Programming Language Implementations with Multiparty Session Types *

Nobuko Yoshida[0000−0002−3925−8557]

University of Oxford, UK

Abstract. Session types provide a typing discipline for communication systems, and a number of programming languages are integrated with session types. This paper provides a survey of programming language implementations which use the structuring mechanism from *multiparty session types* (*MPST*). The theory of MPST guarantees that processes following a predefined communication protocol (a *multiparty session*) are free from communication errors and deadlocks. We discuss the *top-down*, *bottom-up* and *hybrid* MPST frameworks, and compare their positive and negative aspects, through a Rust MPST implementation framework, Rumpsteak. We also survey MPST implementations with dynamic (runtime) verification which target active object programming languages.

1 Introduction

Since the first implementation work which integrates session types [68,27] into the mainstream programming language, Java [32], the session types community has been actively engaged with implementations or integration of session types into various programming languages and tools. This survey focuses on the programming language implementations and tools based on *multiparty session types* (MPST) [28,29].

Initially, session types had a main open problem, repeatedly posed by industry partners and researchers: whether the original *binary* session types [68,27] can be extended to multiparty (i.e. more than two parties). This is a natural question since most of business and distributed protocols and parallel computations are written in multiparty communications. The hint to discover a multiparty session type theory had come from an abstract version of "choreography" developed in W3C Web Service Choreography Description Language (WS-CDL) group [10]. Since the idea was first published in [28], it has been studied and used from many different theoretical and practical aspects in the research community, such as the automata theory, model checking, runtime verification, linear logic, workflows, contracts and mechanisation. With RedHat, multiparty session types

* This research was funded in whole, or in part, by EPSRC EP/T006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462/1, EP/X015955/1, NCSS/EPSRC VeTSS and Horizon EU TaRDIS 101093006.

have opened their way to industry with the new JBoss Scribble Project (a language to *describe* multiparty session types). In the U.S., Ocean Observatories Initiative (OOI) [64] deployed dynamic runtime checking using Scribble for historically large cyberinfrastructures. A new industry-led application domain of MPST is *microservices*– Estafet commercialised a tool which generates Go code for microservices from Scribble [17].

After nearly 15 years from the birth of MPST, as far as we have known, MPST is integrated over 16 different programming languages. Moreover, for some languages, several MPST tools exist: for example, research came up with various MPST tools integrated in Java, and that has led to different MPST-flavoured Java versions or related technologies such as Scribble.

Among the wide range of formal methods for verifying communicating systems, the MPST framework offers a direct link to programming primitives that *digest* the structures and dynamics of multiple communicating components. Specifically:

1. Multiparty session types offer clean abstractions of communicating behaviour as a *protocol*, defining a fundamental *Application Programming Interface* (API) of components, aiding modular development and well-structured engineering;
2. Multiparty session types give a *scalable* automatic verification method without *state-space explosion problems*, extensible to check more advanced/general properties, applying model-checking tools; and
3. Multiparty session types offer a foundation for more refined verification methods, such as the elaboration of components' type signature with assertions and monitoring and tracing behaviours of the systems.

The key element of MPST is a *global type*, which globally (i.e. in a bird's eye view) describes how message exchanges in a conversation (or *session*) proceed among its participants (*end-points*). To obtain the local protocol which an end-point should obey from a global protocol, we project the local portion of a global protocol onto each end-point, giving the end-point's *interface* with respect to that protocol. This local interface generalises the familiar notion of API, which can be regarded as the server-side projection of a two-party call-return protocol. One can then use, at each end-point, these projected local protocols to concurrently build and test an end-point system conforming to the local protocols so that the original global protocols are obeyed in the interactions among these systems.

The first part of this paper outlines three different MPST frameworks using a MPST Rust toolchain, Rumpsteak, as an example. The second part gives a summary of all MPST programming language implementations since 2008 and compares them through several criteria. The first part of this paper is an extended version of a short paper which appeared in [12]. A part of a survey of the top-down framework explained in § 3.1 is an expansion from [41, § 6.2], including the recent MPST implementations published after [41].

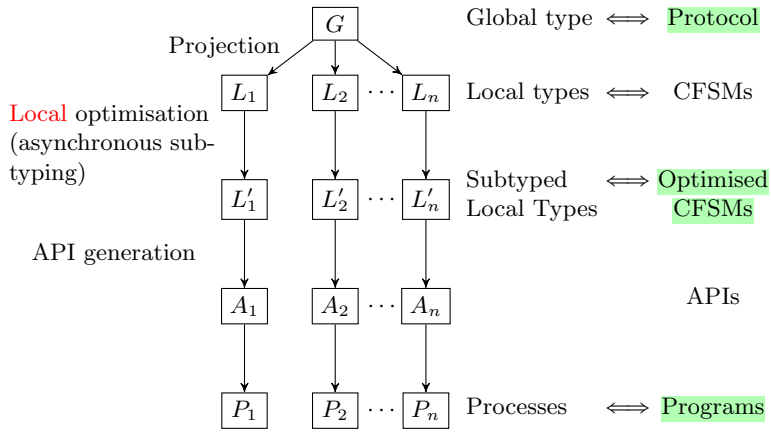


Fig. 1: Top-down MPST methodology: Highlight is supplied/done by user.

2 Multiparty Session Type Frameworks

This section explains the three frameworks of the multiparty session types (MPST), which combine the asynchronous message optimisation. We use the Rust framework, Rumpsteak [13], for the illustration as the toolchain implements the three frameworks. We start from the most standard and commonly used *top-down* framework, which can ensure *correctness by construction*.

2.1 Top-Down Multiparty Session Type Framework

Workflow. Fig. 1 presents the top-down MPST methodology. As the first step, we write a *global type* G to describe the interactions between all roles, and project it onto each role to obtain an *endpoint local type* L_i; then we apply *asynchronous subtyping* [21] to optimise each L_i to obtain L'_i (denoted by $L'_i \leqslant L_i$); and finally, we type-check each process P_i by L'_i. Hence the group of processes $P_1 ... P_n$ created in this way are free from communication errors such as deadlocks.

In the Rumpsteak tool-chain (its stages correspond to the right-hand side in Fig. 1), a global type is written as a *protocol*, each local type is represented as a *communicating finite state machine* (CFSM) [5] (we denote a CFSM by M). The highlight denotes the part supplied by the user. More specifically, the implementation is conducted by the following steps: in

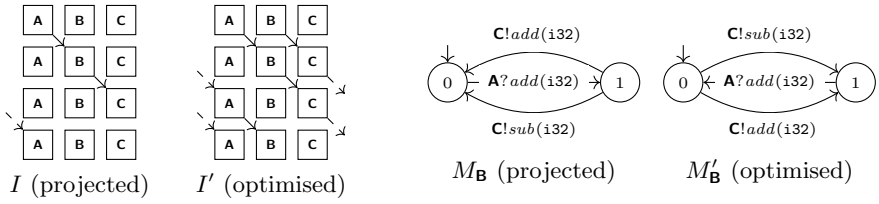
Step 1 we write a protocol to describe the interactions, and project it onto each role to obtain an endpoint communicating finite state machine (CFSM) M_i;
Step 2 we optimise each M_i to obtain M'_i;
Step 3 we *generate* an API A_i from each M'_i; and
Step 4 we use each A_i to create an asynchronous Rust process P_i.

$$G = \mu \mathbf{t}.\mathbf{A} \to \mathbf{B} : \left\{ add(\mathtt{i32}).\mathbf{B} \to \mathbf{C} : \left\{ \begin{array}{l} add(\mathtt{i32}).\mathbf{C} \to \mathbf{A} : \{add(\mathtt{i32}).\mathbf{t}\} \\ sub(\mathtt{i32}).\mathbf{C} \to \mathbf{A} : \{sub(\mathtt{i32}).\mathbf{t}\} \end{array} \right\} \right\}$$

$$L_{\mathbf{B}} = \mu \mathbf{t}.\mathbf{A}?add(\mathtt{i32}).\{\mathbf{C}!add(\mathtt{i32}).\mathbf{t} \oplus \mathbf{C}!sub(\mathtt{i32}).\mathbf{t}\}$$

$$L'_{\mathbf{B}} = \mu \mathbf{t}.\{\mathbf{C}!add(\mathtt{i32}).\mathbf{A}?add(\mathtt{i32}).\mathbf{t} \oplus \mathbf{C}!sub(\mathtt{i32}).\mathbf{A}?add(\mathtt{i32}).\mathbf{t}\}$$

Fig. 2: Global type (top) and the original $L_{\mathbf{B}}$ and optimised $L'_{\mathbf{B}}$ local types (bottom) for the `ring-choice` protocol



I (projected) I' (optimised) $M_{\mathbf{B}}$ (projected) $M'_{\mathbf{B}}$ (optimised)

Fig. 3: Ring protocol: (Left) Projected and optimised interactions; (Right) Projected and optimised session CFSMs.

End-Point Projection. For illustration, we use a ring protocol extended with choice (`ring-choice`) whose global type G is given in Fig. 2 (top). Role **B** chooses between sending an *add* or a *sub* message to role **C**, which must in turn send the same label to role **A**. We then project G into each role to obtain a set of local types. Fig. 2 (bottom) gives a local type of role **B** (denoted by $L_{\mathbf{B}}$) where ! and ? denote send and receive respectively, and $\oplus$ denotes the output (internal) choice.

In the implementation, for **[Step 1]**, Rumpsteak uses νScr [63,75], which is a new lightweight and extensible Scribble toolchain implemented in OCaml. The Scribble language [26,73] is widely used to describe multiparty protocols, agnostic to target languages. Then the tool generates a CFSM for each role. The generated CFSM for role **B** (denoted by $M_{\mathbf{B}}$) is given in Fig. 3 (right).

Asynchronous Message-Reordering Optimisation. A protocol G is *synchronous*—i.e., naïvely projecting it onto **B** produces an overly synchronised local type $L_{\mathbf{B}}$. If **A** is slow to send its value to **B** then the entire interaction is blocked (as shown in I in Fig. 3). Instead, assuming each process begins with its own initial value, **B** could send its value to **C** in the meantime, allowing **C** to begin its next iteration (as shown in I' in Fig. 3).

Therefore, in **[Step 2]**, we transform $L_{\mathbf{B}}$ into the optimal $L'_{\mathbf{B}}$ in Fig. 2. Importantly, we ensure that **(1)** no data dependencies exist between interactions, allowing their order to be changed; and **(2)** $L'_{\mathbf{B}}$ is an *asynchronous subtype* [21] of $L_{\mathbf{B}}$ ($L'_{\mathbf{B}} \leqslant L_{\mathbf{B}}$), allowing it to *safely* be used as a substitution while preserving deadlock-freedom. The CFSM representations of $L_{\mathbf{B}}$ and $L'_{\mathbf{B}}$ are given in $M_{\mathbf{B}}$ and $M'_{\mathbf{B}}$ in Fig. 3, respectively. While the asynchronous subtyping is proven

undecidable [45], Rumpsteak implements the sound decidable algorithm which calculates approximately whether $M'_{\mathbf{B}}$ is a subtype of $M_{\mathbf{B}}$ [13].

Code Generation. While in the theory, we do not have this step, Rumpsteak includes a code generator to produce an API in [**Step 3**]. Listing 1 shows the API $A_{\mathbf{B}}$ corresponding to the CFSM $M'_{\mathbf{B}}$, from which we have elided other participants. To ensure that our API remains readable by developers and to eliminate extensive boilerplate code, we make use of Rust procedural macros [69]. By decorating types with `#[...]`, these macros perform additional compile-time code generation. For each role, we generate a struct storing its communication channels with other roles. For example, **B** (line 3) contains unidirectional channels from **A** and to **C** as per the protocol. We use `#[derive(Role)]` to retrieve channels from the struct.

We build a set of *generic primitives* to construct a simple API—reducing the amount of generated code and avoiding arbitrarily named types. For instance, the `Receive` primitive (line 22) takes a role, label and continuation as generic parameters. For readability, we elide two additional parameters used to store channels at runtime with `#[session]`.

Each choice generates an enum, as seen in `RingBChoice` (line 21), allowing processes to pattern match when branching to determine which label was received. Methods allowing the enum to be used with `Branch` or `Select` primitives are also generated with `#[session]`. An enum is required since Rust's lack of variadic generics means choice cannot be easily implemented as a primitive. We show how the `RingBChoice` type can be used with selection in the `Ring` type (line 18).

Our API requires only one session type for each role, internally sending a `Label` enum (line 9) over reusable channels. We create a type for each label (lines 14 and 15) and use `#[derive(Message)]` to generate methods for converting to and from the `Label` enum.

Process Implementation. In theory, this final step has been done by implementing an end-point process P_i and type-checking it against a local type L_i. In Rumpsteak, we use the API to implement a Rust process. Using the API $A_{\mathbf{B}}$, we give a possible implementation of the process $P_{\mathbf{B}}$, shown in Listing 2, for [**Step 4**]. Linear usage of channels is checked by Rust's *affine type system* to prevent channels from being used multiple times. When a primitive is executed, it consumes itself, preventing reuse, and returns its continuation.

To warn the programmer when a session is discarded without use, we ensure this *statically* by harnessing the type checker. Developers are prevented from constructing primitives directly using visibility modifiers and must instead use `try_session` (line 5). Its closure argument accepts the input session type and returns the terminal type `End`. If a session is discarded, breaking linearity, then the developer will have no `End` to return and the type checker will complain. Even so, we can implement processes with infinitely recursive types (containing no `End`) such as `RingB`.

```
#[derive(Role)]
#[message(Label)]
struct B {
  #[route(A)] a: Receiver,
  #[route(C)] c: Sender,
}

#[derive(Message)]
enum Label {
  Add(Add),
  Sub(Sub),
}

struct Add(i32);
struct Sub(i32);

#[session]
type RingB = Select<C, RingBChoice>;

#[session]
enum RingBChoice {
  Add(Add, Receive<A, Add, RingB>),
  Sub(Sub, Receive<A, Add, RingB>),
}
```

Listing 1: Rust session type API for $M'_{\mathbf{B}}$ ($A_{\mathbf{B}}$)

```
async fn ring_b(
  role: &mut B,
  mut input: i32,
) -> Result<Infallible> {
  try_session(
    role,
    |mut s: RingB<'_, _>| async {
      loop {
        let x = input * 2;
        s = if x > 0 {
          let s = s.select(Add(x)).await?;
          let (Add(y), s) = s.receive().await?;
          input = y + x;
          s
        } else {
          let s = s.select(Sub(x)).await?;
          let (Add(y), s) = s.receive().await?;
          input = y - x;
          s
        };
      }
    },
  ).await
}
```

Listing 2: Possible Rust implementation for process $\mathbf{B}$ ($P_{\mathbf{B}}$) using $A_{\mathbf{B}}$

We use an infinite loop (line 8) which is assigned `Infallible`: Rust's never (or bottom) type. `Infallible` can be implicitly cast to any other type, including `End`, allowing the closure to pass the type checker as before.

We allow roles to be reused across sessions since the channels they contain can be expensive to create. Crucially, to prevent communication mismatches between different sessions, `try_session` takes a *mutable* reference to the role. The same role, therefore, cannot be used multiple times at once because Rust's borrow checker enforces this requirement for mutable references.

2.2 Bottom-Up Multiparty Session Type Framework

A bottom-up framework applies the *global analysis* to check a set of local types or CFSMs satisfy a certain safety property such as communication safety or deadlock-freedom. For this, we require to use an additional general-purpose verification tool such as the *k-multiparty compatibility tool* (KMC) [46] or the mCRL2 [50].

Fig. 4 depicts the two ways to perform the bottom-up strategies. In the left hand side, the user writes local types or CFSMs and generates APIs; and in the right hand side, each CFSM is generated from the API. In this approach, the user does not start from a global protocol, but starts from a set of local types/CFSMs or APIs.

The theory which corresponds to the bottom-up approach is given in [66]. This theory develops both synchronous and asynchronous semantics, but the model checking tool (mCRL2) is only usable for the synchronous version. This is

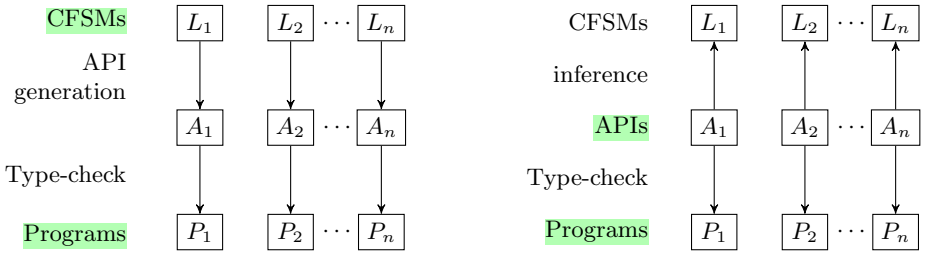


Fig. 4: Bottom-up MPST methodology: The tool globally analyses whether the set $\{L_i\}_{i \in I}$ satisfies a property. (Left) The user writes CFSMs and the tool generates APIs; (Right) the CFMSs are inferred from user-written APIs. Highlight is supplied/done by the user.

because checking a safety property in asynchronous CFSMs with infinite FIFO queues is undecidable.

To realise the bottom-up approach (right) in the Rumpsteak implementation, we first *serialise* each API A_i to obtain a CFSM M'_i. Next, we use KMC on the set of CFSMs $M'_{1\ldots n}$. If they are indeed compatible, then the processes $P_{1\ldots n}$, which implement their respective APIs, are free from communication-mistmatch and deadlocks. KMC takes a set of CFSMs for all participants and verifies deadlock freedom. To perform the serialisation of an API to a CFSM, we provide a Rust function `serialize<S>() -> Fsm` (this is a simplified version). It takes a session type API as a generic type parameter `S` and returns its corresponding CFSM. This CFSM can be printed in a variety of formats and passed into the KMC tool for verification.

Top-Down vs Bottom-Up Frameworks. The benefit of the bottom up approach is that the user does not have to write down a global type. On the other hand, the bottom-up approach has a number of disadvantages:

Complexity KMC and mCRL2 conduct a *global analysis* of a set of CFSMs. The complexity of global verification is high–in general, the complexity of a safety property checking by mCRL2 is exponential w.r.t. the size of CFSMs. Checking k-multiparty compatibility is PPRIME [46]. From the implementation side, analysing the endpoint CFSMs for all participants in the protocol at once is challenging to do scalablely. The asynchronous subtyping checks the optimisation of *a single participant's CFSM in isolation*, performing a local analysis of a single participant. Hence the top-down framework has much less complexity. See [13, Theorem 9] for detailed complexity analysis;

Expressiveness while KMC allows a bounded verification for asynchronous CFSMs, mCRL2 is not applicable to asynchronous CFSMs.

Implementations it is often very tedious to implement a tool which can infer CFSMs or local types from a user-written real-world program [59]. In Rumpsteak, the inference is doable from a specialised API which takes a similar form to a CFSM; and

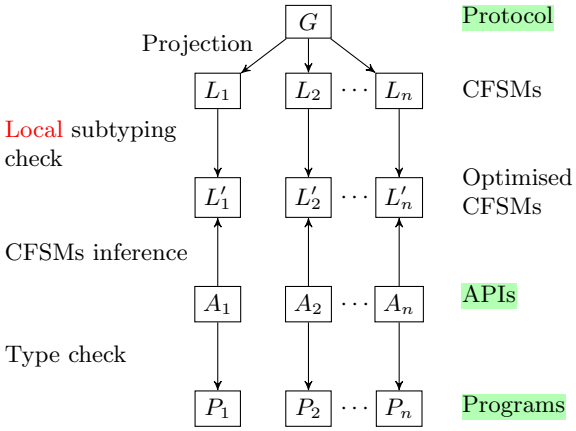


Fig. 5: Hybrid MPST methodology: Highlight is supplied/done by user.

Debugging when a KMC or mCRL2 analysis fails, it is difficult to determine how a programmer should update a complex protocol to make it free from deadlocks. Safety by construction, as used in the top-down approach, is easier to work with since verification is done locally on each participant.

2.3 Hybrid Multiparty Session Type Framework

The third framework, *hybrid*, approach (Fig. 5) is a combination of these two approaches. In this workflow, a global type G is provided by the developer and projected to obtain the CFSMs $M_{1...n}$ as before. Rather than the developers proposing the optimised CFSMs $M'_{1...n}$ directly, they simply write the APIs $A_{1...n}$ (as in the bottom-up approach). These are serialised to $M'_{1...n}$ which can (as in the top-down approach) be checked for safety against $M_{1...n}$ using asynchronous subtyping. In essence, the hybrid approach uses the same theory as the top-down approach, but presents a more programmer-friendly interface that uses serialisation rather than code generation.

The paper [13] gives more detailed complexity analysis and benchmark results which compare the local optimisation (in the top-down and hybrid frameworks) and the global analysis (in the bottom-up approach).

3 Multiparty Session Type Language Implementations

This section gives a survey of the programming language implementations based on multiparty session types (MPST). The previous section has discussed the *static* top-down, bottom-up and and hybrid approaches. The term "*static*" means that we verify safety of a program at the compile time. There is another approach, called *dynamic* where a program conformance against a specification

(session type) is checked at runtime. The dynamic approach is often called *runtime verification*, and this framework also fits well for active object and actor languages. We discuss (1) the static top-down approach (§ 3.1); (2) the dynamic top-down approach (§ 3.2); and (3) the static bottom-up approach (§ 3.3). In (3), we also include the bottom-up tools which use behavioural types.

3.1 Static Top-Down Multiparty Session Type Framework

Table 1 gives a summary of the programming language implementations based on MPST, ordered by date of publication, focusing on statically typed languages.

The table is composed as follows, row by row:

Languages lists the programming languages introduced or used.

Mainstream language states if the language is broadly used among developers or not.

Linearity checking describes whether the linear usage of channels is not checked, checked at compile-time (*static*) or checked at runtime (*dynamic*).

Exhaustive choices check indicates whether the implementation can *statically* enforce the correct handling of potential input types. ✗ denotes implementations that do not support pattern-matching to carry out choices (branching) which are encoded into switch statements on enum types.

Formalism defines the theoretical foundations of the implementations, such as (1) the end point calculus (the π-calculus (noted as π-cal.), FJ [33]) or Mini-MPI; (2) the (global) types formalism without any endpoint calculi (no typing system is given, and no subject reduction theorem is proved); (3) the formalism based on CFSMs or (4) no formalism is given (no theory is developed).

Communication safety outlines the presence or the absence of session type-soundness demonstration. The languages, marked as △, provide the type safety only at type or CFSM level. ✗• means that the theoretical formalism does not provide linear types, therefore only type safety of base values is proved.

Deadlock-freedom is a property guaranteeing that all components are progressing or ultimately terminate (which correspond to deadlock-freedom in MPST). The languages marked by △ proved deadlock-freedom only at the type level. ✓• implies the absence of a formal link with the local configurations reduced from the projection of a global type. [24] did not prove that any typing context reduced from a projection of a well-formed global type satisfies a safety property. Hence, deadlock-freedom is not provided for processes initially typed by a given global type.

Liveness is a property which ensure that all actions are eventually communicated with other parties (unless killed by an exception in those which treat failures [41,3]).

Notice that the *termination* property is a subset of safety but not deadlock-freedom. For example, the ring protocol given in the previous section does not terminate but deadlock-free and live. See [66].

Most of the MPST implementations [31,65,55,6,39,51,76,71,13,41,20,9,3,4] follow the API generation methodology from Scribble introduced by [30], which was explained in § 2.1. One of the main benefits of this methodology [30] is that it empowers IDEs to provide auto-completion for developers. See [51, Fig. 6] for an example.

Notice that the implementations denoted by "dynamic" in the row of "linearity check" are not completely static: they dynamically check linearity of channels at runtime.

The tool [58] automatically generates paralleled endpoint MPI-C programs, using the aspect oriented tool which takes a sequential kernel and a MPST protocol as the input. Another MPI-C implementation [47] uses a global type extended with the indexed dependent types to statically type check the MPI code without the end-point projection (hence two cells are marked as N/A).

The earlier tool [40] implements static type-checking of communication protocols by linking Java classes and their respective typestate definitions generated from Scribble. Objects declaring a typestate should be used linearly, but a linear usage of channels is not statically enforced. Rust implementations in [41,13] can check linearity using the built-in affinity type checking from Rust.

The functional language implementation [35] uses type-level embedding of multiparty channels in OCaml. Their library relies on OCaml-specific parametric polymorphism for variant types to ensure type-safety and the implementation uses a non-trivial, comprehensive encoding of polymorphic variant types and lenses. The survey [39] gives the detailed explanations about the advantages of functional languages to handle linearity of session channels.

Recent works [51,76,71,24,9,3] use the *call-back style* API generations to statically guarantee channel linearity. The recent Scala tool [11] guarantees channel linearity by a new API generation based on the pomsets theory (instead of the FSM-based generation [30] explained in § 2.1), exploring a facility provided by the matched types in Scala 3.

Built on the actor language framework Ensemble, the work [24] builds EnsembleS which generates a skeleton code based on the StMungo tool [40]. Static session typechecking is supported by modifying the original Ensemble typechecker to ensure that each communication action is permitted by the actor's declared session type. Notice that other actor programming languages based on MPST use dynamic verification, and they are discussed in § 3.2.

Other Implementations based on Top-Down Multiparty Session Types. There are several implementations which use the top-down MPST framework, targeting domain-specific applications. The early works in [61,16] implement prototypes of the MPST π-calculus with symmetric sums and dynamic roles in C and Standard ML, respectively.

Apart from the MPI-C implementations [58,47] mentioned above, the MPST is not only effective to provide the specifications of concurrent and distributed message passing programming languages, but also it is useful to provide the guidance to parallelise processes onto the HPC architectures. The earliest work

	[60]	[58]	[47]	[30,31]	[40]	[65]	[55]	[6]	[39]	[35]
Language	C	MPI-C	MPI-C	Java	Java	Scala	F#	Go	PureScript	OCaml
Mainstream language	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Linearity check	✗	✗	N/A	dynamic	✗	dynamic	dynamic	dynamic	static	static
Exhaustive choices check	✓	✗	N/A	✗	✗	✓	✗	✗	✓	✓
Formalism	✗	✗	mini-MPI	types	FJ	π-cal.	✗	types	✗	π-cal.
Comm. safety	✗	✗	✓	△	✓	✓	✗	△	✗	✗•
Deadlock freedom	✗	✗	✗	△	✗	✓	✗	△	✗	✗
Liveness	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

	[51]	[76]	[24]	[71]	[13]	[41]	[11]	[20]	[9]	[3]	[4]
Language	TypeScript	F*	EnsembleS	Scala	Rust	Rust	Scala	TypeScript	Go	Scala	Java
Mainstream language	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
Linearity check	static	static	dynamic	static	static	static	dynamic	static	static	static	static
Exhaustive choices check	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Formalism	types	types	π-cal.	π-cal.	types	π-cal.	✗	CFSMs	types	π-cal.	✗
Comm. safety	△	△	✓	✓	△	✓	✗	△	△	✓	✗
Deadlock freedom	△	△	✓•	✓	△	✓	✗	△	△	✓	✗
Liveness	✗	✗	✗	✗	✓	✗	✗	✗	△	✓	✗

Table 1: MPST top-down implementations

is [74] which maps the double-buffering algorithm specified as a MPST protocol to a multicore architecture. The tool [62] uses Scribble protocols to generate the deadlock-free MPI code to run on the specialised FPGA EURECA architecture. The work [53] designs a typing system inspired by global types for specifying the communication protocols among modern Systems-on-a-Chip (SoC). The algebraic protocol programming of MPST in Haskell is used for compiling sequential functional code into the low-level parallel C code in [8]. The work [7] proposes a cost theory which can predict the cost of message passing by analysing the MPST protocols annotated by the size of data and distance, and compared the difference between the predicted cost and the real execution of the benchmarks in the literature.

The work [25] uses the multiparty session types to implement workflows for healthcare protocols. Recent works in [49,48] develop the concurrent robotics framework where specifications extended from the multiparty session types are compiled into the robotics framework, PGCD [1], which can coordinate physical robots moving around in 3D space. The tool ensures not only deadlock-freedom but also collision-freedom of the concurrent robotics systems.

Another emergent topic of MPST is a *mechanisation*: the Zooid is a domain-specific language for certified asynchronous multiparty session types, embedded in Coq, with fully mechanised metatheory for global and local types. MPGV [36] is a strictly more expressive extension of GV (Wadler's 'Good Variation') [72] to multiparty session types. All results such as type safety and global progress in [36] are mechanised in Coq. A recent work in [70] implements a mechanised proof to proposes the sound and complete inductive endpoint projection algorithm against co-inductive endpoint projection, and proves its correctness by Coq.

3.2 Dynamic Top-Down Multiparty Session Type Framework

The static top-down approaches are suitable for the programming languages with the static type checking. The first application of MPST to the real-world systems was the runtime monitor of the cyberinfrustracture of the Ocean Observatories Initiative [64]. Since their architecture is built on Python, we have developed several dynamic checking systems based on MPST for Python. In essence, the tool monitors sending and receiving messages written in the specialised session APIs (called *conversation APIs*) against the CFSMs to check the local conformance. Along this line, the first work was a development of a monitoring tool in Python with the extensions to interrupts [15].

This Python framework was extended to *the multiparty session-actor framework* in [57]. In the previous work for runtime monitoring discussed above, each end-point process is monitored by a single monitor, which checks messages to conform to its local type. In the actor model, processes (actors) are *event-driven*: upon processing a message from a mailbox, an actor can send messages and *spawn* a set of new actors; and change its behaviour upon receiving the next message. The key point of the framework in [57] that actors are independent entities that can take part in *multiple interleaved sessions*. This enables (1) actors can be involved in multiple sessions (conversations) simultaneously; (2) actors can play multiple roles (one role per each multiparty session); and (3) actors can influence another session by receiving a message from a different session. This Python framework is later extended to the timed MPST in [54].

Later the MPST actor-based framework is applied to Erlang by Folwer [18]. His toolkit handles an extended version of Scribble with *subsessions* [14], which enables to invite new participants midway of the running session. The work [56] develops the sound recovery of supervision trees in Erlang using the causal analysis of the MPST protocols, and builds runtime monitoring.

Another important thread of work in the context of active objects is an application of MPST to the actor domain specific language, *ABS* [37]. The work [22] implements a framework in ABS where local atomic segments are verified *statically*, but global interactions among local objects are monitored *dynamically* against a global type. The work investigates various performance overhead related to object communications, synchronisation between peers, and scheduling.

The implementation faithfully follows a theoretical work [38] which designs the MPST theory targeting a core ABS with futures.[1]

Recent work in [23] proposes the runtime monitoring framework called Discourje (as an extension of Courje) for monitoring more advanced MPST protocols.

3.3 Bottom-Up Behavioural Type Framework

The bottom-up approach uses a general-purpose model checking tool for verifying the properties directly against a set of CFSMs or local types. The first work which uses the bottom-up approach is [59]. This work infers the CFSMs directly from Go source code, and builds a global type so that the constructed global protocol gives the guidance for amending the unsafe code. It uses the GMC Syn tool [44] for synthesising a generalised global type from multiparty compatible CFSMs. However, the tool handles a very limited subset of Go program. The work in [42,43] uses a more general-purpose model-checking tool, mCRL2 [50], to verify properties of Go code such as safety, deadlock-freedom, liveness and termination, inferring *behavioural types* from Go source code. This tool was extended to verify shared memory concurrency in Go in [19]. In general, inferring behavioural types from source code requires non-trivial engineering efforts, and is not straightforward. The work [67] uses mCLR2 to directly verify message-passing behavioural types of a Scala-based DSL to check safety properties. This toolchain corresponds to the l.h.s. in Fig. 4.

The work in [66] extends the MPST theory to adapt the bottom-up approach and develops the verification tool for the MPST π-calculus based on mCRL2. Since this approach does not have to start from the global type, it can type more processes than the top-down approach in [29], but has several disadvantages, see § 2.2. The tool in [66] was extended to verify crash-failure semantics of the MPST π-calculus in [2].

Similarly to Rumpsteak, the Rust toolchain in [41] also includes the bottom-up approach based on the KMC-checker. The OCaml tool in [34] infers local types directly from OCaml source code using the OCaml built-in type inference system, and takes the bottom-up approach applying the KMC-checker to verify safety properties. The tools which use the KMC-checker and Rumpsteak which uses the asynchronous subtyping algorithm are only static behavioural typed programming language tools which can verify asynchronous optimised message-passing programs.

4 Conclusion

This paper gives a short survey of the programming language implementations based on multiparty session types (MPST). There are important related implementations which are not included in this paper—for examples, many works

[1] The work in [22] is categorised as "dynamic verification" as its workflow is close to the approaches by Erlang and Python discussed in this subsection.

using model checking tools of session types, and choreography programming languages [52]. The author wishes to be informed if there is any omission in this survey.

From the author's viewpoint, the most practical innovative idea is the API generation from local CFSMs introduced by [30], which has been adapted to many different mainstream languages. This method is not only engineering useful (for example, integrating with IDEs for the auto-completion), but also theoretically important to motivate the researchers to seek the links between the MPST theory and the CFSM theory [75].

One of the most important future work is a deep adaptation of MPST to active object framework. An effective integration of *futures* and *await* primitives into MPST needs to be investigated. The challenge is to examine a trade-off between low-level preemptive concurrency and fully distributed actors, using the guidance from the MPST specification.

The practical development of MPST is still an infant, and its commercialisation is far beyond the state-of-the-art. We hope that more unforeseen, inventive ideas for "session types in practice" will be emerged from researchers and developers of parallel computing, concurrent and distributed systems.

Acknowledgements We deeply thank the AOL reviewers for helpful and detailed comments, pointing out several missing literature.

References

1. Banusic, G.B., Majumdar, R., Pirron, M., Schmuck, A., Zufferey, D.: PGCD: robot programming and verification with geometry, concurrency, and dynamics. In: Liu, X., Tabuada, P., Pajic, M., Bushnell, L. (eds.) Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2019, Montreal, QC, Canada, April 16-18, 2019. pp. 57–66. ACM (2019)
2. Barwell, A., Scalas, A., Yoshida, N., Zhou, F.: Generalised Multiparty Session Types with Crash-Stop Failures. In: 33rd International Conference on Concurrency Theory. LIPIcs, vol. 243, pp. 35:1–35:25. Dagstuhl (2022)
3. Barwell, A.D., Hou, P., Yoshida, N., Zhou, F.: Designing Asynchronous Multiparty Protocols with Crash-Stop Failures. In: 37th European Conference on Object-Oriented Programming. LIPIcs, Schloss Dagstuhl–Leibniz-Zentrum f'ur Informatik (2023), to appear
4. Bouma, J., de Gouw, S., Jongmans, S.S.: Multiparty session typing in Java, deductively. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 19–27. Springer Nature Switzerland, Cham (2023)
5. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983). https://doi.org/10.1145/322374.322380
6. Castro-Perez, 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), 29:1–29:30 (Jan 2019). https://doi.org/10.1145/3290342

7. Castro-Perez, D., Yoshida, N.: CAMP: Cost-Aware Multiparty Session Protocol. In: OOPSLA 2020: Conference on Object-Oriented Programming Systems, Languages and Applications. PACMPL, vol. 4, pp. 155:1–155:30. ACM (2020)
8. Castro-Perez, D., Yoshida, N.: Compiling First-Order Functions to Session-Typed Parallel Code. In: 29th International Conference on Compiler Construction. pp. 143–154. CC 2020, ACM (2020)
9. Castro-Perez, D., Yoshida, N.: Dynamically updatable multiparty session protocols. In: 37th European Conference on Object-Oriented Programming (ECOOP 2023). Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023), to appear
10. W3C Web Services Choreography. http://www.w3.org/2002/ws/chor/
11. Cledou, G., Edixhoven, L., Jongmans, S.S., Proença, J.: API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming (ECOOP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 222, pp. 27:1–27:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.27, https://drops.dagstuhl.de/opus/volltexte/2022/16255
12. Cutner, Z., Yoshida, N.: Safe Session-Based Asynchronous Coordination in Rust. In: 23rd International Conference on Coordination Models and Languages. LNCS, vol. 12717, pp. 89–80. Springer (2021)
13. Cutner, Z., Yoshida, N., Vassor, M.: Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. vol. abs/2112.12693. ACM (2022)
14. Demangeon, R., Honda, K.: Nested Protocols in Session Types. In: 23rd International Conference on Concurrency Theory. LNCS, vol. 7454, pp. 272–286. Springer (2012)
15. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. Formal Methods in System Design **46**(3), 197–225 (2015). https://doi.org/10.1007/s10703-014-0218-8
16. Deniélou, P., Yoshida, N.: Dynamic multirole session types. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 435–446. ACM (2011). https://doi.org/10.1145/1926385.1926435
17. Estafet: Managing distributed systems using Scribble. https://www.youtube.com/watch?v=_qB2jV5SKwA (2017)
18. Fowler, S.: An Erlang Implementation of Multiparty Session Actors. Electronic Proceedings in Theoretical Computer Science **223**, 36–50 (aug 2016). https://doi.org/10.4204/eptcs.223.3, https://doi.org/10.4204%2Feptcs.223.3
19. Gabet, J., Yoshida, N.: Static Race Detection and Mutex Safety and Liveness for Go Programs. In: 34th European Conference on Object-Oriented Programming. LIPIcs, vol. 166, pp. 4:1–4:30. Schloss Dagstuhl–Leibniz-Zentrum f"ur Informatik (2020)
20. Gheri, L., Lanese, I., Sayers, N., Tuosto, E., Yoshida, N.: Design-by-Contract for Flexible Multiparty Session Protocols. In: 36th European Conference on Object-Oriented Programming. LIPIcs, vol. 222, pp. 8:1–8:28. Schloss Dagstuhl–Leibniz-Zentrum fur Informatik (2022)

21. Ghilezan, S., Pantović, J., Prokić, I., Scalas, A., Yoshida, N.: Precise Subtyping for Asynchronous Multiparty Sessions. ACM Transactions on Computational Logic (2023). https://doi.org/10.1145/3568422
22. Hähnle, R., Haubner, A.W., Kamburjan, E.: Locally Static, Globally Dynamic Session Types for Active Objects. In: de Boer, F.S., Mauro, J. (eds.) Recent Developments in the Design and Implementation of Programming Languages. OpenAccess Series in Informatics (OASIcs), vol. 86, pp. 1:1–1:24. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/OASIcs.Gabbrielli.1, https://drops.dagstuhl.de/opus/volltexte/2020/13223
23. Hamers, R., Jongmans, S.: Discourje: Runtime verification of communication protocols in clojure. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12078, pp. 266–284. Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_15
24. Harvey, P., Fowler, S., Dardha, O., J. Gay, S.: 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, p. 30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.12, https://2021.ecoop.org/details/ecoop-2021-ecoop-research-papers/12/Multiparty-Session-Types-for-Safe-Runtime-Adaptation-in-an-Actor-Language
25. Henriksen, A.S., Nielsen, L., Hildebrandt, T.T., Yoshida, N., Henglein, F.: Trustworthy Pervasive Healthcare Services via Multiparty Session Types. In: Second International Symposium on Foundations of Health Information Engineering and Systems. LNCS, vol. 7789, pp. 124–141. Springer (2012)
26. Honda, K., Mukhamedov, A., Brown, G., Chen, T.C., Yoshida, N.: Scribbling interactions with a formal foundation. In: ICDCIT. LNCS, vol. 6536, pp. 55–75. Springer (2011)
27. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: ESOP. LNCS, vol. 1381, pp. 22–138. Springer (1998). https://doi.org/10.1007/BFb0053567
28. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM Press (2008). https://doi.org/10.1145/1328438.1328472
29. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. JACM **63**, 1–67 (2016)
30. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE. LNCS, vol. 9633, pp. 401–418. Springer (2016)
31. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: FASE. LNCS, vol. 10202, pp. 116–133. Springer (2017)
32. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: ECOOP'08. LNCS, vol. 5142, pp. 516–541. Springer (2008)
33. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a Minimal Core Calculus for Java and GJ. ACM TOPLAS **23**(3), 396–450 (2001), http://doi.acm.org/10.1145/503502.503505
34. Imai, K., Lange, J., Neykova, R.: Kmclib: Automated inference and verification of session types from ocaml programs. In: Fisman, D., Rosu, G. (eds.) Tools and

Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 379–386. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_20

35. 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). Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, pp. 9:1–9:30. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.9, https://drops.dagstuhl.de/opus/volltexte/2020/13166
36. Jacobs, J., Balzer, S., Krebbers, R.: Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom. Proc. ACM Program. Lang. **6**(ICFP) (aug 2022). https://doi.org/10.1145/3547638, https://doi.org/10.1145/3547638
37. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. pp. 142–164. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
38. Kamburjan, E., Din, C.C., Chen, T.C.: Session-based compositional analysis for actor-based languages using futures. In: Ogata, K., Lawford, M., Liu, S. (eds.) Formal Methods and Software Engineering. pp. 296–312. Springer International Publishing, Cham (2016)
39. King, J., Ng, N., Yoshida, N.: Multiparty Session Type-safe Web Development with Static Linearity. In: Programming Language Approaches to Concurrency and Communication-cEntric Software. vol. 291, pp. 35–46. Open Publishing Association (2019)
40. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking Protocols with Mungo and StMungo. In: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming. p. 146–159. PPDP '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2967973.2968595, https://doi.org/10.1145/2967973.2968595
41. Lagaillardie, N., Neykova, R., Yoshida, N.: Stay Safe under Panic: Affine Rust Programming with Multiparty Session Types. In: 36th European Conference on Object-Oriented Programming. LIPIcs, vol. 222, pp. 4:1–4:29. Schloss Dagstuhl–Leibniz-Zentrum f'ur Informatik (2022)
42. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off Go: Liveness and Safety for Channel-based Programming. In: 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 748–761. ACM (2017)
43. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A Static Verification Framework for Message Passing in Go using Behavioural Types. In: 40th International Conference on Software Engineering. pp. 1137–1148. ACM (2018)
44. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: POPL. pp. 221–232 (2015). https://doi.org/10.1145/2676726.2676964
45. Lange, J., Yoshida, N.: On the Undecidability of Asynchronous Session Subtyping. In: 20th International Conference on Foundations of Software Science and Computation Structures. LNCS, vol. 10203, pp. 441–457. Springer (2017)

46. Lange, J., Yoshida, N.: Verifying Asynchronous Interactions via Communicating Session Automata. In: 31st International Conference on Computer-Aided Verification. LNCS (2019)
47. López, H.A., Marques, E.R.B., Martins, F., Ng, N., Santos, C., Vasconcelos, V.T., Yoshida, N.: Protocol-Based Verification of Message-Passing Parallel Programs. In: 2015 ACM International Conference on Object Oriented Programming Systems Languages and Applications / SPLASH '15. pp. 280–298. ACM (2015)
48. Majumdar, R., Pirron, M., Yoshida, N., Zufferey, D.: Motion session types for robotic interactions. In: Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP '19). LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2019)
49. Majumdar, R., Yoshida, N., Zufferey, D.: Multiparty Motion Coordination: From Choreographies to Robotics Programs. In: OOPSLA 2020: Conference on Object-Oriented Programming Systems, Languages and Applications. PACMPL, vol. 4, pp. 134:1–134:30. ACM (2020)
50. MCRL2 home page, https://www.mcrl2.org/web/user_manual/index.html
51. Miu, A., Ferreira, F., Yoshida, N., Zhou, F.: Communication-Safe Web Programming in TypeScript with Routed Multiparty Session Types. In: International Conference on Compiler Construction. pp. 94–106. CC (2021)
52. Montesi, F.: Introduction to Choreographies. CUP (2023)
53. de Muijnck-Hughes, J., Vanderbauwhede, W.: A Typing Discipline for Hardware Interfaces. In: Donaldson, A.F. (ed.) 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, pp. 6:1–6:27. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). https://doi.org/10.4230/LIPIcs.ECOOP.2019.6, http://drops.dagstuhl.de/opus/volltexte/2019/10798
54. Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. Formal Asp. Comput. **29**(5), 877–910 (2017)
55. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#. In: 27th International Conference on Compiler Construction. pp. 128–138. ACM (2018)
56. Neykova, R., Yoshida, N.: Let it recover: Multiparty protocol-induced recovery. In: Compiler Construction. pp. 98–108. ACM (2017)
57. Neykova, R., Yoshida, N.: Multiparty session actors. Logical Methods in Computer Science **13**(1) (2017). https://doi.org/10.23638/LMCS-13(1:17)2017
58. Ng, N., de Figueiredo Coutinho, J.G., Yoshida, N.: Protocols by default - safe MPI code generation based on session types. In: Compiler Construction. LNCS, vol. 9031, pp. 212–232. Springer (2015)
59. Ng, N., Yoshida, N.: Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In: 25th International Conference on Compiler Construction. pp. 174–184. ACM (2016)
60. Ng, N., Yoshida, N., Honda, K.: Multiparty Session C: Safe parallel programming with message optimisation. In: TOOLS. LNCS, vol. 7304, pp. 202–218. Springer (2012)
61. Nielsen, L., Yoshida, N., Honda, K.: Multiparty symmetric sum types. In: Fröschle, S.B., Valencia, F.D. (eds.) Proceedings 17th International Workshop on Expressiveness in Concurrency, EXPRESS'10, Paris, France, August 30th, 2010. EPTCS, vol. 41, pp. 121–135 (2010). https://doi.org/10.4204/EPTCS.41.9

62. Niu, X., Ng, N., Yuki, T., Wang, S., Yoshida, N., Luk, W.: EURECA Compilation: Automatic Optimisation of Cycle-Reconfigurable Circuits. In: 26th International Conference on Field Programmable Logic and Applications. pp. 1–4. IEEE (2016)
63. nuScr home page, http://nuscr.dev/nuscr/
64. Ocean Observatories Initiative home page, https://oceanobservatories.org/
65. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.24
66. Scalas, A., Yoshida, N.: Less is more: Multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30:1–30:29 (Jan 2019). https://doi.org/10.1145/3290343
67. Scalas, A., Yoshida, N., Benussi, E.: Verifying message-passing programs with dependent behavioural types. In: Programming Language Design and Implementation. pp. 502–516. ACM (2019)
68. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and its Typing System. In: PARLE'94. LNCS, vol. 817, pp. 398–413 (1994). https://doi.org/10.1007/3540581847118
69. The Rust Project Developers: Procedural Macros. https://doc.rust-lang.org/reference/procedural-macros.html
70. Tirore, D., Bengtson, J., Carbone, M.: A Sound and Complete Projection for Global Types. In: ITP 2023. LIPIcs, Schloss Dagstuhl (2023)
71. Viering, M., Hu, R., Eugster, P., Ziarek, L.: A multiparty session typing discipline for fault-tolerant event-driven distributed programming. Proceedings of the ACM on Programming Languages **5**(OOPSLA), 1–30 (Oct 2021). https://doi.org/10.1145/3485501
72. Wadler, P.: Propositions as sessions. JFP **24**(2-3), 384–418 (2014). https://doi.org/10.1017/S095679681400001X, http://dx.doi.org/10.1017/S095679681400001X
73. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The Scribble Protocol Language. In: TGC. LNCS, vol. 8358, pp. 22–41. Springer (2013)
74. Yoshida, N., Vasconcelos, V.T., Paulino, H., Honda, K.: Session-Based Compilation Framework for Multicore Programming. In: 7th International Symposium Formal Methods for Components and Objects. LNCS, vol. 5751, pp. 226–246. Springer (2008)
75. Yoshida, N., Zhou, F., Ferreira, F.: Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types. In: 23rd International Symposium on Fundamentals of Computation Theory. LNCS, vol. 12867, pp. 18–35. Springer (2021)
76. Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically Verified Refinements for Multiparty Protocols. Proc. ACM Program. Lang. **4**(OOPSLA) (Nov 2020). https://doi.org/10.1145/3428216