Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types

Nobuko Yoshida(✉), Fangyi Zhou, and Francisco Ferreira

Imperial College London, London, UK
{n.yoshida,fangyi.zhou15,f.ferreira-ruiz}@imperial.ac.uk

Abstract. Multiparty session types (MPST) provide a typing discipline for message passing concurrency, ensuring deadlock freedom for distributed processes. This paper first summarises the relationship between MPST and communicating finite state machines (CFSMs), which offers not only theoretical justifications of MPST but also a guidance to implement MPST in practice. As one of the applications, we present νSCR (NUSCR), an extensible toolchain for MPST-based multiparty protocols. The toolchain can convert multiparty protocols in the SCRIBBLE protocol description language into global types in the MPST theory; global types are projected into local types, and local types are converted to their corresponding CFSMs. The toolchain also generates APIs from CFSMs that implement endpoints in the protocol. Our design allows for language-independent code generation, and opens possibilities to generate APIs in various programming languages. We design our toolchain with modularity and extensibility in mind, so that extensions of core MPST can be easily integrated within our framework. As a case study, we show the implementation of the nested protocol extension in νSCR, to showcase our extensibility.

Keywords: Session Types · Communicating Finite State Machines · Distributed programming · Scribble · Protocols

1 Introduction

In the modern era of distributed and concurrent programming, how to achieve *safety* with minimal effort (i.e. lightweight formal methods) becomes a hot area of research. *Session types* [19] provide a typing discipline for message passing concurrency, by assigning *session types* to communication channels, in terms of a sequence of actions over a channel. Session types, initially only able to describe communications between *two* ends of a channel, are later extended to *multiparty* [20,21], giving rise to the *multiparty session types (MPST)* theory. The MPST typing discipline guarantees that a set of well-typed communicating processes are free from deadlocks or communication mismatches.

1.1 Communicating Finite State Machines and Session Types

Motivation: Why CFSMs? *Communicating Automata* [2], also known as *Communicating Finite State Machines* (CFSMs), are a classical model for protocol specification and verification. Before being used in many industrial contexts, CFSMs have been a pioneer theoretical formalism, in which distributed safety properties could be formalised and studied.

Establishing a formal connection between CFSMs and session types allows the use of CFSMs to build theoretically well-founded tools for MPST. The first work that utilised CFSMs in practice is Demangeon et al. [11], a toolchain for monitoring multiparty communications at runtime for large scientific cyber-infrastructures developed by the Ocean Observatories Initiative [41].

From the theoretical side, the CFSM framework offers canonical justifications for session types to answer open questions which have been asked since [20]. The **1st** question is about *expressiveness*: to which class of CFSMs do session types correspond? The **2nd** question concerns the *semantic correspondence* between session types and CFSMs: how do the safety properties that session types guarantee relate to those of CFSMs? The **3rd** question is about *efficiency*: why do session types provide efficient algorithms for type-checking or verifying distributed programs, while general CFSMs are undecidable? CFSMs can be also seen as generalised endpoint specifications, therefore an excellent target for a common ground for comparing protocol specification languages.

To answer the three questions above, we need to identify a *sound and complete* subset of CFSMs that corresponds to MPST behaviour, which we explain below.

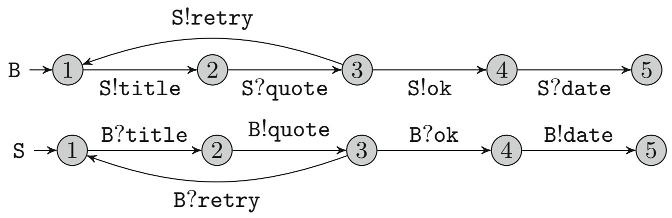


Fig. 1. Two dual communicating automata: the buyer and the seller

Binary Session Types as CFSMs. The subclass that fully characterises binary session types [19] was actually proposed by Gouda, Manning and Yu [16] in a pure automata context (independently from the discovery of session types [45]). Consider a simple business protocol between a Buyer and a Seller. From the Buyer's viewpoint, the Buyer sends the title of a book, then the Seller answers with a quote. If the Buyer is satisfied by the quote, then they send their address and the Seller sends back the delivery date; otherwise they retry the same conversation. This can be specified by the two machines of the Buyer and the Seller in Fig. 1. We can observe that these CFSMs satisfy three conditions: First, the communications are *deterministic*: messages that are part of the same choice, `ok` and `retry` here, are distinct. Secondly, there is no mixed state (each state has either only sending actions, or only receiving actions). Third, these two machines have *compatible* traces (i.e. *dual*): the Seller machine can be defined by exchanging sending and receiving actions of the Buyer machine. Breaking one of these conditions allows deadlock situations and breaking one of the first two conditions makes the compatibility checking undecidable [16].

Essentially, the same characterisation is given in binary session types [19]. Consider the following session type of the Buyer.

$$\mu\mathbf{t}.\oplus title.\,\&quote.\,\oplus\{ok : \oplus addrs.\,\&date.\,\mathtt{end} \quad retry : \mathbf{t}\} \tag{1}$$

The session type above describes the communication pattern using several constructs. The operator $\oplus title$ denotes an output of the title, whereas $\&quote$ denotes an input of a quote. The output choice features the two options *ok* and *retry* and . denotes sequencing. `end` represents the termination of the session, and $\mu\mathbf{t}$ is recursion. The simplicity and tractability of binary sessions come from the notion of *duality* in interactions [15], which corresponds to compatibility of CFSMs. In (1), not only the Buyer's behaviour, but also the whole conversation structure is already represented in this single type: the interaction pattern of the Seller is fully given as the dual of the type in (1) (exchanging input $\oplus$ and output $\&$). When composing two parties, we only have to check they have mutually dual types, and the resulting communication is guaranteed to be deadlock-free.

Multiparty Session Types and CFSMs. The notion of duality is no longer effective in *multiparty* communication, where the whole conversation cannot be reconstructed from only the behaviour of a single machine. Instead of directly trying to decide whether the communication of a system satisfies safety (which is undecidable in the general case), we devise a compatible, decidable condition of a set of machines, which forces them to *collaborate* together. We define a complete characterisation of global type behaviours into CFSMs: a set of CFSMs satisfy some *compatible conditions*, if and only if the CFSMs can mimic the expected behaviour of a given global type. A good global type means the global type can only generate *safe* CFSMs by *endpoint projection*, which satisfies realisability.

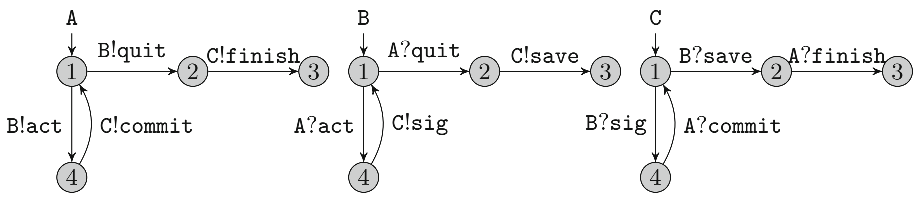


Fig. 2. CFSMs for the commit protocol

We give a simple example to illustrate the proposal. The Commit protocol in Fig. 2 involves three machines: Alice, Bob and Carol. Alice orders Bob to `act` or `quit`. If `act` is sent, Bob sends a `signal` to Carol, and Alice sends a `commitment` to Carol and continues. Otherwise Bob informs Carol to `save` the data, and Alice gives the final notification to Carol to `finish` the protocol.

Deniélou and Yoshida [12] present a decidable notion of *multiparty compatibility* as a generalisation of duality of binary sessions, for a given set of (more than two) CFSMs. The idea is that any single machine can see the rest of the machines as a single machine, up to *unobservable actions* (like a τ-transition in CCS). Therefore, we check the *duality* between each automaton and the rest, up to internal communications (*1-bounded executions* in the terminology of CFSMs) that the other machines will independently perform. For example, in Fig. 2, to check the compatibility of trace $\texttt{AB}!\texttt{quit}\cdot\texttt{AC}!\texttt{finish}$ in Alice, we observe the dual trace $\texttt{AB}?\texttt{quit}\cdot\texttt{AC}?\texttt{finish}$ from Bob and Carol, executing the internal communication between Bob and Carol: $\texttt{BC}!\texttt{save}\cdot\texttt{BC}?\texttt{save}$. If this extended duality is valid for all the machines from any 1-bounded reachable state, then they satisfy multiparty compatibility and can characterise a well-formed global type.

Our motivation to study this general compatibility comes from the need for using global types to develop tools for choreographic distributed testing in web service software [44] and distributed monitoring for large cyber-infrastructures [41], where local specifications are often updated independently and one needs to refine the original global specification according to the local updates.

The 1-boundedness and multiparty compatibility conditions are extended to the k-bounded condition in [28] (called k-*multiparty compatibility*). Another flexible form of safe and more asynchronous CFSMs (which do not rely on duality or buffer bounds) is studied in [7,14] (called *asynchronous subtyping*). Unfortunately, the asynchronous subtyping relation is undecidable, even if limited to only two machines; currently its decidable sound algorithms are restricted to either binary session types [3] or finite MPSTs [6].

Practically, a direct analysis based on CFSMs is computationally expensive, even if the shapes of CFSMs are limited. For building a toolchain for practical programming languages, we take the *safe-by-construction* approaches—we start from specifying a global type, and project it to endpoint types or CFSMs, for code generation into various programming languages, and/or other purposes. Interestingly, multiparty compatibility also helps enlarge the well-formedness condition of global types [23]. See Sect. 5.

1.2 νSCR: An Extensible Toolchain for Multiparty Session Types

We present a new toolchain for multiparty protocols, νSCR (NUSCR), for handling protocols written in the SCRIBBLE language. The implementation of MPST has three main aspects: **(1)** a language for specifying global interactions—specifically, the SCRIBBLE protocol description language [18]; **(2)** a tool to manipulate specifications and generate implementable APIs (SCRIBBLE [44] is already a mature industrial-strength tool able to generate APIs in multiple programming languages); and **(3)** a theory backing the safety guarantees, such as Featherweight SCRIBBLE [35], which minds the gap between the practical SCRIBBLE protocol description language, and the theoretical MPST specifications [21].

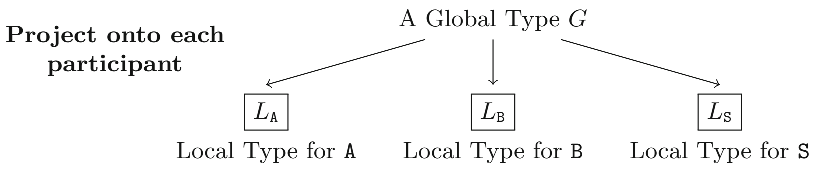


Fig. 3. Top-down methodology

The aim of this implementation is to be *lightweight* and *extensible*. Whilst the SCRIBBLE language describes more expressive protocols than the original MPST theory [21], νSCR handler a *core*, well-defined subset of SCRIBBLE protocols that have a corresponding MPST global type, following the formalisation of [35]. We do so in anticipation that further extensions of the MPST theory can be easily implemented in νSCR, and a researcher can smoothly integrate their own MPST design/theory in νSCR. For this purpose, we use a modular design that does not only enable future extensions, but also makes them easy to implement.

The rest of the paper is structured as follows: Sect. 2 introduces multiparty session types, the theoretical foundation of our tool; Sect. 3 introduces the νSCR toolchain; Sect. 4 presents a case study of extending νSCR with *nested protocols* [10]; and Sect. 5 summarises related work and concludes this paper. νSCR is publicly available at https://github.com/nuscr/nuscr/ under the GPLv3 license.

2 Multiparty Session Types (MPST)

In this section, we introduce the theoretical foundation of our tool—Multiparty Session Types (MPST) [21], a typing discipline for concurrent processes.

The main design philosophy of MPST follows a top-down approach (see Fig. 3): a *global type* describes a global view of a communication protocol between a number of participants. Each participant has their own perspective of the protocol, prescribed by their *local type*, which are obtained via an operation called *projection*. A local type for a participant can be used for code generation or type-checking to ensure that the participating process follows the local type. If all participating processes follow their corresponding local types, obtained via projection from a global type, these processes are free from communication mismatches or deadlocks, guaranteed by the MPST typing discipline.

Global and Local Types. We show the syntax of global types and local types in Fig. 4. The global type $\mathtt{p} \to \mathtt{q}\,\{l_i(S_i).G_i\}_{i \in I}$ is a message from $\mathtt{p}$ to $\mathtt{q}$, where $\mathtt{p} \neq \mathtt{q}$ and $I \neq \emptyset$. The message carries a *label* l_i and payload type S_i, selected from a non-empty index set I, and the protocol continues as G_i. We write $\mathtt{p} \to \mathtt{q} : l(S).G$ when $|I| = 1$. $\mathtt{end}$ denotes a type that is *terminated*. The local type $\mathtt{p}\&\{l_i(S_i).L_i\}_{i \in I}$ (resp. $\mathtt{p}\oplus\{l_i(S_i).L_i\}_{i \in I}$) denotes an *external choice* (resp. *internal choice*), where participant carrying this local type will *receive* (resp.

$$
\begin{array}{llll}
S ::= \texttt{int} \mid \texttt{bool} \mid \ldots & \text{Base Types} & L ::= & \text{Local Types} \\
G ::= & \text{Global Types} & \mid\ \texttt{p}\&\{l_i(S_i).L_i\}_{i\in I} & \text{External Choice} \\
\quad \mid\ \texttt{p} \to \texttt{q}\,\{l_i(S_i).G_i\}_{i\in I} & \text{Message} & \mid\ \texttt{p}\oplus\{l_i(S_i).L_i\}_{i\in I} & \text{Internal Choice} \\
\quad \mid\ \mu\mathbf{t}.G & \text{Recursion} & \mid\ \mu\mathbf{t}.L & \text{Recursion} \\
\quad \mid\ \mathbf{t} \ \mid\ \texttt{end} & \text{Type Var., End} & \mid\ \mathbf{t} \ \mid\ \texttt{end} & \text{Type Var., End}
\end{array}
$$

Fig. 4. Syntax of multiparty session types, in the style of [48]

send) a message from (resp. to) the participant $\texttt{p}$, among the index set I. Recursive types are realised by $\mu\mathbf{t}.G$ (resp. $\mu\mathbf{t}.L$) and $\mathbf{t}$, by taking a equi-recursive view (However, we require types to be contractive, e.g. $\mu\mathbf{t}.\mathbf{t}$ is not allowed).

We can obtain local types by *projecting* a global type upon a participant. Projection is defined as a *partial* function, since not all global types are implementable—these types might be unable to be implemented in a type-safe way. We say a global type is *well-formed*, if the projection of the global type upon all participants are *defined*. Well-formed global types can be implemented by a collection of concurrent processes, each implementing their projected local type. Well-typed processes will enjoy the benefit of the MPST typing discipline, are free from deadlocks or communication mismatches. Curious readers may refer to [48] for more details.

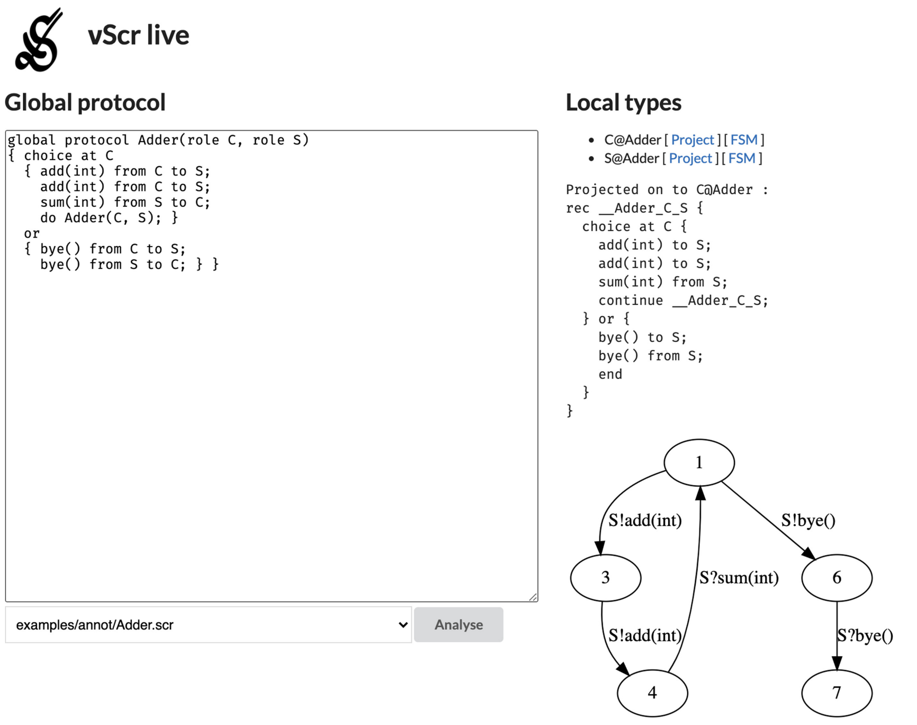
From Local Types to Communicating Finite State Machines. A local type describes the behaviour of a specific role in a given global type, which can be represented by a *communicating finite state machine*[1] *(CFSM)* [2]. As shown by Deniélou and Yoshida [12] and Neykova and Yoshida [35], there is an algorithm to construct a CFSM that is trace-equivalent to the local type.

***Relation to* Scribble.** SCRIBBLE [44,49] is a toolchain for implementing multiparty protocols. In particular, the syntax of the SCRIBBLE protocol description language correlates closely to the theory of MPST. Neykova and Yoshida [35] give a formal description of the SCRIBBLE protocol description language, known as Featherweight SCRIBBLE, and establish a correspondence between global protocols in Featherweight SCRIBBLE and global types in the MPST theory (Sect. 4).

3 νSCR: An Extensible Implementation of Multiparty Session Types in OCAML

In this section, we describe the structure of νSCR and highlight the correspondence to the multiparty session type theory. νSCR is written in OCAML in around 8000 lines of code, implementing the core part of the SCRIBBLE language, with various extensions to the original MPST. νSCR also has a web interface (https://nuscr.dev/), so that users can perform quick prototyping in browsers, saving the need for installation (see Fig. 5 for a screenshot).

[1] Also known as *endpoint finite state machine (EFSM)* [22].



```
global protocol Adder(role C, role S)
{ choice at C
  { add(int) from C to S;
    add(int) from C to S;
    sum(int) from S to C;
    do Adder(C, S); }
  or
  { bye() from C to S;
    bye() from S to C; } }
```

- C@Adder [Project] [FSM]
- S@Adder [Project] [FSM]

```
Projected on to C@Adder :
rec __Adder_C_S {
  choice at C {
    add(int) to S;
    add(int) to S;
    sum(int) from S;
    continue __Adder_C_S;
  } or {
    bye() to S;
    bye() from S;
    end
  }
}
```

Fig. 5. A screenshot of the νScr web interface, showing an `Adder` protocol

Overview. νScr is designed to be extensible, so that researchers working on MPST theories can find it easy to implement their extensions upon the code base of νScr. Inspired by Haskell, we use *language pragmas* to control language extensions, so that users do not need to download different versions of the software for different language extensions. Currently, two major extensions are implemented, namely nested protocols [10,13] and refinement types [51]. Protocols in the Scribble description language are accepted by νScr, and then converted into an MPST global type.

From a global type, νScr is able to project upon a specified participant to obtain their local type, and subsequently obtain the corresponding CFSM. Moreover, νScr is able to generate code for implementing the participant in various programming languages, from their local type or CFSM. νScr can be used either as a standalone command line application, or as an OCaml library for manipulating multiparty protocols.

Code Layout. The codebase of νScr can be briefly split into 4 components: `syntax`, `mpst`, `codegen` and `utils`. We introduce the components in detail.

Syntax. The `syntax` component handles the syntax of the Scribble protocol description language, the core part of which is shown in Fig. 6. We use OCamllex and Menhir to generate the lexer and parser respectively. A Scribble

$$
\begin{array}{rrll}
\text{Protocol Declarations } P & ::= & \texttt{global protocol } p\ (\texttt{role r}_1, \cdots, \texttt{role r}_n)\{G\} & \\
\text{Protocol Constructs } G & ::= & \texttt{l}(S)\ \texttt{from r}_1\ \texttt{to r}_2; G' & \text{Single Message} \\
 & \mid & \texttt{choice at r}\ \{G_1\}\ \texttt{or}\ \cdots\ \texttt{or}\ \{G_n\} & \text{Branches} \\
 & \mid & \texttt{rec}\ X\ \{G'\}\ \mid\ \texttt{continue}\ X & \text{Recursion / Var.} \\
 & \mid & \texttt{end}\ \text{(omitted in practice)} & \text{Termination} \\
 & \mid & \texttt{do}\ p(\texttt{r}_1, \cdots, \texttt{r}_n) & \text{Protocol Call} \\
\text{Base Types } S & ::= & \texttt{int} \mid \texttt{bool} \mid \cdots &
\end{array}
$$

Fig. 6. Syntax of core Scribble language

```
global protocol Adder(role C, role S)
{ choice at C
  { add(int) from C to S;
    add(int) from C to S;
    sum(int) from S to C;
    do Adder(C, S); }
  or
  { bye() from C to S;
    bye() from S to C; } }
```

(a) Adder Protocol in Scribble

$$
G_{\texttt{Adder}} = \mu \mathbf{t}.\texttt{C} \to \texttt{S} \left\{ \begin{array}{l} \mathit{add}(\texttt{int}). \\ \quad \texttt{C} \to \texttt{S} : \mathit{add}(\texttt{int}). \\ \quad \texttt{S} \to \texttt{C} : \mathit{sum}(\texttt{int}). \\ \quad \mathbf{t}; \\ \mathit{bye}(). \\ \quad \texttt{S} \to \texttt{C} : \mathit{bye}(). \\ \quad \texttt{end} \end{array} \right\}
$$

(b) Global Type of Adder Protocol

Fig. 7. Adder protocol and its corresponding global type

module consists of multiple protocol declarations P. In the syntax, protocol names are represented by p, role names by r, label names by l, and recursion variable names by X. The four kinds of names range over string identifiers. They are separated in distinct name spaces in νScr, and appropriately distinguished.

As a running example, we show a simple Scribble protocol describing an `Adder` protocol in Fig. 7a, where a `Client` is able to make various requests to `add` two `int`s, before they decide to finish the protocol with a `bye` message.

Multiparty Session Types. We show the key pipeline of handling multiparty session types in Fig. 8, implemented in the `mpst` component. An input file is parsed into a Scribble *module*, by the `syntaxtree` component described in the previous paragraph. The protocols are then converted into a *global type* (defined in `Gtype` module), which describes an overall protocol between multiple roles. A global type is *projected* into a *local type* (defined in `Ltype` module), given a specified role, which describes the local communication behaviour. We construct a corresponding *communicating finite state machine (CFSM)* [2] (defined in `Efsm` module) for the local type, and it can be used for API generation.

To obtain a global type, we extract it from the syntax tree of the Scribble protocol file. During this extraction process, we perform syntactic checks on the protocol, e.g. validating whether role names, recursion variables, and protocol names have been defined before they are used. We show the global type of the Adder protocol in Fig. 7b.

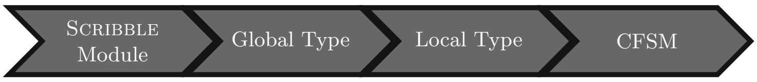


Fig. 8. Workflow of νScr

```
global protocol NonDirected (role A, role B, role C)
{ choice at A // A sends to either B or to C in this choice
     { Foo() from A to B; // either send to B
       Bar() from A to C; }
  or { Bar() from A to C; // or send to C
       Foo() from A to B; } }
```

Fig. 9. Non-directed choice in Scribble

It is important to note that, syntactically correct protocols may fall out of the expressiveness of the original MPST theory, e.g. the protocol shown in Fig. 9. The role `A` makes a choice of sending `Foo` to `B` first, or sending `Bar` to `C` first, which has no corresponding construct in the syntax (Fig. 4). Whilst some protocols fall out of the scope of the core MPST theory, an extension to the core theory may accept such protocols with non-directed choices.

The projection from global types upon participants is implemented in the `Ltype` module, and the projected local types can be converted into their corresponding communicating automata, using the technique described in [12]. We use the graph library OCamlgraph [8], to represent the CFSM as a directed graph. We show the local type for `Client` in Fig. 10a, and its corresponding CFSM in Fig. 10b. Both local types and communicating automata can be used for code generation purposes, which will be introduced in the next paragraph.

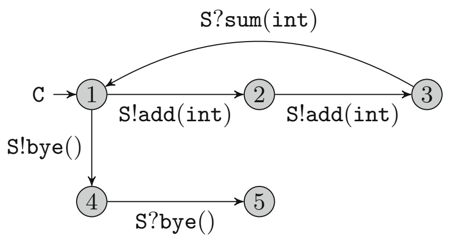
Code Generation. The `codegen` component generates APIs for implementing distributed processes using the MPST theory. Following the MPST design methodology, processes should follow the projected local type from the prescribed global type. By the means of code generation, the processes implemented using generated APIs will be *correct by construction*.

Currently, νScr supports code generation in OCaml, Go (with the nested protocol extension [13]) and F$\star$ (with the refinement type extension [51]). Moreover, νScr can export the CFSM as a GraphViz Dot file, and code generation backends can be implemented separately from νScr. This approach has been used to support code generation in Rust [9], Scala and TypeScript [31].

To generate code in OCaml, we use a CFSM-based generation technique, as proposed in [22]; however, we do not follow the class-based APIs, i.e. states in the CFSMs are classes, and state transitions are methods on classes in [22]. While it would be possible to implement a similar object oriented approach in OCaml, it does not fit well in the functional programming paradigm. νScr uses a *callback-based* approach [51] for API generation, and generates functions for transitions and maintains the state *internally* in a finite state machine runner.

$$L_{\mathtt{C}} = \mu\mathbf{t}.\mathtt{S}\oplus\left\{\begin{array}{l} add(\mathtt{int}). \\ \quad \mathtt{S}\oplus add(\mathtt{int}). \\ \quad \mathtt{S}\& sum(\mathtt{int}). \\ \quad \mathbf{t}; \\ bye(). \\ \quad \mathtt{S}\& bye(). \\ \quad \mathtt{end} \end{array}\right\}$$

(a) Local Type for Role C



(b) CFSM for role C

Fig. 10. Local type and CFSM for role C in the Adder protocol

```
module type Callbacks = sig
  type t (* An abstract type for user-maintained state *)

  (* Sending callbacks return the state and a labelled value to send *)
  val state1Send : t → t * [`bye of unit | `add of int]
  val state2Send : t → t * [`add of int]

  (* Receiving callbacks take received value as arguments,
   * and return the state *)
  val state3Receivesum : t → int → t
  val state4Receivebye : t → unit → t
end
```

Fig. 11. Generated module type in OCaml for role C

API Style. The generated API separates the program logic and communication aspects of the endpoint program, in contrast to existing approaches of code generation [22]. We generate type signatures of *callback functions*, corresponding to state transitions in the CFSM, for handling the program logic. The signatures are collected in the form of a module type, named `Callbacks`. We show the generated module signature in Fig. 11, for implement the Client in the Adder protocol (Fig. 10). Since we use a graph representation for CFSMs, the generation process is done by iterating through the edges of the graph.

For a complete endpoint, We generate an OCaml functor taking a module of type `Callbacks` to an implementation module. The module exposes a runner, which executes the CFSM when provided connections to other communicating roles. The runner handles the communication with other roles, so that the callback module does not need to involve any sending and receiving primitives.

Optional Monadic APIs. To enable asynchronous execution, we optionally generate code compatible with monadic communication primitives. This allows users to implement the endpoint program with popular asynchronous execution libraries in OCaml, such as Lwt [40].

```
(*# NestedProtocols #*)
nested protocol Fork
  (role M; new role W)
{ choice at M
  { Task() from M to W;
    M calls Fork(M);
    Result() from W to M; }
  or
  { End() from M to W; } }
```

```
global protocol ForkJoin
  (role M, role W)
{ choice at M
  { Task() from M to W;
    M calls Fork(M);
    Result() from W to M; }
  or
  { SingleTask() from M to W;
    Result() from W to M; } }
```

Fig. 12. A nested fork join protocol in Scribble [13, Fig. 7.3]

Utilities. The `utils` component contain miscellaneous modules fulfilling various utility functions. A few notable modules in this component, relevant to future extensions of νScr, include:

– `Names` module defines separated namespaces for all kinds of names occurring in global and local types, e.g. payload type names, payload label names, recursion variable names, etc.
– `Err` module defines all kinds of errors that occur throughout all components.
– `Pragma` module defines language pragmas, controlling the enabled extensions.

4 Extending νScr

The modular design of νScr allows extensions of the MPST theory to be implemented easily. The language pragmas, implemented as a special comment at the beginning of an input file, control which extensions are enabled when handling the protocols. So far, two major extensions have been added to νScr: *nested protocols* implemented by Echarren Serrano [13], and *refinement types* implemented by Zhou et al. [51]; and additional extensions are being implemented: *choreography automata* [1] by Neil Sayers, *parallel types* by Francisco Ferreira, and *crash handling* by Adam D. Barwell.

We use the *nested protocol* extension by Echarren Serrano [13] as a case study to demonstrate how an extension can be implemented in νScr. Nested protocols [10] allow dynamic creation of participants and sub-sessions in a protocol, extending the expressiveness of global types. In Fig. 12, we show a fork join protocol, described in Scribble with the nested protocol extension.

Creating a New Pragma. The first line of Fig. 12 enables the nested protocols extension using the *pragma* `NestedProtocols` (wrapped in `(*# #*)`). New pragmas are added in the `Pragma` module in the `utils` component, including a new constructor for the new pragma, and functions to get and set whether the extension is toggled. An implementer may also check for conflicting pragmas when processing all pragmas, so that incompatible extensions are not enabled at the same time. In order to preserve the behaviour when the extension is not enabled, it is essential that subsequent implementations of the extension should query whether the extension is enabled before proceeding.

Extending the Syntax. The extension allows *nested* protocols to be defined using the keyword `nested`, with the possibility to dynamically create new participating roles. In addition, a new construct `calls` is introduced to create a sub-session that follows a nested protocol, where new roles may be created to participate in the sub-session.

To implement these new syntactic constructs, an implementer should extend the `syntaxtree` component. To begin with, the concrete syntax tree (in the `Syntax` module) is to be extended with constructors for the new syntax, e.g. the new `calls` constructs in protocol body. Additional lexing or parsing rules should be added accordingly in the corresponding module.

Extending the MPST Theory. The crucial part is to implement the theory extension in the `mpst` component, where global and local types are defined. Within the component, global (resp. local) types are defined using the OCaml type `Gtype.t` (resp. `Ltype.t`). We add new constructors for new global types (`CallG` for protocol calls) and new local types (`InviteCreateL` for inviting and creating dynamic roles, and `AcceptL` for accepting invitations). Projection can be extended accordingly in the `Ltype` module, which we will not explain in detail.

However, extending the global and local types does not complete the extension—the implementer needs to connect the concrete syntax of Scribble global protocols to the abstract syntax of MPST global types. The extraction is defined at `Gtype.of_protocol`, where a global type is obtained from a global protocol. When processing the new syntactic constructs added by the extension, the implementer should remember to call `Pragma.nested_protocol_enabled` (which will return `true` when the pragma is set) to avoid interference with the core MPST, i.e. when the extension is not enabled.

Extending the Code Generation. Section 3 describes OCaml code generation from CFSMs. However, constructing CFSMs for nested protocols is an open problem. Hence, for this extension the Go code generation is instead based on *local types*. νScr code generation backends in the `codegen` component is free to choose any representation in the `mpst` component, so implementers may pick whichever representation that suits best their code generation approach.

The generated Go APIs also use callbacks, and the message passing primitives in Go with *channels* and concurrent execution with *goroutine*s fit the setup of session types very well. The code generator creates type definitions for different channels used in the communication, message exchanges, and callbacks. New participants in the protocol can be spawned when needed using goroutines.

5 Related and Future Work

Non-*Scribble*-Based MPST Implementations. Scalas and Yoshida [43] (accompanying artefact) implement a toolkit for analysing synchronous multiparty protocols. The underlying theory for this tool is the *generalised* multiparty session types, where the type system is parameterised on a safety property.

The toolkit uses a model checker (mCRL2 [46]) to decide whether the desired safety property holds. A shortcoming of this approach is that **(1)** the verification power is bound by the model checker—for example, mCRL2 allows to verify only finite-controlled local session types (no parallel compositions under recursion) and cannot verify channel passing; and **(2)** the approach is not scalable to asynchronous communication with unbounded buffers (as safety becomes undecidable). Several prototype tools that analyse safety of general forms of local types (or CFSMs) are developed in the context of multiparty CFSMs [27,28] and binary CFSMs [3]. The tool in [27] enables a *bottom-up approach*, which builds a global type from a set of safe local types. These approaches are, in general, high in complexity (requiring a global analysis to a set of CFSMs), and difficult to integrate with real programming languages because of the need to extract local types from source languages. For example, Ng and Yoshida [38] develop a tool (based on [27]) to build a global graph from local session types extracted from Go source codes, in order to check deadlock-freedom. Only a subset of Go syntax is supported [50].

Our top-down approach is based on the original, less general multiparty session type theory, yet we implement an extensible toolchain with possibilities to generate OCaml code for execution. Imai et al. [24] implement multiparty session types in OCaml with protocol *combinators*, whereas our approach takes inputs from Scribble protocols. Their tool uses features such as variant and object types in OCaml to encode external and internal choices in the local types, and supports session delegation. Our callback-based approach does not support delegation, but also does not require sophisticated type system features.

For more advanced applications of MPST, global types with motion primitives of Cyber Physical Systems [29,30] provide a collision freedom guarantee for concurrent robotics applications. Castro-Perez and Yoshida [6] use global types to uniformly predict communication costs of parallel algorithms and distributed protocols implemented in different languages.

Scribble-*Based MPST Implementations and Extensions.* The Scribble toolchain provides a language-agnostic description language for multiparty protocols, targeting a variety of programming languages: Java [22], Scala [42], Go [4], TypeScript [31], PureScript [25], Rust [9,26], F♯ [32], F⋆ [51], Erlang [33], Python [11,34], MPI-C [36,37], C [39], etc.

The Scribble toolchain describes multiparty protocols, including some that are not expressible in the MPST theory, e.g. the `choice` constructs in Scribble name a role making an internal choice, whereas the MPST global type has form $\mathtt{p} \to \mathtt{q}\,\{l_i(S_i).G_i\}$, where two roles are named. The protocol in Fig. 9 is expressible in Scribble, although not in the core MPST theory we implement.

The Scribble toolchain implements a number of extensions of MPST, e.g. explicit connections [23], interruptible protocols [11]. νScr implements the core MPST theory by our design choice, so that extensions of the MPST theory can be easily implemented, with the usual syntactical projections. Whilst the Scribble toolchain uses model checking and other validation techniques to verify the safety of the multiparty protocol to enlarge well-formedness, the same technique might not be applicable when extending the original MPST [21]. νScr keeps an underlying core syntax and its validation faithful to the literature, so that other users can easily integrate their own MPST theory. We demonstrate our extensibility via the case study with nested protocols.

Besides the Scribble toolchain itself, Voinea et al. [47] provide a tool, StMungo, to translate a Scribble multiparty protocol to a typestate specification in Java. The typestate specification can be checked via Mungo, a static typechecker for typestates in Java. Developers can use the generated typestate APIs to implement the multiparty protocol safely. Harvey et al. [17] use the Scribble toolchain with explicit connections [23] to develop a tool for affine multiparty session types with adaptations.

Future Work. Recently, Castro-Perez et al. [5] propose Zooid, a domain specific language for certified multiparty communication, embedded in Coq and implemented atop their mechanisation framework of asynchronous MPST. For future work, we would like to produce a certified version of Scribble—CertiScr, extending the Zooid framework, for the core of νScr. We would like to mechanise the extraction process from Scribble to global types in [5], and the CFSM construction process from local types in Coq, completing the picture of fully mechanised toolchain from Scribble protocols to CFSMs. The Coq code can then be extracted into OCaml to produce a formally verified νScr core.

Acknowledgements. We thank Simon Castellan for the initial collaboration on νScr project. The work is supported by EPSRC EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1, EP/T014709/1 and EP/V000462/1, and NCSS/EPSRC VeTSS.

References

1. Barbanera, F., Lanese, I., Tuosto, E.: Choreography automata. In: Bliudze, S., Bocchi, L. (eds.) COORDINATION 2020. LNCS, vol. 12134, pp. 86–106. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50029-0_6
2. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983). https://doi.org/10.1145/322374.322380
3. Bravetti, M., Carbone, M., Lange, J., Yoshida, N., Zavattaro, G.: A sound algorithm for asynchronous session subtyping and its implementation. Log. Methods Comput. Sci. **17**(1), March 2021. https://lmcs.episciences.org/7238
4. Castro, D., Hu, R., Jongmans, S.S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. Proc. ACM Program. Lang. 3 (POPL), January 2019. https://doi.org/10.1145/3290342
5. Castro-Perez, D., Ferreira, F., Gheri, L., Yoshida, N.: Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, New York, NY, USA, pp. 237–251. Association for Computing Machinery (2021). https://doi.org/10.1145/3453483.3454041

6. Castro-Perez, D., Yoshida, N.: CAMP: cost-aware multiparty session protocols. Proc. ACM Program. Lang. 4 (OOPSLA), November 2020. https://doi.org/10.1145/3428223
7. Chen, T.C., Dezani-Ciancaglini, M., Scalas, A., Yoshida, N.: On the preciseness of subtyping in session types. Log. Methods Comput. Sci. **13**(2), June 2017. https://lmcs.episciences.org/3752
8. Conchon, S., Filliâtre, J.C., Signoles, J.: OCamlgraph: An OCaml Graph Library (2017). http://ocamlgraph.lri.fr/index.en.html. Accessed 21 May 2021
9. Cutner, Z., Yoshida, N.: Safe session-based asynchronous coordination in rust. In: Damiani, F., Dardha, O. (eds.) COORDINATION 2021. LNCS, vol. 12717, pp. 80–89. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78142-2_5
10. Demangeon, R., Honda, K.: Nested protocols in session types. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 272–286. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_20
11. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. Formal Methods Syst. Des. **46**(3), 197–225 (2014). https://doi.org/10.1007/s10703-014-0218-8
12. Deniélou, P.-M., Yoshida, N.: Multiparty compatibility in communicating automata: characterisation and synthesis of global session types. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013. LNCS, vol. 7966, pp. 174–186. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39212-2_18
13. Echarren Serrano, B.: Nested multiparty session programming in go. Master's thesis, Imperial College London (2020). https://becharrens.files.wordpress.com/2020/07/final_report.pdf
14. Ghilezan, S., Pantović, J., Prokić, I., Scalas, A., Yoshida, N.: Precise subtyping for asynchronous multiparty sessions. Proc. ACM Program. Lang. 5 (POPL), January 2021. https://doi.org/10.1145/3434297
15. Girard, J.Y.: Linear logic. Theor. Comput. Sci. **50**(1), 1–101 (1987). https://www.sciencedirect.com/science/article/pii/0304397587900454
16. Gouda, M.G., Manning, E.G., Yu, Y.T.: On the progress of communication between two machines. In: Maekawa, M., Belady, L.A. (eds.) IBM 1980. LNCS, vol. 143, pp. 369–389. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-11604-4_62
17. Harvey, P., Fowler, S., Dardha, O., Gay, S.J.: Multiparty session types for safe runtime adaptation in an actor language. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming (ECOOP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 194, pp. 10:1–10:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://drops.dagstuhl.de/opus/volltexte/2021/14053
18. Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N.: Scribbling interactions with a formal foundation. In: Natarajan, R., Ojo, A. (eds.) ICDCIT 2011. LNCS, vol. 6536, pp. 55–75. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19056-8_4
19. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053567

20. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, New York, NY, USA, pp. 273–284. ACM (2008). http://doi.acm.org/10.1145/1328438.1328472
21. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**, 1–67 (2016)
22. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: Stevens, P., Wasowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 401–418. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_24
23. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 116–133. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_7
24. Imai, K., Neykova, R., Yoshida, N., Yuen, S.: Multiparty session programming with global protocol combinators. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming (ECOOP 2020), pp. 9:1–9:30. Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://drops.dagstuhl.de/opus/volltexte/2020/13166
25. King, J., Ng, N., Yoshida, N.: Multiparty session type-safe web development with static linearity. In: Martins, F., Orchard, D. (eds.) Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, Prague, Czech Republic, 7th April 2019. Electronic Proceedings in Theoretical Computer Science, vol. 291, pp. 35–46. Open Publishing Association (2019)
26. Lagaillardie, N., Neykova, R., Yoshida, N.: Implementing multiparty session types in rust. In: Bliudze, S., Bocchi, L. (eds.) COORDINATION 2020. LNCS, vol. 12134, pp. 127–136. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50029-0_8
27. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, New York, NY, USA, pp. 221–232. Association for Computing Machinery (2015). https://doi.org/10.1145/2676726.2676964
28. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 97–117. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_6
29. Majumdar, R., Pirron, M., Yoshida, N., Zufferey, D.: Motion session types for robotic interactions (brave new idea paper). In: Donaldson, A.F. (ed.) 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, pp. 28:1–28:27. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). http://drops.dagstuhl.de/opus/volltexte/2019/10820
30. Majumdar, R., Yoshida, N., Zufferey, D.: Multiparty motion coordination: from choreographies to robotics programs. Proc. ACM Program. Lang. 4 (OOPSLA), November 2020. https://doi.org/10.1145/3428202
31. Miu, A., Ferreira, F., Yoshida, N., Zhou, F.: Communication-safe web programming in TypeScript with routed multiparty session types. In: Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction, CC 2021, New York, NY, USA, pp. 94–106. Association for Computing Machinery (2021). https://doi.org/10.1145/3446804.3446854

32. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in F#. In: Proceedings of the 27th International Conference on Compiler Construction, CC 2018, New York, NY, USA, pp. 128–138. ACM (2018). http://doi.acm.org/10.1145/3178372.3179495
33. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: Proceedings of the 26th International Conference on Compiler Construction, CC 2017, New York, NY, USA, pp. 98–108. Association for Computing Machinery (2017). https://doi.org/10.1145/3033019.3033031
34. Neykova, R., Yoshida, N.: Multiparty session actors. Log. Methods Comput. Sci. **13**(1), March 2017. https://lmcs.episciences.org/3227
35. Neykova, R., Yoshida, N.: Featherweight scribble. In: Boreale, M., Corradini, F., Loreti, M., Pugliese, R. (eds.) Models, Languages, and Tools for Concurrent and Distributed Programming. LNCS, vol. 11665, pp. 236–259. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21485-2_14
36. Ng, N., de Figueiredo Coutinho, J.G., Yoshida, N.: Protocols by default. In: Franke, B. (ed.) CC 2015. LNCS, vol. 9031, pp. 212–232. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46663-6_11
37. Ng, N., Yoshida, N.: Pabble: parameterised Scribble. SOCA **9**(3), 269–284 (2014). https://doi.org/10.1007/s11761-014-0172-8
38. Ng, N., Yoshida, N.: Static deadlock detection for concurrent go by global session graph synthesis. In: Proceedings of the 25th International Conference on Compiler Construction, CC 2016, New York, NY, USA, pp. 174–184. Association for Computing Machinery (2016). https://doi.org/10.1145/2892208.2892232
39. Ng, N., Yoshida, N., Honda, K.: Multiparty session C: safe parallel programming with message optimisation. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 202–218. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30561-0_15
40. Ocsigen: Lwt Manual (2021). https://ocsigen.org/lwt/latest/manual/manual. Accessed 21 May 2021
41. OOI: Ocean Observatories Initiative (2020). http://www.oceanobservatories.org/
42. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: Müller, P. (ed.) 31st European Conference on Object-Oriented Programming (ECOOP 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 74, pp. 24:1–24:31. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). http://drops.dagstuhl.de/opus/volltexte/2017/7263
43. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. 3 (POPL), January 2019. https://doi.org/10.1145/3290343
44. Scribble Authors: Scribble: Describing Multi Party Protocols (2015). http://www.scribble.org/. Accessed 21 May 2021
45. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Maritsas, D., Philokyprou, G., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58184-7_118
46. Technische Universiteit Eindhoven: mCRL2 (2018). https://www.mcrl2.org/web/user_manual/index.html
47. Voinea, A.L., Dardha, O., Gay, S.J.: Typechecking Java protocols with [St]Mungo. In: Gotsman, A., Sokolova, A. (eds.) FORTE 2020. LNCS, vol. 12136, pp. 208–224. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50086-3_12

48. Yoshida, N., Gheri, L.: A very gentle introduction to multiparty session types. In: Hung, D.V., D'Souza, M. (eds.) ICDCIT 2020. LNCS, vol. 11969, pp. 73–93. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-36987-3_5
49. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The Scribble protocol language. In: Abadi, M., Lluch Lafuente, A. (eds.) TGC 2013. LNCS, vol. 8358, pp. 22–41. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05119-2_3
50. Yuan, T., Li, G., Lu, J., Liu, C., Li, L., Xue, J.: GoBench: a benchmark suite of real-world go concurrency bugs. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 187–199 (2021)
51. Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically verified refinements for multiparty protocols. Proc. ACM Program. Lang. 4 (OOPSLA), November 2020. https://doi.org/10.1145/3428216