Multiparty Session Type-safe Web Development in TypeScript

Anson Miu
Imperial College London

Francisco Ferreira
Imperial College London

Nobuko Yoshida
Imperial College London

Fangyi Zhou
Imperial College London

This is a sentence in the abstract. This is another sentence in the abstract. This is yet another sentence in the abstract. This is the final sentence in the abstract.

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 a multiplayer game) 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 [4] 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 (such as real-time multiplayer games), but introduces additional concerns to the developer with respect to implementation correctness.

Consider a classic turn-based board game of *Noughts and Crosses* between two players. Both players are identified by either noughts or crosses respectively, and 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).

Figure **??** illustrates a possible implementation of the game as an interactive real-time multiplayer web application. Both players are connected to the game server by bidirectional WebSocket connections and interact with the game from their web browser, which serve a *single-page application* (SPA) of the game client. SPAs feature a single HTML page and dynamically renders content via JavaScript in the browser. Players take turns to make a move on the game board and the server implements the game logic to progress the game forward until a result (either a win/loss or draw) can be declared.

Whilst WebSockets make this web-based implementation possible, it introduces 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, which correlate to the complexity of tests required against these errors, scale with the complexity of the communication patterns involved.

Multiparty Session Types (MPST) [3] 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 a protocol and verifying the implementations against the protocol for well-formedness, MPST theory guarantees well-formed implementations to be free from communication errors.

Submitted to:
PLACES 2020

This MPST methodology has been applied to target applications written in both statically and dynamically typed languages in [7, 10], but do not apply to the event-driven paradigm of interactive web applications. A proposal that caters for web applications in [8] targets PureScript to maximise the static guarantees it can provide, but the strong type system make it difficult to extract common code and the functional paradigm is arguably unidiomatic relative to mainstream web development in JavaScript, which limits developer productivity. JavaScript began as the language of the web browser, but a shallow learning curve, an ever-growing ecosystem of libraries and frameworks (such as React.js for writing SPAs) and a successful server-side single-threaded event loop implementation in *Node.js* has propelled it as the de-facto choice for full-stack web application development in a single language.

Whilst the proposal in [10] could be adapted to verify interactive web applications written in JavaScript, the emergence of Microsoft's TypeScript [1] with a gradual type system motivates the need for a MPST-based workflow for verifying full-stack TypeScript web applications, such that it is flexible with respect to idiomatic practices and frameworks to be useful in industry whilst also leveraging the language features to maximise the static guarantees it is able to provide. Such a workflow would ultimately decrease the overhead for incorporating MPST into mainstream web development, which reduces development time by programmatically verifying implementations for communication correctness.

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.

Figure **??** illustrates our proposed development workflow: we extend the Scribble framework (§ 2) and generate TypeScript endpoints for server-side (§ 3.1) and browser-side (§ 3.2) targets. We will use the *Noughts and Crosses* game as our running example in the rest of the paper and show how our approach is a practical compromise that combines benefits from static session typing and being compatible with common libraries and idiomatic JavaScript practices (§ 4).

2 The Scribble Framework

Development begins by specifying the permitted communications between participants as a *global protocol* in Scribble, a MPST-based framework introduced in [15] which features a protocol specification language and code generation toolchain. This means that a protocol specified in Scribble maps to some *global type* in multiparty session type theory [3].

We specify the *Noughts and Crosses* game as a Scribble protocol in Listing 1 and explain the syntax of the language through this example. Scribble protocols are organised into *modules*: line 1 declares the name of the module. Line 2 declares a message type to be used as payload types in the global protocol: this would correspond to an `import { Coordinate as Point } from './Types'` statement in TypeScript, where the developer provides the definition of the `Coordinate` interface under a `Types.ts` file. Line 4 declares the name of the global protocol (`Game`) and specifies the participants involved in the protocol as *roles*; the protocol body is defined by lines 5-16. A message passing statement from a sender (e.g. `P1`) to a receiver (e.g. `Svr`) require a *label identifier* followed by a list of *payload types* in parentheses (e.g. `Pos(Point) from P1 to Svr` on line 5). Semicolons denote continuations for message passing statement in the same way that they express sequential statement execution in programming languages such as Java.

```
1  module NoughtsAndCrosses;
2  type <typescript> "Coordinate" from "./Types" as Point; // Position on board
3
4  global protocol Game(role Svr, role P1, role P2) {
5    Pos(Point) from P1 to Svr;
6    choice at Svr {
7      Lose(Point) from Svr to P2;
8      Win(Point) from Svr to P1;
9    } or {
10     Draw(Point) from Svr to P2;
11     Draw(Point) from Svr to P1;
12   } or {
13     Update(Point) from Svr to P2;
14     Update(Point) from Svr to P1;
15     do Game(Svr, P2, P1);
16   }
17 }
```

Listing 1: Main body of the *Noughts and Crosses* protocol.

The `Game` protocol involves three roles: the server (`Svr`) and two players (`P1` and `P2`). First, `P1` makes a move by sending a coordinate to place the marker on the game board to `Svr`. Upon receiving the `Pos` message, `Svr` will implement the game logic to decide the outcome of `P1`'s move and communicate with `P2` accordingly. We express this as a `choice` statement (lines 6-15), where `Svr` decides whether `P1`'s move resulted in a loss for `P2` (lines 7-8), stalemate (lines 10-11) or an unfinished game (lines 13-15).

If `P1` performed a winning move (i.e. the latest coordinate resulted in three-in-a-row for `P1`), then `Svr` sends a `Lose` message to `P2` and a `Win` message to `P1` with the coordinates of the latest move. These coordinates are sent to allow the applications running in the browser for both players to update their version of the game state respectively; how they choose to render this on the DOM is up to their implementation, independent of the protocol. Similar message passing actions are sent in the case of a draw, just with updated labels for `Draw`.

Otherwise, the game has not ended, so `Svr` updates both players with the coordinates of the latest move by sending an `Update` message, and the game proceeds to allow `P2` to make a move. We express this as a recursive call to the `Game` protocol using a `do` statement: note that we swap the role parameters `P2` and `P1` to express the turn-taking semantics of the game. In short, the global protocol describes `P1` and `P2` taking turns until `Svr` declares a winner/loser or draw.

We leverage the Scribble toolchain to check for protocol well-formedness. This directly corresponds to the multiparty session type theory of verifying valid local type *projections* for all participants of a global type. The implementation of this validation by the toolchain follow from the algorithmic projection procedures defined in [3]. We obtain *endpoint protocols* for each role in a well-formed global protocol. An endpoint protocol only preserves the interactions defined by the global protocol in which the target role is involved. For example, the endpoint protocol for `P1` will not include the exchange of messages between `Svr` and `P2`. An endpoint protocol corresponds to an equivalent *Endpoint Finite State Machine* (EFSM) in the Scribble toolchain. We adopt the formalism of EFSMs in [7], and use the EFSMs as a basis for API generation.

3 Encoding Session Types in TypeScript

Developers can validate the communication aspects of their implementation against their EFSM to verify protocol conformance. Our approach integrates the EFSM into the development workflow by encoding session types as TypeScript types. We appreciate the inherent differences in development considerations between front- and back-end web development: business logic tends to initiate communication in the back-end, whilst communication in the front-end is mostly driven by user interactions with the browser. This motivates our design choice of generating different session types encodings for the target application on the server (§ 3.1) and in the browser (§ 3.2).

3.1 Server-side API generation

We will refer to the `Svr` EFSM (fig. 1) as a running example in this section. For server-side targets, we encode EFSM states as TypeScript types depending on the type of state. We adopt the EFSM definition presented in [7] to consider receive and send states separately. We assign each TypeScript encoding its state identifier from the EFSM as its type alias, thus providing syntactic sugar when referring to the successor state in the TypeScript encoding of the current state. For any state S in the EFSM, we refer to the TypeScript type alias of its encoding as $\llbracket S \rrbracket$.

We make a key design decision *not* to expose communication channels in the TypeScript session type encodings to provide linearity guarantees (§ 3.1.2). Our encodings sufficiently exposes seams for the developer to inject their business logic, whilst the generated session API (§ 3.1.1) handles the sending and receiving of messages; as a result, the encodings do not concern with the role involved in the send/receive action. We outline the encodings below using examples from the *Noughts and Crosses* game server (listing 2).

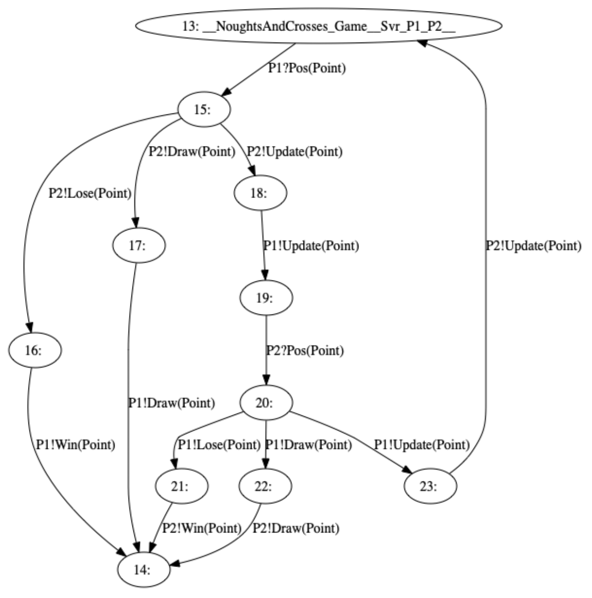

Figure 1: EFSM for `Svr` in *Noughts and Crosses*.

Branching state We consider a receive state as a unary branching state for conciseness. A branching state is encoded as an *object literal* [12], with each branch 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 $\texttt{(payload:T}_i\texttt{) =>}$ $\llbracket S_i \rrbracket$. 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 send state as a unary selection state for conciseness. A selection state is encoded as a *union type* [12] of internal choice encodings: each internal choice 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 $\texttt{[Labels.label}_i\texttt{, T}_i\texttt{, }\llbracket S_i \rrbracket\texttt{]}$. 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, hence the enum member access in the first element of the tuple type.

./NoughtsAndCrosses/Game.ts

```
import { Coordinate as Point } from "./Types";
import { Labels } from "./Constants";

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.

3.1.1 Session Runtime and Protocol API

Our session runtime performs communication in a way that conforms to the protocol specification. The runtime listens to message (receive) events on the communication channel, invokes the corresponding callback to obtain the value to send next, and performs the send.

The *Protocol API* is a class that interfaces between the session runtime and the user implementation of the EFSM state encodings. Upon construction, the Protocol class waits for all client roles to connect via WebSocket, then proceeds to the initial state of the EFSM and performs state transitions depending on the permitted communication patterns from the EFSM. The Protocol class constructor requires the WebSocket server and the following handlers:

Initial state When all roles have joined the session, the Protocol class needs to know what to do in the initial state to start communication, be it sending a message to a client or offering branches to a client. For the *Noughts and Crosses* game server, this would be a callback to handle the receipt of the `Pos(Point)` message from `P1`.

Receive handlers for each role The runtime needs to bind a message event listener to the WebSocket between the server and each (client) role, and needs to know how to handle the incoming messages. For each role, the user is required to pass callbacks that encode the receive states for that role. The type alias for such a handler is the *intersection type* [12] of all receive states concerning that role. Considering a hypothetical web-service providing addition and square root functionality to a client role `C`, if `S1` is a state expecting to receive `Add(int, int)` from `C` and `S4` is a state expecting to receive `Sqrt(int)` from `C`, we define `type Receive_C = S1 & S4` as the receive handler type alias and the user needs to pass some value of `Receive_C` when instantiating the session through the Protocol API.

3.1.2 Linear channel usage

Our callback-oriented design for server-side API generation provides guarantees on state channel linearity by preventing the two properties detailed below. Here, channel linearity is guaranteed based on the correctness of our library design and session runtime implementation, which means a faulty implementation could violate this, but is up to the library author to verify once rather than the end user.

Repeat use By adopting a callback-oriented design for server-side API generation, channels are not directly accessed by the programmer which makes *reuse* impossible; only the Protocol API is exposed but channel operations are concealed in a private class. Consider the *Noughts and Crosses* server endpoint: the programmer must pass a callback that handles the receipt of a `Pos(Point)` message and returns either a `Lose(Point)`, `Draw(Point)` or `Update(Point)` message with the appropriate continuation. Upon receiving the `Pos(Point)` message from a player, our lightweight runtime invokes the callback to get `Svr`'s choice from the return value, and perform the appropriate send. This is transparent to the programmer and it is impossible for the programmer to send or receive messages more than once, by design.

Unused The initial state must be supplied to the Protocol API constructor in order to instantiate the session; this initial state is defined in terms of the successor states, which in turn has references to its successors and so forth. This encoding approach will cover the terminal state (if it exists in the EFSM, as [7] notes that an EFSM contains at most one terminal state), and the session runtime guarantees this terminal state, if it exists, will be reached by construction.

3.2 Browser-side API generation

We will refer to the `P1` EFSM (figure 2) as a running example in this section. Preserving behavioural typing and channel linearity are known to be challenging for browser-side applications due to the event-driven nature of user interaction: 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 APIs we generate.

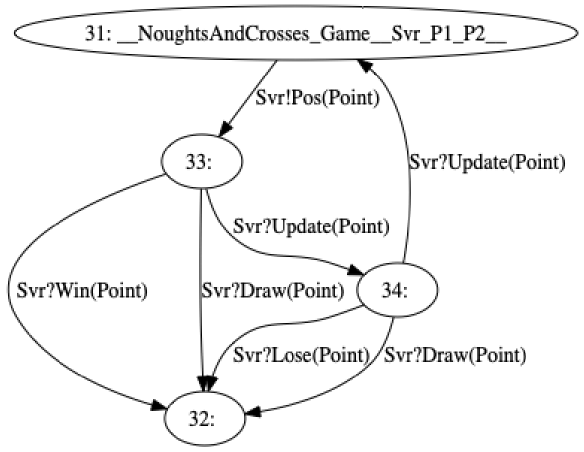

Figure 2: EFSM for `P1` in *Noughts and Crosses*.

For browser-side targets, we extend the approach presented in [6] on *multiple model types* motivated by the *Model-View-Update* (MVU) architecture. Each state in the EFSM is a model type and uniquely defines a *view function*, set of *messages* and *update function*: the view function defines what to render to the DOM, the set of messages define the possible (IO) actions available at that state, and the update function defines which successor state to transition to, given some supported IO action at this state.

3.2.1 The React Framework

Rather than using the LINKS web programming language from [6], we apply the multiple model types approach to the React framework [14] developed by Facebook. 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 below.

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. Components can keep *state* and the `render()` function is invoked upon a change of state. A simple counter can be implemented as a component, with the count stored as state, a button which increments the count when clicked and a `div` that renders the current count. Components can also render other components, which gives rise to a parent/child relationship between components. Parents can pass data to children as *props*: going back to the aforementioned example, the 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).

Virtual DOM React components are rendered on a logical abstraction of the DOM, which in turn performs a `diff` on the current browser DOM and patches the delta accordingly. This allows the programmer to directly specify what should be rendered without having to worry about which elements to append or remove from the browser DOM.

3.2.2 Model types in React

We leverage React to express the model types approach in [6] rather than the *Model-View-Controller* (MVC) architecture it was intended for.

State An EFSM state is encoded as an *abstract* React component. This is an abstract class to require the programmer to provide their own view function, which translates conveniently to the `render()` function of React components. Our session runtime (§ 3.2.3) "executes" the EFSM and renders the current state, so upon transitioning to a successor state, the successor's view function will be invoked, as per the semantics expressed in [6].

Model transitions Transitions are encoded as React component props onto the encoded states by the session runtime (§ 3.2.3). We motivate the design choice of not exposing channel resources to provide guarantees on channel usage. React components in TypeScript are *generics* [12] which are parameterised by the permitted types of prop and state, which allows 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.

Send 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 programmer 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 a 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.

We clarify the semantics using the *Noughts and Crosses* example in listing 3. `this.props.Pos` is the factory function prop passed from the session runtime. For each x-y coordinate on the game

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

Listing 3: Example fragment of *Noughts and Crosses* `P1` implementation.

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 a HTML table) inside the `SelectPoint` component to bind the click event on the table cell.

Receive The payload of a message receive transition is passed as a prop to the successor state and can be handled in the successor’s view function. Upon receiving the message from the channel, the session runtime renders the successor of the receive state and propagates the payload as the prop.

3.2.3 Session runtime

...

The session runtime component is rendered by the programmer and requires the following properties to instantiate the session and execute the EFSM.

Endpoint URL

Concrete state components The programmer 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.

3.2.4 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 supposed to *exactly once*.

Repeat use Communication channels are not exposed to the programmer so multiple sends are impossible. This does not restrict the programmer from binding the send action to exactly one UI event: for the *Noughts and Crosses* game, we bind the `Pos(Point)` send action to each unoccupied cell on the game board, but the generated runtime ensures that, once an event bound to the send action is triggered, the

./app.ts

```
const handleP1Move: Receive_P1 = (move: Point) => {
    // User-implemented logic
    board.P1(move)
    if (board.won()) { return [Labels.Lose, move, move] }
    else if (board.draw()) { return [Labels.Draw, move, move] }
    else { return [Labels.Draw, move, handleP2Move] }
}

const handleP2Move: Receive_P2 = ...

// Instantiate session
new NoughtsAndCrosses.Svr(webSocketServer, handleP1Move, handleP2Move)
```

Listing 4: Example fragment of *Noughts and Crosses* `Svr` implementation.

send is only performed once and the successor state is rendered on the DOM to prevent multiple sends initiated by the user.

Unused 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. Consider a hypothetical protocol where, in the current state, sending `Quit()` to some server role `S` transitions to the terminal state: ...

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 [9]. In addition to showing the multiparty session type safety achieved by our generated APIs, we also show that our library design welcomes idiomatic JavaScript practices in the user implementation and is interoperable with common front- and back-end frameworks.

4.1 Game server

We set up the WebSocket server inside an Express.js [5] application on top of a Node.js [11] 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 `Receive_P1` and `Receive_P2` handlers (listing 4) are defined using

4.2 Game clients

Implement states ...

Register states ...

The beauty of the MPST framework is that implementation is decoupled from protocol conformance, so it would be possible to have different front-end implementations for both players; but for the sake of sharing the same game client implementation between both players for demonstration purposes, [9] uses *higher-order components* (HOC) to build the correct state implementations depending on which player the user chooses to be.

We also illustrate the flexibility of our approach in being interoperable with other libraries and idiomatic JavaScript practices. Our generated APIs provide an implementation of the communication state machine which conforms to the protocol, but [9] also integrates with the *Redux* state management library to uniformly keep track of the game state across the game client - for simplicity, we save game state into browser local storage.

5 Related Works

Endpoint API generation ...

Session types in web development ...

Typestate programming ...

6 Conclusion and Future Works

We have presented a MPST-based framework for developing full-stack interactive TypeScript applications with WebSocket communications that conform to a protocol.

Future works include ...

References

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

[2] Ezra Cooper, Sam Lindley, Philip Wadler & Jeremy Yallop (2007): *Links: Web Programming Without Tiers*, p. 266–296. 4709, Springer Berlin Heidelberg, doi:10.1007/978-3-540-74792-5_12. Available at `http://link.springer.com/10.1007/978-3-540-74792-5_12`.

[3] 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.

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

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

[6] 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.

[7] 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.

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

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

[10] 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.

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

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

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

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

[15] 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.