l.\ (\mu \mathtt{t}.G \xrightarrow{l} G')$.

2. $G = \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$

   Apply [GR1] to get $G \xrightarrow{\mathtt{pq}!j} \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}$.

3. $G = \mathtt{p} \underset{\mathtt{r}}{\to} \mathtt{q} : \{l_i : G_i\}_{i \in I}$

   By assumption, $\mathtt{wellFormed}(G, \mathtt{s})$, so $\mathtt{r} = \mathtt{s}$.

   Apply [GR6] to get $G \xrightarrow{\mathtt{via}_{\mathtt{s}}(\mathtt{pq}!j)} \mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}$.

4. $G = \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}$

   Apply [GR2] to get $G \xrightarrow{\mathtt{pq}?j} G_j$.

5. $G = \mathtt{p} \underset{\mathtt{r}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}$

   By assumption, $\mathtt{wellFormed}(G, \mathtt{s})$, so $\mathtt{r} = \mathtt{s}$.

   Apply [GR7] to get $G \xrightarrow{\mathtt{via}_{\mathtt{s}}(\mathtt{pq}?j)} G_j$.

□

Theorem 8.2 (Deadlock Freedom)**.** *Let G be a global type. Suppose G is well-formed with respect to some router* s*, i.e.* $\mathtt{wellFormed}(G, \mathtt{s})$.

$$\forall G'.\ \left(G \to^* G' \implies (G' = \mathtt{end}) \vee \exists G'', l.\ (G' \xrightarrow{l} G'')\right)$$

Proof. Direct consequence of Lemmas 8.2 and 8.3. □

8.4 From Canonical MPST to RoutedSessions

We present an encoding from canonical MPST theory to RoutedSessions. This encoding is *parameterised* by the router role (conventionally denoted as `s`); the intuition is that we encode all communication interactions to involve `s`. For example, as motivated in Chapter 7, we can encode the Two Buyer protocol into routed multiparty session types with respect to the Seller role as the router. When we extend SessionTS to implement RoutedSessions in Chapter 9, the developer can implement the Two Buyer protocol as an interactive web application over WebSocket transport.

We define the encoding in Section 8.4.1. More importantly, we prove that the encoded routed communications preserve the communication safety properties (Section 8.4.2) and communication structure (Section 8.4.3) of the original communication. This directly addresses the challenges discussed in Section 7.3

8.4.1 Router-Parameterised Encoding

We define the router-parameterised encoding on global types, local types and LTS labels in the canonical MPST theory.

We start with global types, as presented in Definition 8.9. The main rule is [Enc-G-Comm] : if the communication did not go through `s`, then the encoded communication involves `s` as the router. The remaining rules are self-explanatory.

Definition 8.9 (Encoding on Global Types)**.**

$$
\begin{aligned}
[\![\mathtt{end},\ \mathtt{s}]\!] &= \mathtt{end} && \text{[Enc-G-End]}\\
[\![\mathtt{t},\ \mathtt{s}]\!] &= \mathtt{t} && \text{[Enc-G-RecVar]}\\
[\![\mu \mathtt{t}.G,\ \mathtt{s}]\!] &= \mu \mathtt{t}.[\![G,\ \mathtt{s}]\!] && \text{[Enc-G-Rec]}\\
[\![\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i\in I},\ \mathtt{s}]\!] &= \begin{cases} \mathtt{p} \to \mathtt{q} : \{l_i : [\![G_i,\ \mathtt{s}]\!]\}_{i\in I} & \text{if } \mathtt{s} \in \{\mathtt{p},\mathtt{q}\} \\ \mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \{l_i : [\![G_i,\ \mathtt{s}]\!]\}_{i\in I} & \text{otherwise} \end{cases} && \text{[Enc-G-Comm]}
\end{aligned}
$$

Readers may observe striking similarities between the encoding on global types and the definition of a centroid in routed communication (Definition 8.4). In fact, the two are the same – the encoding on global types simply expresses the centroid predicate as a function. We arrive at Lemma 8.4 – the proof is straightforward by induction on global types.

Lemma 8.4 (Encoding Defines Centroid)**.** *Given an encoding of global type G with respect to the router role* s*, the router role is the centroid of the encoded communication.*

$$[\![G,\ \mathtt{s}]\!] \circledast \mathtt{s}$$

Proof. By induction on the structure of G. □

We now define the encoding on local types in Definition 8.10. Local types express communication from the perspective of a particular role, say q. We need to capture this information in the encoding to accurately encode into the routed communication, so the encoding on local types is parameterised by *two* roles.

Definition 8.10 (Encoding on Local Types)**.** The encoding of interpreting local type T (from the perspective of role q) with respect to the router role s, or $[\![T,\ \mathtt{q},\ \mathtt{s}]\!]$, is defined as:

$$[\![\mathtt{end},\ \mathtt{q},\ \mathtt{s}]\!] = \mathtt{end} \qquad \text{[Enc-L-End]}$$

$$[\![\mathtt{t},\ \mathtt{q},\ \mathtt{s}]\!] = \mathtt{t} \qquad \text{[Enc-L-RecVar]}$$

$$[\![\mu\mathtt{t}.T,\ \mathtt{q},\ \mathtt{s}]\!] = \mu\mathtt{t}.[\![T,\ \mathtt{q},\ \mathtt{s}]\!] \qquad \text{[Enc-L-Rec]}$$

$$[\![\mathtt{p} \oplus \{l_i : T_i\}_{i\in I},\ \mathtt{q},\ \mathtt{s}]\!] = \begin{cases} \mathtt{p} \oplus \{l_i : [\![T_i,\ \mathtt{q},\ \mathtt{s}]\!]\}_{i\in I} & \text{if } \mathtt{s} \in \{\mathtt{p},\mathtt{q}\} \\ \mathtt{p}_\mathtt{s} \oplus \{l_i : [\![T_i,\ \mathtt{q},\ \mathtt{s}]\!]\}_{i\in I} & \text{otherwise} \end{cases} \qquad \text{[Enc-L-Sel]}$$

$$[\![\mathtt{p} \,\&\, \{l_i : T_i\}_{i\in I},\ \mathtt{q},\ \mathtt{s}]\!] = \begin{cases} \mathtt{p} \,\&\, \{l_i : [\![T_i,\ \mathtt{q},\ \mathtt{s}]\!]\}_{i\in I} & \text{if } \mathtt{s} \in \{\mathtt{p},\mathtt{q}\} \\ \mathtt{p}_\mathtt{s} \,\&\, \{l_i : [\![T_i,\ \mathtt{q},\ \mathtt{s}]\!]\}_{i\in I} & \text{otherwise} \end{cases} \qquad \text{[Enc-L-Bra]}$$

Similarly, if the original communication does not involve the router from a local perspective, the local endpoint will carry out the communication through the router s in the encoded local type. This is expressed in both [Enc-L-Sel] and [Enc-L-Bra] .

We establish a correspondence between the encodings for global types and local types.

$$\forall \mathtt{r}, \mathtt{s}, G.\ (\mathtt{r} \neq \mathtt{s} \Longrightarrow [\![G,\ \mathtt{s}]\!] \upharpoonright \mathtt{r} = [\![G \upharpoonright \mathtt{r},\ \mathtt{r},\ \mathtt{s}]\!])$$

We formalise this as Lemma A.6. The constraint $\mathtt{r} \neq \mathtt{s}$ is necessary because we would otherwise lose information on the right-hand side of the equality: the projection of s in the original communication will not contain the routed interactions, so applying the local type encoding cannot recover this information. We prove Lemma A.6 by induction on the structure of G; the proof is mechanical and is left in Appendix A.1 for the interested reader.

This correspondence lets us prove a more useful lemma about how our encoding preserves the equality of projections, as presented in Lemma A.7.

We also encode LTS actions using the same idea. This is presented in Definition 8.11, and is required for proofs in Section 8.4.3.

Definition 8.11 (Encoding on LTS Actions)**.**

$$[\![\mathtt{pq}!j,\ \mathtt{s}]\!] = \begin{cases} \mathtt{pq}!j & \text{if } \mathtt{s} \in \{\mathtt{p},\mathtt{q}\} \\ \mathtt{via}_{\mathtt{s}}(\mathtt{pq}!j) & \text{otherwise} \end{cases} \quad [\textsc{Enc-L-Out}]$$

$$[\![\mathtt{pq}?j,\ \mathtt{s}]\!] = \begin{cases} \mathtt{pq}?j & \text{if } \mathtt{s} \in \{\mathtt{p},\mathtt{q}\} \\ \mathtt{via}_{\mathtt{s}}(\mathtt{pq}?j) & \text{otherwise} \end{cases} \quad [\textsc{Enc-L-In}]$$

8.4.2 Preserving Well-formedness

We show that our encoding preserves the well-formedness of global types in Theorem 8.3. Well-formedness in RoutedSessions defines a constraint on the centroid of the routed communication. As this directly follows from Lemma 8.4, it is sufficient to prove that our encoding preserves projection (Lemma 8.5), then arrive at Theorem 8.3 by consequence.

Lemma 8.5 (Encoding Preserves Projection)**.** *Let G be a global type. Take arbitrary roles* $\mathtt{r}, \mathtt{s}$.

$$G \upharpoonright \mathtt{r} \textit{ exists} \implies [\![G,\ \mathtt{s}]\!] \upharpoonright \mathtt{r} \textit{ exists}$$

Proof. By induction on the structure of G.

1. $G = \mathtt{end}$, $G = \mathtt{t}$

   As $[\![G,\ \mathtt{s}]\!] = G$, if $G \upharpoonright \mathtt{r}$ exists, so does $[\![G,\ \mathtt{s}]\!] \upharpoonright \mathtt{r}$.

2. $G = \mu\mathtt{t}.G'$

   By assumption, $\mu\mathtt{t}.G' \upharpoonright \mathtt{r}$ exists. Note that $G' \upharpoonright \mathtt{r}$ exists regardless of $\mathtt{r}$'s participation in G'.

   By induction, $[\![G',\ \mathtt{s}]\!] \upharpoonright \mathtt{r}$ exists.

   To show $[\![\mu\mathtt{t}.G',\ \mathtt{s}]\!] \upharpoonright \mathtt{r}$ exists, consider $\mathtt{r}$ by case:

   - $\mathtt{r} \in \mathtt{pt}([\![G',\ \mathtt{s}]\!])$:

     $$[\![\mu\mathtt{t}.G',\ \mathtt{s}]\!] \upharpoonright \mathtt{r} = \mu\mathtt{t}.[\![G',\ \mathtt{s}]\!] \upharpoonright \mathtt{r} = \mu\mathtt{t}.[\![G',\ \mathtt{s}]\!] \upharpoonright \mathtt{r}$$

     As $[\![G',\ \mathtt{s}]\!] \upharpoonright \mathtt{r}$ exists, so does $[\![\mu\mathtt{t}.G',\ \mathtt{s}]\!] \upharpoonright \mathtt{r}$.

- $\mathtt{r} \notin \mathtt{pt}(\llbracket G', \mathtt{s} \rrbracket)$:

$$\llbracket \mu \mathtt{t}.G', \mathtt{s} \rrbracket \upharpoonright \mathtt{r} = \mu \mathtt{t}.\llbracket G', \mathtt{s} \rrbracket \upharpoonright \mathtt{r} = \mathtt{end}$$

3. $G = \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$

   To determine $\llbracket G, \mathtt{s} \rrbracket$, consider $\mathtt{s}$ by case:

   - $\mathtt{s} \in \{\mathtt{p}, \mathtt{q}\}$:
     Then $\llbracket G, \mathtt{s} \rrbracket = \mathtt{p} \to \mathtt{q} : \{l_i : \llbracket G_i, \mathtt{s} \rrbracket\}_{i \in I}$.
     To show $\llbracket G, \mathtt{s} \rrbracket \upharpoonright \mathtt{r}$ exists, consider $\mathtt{r}$ by case:
     - $\mathtt{r} = \mathtt{p}$: Then $G \upharpoonright \mathtt{p} = \mathtt{q} \oplus \{l_i : G_i \upharpoonright \mathtt{p}\}_{i \in I}$.
       By induction, $\llbracket G_i, \mathtt{s} \rrbracket \upharpoonright \mathtt{p}$ exists for $i \in I$.
       $\llbracket G, \mathtt{s} \rrbracket \upharpoonright \mathtt{p} = \mathtt{q} \oplus \{l_i : \llbracket G_i, \mathtt{s} \rrbracket \upharpoonright \mathtt{p}\}_{i \in I}$.
       As projections of the encoded continuations exist, so does $\llbracket G, \mathtt{s} \rrbracket \upharpoonright \mathtt{p}$.
     - $\mathtt{r} = \mathtt{q}$: Follows similarly from above.
     - $\mathtt{r} \notin \{\mathtt{p}, \mathtt{q}\}$: Then $G \upharpoonright \mathtt{r} = \bigsqcap_{i \in I} G_i \upharpoonright \mathtt{r}$, so the merge exists.
       By induction, $\llbracket G_i, \mathtt{s} \rrbracket \upharpoonright \mathtt{r}$ exists for $i \in I$.
       $\llbracket G, \mathtt{s} \rrbracket \upharpoonright \mathtt{r} = \bigsqcap_{i \in I} \llbracket G_i, \mathtt{s} \rrbracket \upharpoonright \mathtt{r}$, and this merge exists by Lemma A.8.
   - $\mathtt{s} \notin \{\mathtt{p}, \mathtt{q}\}$:
     Then $\llbracket G, \mathtt{s} \rrbracket = \mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \{l_i : \llbracket G_i, \mathtt{s} \rrbracket\}_{i \in I}$.
     To show $\llbracket G, \mathtt{s} \rrbracket \upharpoonright \mathtt{r}$ exists, consider $\mathtt{r}$ by case:
     - $\mathtt{r} = \mathtt{p}$: Then $G \upharpoonright \mathtt{p} = \mathtt{q} \oplus \{l_i : G_i \upharpoonright \mathtt{p}\}_{i \in I}$.
       By induction, $\llbracket G_i, \mathtt{s} \rrbracket \upharpoonright \mathtt{p}$ exists for $i \in I$.
       $\llbracket G, \mathtt{s} \rrbracket \upharpoonright \mathtt{p} = \mathtt{q}_\mathtt{s} \oplus \{l_i : \llbracket G_i, \mathtt{s} \rrbracket \upharpoonright \mathtt{p}\}_{i \in I}$.
       As projections of the encoded continuations exist, so does $\llbracket G, \mathtt{s} \rrbracket \upharpoonright \mathtt{p}$.
     - $\mathtt{r} = \mathtt{q}$: Follows similarly from above.
     - $\mathtt{r} = \mathtt{s}$: Then $G \upharpoonright \mathtt{s} = \bigsqcap_{i \in I} G_i \upharpoonright \mathtt{s}$.
       By induction, $\llbracket G_i, \mathtt{s} \rrbracket \upharpoonright \mathtt{p}$ exists for $i \in I$.
       $\llbracket G, \mathtt{s} \rrbracket \upharpoonright \mathtt{s} = \mathtt{p} \hookrightarrow \mathtt{q} : \{l_i : \llbracket G_i, \mathtt{s} \rrbracket \upharpoonright \mathtt{s}\}_{i \in I}$.
       As projections of the encoded continuations exist, so does $\llbracket G, \mathtt{s} \rrbracket \upharpoonright \mathtt{s}$.
     - $\mathtt{r} \notin \{\mathtt{p}, \mathtt{q}, \mathtt{s}\}$: Then $G \upharpoonright \mathtt{r} = \bigsqcap_{i \in I} G_i \upharpoonright \mathtt{r}$, so the merge exists.
       By induction, $\llbracket G_i, \mathtt{s} \rrbracket \upharpoonright \mathtt{r}$ exists for $i \in I$.
       $\llbracket G, \mathtt{s} \rrbracket \upharpoonright \mathtt{r} = \bigsqcap_{i \in I} \llbracket G_i, \mathtt{s} \rrbracket \upharpoonright \mathtt{r}$, and this merge exists by Lemma A.8.

□

Theorem 8.3 (Encoding Preserves Well-Formedness)**.** *Let G be a global type, and $\mathtt{s}$ be a role.*

$$\mathtt{wellFormed}(G) \Longrightarrow \mathtt{wellFormed}(\llbracket G, \mathtt{s} \rrbracket, \mathtt{s})$$

Proof. Direct consequence of Lemmas 8.4 and 8.5 and Definition 8.5. □

8.4.3 Preserving Communication

We present a crucial result that directly addresses the pitfalls discussed in Section 7.3 – namely, that our encoding does not over-serialise the original communication. We show this through proving that our encoding preserves the LTS semantics over global types – or more precisely, we can use the encodings over global types and LTS actions to encode all possible transitions in the LTS for global types in the canonical MPST theory.

Theorem 8.4 (Encoding Preserves Semantics)**.** *Let G, G' be global types such that $G \xrightarrow{l} G'$ for some label l.*

$$\forall l, \mathtt{s}.\ \left(G \xrightarrow{l} G' \Longrightarrow [\![G,\ \mathtt{s}]\!] \xrightarrow{[\![l,\ \mathtt{s}]\!]} [\![G',\ \mathtt{s}]\!] \right)$$

Proof. By induction on the structure of $G \xrightarrow{l} G'$. Take arbitrary router role $\mathtt{s}$.

- [GR1] , where $G = \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}, G' = \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}, l = \mathtt{pq}!j$.

  To show $[\![G,\ \mathtt{s}]\!] \xrightarrow{[\![l,\ \mathtt{s}]\!]} [\![G',\ \mathtt{s}]\!]$, consider $\mathtt{s}$ by case:

  - $\mathtt{s} \in \{\mathtt{p}, \mathtt{q}\}$: Then we have

    $$\begin{aligned} [\![G,\ \mathtt{s}]\!] &= \mathtt{p} \to \mathtt{q} : \{l_i : [\![G_i,\ \mathtt{s}]\!]\}_{i \in I} \\ [\![G',\ \mathtt{s}]\!] &= \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : [\![G_i,\ \mathtt{s}]\!]\}_{i \in I} \\ [\![l,\ \mathtt{s}]\!] &= \mathtt{pq}!j \end{aligned}$$

    The encoded transition is possible using [GR1] .
  - $\mathtt{s} \notin \{\mathtt{p}, \mathtt{q}\}$: Then we have

    $$\begin{aligned} [\![G,\ \mathtt{s}]\!] &= \mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \{l_i : [\![G_i,\ \mathtt{s}]\!]\}_{i \in I} \\ [\![G',\ \mathtt{s}]\!] &= \mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : [\![G_i,\ \mathtt{s}]\!]\}_{i \in I} \\ [\![l,\ \mathtt{s}]\!] &= \mathtt{via}_{\mathtt{s}}(\mathtt{pq}!j) \end{aligned}$$

    The encoded transition is possible using [GR6] .

- [GR2] , where $G = \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}, G' = G_j, l = \mathtt{pq}?j$.

  We know $[\![G',\ \mathtt{s}]\!] = [\![G_j,\ \mathtt{s}]\!]$

  To show $[\![G,\ \mathtt{s}]\!] \xrightarrow{[\![l,\ \mathtt{s}]\!]} [\![G',\ \mathtt{s}]\!]$, consider $\mathtt{s}$ by case:

– $\mathtt{s} \in \{\mathtt{p}, \mathtt{q}\}$: Then we have

$$[\![G,\ \mathtt{s}]\!] = \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : [\![G_i,\ \mathtt{s}]\!]\}_{i \in I}$$
$$[\![l,\ \mathtt{s}]\!] = \mathtt{pq}?j$$

The encoded transition is possible using [GR2] .

– $\mathtt{s} \notin \{\mathtt{p}, \mathtt{q}\}$: Then we have

$$[\![G,\ \mathtt{s}]\!] = \mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{q}.\ j : \{l_i : [\![G_i,\ \mathtt{s}]\!]\}_{i \in I}$$
$$[\![l,\ \mathtt{s}]\!] = \mathtt{via}_{\mathtt{s}}(\mathtt{pq}?j)$$

The encoded transition is possible using [GR7] .

- [GR3] , where $G = \mu \mathtt{t}.G''$.

  By hypothesis, $G''[\mu \mathtt{t}.G''/\mathtt{t}] \xrightarrow{l} G'$.

  By induction, $[\![G''[\mu \mathtt{t}.G''/\mathtt{t}],\ \mathtt{s}]\!] \xrightarrow{[\![l,\ \mathtt{s}]\!]} [\![G',\ \mathtt{s}]\!]$.

  By Lemma A.3, $[\![G''[\mu \mathtt{t}.G''/\mathtt{t}],\ \mathtt{s}]\!] = [\![G'',\ \mathtt{s}]\!][\mu \mathtt{t}.[\![G'',\ \mathtt{s}]\!]/\mathtt{t}]$.

  We know $[\![G,\ \mathtt{s}]\!] = [\![\mu \mathtt{t}.G'',\ \mathtt{s}]\!] = \mu \mathtt{t}.[\![G'',\ \mathtt{s}]\!]$.

  The encoded transition is possible using [GR3] as shown:

$$\frac{[\![G'',\ \mathtt{s}]\!][\mu \mathtt{t}.[\![G'',\ \mathtt{s}]\!]/\mathtt{t}] \xrightarrow{[\![l,\ \mathtt{s}]\!]} [\![G',\ \mathtt{s}]\!]}{\mu \mathtt{t}.[\![G'',\ \mathtt{s}]\!] \xrightarrow{[\![l,\ \mathtt{s}]\!]} [\![G',\ \mathtt{s}]\!]}\ [\mathrm{GR3}]$$

- [GR4] , where $G = \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}, G' = \mathtt{p} \to \mathtt{q} : \left\{l_i : G_i'\right\}_{i \in I}$.

  By hypothesis, $\forall i \in I.\ G_i \xrightarrow{l} G_i'$ and $\mathrm{subj}(l) \notin \{\mathtt{p}, \mathtt{q}\}$.

  By induction, $\forall i \in I.\ \left([\![G_i,\ \mathtt{s}]\!] \xrightarrow{[\![l,\ \mathtt{s}]\!]} [\![G_i',\ \mathtt{s}]\!]\right)$.

  By Definition 8.6, $\mathrm{subj}([\![l,\ \mathtt{s}]\!]) \notin \{\mathtt{p}, \mathtt{q}\}$.

  To show $[\![G,\ \mathtt{s}]\!] \xrightarrow{[\![l,\ \mathtt{s}]\!]} [\![G',\ \mathtt{s}]\!]$, consider $\mathtt{s}$ by case:

  – $\mathtt{s} \in \{\mathtt{p}, \mathtt{q}\}$: Then we have

$$[\![G,\ \mathtt{s}]\!] = \mathtt{p} \to \mathtt{q} : \{l_i : [\![G_i,\ \mathtt{s}]\!]\}_{i \in I}$$

$$[\![G',\ \mathtt{s}]\!] = \mathtt{p} \to \mathtt{q} : \left\{ l_i : [\![G'_i,\ \mathtt{s}]\!] \right\}_{i \in I}$$

The encoded transition is possible using [GR4] .

– $\mathtt{s} \notin \{\mathtt{p}, \mathtt{q}\}$: Then we have

$$[\![G,\ \mathtt{s}]\!] = \mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \{ l_i : [\![G_i,\ \mathtt{s}]\!] \}_{i \in I}$$
$$[\![G',\ \mathtt{s}]\!] = \mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \left\{ l_i : [\![G'_i,\ \mathtt{s}]\!] \right\}_{i \in I}$$

The encoded transition is possible using [GR8] .

- [GR5] , where $G = \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{l_i : G_i\}_{i \in I}, G' = \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \left\{ l_i : G'_i \right\}_{i \in I}$.

  By hypothesis, $G_j \xrightarrow{l} G'_j$, $\mathtt{p}' \neq \mathrm{subj}(l)$, and $\forall i \in I \setminus \{j\}.\ G'_i = G_i$.

  By induction, $[\![G_j,\ \mathtt{s}]\!] \xrightarrow{[\![l,\ \mathtt{s}]\!]} [\![G'_j,\ \mathtt{s}]\!]$.

  By Definition 8.6, $\mathrm{subj}([\![l,\ \mathtt{s}]\!]) \neq \mathtt{q}$.

  To show $[\![G,\ \mathtt{s}]\!] \xrightarrow{[\![l,\ \mathtt{s}]\!]} [\![G',\ \mathtt{s}]\!]$, consider $\mathtt{s}$ by case:

  – $\mathtt{s} \in \{\mathtt{p}, \mathtt{q}\}$: Then we have

$$[\![G,\ \mathtt{s}]\!] = \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \{ l_i : [\![G_i,\ \mathtt{s}]\!] \}_{i \in I}$$
$$[\![G',\ \mathtt{s}]\!] = \mathtt{p} \rightsquigarrow \mathtt{q}.\ j : \left\{ l_i : [\![G'_i,\ \mathtt{s}]\!] \right\}_{i \in I}$$

  The encoded transition is possible using [GR5] .

  – $\mathtt{s} \notin \{\mathtt{p}, \mathtt{q}\}$: Then we have

$$[\![G,\ \mathtt{s}]\!] = \mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{q}.\ j : \{ l_i : [\![G_i,\ \mathtt{s}]\!] \}_{i \in I}$$
$$[\![G',\ \mathtt{s}]\!] = \mathtt{p} \underset{\mathtt{s}}{\rightsquigarrow} \mathtt{q}.\ j : \left\{ l_i : [\![G'_i,\ \mathtt{s}]\!] \right\}_{i \in I}$$

  The encoded transition is possible using [GR9] .

□

We motivated the risk of over-serialising communication with a naive routing definition in Section 7.3. We revisit the example and show that our encoding preserves valid traces in the original communication.

Example: Encoding Preserves Semantics
Consider the global type

$$G = \mathtt{p} \to \mathtt{q} : \mathtt{M1} \,.\, \mathtt{s} \to \mathtt{q} : \mathtt{M2} \,.\, \mathtt{end}$$

We apply our encoding with respect to the router role `s`

$$[\![G,\ \mathtt{s}]\!] = \mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \mathtt{M1} \,.\, \mathtt{s} \to \mathtt{q} : \mathtt{M2} \,.\, \mathtt{end}$$

Recall that $l = \mathtt{sq!M2}$ can reduce G.

$$\dfrac{\dfrac{}{(\mathtt{s} \to \mathtt{q} : \mathtt{M2} \,.\, \mathtt{end}) \xrightarrow{l} \mathtt{end}}\,[\textsc{Gr}1] \qquad \mathrm{subj}(l) = \mathtt{s} \notin \{\mathtt{p}, \mathtt{q}\}}{(\mathtt{p} \to \mathtt{q} : \mathtt{M1} \,.\, \mathtt{s} \to \mathtt{q} : \mathtt{M2} \,.\, \mathtt{end}) \xrightarrow{l} (\mathtt{p} \to \mathtt{q} : \mathtt{M1} \,.\, \mathtt{s} \rightsquigarrow \mathtt{q} : \mathtt{M2} \,.\, \mathtt{end})}\,[\textsc{Gr}4]$$

We show that $[\![l,\ \mathtt{s}]\!] = l$ can reduce $[\![G,\ \mathtt{s}]\!]$.

$$\dfrac{\dfrac{}{(\mathtt{s} \to \mathtt{q} : \mathtt{M2} \,.\, \mathtt{end}) \xrightarrow{l} \mathtt{end}}\,[\textsc{Gr}8] \qquad \mathrm{subj}(l) = \mathtt{s} \notin \{\mathtt{p}, \mathtt{q}\}}{\left(\mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \mathtt{M1} \,.\, \mathtt{s} \to \mathtt{q} : \mathtt{M2} \,.\, \mathtt{end}\right) \xrightarrow{l} \left(\mathtt{p} \underset{\mathtt{s}}{\to} \mathtt{q} : \mathtt{M1} \,.\, \mathtt{s} \rightsquigarrow \mathtt{q} : \mathtt{M2} \,.\, \mathtt{end}\right)}\,[\textsc{Gr}4]$$

$\mathtt{sq!M2}$ is a prefix of a valid execution trace for G, given below.

$$G \xrightarrow{\mathtt{sq!M2}} \xrightarrow{\mathtt{pq!M1}} \xrightarrow{\mathtt{pq?M1}} \xrightarrow{\mathtt{sq?M2}} \mathtt{end}$$

Interested readers can verify that the encoded trace (given below) is a valid execution trace for $[\![G,\ \mathtt{s}]\!]$.

$$[\![G,\ \mathtt{s}]\!] \xrightarrow{\mathtt{sq!M2}} \xrightarrow{\mathtt{via_s(pq!M1)}} \xrightarrow{\mathtt{via_s(pq?M1)}} \xrightarrow{\mathtt{sq?M2}} \mathtt{end}$$

8.5 Summary

We have presented RoutedSessions, a variant of the canonical MPST theory to express *routed communication*. We introduced extensions to syntax and semantics, and proved that our extended semantics are sound and complete, and preserve deadlock-freedom for well-formed protocols. We defined an encoding from the canonical theory onto RoutedSessions, and proved the preservation of well-formedness and communication.

Chapter 9

Implementing RoutedSessions in SessionTS

In this chapter, we explain how SessionTS is extended to implement RoutedSessions. As motivated in Section 7.2, the routing mechanism is intended to be transparent to the developer. Hence, there are *no changes* made to the generated APIs; we only need to adapt how the session runtime of the respective endpoints perform send and receive actions. We present the changes made to the session runtime generated by NodeMPST and ReactMPST in Sections 9.1 and 9.2 respectively.

9.1 Extending NodeMPST

By enabling peer-to-peer communication between browser endpoints through the server, not all messages received by the server are intended for the server role. We distinguish these by requiring all messages sent by browser endpoints to specify the intended recipient under the `role` property. We specify this through constructing an interface defining the structure of a "general" message received by the server. Since this interface must match *both* routed messages and messages intended for the server, we have to respect the fact that the server does not need to know the contents of routed messages, so the existing `label` and `payload` properties must be loosely typed to respect this notion of privacy.

```
// Inside the Message namespace...
export interface Channel {
    role: Roles.All, label: string, payload: any[]
};

// Build message object to send through WebSocket
export const toChannel =
    (role: Roles.All, label: string, payload: any[]) => ({
    role, label, payload
});
```

We can now adapt the `onmessage` WebSocket event listener defined at the server endpoint to route messages not intended for the server role, as shown in Listing 9.1. By construction of our API generation strategy, the incoming message must match

the interface defined above, so it is a safe deserialisation step. This preliminary deserialisation allows the server to identify *routed messages*, and proceed to send it to the intended recipient.

```
// Top-level parsing to distinguish between routed messages
const { role, label, payload } =
    JSON.parse(data) as Message.Channel;
if (role !== Roles.Self) {
    // Route message
    this.send(role, label, payload, from);
} else {
    // Invoke message handler as before, see Section 4.4.4
}
```

Listing 9.1: Modified `onmessage` Event Listener for NODEMPST for Routing

Browser endpoints also expect to receive messages with the original sender explicitly marked under the `role` property, so we modify the `send()` method generated for the server endpoint to attach this extra piece of information. To maximise compatibility with the existing runtime implementation, we define the server as the default argument for the `from` parameter of the `send()` method, as highlighted below.

```
send(to: Roles.Peers, label: string, payload: any[],
    from: Roles.All = Roles.Self) {
    const message = Message.toChannel(from, label, payload);
    this.roleToSocket[to].send(JSON.stringify(message));
}
```

ROUTEDSESSIONS introduces an additional concern for managing the terminal state of the server's EFSM: when the server's EFSM reaches its terminal state, it does not mean other endpoints have reached their terminal state, as they can still interact amongst themselves. The server will need to remain connected to perform its routing duties. As our existing design choice already specifies that browser endpoints initiate the WebSocket close event when they reach their terminal state, we do not need to worry about this additional concern as the server will remain connected when it reaches its terminal state to carry out its routing duties anyway.

9.2 Extending REACTMPST

Similarly, the session runtime for browser endpoints need to specify the intended recipient when sending a message through the WebSocket connection, in order for the server endpoint to route the message accordingly. Likewise, when registering receive handlers, the browser endpoint also needs to specify the role to receive from in addition to the handler.

As explained in Section 5.4.4, the current implementation of REACTMPST has a pair of queues for messages waiting for handlers and vice-versa. Because ROUTEDSESSIONS allows browser endpoints to interact with non-server endpoints, this existing framework may lead to communication mismatch.

For example, consider the sequence of routed interactions between server endpoint S and *browser* endpoints A, B and C:

$$A \underset{S}{\rightarrow} B : \texttt{M1(number)}.C \underset{S}{\rightarrow} B : \texttt{M2(string)}.\texttt{end}$$

From the perspective of B, B will register a receive handler expecting to process `M1(number)` from A first. However, communication will be routed by some server endpoint S, and by RoutedSessions theory, S can route C's `M2(string)` message to B before A's `M1(number)` message, so the existing message receiving framework will use the receive handler defined for `M1(number)` to process the received `M2(string)`, which is a mismatch.

Hence, we need to implement the message receiving framework outlined in Section 4.4.4 instead, which generalises the pair of message/handler queues to two mappings to provide a pair of message/handler queues for each endpoint. The runtime instantiates the two mappings of queues, and receive handler registration now targets the pair of queues specified by the role to receive from. We illustrate this in Listing 9.2

```
class Client extends React.Component<...> {
    private messageQueue: RoleToMessageQueue;
    private handlerQueue: RoleToHandlerQueue;

    // Omitting unchanged methods

    private registerReceiveHandler(
        role: Roles.Peers, handle: ReceiveHandler) {
        const message = this.messageQueue[role].shift();
        if (message !== undefined) {
            // Message received already -- process as before
        } else {
            // No message received -- enqueue handler
            this.handlerQueue[role].push(handle);
        }
    }
}
```

Listing 9.2: Runtime Extensions for ReactMPST to Implement RoutedSessions

Referring to the previous example under this framework, S can route C's message to B first, but `M2(string)` will be stored under the message queue mapped to C, so it will not be processed until B registers the receive handler for processing C's message. This cannot happen until B processes `M1(number)` from A, so the order of communication specified in the global protocol is respected.

Part III

Evaluation and Conclusion

Chapter 10

Evaluation

In this chapter, we evaluate the expressiveness and performance of our generated APIs with respect to the main objective of our project in providing developers with a development workflow that offers communication safety guarantees in modern web programming through multiparty session types.

Expressiveness

To evaluate the expressiveness of our work, we use two case studies of protocols found in web services to demonstrate strengths and weaknesses of our work:

- **NOUGHTS AND CROSSES Game, as introduced in [40]:**

  In Section 10.1, we implement a classic multiplayer game which involves a game server and two players interacting with the game via the browser, to show that our work is compatible with multiparty sessions, which is not the case in [19, 20]. We show how our generated APIs are compatible with the state management solution used by the game, and give examples of how the generated APIs empower the developer to intuitively implement the game interface.

- **TWO BUYERS Protocol, as introduced in Section 7.1 and [25, 26]:**

  In Section 10.2, we walk through an implementation of the TWO BUYERS protocol. This *cannot* be implemented in [19, 20, 32], which illustrates the novelty of our ROUTEDSESSIONS theory on routed multiparty session types. We emphasise how the routing mechanism is transparent to the developer implementing the protocol, which demonstrates the practicality of our extensions to SESSIONTS to support ROUTEDSESSIONS.

Performance

To evaluate the performance of our work, we run micro-benchmarks on a variant of the PING PONG protocol parameterised by the number of round trips to analyse the overhead of our implementation, compared with implementations written without using our generated APIs, as the number of round trips increase. We describe the experiment methodology and comment on our findings in Section 10.3.

10.1 Multiparty Sessions: NOUGHTS AND CROSSES

We implement the classic turn-based board game of *Noughts and Crosses* between two players, as introduced in [40]. Both players, identified by either *noughts (O's)* or *crosses (X's)* respectively, take turns to place a mark on an unoccupied cell of a 3-by-3 grid until one player wins (when their markers form one straight line on the board) or a stalemate is reached (when all cells are occupied and no one wins).

We formalise the game interactions using a Scribble protocol presented in Listing 10.1. The GAME protocol describes one turn: `P1` makes a move by sending the coordinates of a vacant cell on the game board to `Svr`, then `Svr` reports the outcome of that move to both players. If another round is required to determine the game result, the GAME protocol is recursively invoked (Line 18) with roles `P1` and `P2` swapped.

```
1  module NoughtsAndCrosses;
2
3  // TypeScript definition:
4  // interface Point {x: number, y: number}
5  type <typescript> "Coordinate" from "./Types" as Point;
6
7  global protocol Game(role Svr, role P1, role P2) {
8      Pos(Point) from P1 to Svr;
9      choice at Svr {
10         Lose(Point)   from Svr to P2;
11         Win(Point)    from Svr to P1;
12     } or {
13         Draw(Point)   from Svr to P2;
14         Draw(Point)   from Svr to P1;
15     } or {
16         Update(Point) from Svr to P2;
17         Update(Point) from Svr to P1;
18         do Game(Svr, P2, P1);
19     }
20 }
```

Listing 10.1: The NOUGHTS AND CROSSES Protocol

We focus on the implementation details that best illustrate the expressiveness of our work; the interested reader can find the full implementation on GitHub[1], and consult `README.md` to navigate between the generated code and developer implementation.

10.1.1 Game Server

We set up the WebSocket server as an *Express.js*[2] [18] application on top of the Node.js runtime. We define our own game logic in a `Board` class to keep track of the

[1] `https://github.com/ansonmiu0214/SessionTS-Examples/NoughtsAndCrosses`
[2] Express is a commonly used library for writing lightweight web servers in JavaScript.

game state and expose methods to query the result – the implementation for `Board` is included in Appendix B.1. This custom logic is integrated into the `handleP1Move` and `handleP2Move` handlers implemented by the developer, defined to handle the moves made by `P1` and `P2` respectively. We illustrate this in Listing 10.2.

```
const handleP1Move = new Implementation.S13({
  Pos: async (move: Point) => {
    // `board` manages game state;
    // `board.P1` registers the move and returns the game result
    const result = await board.P1(move);
    switch (result) {
      case MoveResult.Win: {
        return new Implementation.S15([
          // Send losing result to P2
          [Labels.S15.Lose, [move], new Implementation.S16(
            // Send winning result to P1
            [Labels.S16.Win, [move], new Implementation.Terminal()]
          )]
        ]);
      }
      case MoveResult.Draw: { ... }
      case MoveResult.Continue: {
        return new Implementation.S15(
          // Notify both players and proceed
          // with next round using `handleP2Move`
          [Labels.S15.Update, [move], new Implementation.S18(
            [Labels.S18.Update, [move], handleP2Move]
          )]
        );
      }
    }
  }
});

const handleP2Move = ...    // Defined similarly as handleP1Move

const cancellation = (role: Roles.All, reason: string) => {
  console.log(`${role} cancelled because of ${reason}`);
}

new Svr(
  wss,              // WebSocket server
  handleP1Move,     // Game logic
  cancellation      // Error handler
);
```

Listing 10.2: Implementing NOUGHTS AND CROSSES Game Server

When the server receives a move, it notifies the game logic to update the game state and return the game result caused by that move. The game logic is likely to keep track of move history using a database; we simulate this with a delay, so the game result returned by the game logic is a `Promise`. The expressiveness of our generated APIs enable the developer to define the handlers as `async` functions

to use the asynchronous game logic API intuitively – this is something prevalent in modern web programming, but not directly addressed in existing session type implementations for web development [19, 32].

10.1.2 Game Players

For simplicity, our game uses the same implementation for both `P1` and `P2`, although they can be different in theory – the developer could implement `P1` using a GUI and provide `P2` with a text-based game experience on the browser.

The main implementation detail for players is to make moves. Intuitively, the developer implements a grid and binds a handler to the `'onClick'` event of each vacant cell to send that cell's coordinate in a `Pos(Point)` message to the game server. A common source of bugs would be not preventing the user from selecting a second cell when waiting for the game server's response, which violates the game rules (and the global protocol).

Our approach of providing *component factories* for send states in ReactMPST makes this very intuitive and guarantees communication safety. First, it gives the developer the flexibility to trigger the same send action (in this case, `Pos(Point)` via multiple UI elements – the developer can generate a send action wrapper component for each vacant cell on the game board. Moreover, each generated wrapper component sends a different payload corresponding to the coordinates of the cell: our generated APIs support this as the handler supplied to the send component factory can access the cell's coordinates in the closure. Finally, the send action is always followed by a transition to the receive state component, so the user cannot violate channel linearity by selecting two cells.

We demonstrate how this works in Listing 10.3. The factory function for binding the `Pos(Point)` send action is defined under `this.props.Pos`. For each x-y coordinate on the game board, if the cell is vacant, we create a `<SelectPoint>` React component from the component factory function (which reads "build a react component that sends the `Pos` message with x-y coordinate as payload when the user clicks on it"), and we wrap a `<td>` table cell (since the game board is rendered as an HTML table) inside the generated component to bind the click event to the table cell.

The session cancellation handler allows the developer to render useful messages to the player, since a different component can be rendered depending on whether the server or the opposition has disconnected, and make *application-specific* interpretations of the cancellation. For example, if the opposition has disconnected, the developer can interpret this as a forfeit and render a winning message to the user.

10.1.3 Summary

We demonstrated how the developer can use the generated APIs from SessionTS to implement a complex multiparty protocol which features branching, selection and recursion. We highlighted specific features in the generated APIs for both server and browser endpoints that allow the developer to intuitively implement their application logic. In particular, we observed that the extensions introduced in Chapter 6

```
// Inside some render() function...
{board.map((row, x) =>
  <tr>
  {row.map((cell, y) => {
    if (cell === Cells.VACANT) {
      const sendPoint = (event: React.MouseEvent) => {
        return { x, y };
      });
      const SelectPoint = this.pros.Pos('onClick', sendPoint);
      return <SelectPoint><td>{cell}</td></SelectPoint>
    } else {
      // Render nought or cross,
      // but clicking on this cell will *not* send anything
      return <td>{cell}</td>
    }
  })}
  </tr>
)}
```

Listing 10.3: Safely Binding Send Actions to NOUGHTS AND CROSSES Game Board

play crucial roles in improving the usability of the generated APIs when compared with existing work on session types for web development [5, 19, 32].

Code is available at `https://github.com/ansonmiu0214/SessionTS-Examples/NoughtsAndCrosses`.

10.2 Routed Multiparty Sessions: TWO BUYERS

We implement the TWO BUYER protocol introduced in Section 7.1. This protocol cannot be implemented using existing proposals [19, 32] for integrating session types into web development. SESSIONTS overcomes the limitations of existing work through implementing novel theory of routed multiparty session types formalised in Chapter 8; the implementation of ROUTEDSESSIONS in SESSIONTS is explained in Chapter 9.

We present the implementation of the `Seller` in Listing 10.4, and the implementation of the peer-to-peer interaction by buyer `A` in Listing 10.5 The main point to note is that the routing mechanism is completely transparent to the developer, which shows the elegance of our solution.

The ROUTEDSESSIONS implementation does not affect the compatibility of our generated APIs with external libraries. The `Seller` endpoint is set up as an Express.js application, and both buyers use the *React Context API*[3] for application state management.

Interested readers can find the full implementations for the other endpoints alongside the generated code on GitHub[4].

[3] React Contexts allow components to pass data (such as application state) through the component tree without having to propagate via props at every level.

[4] `https://github.com/ansonmiu0214/SessionTS-Examples/TwoBuyer`

```
new Implementation.Initial({
  [Labels.S32.title]: async (title) => {
    const quote = await db.getQuote(title);
    return new Implementation.S34([
      Labels.S34.quote, [quote], new Implementation.S35([
        Labels.S35.quote, [quote], new Implementation.S36({
          [Labels.S36.buy]: () => {
            return new Implementation.Terminal();
          },
          [Labels.S36.cancel]: () => {
            return new Implementation.Terminal();
          }
        })
      ])
    ])
  }
})
```

Listing 10.4: Two Buyer Seller Implementation

```
type State = { split: number };

class ProposeSplit extends S11<State> {
    state = { split: 0 };

    render() {
        const SendSplit = this.split('onClick', ev => {
            return [this.state.split];
        });

        return <div>
            <input
                type='number'
                value={this.state.split}
                onChange={ev => this.setState({
                    split: Number(ev.target.value)
                })}
                placeholder='Enter split'
            />
            <SendSplit><button>Propose</button></SendSplit>
        </div>;
    }
}
```

Listing 10.5: Developer Implementation of Peer-to-Peer Interaction in Two Buyer

10.3 Performance Benchmarks

Whilst web applications implementing our generated APIs enjoy communication safety guarantees, the presence of the session runtime acts as an additional layer of abstraction between the application logic and the WebSocket transport, which

presents a performance trade-off; we observe an increase in the time taken to perform each channel action, as a result of the session runtime intercepting channel actions and performing additional logic to ensure that the endpoint conforms to the protocol.

```
global protocol PingPong(role Client, role Svr) {
    PING(number) from Client to Svr;
    choice at Svr {
        PONG(number) from Svr to Client;
        do PingPong(Client, Svr);
    } or {
        BYE(number) from Svr to Client;
    }
}
```

Listing 10.6: The PING PONG Protocol

To measure the overhead of our implementation, we compare the execution time of web-based implementations of the Ping Pong protocol (Listing 10.6) with and without our generated APIs.

We parameterise the PING PONG protocol by $n > 0$, the number of round-trip messages. This is standardised in the application logic across experiments. Upon establishing a connection, the experiment proceeds as follow:

1. `Client` sends `PING(`m`:number)` to `Svr`, with $m = 0$ initially.
2. `Svr` receives `PING(`m`:number)`, and conditionally responds based on n:
   (a) If $m + 1 < n$, then `Svr` replies `PONG(`$m+1$`)`. `Client` responds to `PONG` by returning to step 1 with m set as the payload from `PONG`.
   (b) Otherwise, $m+1 = n$, then `Svr` responds with `BYE(`$m+1$`)`, as n round trips have taken place. `Client` responds to `BYE` by closing the connection, thus ending the experiment.

We note that the PING PONG protocol implements a *binary* session. It would be interesting to observe the overhead in a *multiparty* context, but due to limited time constraints, our benchmarking suite does not support multiple browser targets. Benchmarking multiparty protocols would also require writing multiple distinct React applications using the generated APIs – as this is currently a manual process, doing this for multiple roles requires more time than available.

10.3.1 Setup

In order to measure the overhead as accurately as possible, we outline the logic that all implementations must follow:

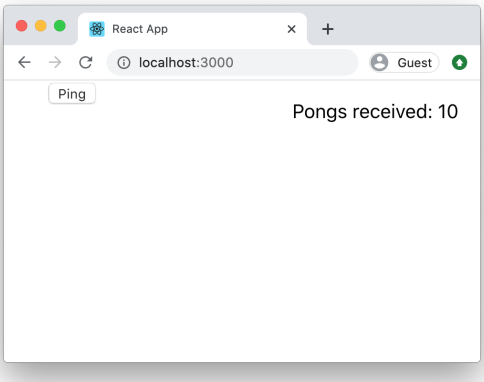


Figure 10.1: User Interface of `Client` Endpoint in PING PONG Protocol

PING PONG `Client` on React:

- All `Clients` implement the same user interface (Figure 10.1), rendering a `<button>` which triggers the send, and a `<div>` captioned with the number of `PONG`s received.
- `Clients` will use the React Context API [8] for application state management, i.e. the number of `PONG`s received. We wrap the session logic in a `<Benchmark>` component that acts as the `ContextProvider` using its component state.
- To automate the benchmark, we use the React Refs API [11] to access the DOM `<button>` node programmatically, in order to simulate the click event and send a `PING` message upon establishing the WebSocket connection, or upon receiving a `PONG`.
- We use the production build generated by *create-react-app* [14] for all experiments, which performs the compilation into JavaScript. We serve the production build using the *serve* package [54] available on *npm*.

PING PONG `Svr` on Node:

- We use the built-in `console.time` function to record the execution time of all experiments. The timer starts when a WebSocket connection has been established at `Server`, and stops when on a `CloseEvent`.
- To observe the execution pattern, the `Svr` will log the running elapsed time for every `PING` message received.
- All `Svrs` run the benchmarks without a real web browser, using headless browsing functionality from the *Zombie.js* [1] package.
- `Svr` logic is parameterised by the number of round trips, n, configured through an environment variable passed through the command line.

- We use the compiled JavaScript versions of all `Svr`s for the experiments.

Code is available at `https://github.com/ansonmiu0214/SessionTS-Benchmarks`. Interested readers may follow the `README.md` to run the benchmarks and visualise the logs using the interactive notebook in the same directory.

We run the experiments under a network of latency 0.165ms (64 bytes ping), and repeat each experiment 20 times. Execution time measurements are taken using a machine equipped with Intel i7-4850HQ CPU (2.3 GHz, 4 cores, 8 threads), 16 GB RAM, macOS operating system version 10.15.4, Node.js runtime version 12.12.0, and TypeScript compiler version 3.7.4. We standardise all packages used in the front- and back-end implementations across experiments. Details can be found in Appendix B.2.

The benchmark compares three implementations of the PING PONG protocol:

bare: The `bare` implementation directly interfaces with WebSocket primitives for sending and receiving. The implementation executes the PING PONG protocol, but does not guarantee communication safety by construction – e.g. the user can click the `PING` button multiple times before a `PONG` message is received, violating channel linearity. This represents the typical developer implementation without using the MPST framework.

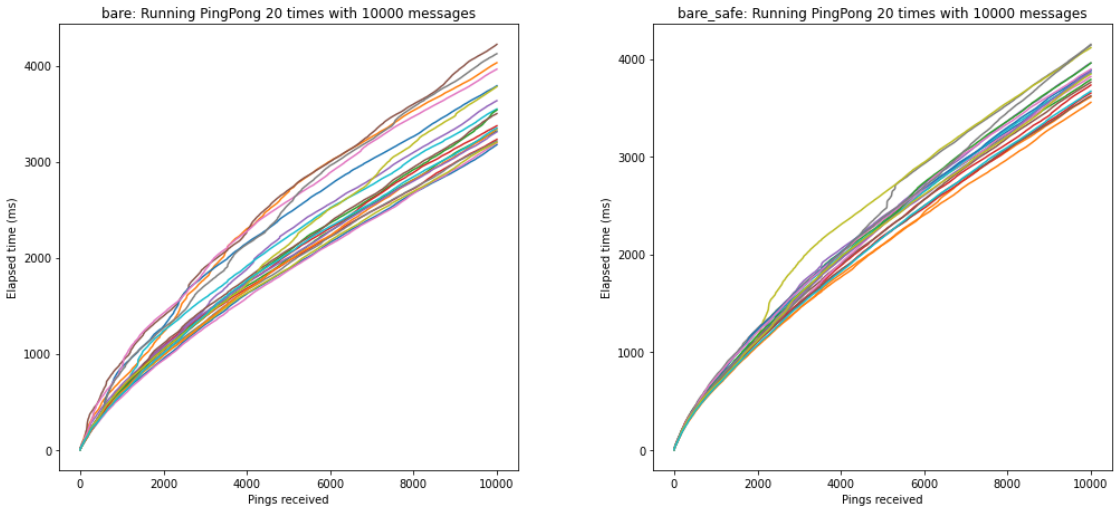
bare_safe: The `bare_safe` implementation also directly interfaces with WebSocket primitives for communication, but assumes the developer implements minimal viable workarounds to address the lack of communication safety. Here, the developer renders an inactive version of the `PING` button when the `PING` message has been sent but a response has yet to be received; a `visible` boolean flag is used to explicitly manage which `<button>` to render.

mpst: The `mpst` implementation uses the APIs generated from SESSIONTS, so it enjoys the communication safety guarantees from our methodology.

10.3.2 Execution Pattern

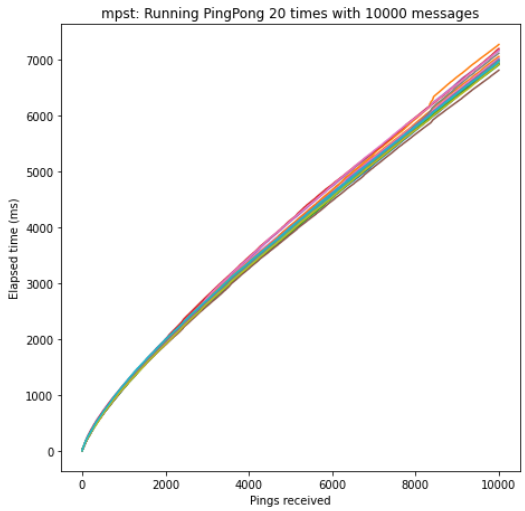
We compare the execution patterns of exchanging 10,000 Ping-Pongs throughout 20 repeated experiments across the three implementations. We visualise the elapsed time with respect to the number of `PING`s received in Figure 10.2: each line represents one execution of the benchmark.

Relative to the two `bare` implementations, the `mpst` version performs more consistently, perhaps as a result of the session runtimes handling all WebSocket interactions in a systematic way. The gradient of the graph represents the rate at which a Ping-Pong round trip takes place. We observe a steeper gradient when the protocol begins, which illustrates the overhead incurred in the session joining phase in our generated APIs. Aside from these factors, all three implementations generally share similar characteristics in their protocol execution.



(a) `bare`

(b) `bare_safe`



(c) `mpst`

Figure 10.2: Comparison of Execution Pattern for 10,000 Ping-Pongs

10.3.3 Overhead

The execution time is the total time taken for the `Client` to connect to the `Server` *and* complete the parameterised number of round trips. We compare the total execution time (*Exec. Time*) and execution time per round trip (*Exec. Time / Ping-Pong*) – averaged over 20 repeated experiments – across the three implementations, for $n \in \{10^2, 10^3, 10^4\}$. We summarise the results in Table 10.1.

We note that the addition of a session runtime for all roles in the `mpst` implementation *does* incur a performance overhead. This is made apparent when looking closely at *Exe. Time / Ping-Pong*; we visualise this in Figure 10.3.

The `mpst` implementation records greater round trip times compared to both `bare`

n	Exec. Time			Exec. Time / Ping-Pong		
	`bare`	`bare_safe`	`mpst`	`bare`	`bare_safe`	`mpst`
10^2	89.64ms	107.09ms	186.23ms	0.90ms	1.07ms	1.86ms
10^3	642.92ms	663.91ms	1155.48ms	0.64ms	0.66ms	1.16ms
10^4	3542.16ms	3837.97ms	7015.25ms	0.35ms	0.38ms	0.70ms

Table 10.1: Comparison of Execution Time for 100, 1,000 and 10,000 Ping-Pongs

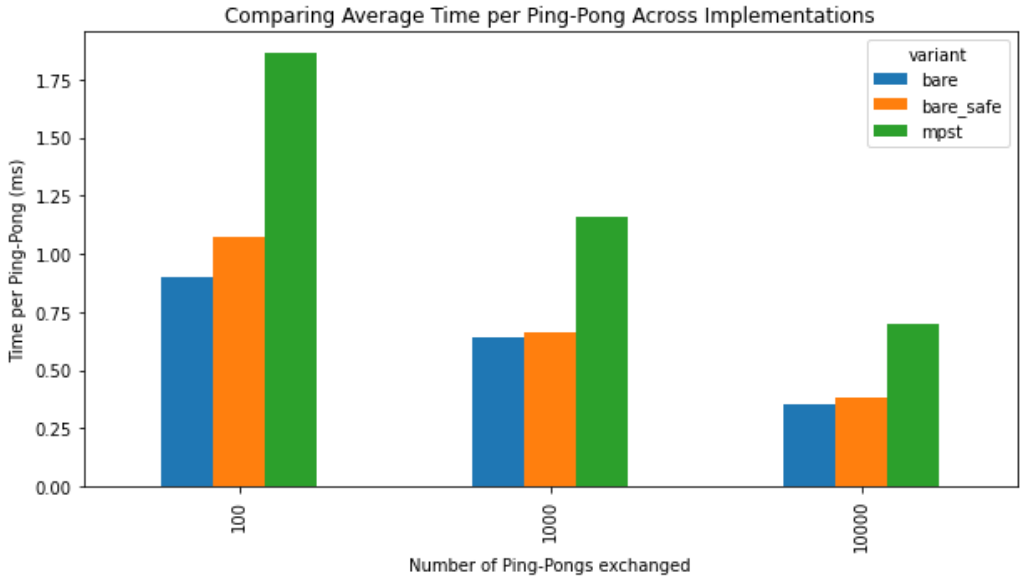


Figure 10.3: Comparing Average Time per Ping-Pong Across Implementations

and `bare_safe` variants. This is expected, as the runtime for *each* role performs additional logic during *both* the sending and receiving of messages. For example, receiving a message involves performing (albeit $O(1)$ time complexity) operations on the message queue and handler queue. As for browser-side implementations, every EFSM transition invokes an updated `render()` on the VDOM, which requires reconciliation internally by React to update the browser DOM accordingly.

We also observe that the (round trip time) performance gap between the `mpst` implementation and the two `bare` implementations narrows as the number of round trips increase. This suggests that the session joining logic in our implementation yields greater overhead than the additional processing logic injected during communication, which is consistent with our findings in Section 10.3.2.

We interpret the overhead as a trade-off between maximising performance and maximising the **static** communication safety guarantees for web applications. In particular, the event-driven nature of browser-side logic makes it highly challenging to guarantee communication safety, as having "active" channel actions on the DOM

allows linearity to be violated by the user, even if one verifies linear channel usage in the source code. The `mpst` implementation properly addresses the problem by statically guaranteeing, by construction, that the UI component rendered on the DOM will only contain channel actions permitted at that state in the EFSM execution.

One may argue that the `bare_safe` implementation also provides said guarantee but adds negligible overhead. By inspecting the workaround in the `bare_safe` implementation (Listing 10.7), it is clear that this does not scale for more complicated protocols with larger EFSMs.

```
click() {
  this.state.ws?.send(JSON.stringify({
    label: 'PING',
    payload: [this.context.count],
  }));
  this.setState({ visible: false });
}

render() {
  return this.state.visible
    ? <button
       ref={this.state.button}
       onClick={this.click.bind(this)}
       >Ping</button>
    : <button>Ping</button>;
}
```

Listing 10.7: Preventing Channel Linearity Violation in `bare_safe` PING PONG

The workaround is tailored to the PING PONG protocol and toggles the `visible` flag to hide the button that triggers a send action when `Client` transitions to a receive state. In fact, this "workaround" precisely generalises to our `mpst` implementation of having some form of wrapper component that renders the UI component corresponding to the current EFSM state. Our API generation strategy formally implements the EFSM for browser-side logic.

We also compare the server-side logic between the `base` and `mpst` implementations in Figure 10.4, and observe that our generated APIs (Figure 10.4b) allow the developer to focus on the implementation detail, whilst the naive bare implementation (Figure 10.4a) chooses to interleave communication logic with application logic, which arguably contributes towards a hidden source of bugs found when implementing more complex protocols.

10.4 Summary

We have demonstrated the expressiveness of our generated APIs through implementing two web services describing multiparty sessions. We highlighted how our approach towards session-typed GUI programming prevents channel reuse and makes it intuitive to implement the game board of NOUGHTS AND CROSSES. We also showed

```
socket.onmessage = ({ data }) => {
  const { label, payload } = JSON.parse(data.toString());
  if (label === 'PING') {
    let count: number = payload[0];
    console.timeLog(LABEL, ++count);
    if (count === MSGS) {
      socket.send(JSON.stringify({
        label: 'BYE',
        payload: [count],
      }));
    } else {
      socket.send(JSON.stringify({
        label: 'PONG',
        payload: [count],
      }));
    }
  } else { throw new Error(`Unrecognised label: ${label}`); }
}
```

(a) `bare` Implementation of PING PONG Svr

```
const logic = new Implementation.S14({
  PING: (count) => {
    console.timeLog(LABEL, ++count);
    if (count === MSGS) {
      return new Implementation.S16([
        Labels.S16.BYE, [count], new Implementation.S15()
      ]);
    } else {
      return new Implementation.S16([
        Labels.S16.PONG, [count], logic
      ]);
    }
  }
});
```

(b) `mpst` Implementation of PING PONG Svr

Figure 10.4: Comparison of `bare` and `mpst` Implementations for PING PONG Svr

that the extensions added to NODEMPST and REACTMPST to support ROUTEDSESSIONS are transparent to the developer.

Through performance benchmarks against a baseline implementation of the PING PONG protocol, we analysed the overhead of our implementation and reasoned that this overhead will minimise when compared with baseline implementations of more complex, multiparty protocols involving routed communication.

Chapter 11

Conclusion

11.1 Contributions

In this project, we develop and present a novel MPST-based framework for developing full-stack interactive TypeScript applications over WebSocket transport.

We motivate our API generation approach from [28, 32] to generate TypeScript APIs from a Scribble protocol specification. By writing their full-stack applications using the generated APIs, developers enjoy communication safety guarantees in their endpoint applications by construction. APIs generated for server-side endpoints are compatible with the Node.js runtime, as detailed in Chapter 4. For browser-side endpoints, we present a novel approach for integrating session types into web-based GUI programming based on translating the theory on model types [19] to idiomatic practices on the React.js framework; this is highlighted in Chapter 5. With respect to session type theory, implementations using the generated APIs statically enjoy linear channel usage guarantees and affine channel usage guarantees for back-end and front-end targets respectively.

Compared to previous work [5, 19, 20, 32] on session-typed web development, we are not only able to statically provide the same level of communication safety guarantees, but we do so using modern web programming practices to increase the relevance and usability of our work in industry: our work targets TypeScript, Node.js and React.js, and we explain in Chapter 6 how we support advanced web development idioms (such as asynchronous implementations) and handle premature disconnections gracefully.

We also do not limit support to protocols that describe a server-centric topology: we formalise RoutedSessions, a theory of routed multiparty session types, in Chapter 8 to prove that it is possible to relax the server-centric topology assumption over WebSocket transport in a way that preserves communication along with the communication safety properties inherited from canonical session type theory.

Our work has received positive feedback from academia. The initial part of this project was published as **Generating Interactive WebSocket Applications in TypeScript** in Volume 314 of the Electronic Proceedings in Theoretical Computer Science (EPTCS) [40], in which I am the first and lead author. The materials in Chapters 3 to 5 are expanded from [40]. The materials in Chapters 6 to 10 are developed exclusively in this project by myself. We were also invited to present a part of this

work at the 12th International Workshop on Programming Language Approaches to Concurrency- & Communication-cEntric Software (PLACES 2020).

Overall, we offer an end-to-end solution for integrating multiparty session type theory into modern full-stack web programming practices, in a way that actively encourages developers to design their application with communication safety in mind. Our formalism of routed multiparty session types enables our API generation tool to support a wider range of protocols, and further reveals the potential for implementing dynamic communication structures over centralised network topologies.

11.2 Future Work

We highlight several areas of future work to support a wider range of communication protocols for web applications and increase the industrial relevance of our work.

Session Delegation Delegation allows channels to be sent between endpoints. Using the NOUGHTS AND CROSSES game from Section 10.1, the developer may want the `Svr` to send a private channel to both players to play the game, whilst concurrently accepting new connections.

The API generation solution presented by Scalas et al. [49] supports *distributed multiparty delegation* in Scala. Their work also uses the Scribble toolchain for protocol specification. Future work can apply the findings from [49] and extend SESSIONTS to support channel delegation in the generated handler-style APIs.

Explicit Connection Actions Hu and Yoshida [29] extended MPST theory with *explicit connection actions* to support protocols with optional participants. Supporting optional participants is a relevant concept for web applications, particularly for browser-side endpoints.

Our work defines a way to handle connection requests and detect disconnection events. The implementation assumes that all participants join the session when the protocol begins. We also focus on disconnection events as a result of session cancellation rather than those stated in the protocol.

The work of Hu and Yoshida [29] uses a variant of the Scribble protocol language that supports explicit connection actions. Future work can use that as a basis for code generation, and generalise our existing mechanisms for session connection and cancellation to support explicit connection actions.

Transport Abstraction The MPST-based web development framework presented by King et al. [32] parameterises the transport mechanism in the session runtime definition. The developer can run the session over different transportation, provided that the custom transport implements the behaviour of the `Transport` type class.

Future work can generalise the session runtime for NODEMPST and REACTMPST to provide similar abstractions. A successful implementation could possibly allow direct interactions between two Node.js-based endpoints, but this will also require

the routed MPST theory to be adapted accordingly to reason about such interactions in a web-based context.

EFSM Encoding as Typestates The concept of *typestates* define the interface of an object to be dependent on its private state, meaning it can change at runtime. This is compatible with the EFSM abstraction from the MPST framework: the permitted methods (i.e. channel actions) of the EFSM depend on its current state. In fact, Kouzapas et al. [33] presents a way to generate typestate specifications for each endpoint of a Scribble protocol; Gay et al. [21] proposes an encoding of session types in object-oriented languages and discusses their approach with respect to typestates.

Future work can explore implementing typestates in TypeScript. A linear type system would be useful towards implementing typestates, but this is lacking in TypeScript. One interesting possibility for approximating this would be to combine *decorators* with *transformers*: the former lets the developer define typestate-related metadata, whilst the latter can parse the metadata and transform the source code into another TypeScript program that acts as an "intermediate representation", such that only implementations that respect the typestate specification will type-check with its intermediate representation.

Bibliography

[1] ARKIN, A. *assaf/zombie*. Jun 2020. Accessed on 14th June 2020. pages 139

[2] BIERMAN, G., ABADI, M., AND TORGERSEN, M. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming* (2014), R. Jones, Ed., Lecture Notes in Computer Science, Springer, p. 257–281. Accessed on 24th November 2019. pages 12, 31, 32

[3] CAIRES, L., AND PÉREZ, J. A. Multiparty Session Types Within a Canonical Binary Theory, and Beyond. In *Formal Techniques for Distributed Objects, Components, and Systems* (2016), E. Albert and I. Lanese, Eds., Lecture Notes in Computer Science, Springer International Publishing, p. 74–95. Accessed on 23rd April 2020. pages 96

[4] CARBONE, M., LINDLEY, S., MONTESI, F., SCHÜRMANN, C., AND WADLER, P. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *27th International Conference on Concurrency Theory (CONCUR 2016)* (2016), J. Desharnais and R. Jagadeesan, Eds., vol. 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, p. 33:1–33:15. Accessed on 23rd April 2020. pages 96

[5] COOPER, E., LINDLEY, S., WADLER, P., AND YALLOP, J. *Links: Web Programming Without Tiers*, vol. 4709. Springer Berlin Heidelberg, 2007, p. 266–296. Accessed on 21st January 2020. pages 11, 13, 31, 64, 136, 145

[6] COPPO, M., DEZANI-CIANCAGLINI, M., PADOVANI, L., AND YOSHIDA, N. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming* (2015), vol. 9104 of *LNCS*, Springer, pp. 146–178. Accessed on 24th November 2019. pages 24

[7] DENIÉLOU, P.-M., AND YOSHIDA, N. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. *arXiv:1304.1902 [cs]* (Apr 2013). arXiv: 1304.1902. Accessed on 5th May 2020. pages 96, 103, 104, 107, 108, 109, 110, 112

[8] FACEBOOK OPEN SOURCE. Context – react. `https://reactjs.org/docs/context.html`. Accessed on 14th June 2020. pages 139

[9] FACEBOOK OPEN SOURCE. Introducing JSX – React. `https://reactjs.org/docs/introducing-jsx.html`. Accessed on 22nd January 2020. pages 65, 66

[10] FACEBOOK OPEN SOURCE. React – A JavaScript library for building user interfaces. `https://reactjs.org/`. Accessed on 21st January 2020. pages 9, 63, 64

[11] FACEBOOK OPEN SOURCE. Refs and the DOM – React. `https://reactjs.org/docs/refs-and-the-dom.html`. Accessed on 14th June 2020. pages 139

[12] FACEBOOK OPEN SOURCE. State and Lifecycle – React. `https://reactjs.org/docs/state-and-lifecycle.html`. Accessed on 7th June 2020. pages 78

[13] FACEBOOK OPEN SOURCE. Strict Mode – React. `https://reactjs.org/docs/strict-mode.html`. Accessed on 7th June 2020. pages 75

[14] FACEBOOK OPEN SOURCE. *facebook/create-react-app*. Facebook, Jun 2020. Accessed on 14th June 2020. pages 139

[15] FEATHERS, M. *Working Effectively with Legacy Code*, 1 edition ed. Prentice Hall, Oct 2004. Accessed on 12th June 2020. pages 44

[16] FENG, R. The Benefits of Migrating From JavaScript to TypeScript - DZone Performance. `https://dzone.com/articles/the-benefits-of-migrating-from-javascript-to-types`. Accessed on 4th June 2020. pages 52

[17] FETTE, I., AND MELNIKOV, A. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011. Accessed on 28th December 2019. pages 9, 15, 88

[18] FOUNDATION, N. Express - Node.js web application framework. `https://expressjs.com/`. Accessed on 14th June 2020. pages 133

[19] FOWLER, S. Model-View-Update-Communicate: Session Types meet the Elm Architecture. *arXiv:1910.11108 [cs]* (Jan 2020). arXiv: 1910.11108. Accessed on 29th May 2020. pages 10, 11, 12, 13, 30, 31, 63, 64, 80, 132, 135, 136, 145

[20] FOWLER, S., LINDLEY, S., MORRIS, J. G., AND DECOVA, S. Exceptional Asynchronous Session Types: Session Types without Tiers. *Proceedings of the ACM on Programming Languages 3*, POPL (Jan 2019), 1–29. Accessed on 28th January 2020. pages 13, 31, 81, 85, 86, 87, 132, 145

[21] GAY, S. J., GESBERT, N., RAVARA, A., AND VASCONCELOS, V. T. Modular session types for objects. *Logical Methods in Computer Science 11*, 4 (Dec 2015), 12. arXiv: 1205.5344. Accessed on 8th June 2020. pages 147

[22] GERBO, R., AND PADOVANI, L. Concurrent Typestate-Oriented Programming in Java. *Electronic Proceedings in Theoretical Computer Science 291* (Apr 2019), 24–34. arXiv: 1904.01286. Accessed on 3rd March 2020. pages 61

[23] GRAPHVIZ. The DOT Language. `https://graphviz.org/doc/info/lang.html`. Accessed on 14th June 2020. pages 36

[24] HONDA, K., AND TOKORO, M. An object calculus for asynchronous communication. In *ECOOP'91 European Conference on Object-Oriented Programming* (1991), P. America, Ed., Springer Berlin Heidelberg, p. 133–147. Accessed on 26th December 2019. pages 16

[25] HONDA, K., YOSHIDA, N., AND CARBONE, M. Multiparty Asynchronous Session Types. In *35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2008), ACM, pp. 273–284. Accessed on 24th November 2019. pages 22, 94, 132

[26] HONDA, K., YOSHIDA, N., AND CARBONE, M. Multiparty Asynchronous Session Types. *Journal of the ACM 63* (2016), 1–67. Accessed on 25th November 2019. pages 94, 132

[27] HU, R. Distributed Programming Using Java APIs Generated from Session Types. 22. Accessed on 18th December 2019. pages 28, 42

[28] HU, R., AND YOSHIDA, N. Hybrid Session Verification through Endpoint API Generation. In *19th International Conference on Fundamental Approaches to Software Engineering* (2016), vol. 9633 of *LNCS*, Springer, pp. 401–418. Accessed on 18th December 2019. pages 25, 27, 28, 34, 35, 61, 62, 85, 145

[29] HU, R., AND YOSHIDA, N. *Explicit Connection Actions in Multiparty Session Types*, vol. 10202. Springer Berlin Heidelberg, 2017, p. 116–133. Accessed on 3rd January 2020. pages 146

[30] JAIN, A. *ajnsit/concur*. May 2020. Accessed on 5th June 2020. pages 30

[31] JINJA. Jinja — Jinja Documentation (2.11.x). `https://jinja.palletsprojects.com/en/2.11.x/`. Accessed on 13th June 2020. pages 37

[32] KING, J., NG, N., AND YOSHIDA, N. Multiparty Session Type-safe Web Development with Static Linearity. *Electronic Proceedings in Theoretical Computer Science 291* (Apr 2019), 35–46. Accessed on 18th December 2019. pages 10, 11, 13, 15, 22, 28, 29, 30, 31, 34, 35, 42, 64, 80, 132, 135, 136, 145, 146

[33] KOUZAPAS, D., DARDHA, O., PERERA, R., AND GAY, S. J. Typechecking protocols with Mungo and StMungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming - PPDP '16* (2016), ACM Press, p. 146–159. Accessed on 15th January 2020. pages 147

[34] MDN CONTRIBUTORS. Fetch API. `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API`. Accessed on 10th June 2020. pages 82

[35] MDN CONTRIBUTORS. Signaling and video calling. `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling`. Accessed on 14th June 2020. pages 95

[36] MDN CONTRIBUTORS. Window.sessionStorage. `https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage`. Accessed on 6th June 2020. pages 71

[37] MERKEL, D. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal 2014*, 239 (Mar 2014), 2:2. Accessed on 12th June 2020. pages 38

[38] MILNER, R. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, New York, NY, USA, 1999. Accessed on 26th December 2019. pages 15

[39] MIU, A. *ansonmiu0214/TypeScript-Multiparty-Sessions*. Jun 2020. Accessed on 14th June 2020. pages 34

[40] MIU, A., FERREIRA, F., YOSHIDA, N., AND ZHOU, F. Generating Interactive WebSocket Applications in TypeScript. *Electronic Proceedings in Theoretical Computer Science 314* (Apr 2020), 12–22. arXiv: 2004.01321. Accessed on 6th April 2020. pages 14, 132, 133, 145

[41] MOSTROUS, D., AND VASCONCELOS, V. T. Affine Sessions. *Logical Methods in Computer Science 14* (2018), Issue 4; 18605974. arXiv: 1809.02781. Accessed on 11th June 2020. pages 86

[42] NEYKOVA, R. Session Types Go Dynamic or How to Verify Your Python Conversations. *Electronic Proceedings in Theoretical Computer Science 137* (Dec 2013), 95–102. arXiv: 1312.2704. Accessed on 20th December 2019. pages 15, 28, 29, 35

[43] NODE.JS. Node.js. `https://nodejs.org/en/`. Accessed on 22nd January 2020. pages 9, 41

[44] OGDEN, M. *maxogden/callback-hell*. Jun 2020. Accessed on 10th June 2020. pages 82

[45] ORACLE. Future (Java Platform SE 7). `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html`. Accessed on 10th June 2020. pages 83

[46] PEZOA, F., REUTTER, J. L., SUAREZ, F., UGARTE, M., AND VRGOČ, D. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web* (2016), International World Wide Web Conferences Steering Committee, pp. 263–273. Accessed on 14th June 2020. pages 54

[47] PURESCRIPT. *purescript/purescript*. PureScript, Jun 2020. Accessed on 5th June 2020. pages 11, 29

[48] PYDOT. *pydot/pydot*. pydot, Jun 2020. Accessed on 14th June 2020. pages 36

[49] SCALAS, A., DARDHA, O., HU, R., AND YOSHIDA, N. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)* (Dagstuhl, Germany, 2017), P. Müller, Ed., vol. 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 24:1–24:31. Accessed on 17th April 2020. pages 28, 34, 35, 96, 146

[50] SCALAS, A., AND YOSHIDA, N. Less Is More: Multiparty Session Types Revisited. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages* (2019), vol. 3, ACM, pp. 30:1–30:29. Accessed on 14th June 2020. pages 23, 98

[51] SIEK, J., AND TAHA, W. Gradual Typing for Objects. In *ECOOP 2007 – Object-Oriented Programming* (2007), E. Ernst, Ed., Springer Berlin Heidelberg, p. 2–27. Accessed on 17th June 2020. pages 32

[52] TECHOPEDIA. Thunk - Definition from Techopedia. `https://www.techopedia.com/definition/2818/thunk-computing`. Accessed on 10th June 2020. pages 84

[53] UBERTI, J., AND THATCHER, P. Webrtc. `https://webrtc.org/`, 2011. Accessed on 14th June 2020. pages 95

[54] VERCEL. *vercel/serve_2020*. Vercel (formerly ZEIT), Jun 2020. Accessed on 14th June 2020. pages 139

[55] XI, H. Applied Type System: An Approach to Practical Programming with Theorem-Proving. *Journal of Functional Programming* (2016), 30. Accessed on 26th December 2019. pages 15

[56] YOSHIDA, N. Lecture Notes in CO406 Concurrent Processes, October 2019. Accessed on 26th December 2019. pages 16, 17, 18, 19, 23

[57] YOSHIDA, N., HU, R., NEYKOVA, R., AND NG, N. The Scribble Protocol Language. In *8th International Symposium on Trustworthy Global Computing* (2013), vol. 8358 of *LNCS*, Springer, pp. 22–41. Accessed on 18th December 2019. pages 12, 25, 26, 35

[58] YOSHIDA, N., AND LORENZO, G. A Very Gentle Introduction to Multiparty Session Types. Accessed on 24th November 2019. pages 10, 12, 17, 21, 22, 23

Appendix A

Lemmas and Proofs

A.1 Lemmas and Proofs for Chapter 8

Lemma A.1 (Local LTS Preserves Merge)**.** *Let T_1, T_2 be local types. Suppose $T_1 \sqcap T_2$ exists.*

$$\forall l, T_1', T_2'.\ \left((T_1 \xrightarrow{l} T_1') \wedge (T_2 \xrightarrow{l} T_2') \Longrightarrow (T_1' \sqcap T_2')\ \mathit{exists}\right)$$

Proof. By simultaneous induction on $T_1 \sqcap T_2$, $T_1 \xrightarrow{l} T_1'$, and $T_2 \xrightarrow{l} T_2'$.

□

Lemma A.2 (Projection and Participation)**.**

$$\forall G, \mathtt{p}.\ (G \upharpoonright \mathtt{p} = \mathtt{end} \Longleftrightarrow \mathtt{p} \notin \mathtt{pt}(G))$$

Proof. Prove ($\Longrightarrow$) by induction on the structure of G. Prove ($\Longleftarrow$) using the contrapositive (stated below) by induction on the derivation of $\mathtt{pt}(G)$.

$$\mathtt{p} \in \mathtt{pt}(G) \Longrightarrow G \upharpoonright \mathtt{p} \neq \mathtt{end}$$

□

Lemma A.3 (Commutativity between Encoding and Substitution)**.** *Let G, G' be global types, and $\mathtt{s}$ be a role.*

$$[\![G[G'/\mathtt{t}],\ \mathtt{s}]\!] = [\![G,\ \mathtt{s}]\!][[\![G',\ \mathtt{s}]\!]/\mathtt{t}]$$

Proof. By induction on the structure of G. □

Lemma A.4 (Encoding Preserves Participants)**.**

$$\forall G, \mathtt{s}.\ (\mathtt{pt}(G) \subseteq \mathtt{pt}([\![G,\ \mathtt{s}]\!]))$$

Proof. The following is logically equivalent.

$$\forall \mathrm{r}, \mathrm{s}.\ (\mathrm{r} \in \mathtt{pt}(G) \Longrightarrow \mathrm{r} \in \mathtt{pt}([\![G,\ \mathrm{s}]\!]))$$

We prove this by induction on the structure of G. □

Lemma A.5 (Encoding Preserves Privacy)**.** *The encoding on global types will not introduce non-server roles that were not participants of the original communication.*

$$\forall \mathrm{r}, \mathrm{s}, G.\ (\mathrm{r} \neq \mathrm{s} \wedge \mathrm{r} \notin \mathtt{pt}(G) \Longrightarrow \mathrm{r} \notin \mathtt{pt}([\![G,\ \mathrm{s}]\!]))$$

Proof. The following is logically equivalent.

$$\forall \mathrm{r}, \mathrm{s}, G.\ (\mathrm{r} \neq \mathrm{s} \wedge \mathrm{r} \notin \mathtt{pt}(G) \Longrightarrow \mathrm{r} \notin \mathtt{pt}([\![G,\ \mathrm{s}]\!]))$$

We prove this by induction on the structure of G, assuming that $\mathrm{r} \neq \mathrm{s}$ for arbitrary roles r, s. □

Lemma A.6 (Correspondence between Encodings on Global Types and Local Types)**.**

$$\forall \mathrm{r}, \mathrm{s}, G.\ (\mathrm{r} \neq \mathrm{s} \Longrightarrow [\![G,\ \mathrm{s}]\!] \upharpoonright \mathrm{r} = [\![G \upharpoonright \mathrm{r},\ \mathrm{r},\ \mathrm{s}]\!])$$

Proof. By induction on the structure of G, Lemmas A.4 and A.5. □

Lemma A.7 (Local Type Encoding Preserves Equality of Projection)**.**

$$\forall G_1, G_2, \mathrm{r}, \mathrm{s}.\ ((G_1 \upharpoonright \mathrm{r}) = (G_2 \upharpoonright \mathrm{r}) \wedge \mathrm{r} \neq \mathrm{s} \Longrightarrow [\![G_1,\ \mathrm{s}]\!] \upharpoonright \mathrm{r} = [\![G_2,\ \mathrm{s}]\!] \upharpoonright \mathrm{r})$$

Proof. By consequence from Lemma A.6.

Take $G_1, G_2, \mathrm{r}, \mathrm{s}$ arbitrarily. Assume $(G_1 \upharpoonright \mathrm{r}) = (G_2 \upharpoonright \mathrm{r})$ and $\mathrm{r} \neq \mathrm{s}$.

We need to show $[\![G_1,\ \mathrm{s}]\!] \upharpoonright \mathrm{r} = [\![G_2,\ \mathrm{s}]\!] \upharpoonright \mathrm{r}$, but by Lemma A.6, it is sufficient to show

$$[\![G_1 \upharpoonright \mathrm{r},\ \mathrm{r},\ \mathrm{s}]\!] = [\![G_2 \upharpoonright \mathrm{r},\ \mathrm{r},\ \mathrm{s}]\!]$$

Define an "inner" function $f(z) = [\![z,\ \mathrm{r},\ \mathrm{s}]\!]$.

By definition, $\forall x, y.(x = y \Longrightarrow f(x) = f(y))$.

We have $(G_1 \upharpoonright \mathrm{r}) = (G_2 \upharpoonright \mathrm{r})$ by assumption, so we can conclude.

$$\begin{aligned}(G_1 \upharpoonright \mathrm{r}) = (G_2 \upharpoonright \mathrm{r}) &\Longrightarrow f(G_1 \upharpoonright \mathrm{r}) = f(G_2 \upharpoonright \mathrm{r}) \\ &\Longrightarrow [\![G_1 \upharpoonright \mathrm{r},\ \mathrm{r},\ \mathrm{s}]\!] = [\![G_2 \upharpoonright \mathrm{r},\ \mathrm{r},\ \mathrm{s}]\!]\end{aligned}$$

□

Lemma A.8 (Encoding on Global Types Preserves Merge)**.** *Take global types G_1, G_2 and roles r, s such that $\mathrm{r} \neq \mathrm{s}$. Suppose $G_1 \upharpoonright \mathrm{r}$ and $G_2 \upharpoonright \mathrm{r}$ exist.*

$$(G_1 \upharpoonright \mathrm{r}) \sqcap (G_2 \upharpoonright \mathrm{r}) \textit{ exists} \Longrightarrow ([\![G_1,\ \mathrm{s}]\!] \upharpoonright \mathrm{r}) \sqcap ([\![G_2,\ \mathrm{s}]\!] \upharpoonright \mathrm{r}) \textit{ exists}$$

Proof. By induction on the structure of $T_1 \sqcap T_2$, Lemmas A.2 and A.7

□

Appendix B

Artefacts for Evaluation

B.1 Implementation for Noughts and Crosses

```
/**
 * User implementation of game board state.
 * Export game board instance.
 */

import { Coordinate as Point } from './Game/GameTypes';

enum Cell { Empty, P1, P2 };
export enum MoveResult { Win, Draw, Continue };

class Board {

  private _board: Array<Array<Cell>>;
  private _emptyCellCount: number;

  constructor() {
    this._board = [[Cell.Empty, Cell.Empty, Cell.Empty],
    [Cell.Empty, Cell.Empty, Cell.Empty],
    [Cell.Empty, Cell.Empty, Cell.Empty]];
    this._emptyCellCount = 9;
  }

  // Factory for generating a function that, given a player role,
  // places the marker and returns the game result with respect to
  // that player.
  private _makeMove = (marker: Cell) => ({ x: row, y: col }: Point)
    => {
    return new Promise<MoveResult>((resolve, _) => {
      setTimeout(() => {
        // Update board state
        this._board[row][col] = marker;
        this._emptyCellCount--;

        // Check for winning move
        if (this._board[row].every(cell => cell === marker))
          resolve(MoveResult.Win);

```

```
        /* Winning column */
        if (this._board.every(row => row[col] === marker))
          resolve(MoveResult.Win);

        /* Placed middle marker - check corners */
        if (row === 1 && col === 1)
          if ((this._board[0][0] === marker
            && this._board[2][2] === marker) ||
            (this._board[0][2] === marker
            && this._board[2][0] === marker))
            resolve(MoveResult.Win);

        /* Placed corner marker - check diagonals */
        if (row !== 1 && col !== 1)
          if (this._board[1][1] === marker
            && this._board[2 - row][2 - col] === marker)
            resolve(MoveResult.Win);

        resolve(this._emptyCellCount === 0 ?
          MoveResult.Draw : MoveResult.Continue);
      }, 2000);
    })

  }

  p1(move: Point) { return this._makeMove(Cell.P1)(move); }
  p2(move: Point) { return this._makeMove(Cell.P2)(move); }

  clear() {
    this._board = [[Cell.Empty, Cell.Empty, Cell.Empty],
    [Cell.Empty, Cell.Empty, Cell.Empty],
    [Cell.Empty, Cell.Empty, Cell.Empty]];
    this._emptyCellCount = 9;
  }

}

// Initialise state
export const board = new Board();
```

B.2 Package Dependencies for Benchmarks

Packages for Ping Pong Client

```
{
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.3.2",
    "@testing-library/user-event": "^7.1.2",
    "@types/jest": "^24.0.0",
    "@types/node": "^12.0.0",
    "@types/react": "^16.9.0",
```

```
    "@types/react-dom": "^16.9.0",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "3.4.1",
    "typescript": "~3.7.2"
  },
  "devDependencies": {
    "@babel/preset-typescript": "^7.10.1",
    "customize-cra": "^1.0.0",
    "react-app-rewired": "^2.1.6"
  }
}
```

Packages for Ping Pong Server

```
{
  "dependencies": {
    "express": "^4.17.1",
    "ws": "^7.3.0",
    "zombie": "^6.1.4"
  },
  "devDependencies": {
    "@types/express": "^4.17.6",
    "@types/ws": "^7.2.4",
    "ts-node": "^8.10.2",
    "typescript": "^3.9.3"
  }
}
```