Generating Interactive WebSocket Applications in TypeScript

Anson Miu
Imperial College London

Francisco Ferreira
Imperial College London

Nobuko Yoshida
Imperial College London

Fangyi Zhou
Imperial College London

Advancements in mobile device computing power have made interactive web applications possible, allowing the web browser to render contents dynamically and support low-latency communication with the server. This comes at a cost to the developer, who now needs to reason more about correctness of communication patterns in their application as web applications support more complex communication patterns.

Multiparty session types (MPST) provide a framework for verifying conformance of implementations to their prescribed communication protocol. Existing proposals for applying the MPST framework in application developments either neglect the event-driven nature of web applications, or lack compatibility with industry tools and practices, which discourages mainstream adoption by web developers.

In this paper, we present an implementation of the MPST framework for developing interactive web applications using familiar industry tools using TypeScript and the *React.js* framework. The developer can use the Scribble protocol language to specify the protocol and use the Scribble toolchain to validate and obtain the *local protocol* for each role. The local protocol describes the interactions of the global communication protocol observed by the role. We encode the local protocol into TypeScript types, catering for server-side and client-side targets separately. We show that our encoding guarantees that only implementations which conform to the protocol can type-check. We demonstrate the effectiveness of our approach through a web-based implementation of the classic *Noughts and Crosses* game from an MPST formalism of the game logic.

1 Introduction

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 (such as real-time multiplayer games) introduces undesirable performance overhead. Developers have since adopted other communication transport abstractions over HTTP connections such as the WebSockets protocol [7] 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 correctness concerns to the developer.

Consider a classic turn-based board game of *Noughts and Crosses* between two players. 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). A web-based implementation may involve players connected to a game server via WebSocket connections. The players interact with the game from their web browser, which shows a *single-page application* (SPA) of the game client written in a popular framework like *React.js* [20]. SPAs feature a single HTML page and dynamically render content via JavaScript in the browser. Players take turns to make a move on the game board, which sends a message to the server. The server implements the game logic to progress the game forward until

S. Balzer, L. Padovani (Eds.): Programming Language Approaches
to Concurrency- & Communication-cEntric Software (PLACES 2020)
EPTCS 314, 2020, pp. 12–22, doi:10.4204/EPTCS.314.2

a result (either a win/loss or draw) can be declared, where either the move of the other player or the game result is sent to players.

Whilst WebSockets make this web-based implementation possible, they introduce the developer to a new family of communication errors, even for this simple game. In addition to the usual testing for game logic correctness, the developer needs to test against *deadlocks* (e.g. both players waiting for each other to make a move at the same time) and *communication mismatches* (e.g. player 1 sending a boolean to the game server instead of the board coordinates). The complexity of these errors correlates to the complexity of the required tests and scales with the complexity of communication patterns involved.

Multiparty Session Types (MPST) [5] 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 game as an MPST protocol and verifying the implementations against the protocol for conformance, MPST theory guarantees well-formed implementations to be free from communication errors.

We see the application of the MPST methodology to generating interactive TypeScript web applications to be an interesting design space — to what extent can the MPST methodology be applied to deliver a workflow where developers use the generated TypeScript APIs in their application to guarantee protocol conformance by construction? Such a workflow would ultimately decrease the overhead for incorporating MPST into mainstream web development, which reduces development time by programmatically verifying communication correctness of the implementation.

Contributions This paper presents a workflow for developing type-safe interactive SPAs motivated by the MPST framework: **(1)** An endpoint API code generation workflow targeting TypeScript-based web applications for multiparty sessions; **(2)** An encoding of session types in server-side TypeScript that enforces static linearity; and **(3)** An encoding of session types in browser-side TypeScript using the React framework that guarantees affine usage of communication channels.

2 The Scribble Framework

Development begins with specifying the expected communications between participants as a *global protocol* in Scribble [23], a MPST-based protocol specification language and code generation toolchain. We specify the *Noughts and Crosses* game as a Scribble protocol in Listing 1. In the protocol, the role `Svr` stands for the Server, and the roles `P1` and `P2` stand for the two Players respectively.

We leverage the Scribble toolchain to check for protocol well-formedness. This directly corresponds to multiparty session type theory [16]: a Scribble protocol maps to some *global type*, and the Scribble toolchain implements the algorithmic projection defined in [5] to derive valid local type *projections* for all participants. We obtain a set of *endpoint protocols* (corresponds to *local types*) — one for each role from a well-formed global protocol. An endpoint protocol only preserves the interactions defined by the global protocol in which the target role is involved, and corresponds to an equivalent *Endpoint Finite State Machine* (EFSM) [6]. The EFSM holds information about the permitted IO actions for the role. We use the EFSMs as a basis for API generation and adopt the formalisms in [11].

3 Encoding Session Types in TypeScript

Developers can implement their application using APIs generated from the EFSM to guarantee correctness by construction. Our approach integrates the EFSM into the development workflow by encoding

```
module NoughtsAndCrosses;
type <typescript> "Coordinate" from "./Types" as Point; // Position on board

global protocol Game(role Svr, role P1, role P2) {
  Pos(Point) from P1 to Svr;
  choice at Svr {
    Lose(Point) from Svr to P2; Win(Point) from Svr to P1;
  } or {
    Draw(Point) from Svr to P2; Draw(Point) from Svr to P1;
  } or {
    Update(Point) from Svr to P2; Update(Point) from Svr to P1;
    do Game(Svr, P2, P1); // Continue the game with player roles swapped
  }
}
```

Listing 1: *Noughts and Crosses* in a Scribble protocol.

session types as TypeScript types. Communication over the WebSocket protocol introduces additional constraints: communication is always initiated in the front-end and driven by user interactions, whilst back-end roles can only accept connections. This motivates our design of encoding the session types differently for server (Section 3.1) and client (Section 3.3) targets.

3.1 Server-Side API Generation

We refer to the Svr EFSM (Figure 1) as a running example in this section. For server-side targets, we encode EFSM states into TypeScript types and consider branching (receiving) and selection (sending) states separately. We assign TypeScript encodings of states to their state identifiers in the EFSM, providing syntactic sugar when referring to the successor state when encoding the current state. For any state S in the EFSM, we refer to the TypeScript type alias of its encoding as $[\![S]\!]$. We outline the encoding below using examples from the *Noughts and Crosses* game (Listing 2).

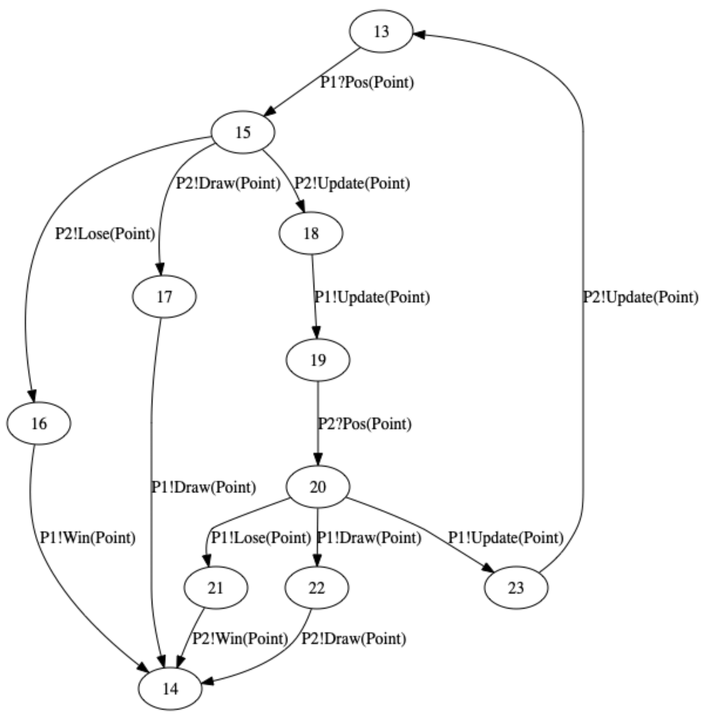

Figure 1: EFSM for Svr.

Branching State We consider a receiving state as a unary branching state for conciseness. A branching state is encoded as an *object literal* [18] (a record type), with each branch $i \in I$ (I denoting set of all branches), corresponding to a member field. A branch expecting to receive a message labelled $\texttt{label}_i$ carrying payload of type $\texttt{T}_i$ with successor state S_i is encoded as an *member field* named $\texttt{label}_i$ of function type $(payload : \texttt{T}_i) \to [\![S_i]\!]$. The developer implements a branching operation by passing callbacks for each branch, parameterised by the expected message payload type for that branch.

Selection State We consider a sending state as a unary selection state for conciseness. A selection state is encoded as a *union type* [18] of internal choice encodings: each internal choice $i \in I$ (I denoting set of all choices), sending a message labelled $\texttt{label}_i$ carrying payload of type $\texttt{T}_i$ with successor state S_i is encoded as a *tuple type* of $\texttt{[Labels.label}_i\texttt{, T}_i\texttt{, } [\![S_i]\!]\texttt{]}$. The developer implements a selection operation by passing the selected label and payload to send in the message. We generate a *string enum* (named `Labels`) wrapping the labels in the protocol.

```
export type S13 = { Pos: (payload: Point) => S15 };
export type S15 = [ Labels.Lose, Point, S16 ]
                | [ Labels.Draw, Point, S17 ]
                | [ Labels.Update, Point, S18 ];
```

Listing 2: Example encodings from *Noughts and Crosses* `Svr` EFSM.

In the case of Listing 2, the developer is expected to implement `S13` which handles the `Pos` message sent by `P1`, and the code in `S13` returns a value of type `S15`, which corresponds to a selection of messages to send to `P2`. Listing 4 illustrates how the developer may implement these types.

We make a key design decision *not* to expose communication channels in the TypeScript session type encodings to provide *static* linearity guarantees (Section 3.1.2). Our encoding sufficiently exposes seams for the developer to inject their program logic, whilst the generated session API (Section 3.1.1) handles the sending and receiving of messages.

3.1.1 Session Runtime

The generated code for our session runtime performs communication in a protocol-conformant manner, but does not expose these IO actions to the developer by delegating the aforementioned responsibilities to an inner class. The runtime executes the EFSM by keeping track of the current state (similar to the generated code in [10]) and only permitting the specified IO actions at the current state. The runtime listens to message (receiving) events on the communication channel, invokes the corresponding callback to obtain the value to send next, and performs the sending. The developer instantiates a session by constructing an instance of the session runtime class, providing the WebSocket endpoint URL (to open the connection) and the initial state (to execute the EFSM).

3.1.2 Linear Channel Usage

Developers writing their implementation using the generated APIs enjoy channel linearity by construction. Our library design prevents the two conditions detailed below:

Repeated Usage We do not expose channels to the developer, which makes *reusing channels* impossible. For example, to send a message, the generated API only requires the payload that needs to be sent, and the session runtime performs the send internally, guaranteeing this action is done *exactly once* by construction.

Unused Channels The initial state must be supplied to the session runtime constructor in order to instantiate a session; this initial state is defined in terms of the successor states, which in turn has references to its successors and so forth. The developer's implementation will cover the terminal state (if it exists), and the session runtime guarantees this terminal state will be reached by construction.

3.2 The React Framework

Our browser-side session type encodings for browser-side targets build upon the *React.js* framework, developed by Facebook [20] for the *Model-View-Controller* (MVC) architecture. React is widely used in industry to create scalable single-page TypeScript applications, and we intend for our proposed workflow to be beneficial in an industrial context. We introduce the key features of the framework.

Components A component is a reusable UI element which contains its own markup and logic. Components implement a `render()` function which returns a React element, the smallest building blocks of a React application, analogous to the view function in the MVU architecture. Components can keep *state*s and the `render()` function is invoked upon a change of state.

For example, a simple counter can be implemented as a component, with its `count` stored as state. When rendered, it displays a button which increments `count` when clicked and a `div` that renders the current `count`. If the button is clicked, the `count` is incremented, which triggers a re-rendering (since the state has changed), and the updated `count` is displayed.

Components can also render other components, which gives rise to a parent/child relationship between components. Parents can pass data to children as *props* (short for properties). Going back to the aforementioned example, the counter component could render a child component `<StyledDiv count={this.state.count} />` in its `render()` function, propagating the `count` from its state to the child. This enables reusability, and for our use case, gives control to the parent on what data to pass to its children (e.g. pass the payload of a received message to a child to render).

3.3 Browser-Side API Generation

We refer to the `P1` EFSM (Figure 2) as a running example in this section. Preserving behavioural typing and channel linearity is challenging for browser-side applications due to EFSM transitions being triggered by user events: in the case of *Noughts and Crosses*, once the user makes a move by clicking on a cell on the game board, this click event must be deactivated until the user's next turn, otherwise the user can click again and violate channel linearity. Our design goal is to enforce this statically through the generated APIs.

For browser-side targets, we extend the approach presented in [9] on *multiple model types* motivated by the *Model-View-Update* (MVU) architecture. An MVU application features a *model* encapsulating application state, a *view function* rendering the state on the Document Object Model (DOM), and an *update function* handling *messages* produced by the rendered model to produce a new model. 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 (IO) actions available at that state, and the update function defines which successor state to transition to, given the supported IO actions at this state.

3.3.1 Model Types in React

State An EFSM state is encoded as an *abstract* React component. This is an abstract class to require the developer to provide their own view function, which translates conveniently to the `render()` function of React components. Our session runtime (Section 3.3.2) "executes" the EFSM and renders the current state. Upon transitioning to a successor state, the successor's view function will be invoked, as per the semantics expressed in [9].

Model Transitions Transitions are encoded as React component props onto the encoded states by the session runtime (Section 3.3.2). We motivate the design choice of not exposing channel resources to provide guarantees on channel usage. React components in TypeScript are *generic* [18], parameterised by the permitted types of prop and state. The parameters allow us to leverage the TypeScript compiler to verify that the props for model transitions stay local to the state they are defined for. The model transitions for EFSMs are message send and receive.

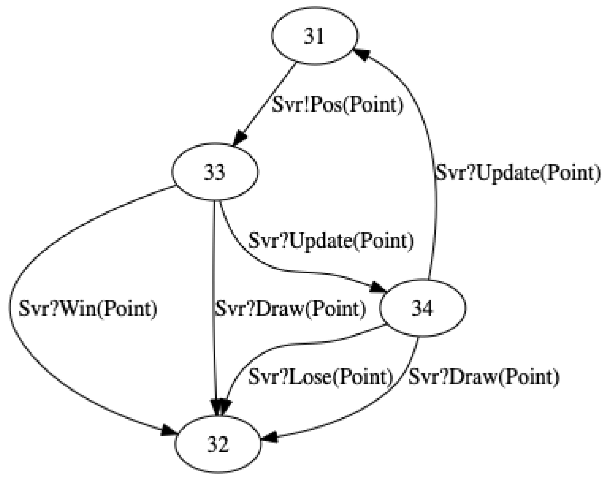

Figure 2: EFSM for `P1`.

Sending We make the assumption that message sending is triggered by some user-driven UI event (e.g. clicking a button, pressing a key on the keyboard) which interacts with some DOM element. We could pass a `send()` function as a prop to the sending state, but the developer would be free to call the function multiple times which makes channel reuse possible. Instead, we pass a *factory function* as a prop, which will, given an HTML event and an event handler function, return a fresh React component that binds the sending action on construction. So once the bound event is triggered, our session runtime executes the event handler function to obtain the payload to send, perform the send *exactly once* and transition to (which, in practice, means render) the successor state.

```
// Inside some render() function..
{board.map((row, x) => (
    row.map((col, y) => {
        const SelectPoint = this.props.Pos('click', (event: UIEvent) => {
            event.preventDefault();
            return { x: x, y: y };}
        return <SelectPoint><td>.</td></SelectPoint>;
});}
```

Listing 3: Model transition for message sending in *Noughts and Crosses* `P1` implementation.

We demonstrate the semantics using the *Noughts and Crosses* example in Listing 3. The session runtime passes the factory function `this.props.Pos` as a prop. For each x-y coordinate on the game board, we create a `SelectPoint` React component from the factory function (which reads "build a React component that sends the `Pos` message with `x-y` coordinates as payload when the user clicks on it") and we wrap a table cell (the game board is rendered as an HTML table) inside the `SelectPoint` component to bind the click event on the table cell.

Receiving The React component for a receiving state is required to define a handler for each supported branch. Upon a message receive event, the session runtime invokes the handler of the corresponding branch with the message payload and renders the successor state upon completion.

3.3.2 Session Runtime

The session runtime can be interpreted as an abstraction on top of the React VDOM that implements the EFSM by construction. The session runtime itself is a React component too, named after the endpoint role identifier: it opens the WebSocket connection to the server, keeps track of the current EFSM state as part of its React component state, and most importantly, renders the React component encoding of the active EFSM state. Channel communications are managed by the runtime, which allows it to render the successor of a receive state upon receiving a message from the channel. Similarly, the session runtime is responsible for passing the required props for model transitions to EFSM state React components. The session runtime component is rendered by the developer and requires, as props, the *endpoint URL* (so it can open the connection) and a list of *concrete state components*.

The developer writes their own implementation of each state (mainly to customise how the state is rendered and inject business logic into state transitions) by extending the abstract React class components. The session runtime requires references to these concrete components in order to render the user implementation accordingly.

3.3.3 Affine Channel Usage

A limitation of our browser-side session type encoding is only being able to guarantee that channel resources are used *at most once* as opposed to *exactly once*.

Communication channels are not exposed to the developer so multiple sends are impossible. This does not restrict the developer from binding the send action to exactly one UI event: for *Noughts and Crosses*, we bind the `Pos(Point)` send action to each unoccupied cell on the game board, but the generated runtime ensures that, once the cell is clicked, the send is only performed once and the successor state is rendered on the DOM, so the channel resource used to send becomes unavailable.

However, our approach *does not* statically detect whether all transitions in a certain state are bound to some UI event. This means that it is possible for an implementation to *not* handle transitions to a terminal state but still type-check, so we cannot prevent unused states. Equally, our approach does not prevent a client closing the browser, which would drop the connection.

4 Case Study

We apply our framework to implement a web-based implementation of the *Noughts and Crosses* running example in TypeScript; the interested reader can find the full implementation in [14]. In addition to MPST-safety, we show that our library design welcomes idiomatic JavaScript practices in the user implementation and is interoperable with common front- and back-end frameworks.

```
1  const handleP1Move: S13 = (move: Point) => {
2      board.P1(move);          // User logic
3      if (board.won()) {
4          return [Labels.Lose, move, [Labels.Win, move]];
5      } else if (board.draw()) {
6          return [Labels.Draw, move, [Labels.Draw, move]];
7      } else {
8          return [Labels.Update, move, [Labels.Update, move, handleP2Move]];
9      }
10 }
11
12 // Instantiate session - `handleP2Move` defined similarly as S19
13 new NoughtsAndCrosses.Svr(webSocketServer, handleP1Move);
```

Listing 4: Session runtime instantiation for *Noughts and Crosses* `Svr`.

Game Server We set up the WebSocket server as an Express.js [8] application on top of a Node.js [17] runtime. We define our own game logic in a `Board` class to keep track of the game state and expose methods to query the result. This custom logic is integrated into our `handleP1Move` and `handleP2Move` handlers (Listing 4), so the session runtime can handle `Pos(Point)` messages from players and transition to the permitted successor states (Listing 1) according to the injected game logic: if `P1` played a winning move (Line 4), `Svr` sends a `Lose` message to `P2` with the winning move, and also sends a `Win` message to `P1`; if `P1`'s move resulted in a draw (Line 6), `Svr` sends `Draw` messages to both `P2` and `P1`; otherwise, the game continues (Line 8), so `Svr` updates both `P2` and `P1` with the latest move and proceeds to handle `P2`'s turn.

Note that, by TypeScript's structural typing [18], replacing `handleP2Move` on Line 8 with a recursive occurrence of `handleP1Move` would be type-correct — this allows for better code reuse as opposed to defining additional abstractions to work around the limitations of nominal typing in [11]. There is also full type erasure when transpiling to JavaScript to run the server code, so the types defined in TypeScript will not appear in the JavaScript after type-checking. This means state space explosion is not a runtime consideration.

Game Clients We implement the game client for `P1` and `P2` by extending from the generated abstract React (EFSM state) components and registering those to the session runtime component.

For the sake of code reuse, [14] uses *higher-order components* (HOC) to build the correct state implementations depending on which player the user chooses to be. We leverage the *Redux* [1] state management library to keep track of the game state, thus showing the flexibility of our library design in being interoperable with other libraries and idiomatic JavaScript practices. Our approach encourages the separation of concerns between the communication logic and program logic — the generated session runtime keeps track of the state of the EFSM to ensure protocol conformance by construction, whilst *Redux* solely manages our game state.

5 Related Work

The two main approaches for incorporating our MPST workflow into application development are native language support for first-class linear channel resources [22] and code generation. The latter closely relates to our proposal; we highlight two areas of existing work under this approach that motivate our design choice.

Endpoint API Generation Neykova and Yoshida targeted Python applications and the generation of runtime monitors [15] to dynamically verify communication patterns. Whilst the same approach could be applied to JavaScript, we can provide more static guarantees with TypeScript's gradual typing system. Scribble-Java [11] proposed to encode the EFSM states and transitions as classes and instance methods respectively, with behavioural typing achieved statically by the type system and channel linearity guarantees achieved dynamically since channels are exposed and aliasing is not monitored. Scribble-Java can generate callback-style APIs similar to the approach we present, but this approach is arguably less idiomatic for Java developers.

Session Types in Web Development King et al. [13] targeted web development in PureScript using the *Concur UI* framework and proposed a type-level encoding of EFSMs as multi-parameter type classes. However, it presents a trade-off between achieving static linearity guarantees from the type-level EFSM encoding under the expressive type system and providing an intuitive development experience to developers, especially given the prevalence of JavaScript and TypeScript applications in industry. Fowler [9] focused on applying binary session types in front-end web development and presented approaches that tackle the challenge of guaranteeing linearity in the event-driven environment, whereas our work is applicable to multiparty scenarios.

Our work applies the aforementioned approaches in a *multiparty* context using industrial tools and practices to ultimately encourage MPST-safe web application development workflows in industry.

6 Conclusion and Future Work

We have presented an MPST-based framework for developing full-stack interactive TypeScript applications with WebSocket communications. The implementation conforms to a specified protocol, statically providing linear channel usage guarantees and affine channel usage guarantees for back-end and front-end targets respectively.

Future work includes incorporating *explicit connection actions* introduced in [12] in our API generation to better model real-world communication protocols that may feature in interactive web applications. Server-side implementations may perform asynchronous operations on the received messages, so supporting asynchronous values (such as JavaScript *Promises* [3]) in IO actions would be a welcome addition. Whilst our approach supports multiparty sessions, the nature of WebSockets require some server-based role in the communication protocol and clients to interact via the server. Extending support to WebRTC [21] would cater for peer-to-peer communication between browsers, which further opens up possibilities for communication protocols supported by our approach.

Acknowledgements

We thank the anonymous reviewers for their feedback.
This work was supported in part by EPSRC projects EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, and EP/T006544/1.

References

[1] Dan Abramov (2015): *Redux - A predictable state container for JavaScript apps*. Available at `https://redux.js.org/`.

[2] Gavin Bierman, Martn Abadi & Mads Torgersen (2014): *Understanding TypeScript*. In RichardEditor Jones, editor: *ECOOP 2014 Object-Oriented Programming*, Lecture Notes in Computer Science, Springer, pp. 257–281, doi:10.1007/978-3-662-44202-9_11.

[3] MDN contributors (2020): *Promise*. Available at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`. Library Catalog: developer.mozilla.org.

[4] Ezra Cooper, Sam Lindley, Philip Wadler & Jeremy Yallop (2007): *Links: Web Programming Without Tiers*, pp. 266–296. 4709, Springer Berlin Heidelberg, doi:10.1007/978-3-540-74792-5_12.

[5] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani & Nobuko Yoshida (2015): *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*, *LNCS* 9104, Springer, pp. 146–178, doi:10.1007/978-3-319-18941-3_4.

[6] Pierre-Malo Denilou & Nobuko Yoshida (2013): *Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types*. In: *40th International Colloquium on Automata, Languages and Programming*, *LNCS* 7966, Springer, Berlin, Heidelberg, pp. 174–186, doi:10.1007/978-3-642-39212-2_18.

[7] I. Fette & A. Melnikov (2011): *The WebSocket Protocol*. RFC 6455, RFC Editor. Available at `https://www.rfc-editor.org/rfc/rfc1654.txt`.

[8] Node.js Foundation: *Express - Node.js web application framework*. Available at `https://expressjs.com/`.

[9] Simon Fowler (2019): *Model-View-Update-Communicate: Session Types meet the Elm Architecture*. *arXiv:1910.11108 [cs]*. Available at `http://arxiv.org/abs/1910.11108`. ArXiv: 1910.11108.

[10] Rosita Gerbo & Luca Padovani (2019): *Concurrent Typestate-Oriented Programming in Java*. *Electronic Proceedings in Theoretical Computer Science* 291, pp. 24–34, doi:10.4204/EPTCS.291.3. ArXiv: 1904.01286.

[11] Raymond Hu & Nobuko Yoshida (2016): *Hybrid Session Verification through Endpoint API Generation*. In: *19th International Conference on Fundamental Approaches to Software Engineering*, *LNCS* 9633, Springer, pp. 401–418, doi:10.1007/978-3-662-49665-7_24.

[12] Raymond Hu & Nobuko Yoshida (2017): *Explicit Connection Actions in Multiparty Session Types*, pp. 116–133. 10202, Springer Berlin Heidelberg, doi:10.1007/978-3-662-54494-5_7.

[13] Jonathan King, Nicholas Ng & Nobuko Yoshida (2019): *Multiparty Session Type-safe Web Development with Static Linearity*. *Electronic Proceedings in Theoretical Computer Science* 291, pp. 35–46, doi:10.4204/EPTCS.291.4.

[14] Anson Miu (2020): *ansonmiu0214/scribble-noughts-and-crosses*. Available at `https://github.com/ansonmiu0214/scribble-noughts-and-crosses`.

[15] Rumyana Neykova & Nobuko Yoshida (2017): *How to Verify Your Python Conversations*. *Behavioural Types: from Theory to Tools*, pp. 77–98, doi:10.13052/rp-9788793519817.

[16] Rumyana Neykova & Nobuko Yoshida (2019): *Featherweight Scribble*, pp. 236–259. 11665, Springer International Publishing, doi:10.1007/978-3-030-21485-2_14.

[17] Node.js: *Node.js*. Available at `https://nodejs.org/en/`.

[18] Microsoft Research: *TypeScript Language Specification*. Available at `https://github.com/microsoft/TypeScript`.

[19] Facebook Open Source: *Introducing JSX React*. Available at `https://reactjs.org/docs/introducing-jsx.html`.

[20] Facebook Open Source: *React A JavaScript library for building user interfaces*. Available at `https://reactjs.org/`.

[21] Justin Uberti & Peter Thatcher (2011): *WebRTC*. Available at `https://webrtc.org/`.

[22] Hongwei Xi (2017): *Applied Type System: An Approach to Practical Programming with Theorem-Proving*. *arXiv:1703.08683 [cs]*. Available at `http://arxiv.org/abs/1703.08683`. ArXiv: 1703.08683.

[23] Nobuko Yoshida, Raymond Hu, Rumyana Neykova & Nicholas Ng (2013): *The Scribble Protocol Language*. In: *8th International Symposium on Trustworthy Global Computing*, *LNCS* 8358, Springer, pp. 22–41, doi:10.1007/978-3-319-05119-2_3.