A Synthetic Reconstruction of Multiparty Session Types

DAVID CASTRO-PEREZ, University of Kent, UK
FRANCISCO FERREIRA, Royal Holloway, University of London, UK
SUNG-SHIK JONGMANS, University of Groningen, The Netherlands

Multiparty session types (MPST) provide a rigorous foundation for verifying the safety and liveness of concurrent systems. However, existing approaches often force a difficult trade-off: classical, projection-based techniques are compositional but limited in expressiveness, while more recent techniques achieve higher expressiveness by relying on non-compositional, whole-system model checking, which scales poorly.

This paper introduces a new approach to MPST that delivers both expressiveness and compositionality, called the synthetic approach. Our key innovation is a type system that verifies each process directly against a global protocol specification, represented as a labelled transition system (LTS) in general, with global types as a special case. This approach uniquely avoids the need for intermediate local types and projection.

We demonstrate that our approach, while conceptually simpler, supports a benchmark of challenging protocols that were previously beyond the reach of compositional techniques in the MPST literature. We generalise our type system, showing that it can validate processes against any specification that constitutes a "well-behaved" LTS, supporting protocols not expressible with the standard global type syntax. The entire framework, including all theorems and many examples, has been formalised and mechanised in Agda, and we have developed a prototype implementation as an extension to VS Code.

CCS Concepts: • **Theory of computation** → **Type theory**; **Process calculi**.

Additional Key Words and Phrases: Multiparty session typing, behavioural typing, choreographies

ACM Reference Format:
David Castro-Perez, Francisco Ferreira, and Sung-Shik Jongmans. 2018. A Synthetic Reconstruction of Multiparty Session Types. *J. ACM* 37, 4, Article 111 (August 2018), 31 pages. https://doi.org/XXXXXXX.XXXXXXX

Acknowledgments

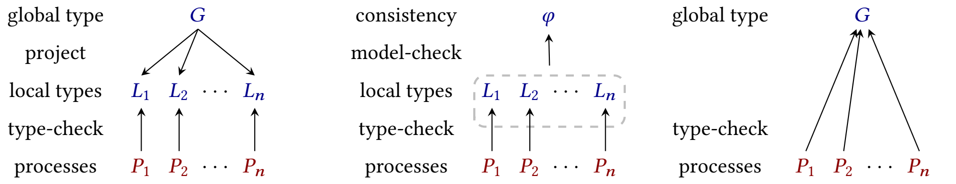
This work is partially supported by the EPSRC grants EP/Y00339X/1 and EP/T014512/1. For the purpose of open access, the author(s) has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising.

1 Introduction

Programming of concurrent systems is hard. One of the challenges is to prove—broadly construed—that implementations of *protocols* among message-passing processes are *safe* and *live* relative to specifications. Safety means that "bad" communications never happen: if a communication happens in the implementation, then it is allowed to happen by the specification. Liveness means that "good" communications eventually happen. *Multiparty session typing* (MPST) [15] is a method to automatically prove the safety and liveness of protocol implementations relative to specifications.

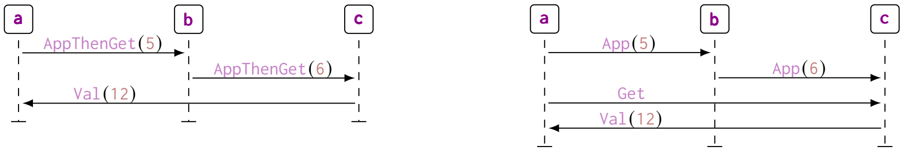
Authors' Contact Information: David Castro-Perez, University of Kent, Canterbury, UK, d.castro-perez@kent.ac.uk; Francisco Ferreira, Royal Holloway, University of London, Egham, UK, francisco.ferreiraruiz@rhul.ac.uk; Sung-Shik Jongmans, University of Groningen, Groningen, The Netherlands, s.s.t.q.jongmans@rug.nl.

ACM 1557-735X/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX



(a) Classical [15] (b) "Less Is More" [29] (c) Synthetic [this paper]

Fig. 1. MPST approaches



(a) Final number pushed by Carol (b) Final number pulled by Alice

Fig. 2. Example runs of the Ring protocol

The idea is to write specifications as *behavioural types* [1, 16] against which implementations are type-checked. Well-typedness, then, implies safety and liveness.

In this paper, we present **a new approach to MPST, called the *synthetic approach***. Inspired by the recent concept of *synthetic behavioural typing* [19], the synthetic approach to MPST has **a unique way of combining high expressiveness and compositional verification**, significantly beyond the state of the art in the MPST literature. Moreover, we show that the synthetic approach can be generalised to verify protocol implementations relative to specifications expressed as *labelled transition systems* (semantic objects) instead of as behavioural types (syntactic objects). This makes the synthetic approach very broadly applicable.

In the rest of this section, we explain in more detail the MPST method (Section 1.1), the state-of-the-art (Section 1.2), and our contributions (Section 1.3).

1.1 Multiparty Session Typing (MPST) – *Classical Approach*

To explain the MPST method, Figure 1a visualises the idea (while Figure 1b and Figure 1c are discussed in Section 1.2 and Section 2):

(1) First, a protocol among roles $\mathsf{r}_1, \ldots, \mathsf{r}_n$ is implemented as a family of **processes** $P_1, \ldots, P_n$, while it is specified as a **global type** G. The global type models the behaviour of all processes together (e.g., "a number from Alice to Bob, followed by a boolean from Bob to Carol").
(2) Next, G is decomposed into a family of **local types** $L_1, \ldots, L_n$ by **projecting** G onto each role. Each local type models the behaviour of one process alone (e.g., for Bob, "a number from Alice, followed by a boolean to Carol").
(3) Last, the family of processes is verified by **type-checking** P_i against L_i for each role. The main result is that well-typedness implies safety and liveness: if each process is statically well-typed at compile-time, then the parallel composition of the family of processes is dynamically safe and live at run-time.

The following example demonstrates the MPST method.

Example 1.1. The *Ring* protocol consists of roles *Alice*, *Bob*, and *Carol*:

- Alice sends initial number n to Bob.

- Bob receives n, applies function f (e.g., increment), and sends $f(n)$ to Carol.
- Carol receives $f(n)$, applies function g (e.g., double), and sends $g(f(n))$ to Alice.
- Alice receives final number $g(f(n))$.

There are two “modes” in which the protocol can run: either Carol *pushes* the final number to Alice immediately after applying g, or Alice *pulls* the final number from Carol sometime later. The choice between the modes is Alice’s and communicated along the ring. Figure 2 visualises example runs.

The following **global type** specifies the protocol:

$$G^{\text{Ring}} = \mathsf{a}\rightarrow\mathsf{b}{:}\begin{cases} \mathtt{AppThenGet}(\mathtt{Nat}) \,.\, \mathsf{b}\rightarrow\mathsf{c}{:}\mathtt{AppThenGet}(\mathtt{Nat}) \,.\, \mathsf{c}\rightarrow\mathsf{a}{:}\mathtt{Val}(\mathtt{Nat}) \,.\, \mathbf{end} & (\textit{push}) \\ \mathtt{App}(\mathtt{Nat}) \,.\, \mathsf{b}\rightarrow\mathsf{c}{:}\mathtt{App}(\mathtt{Nat}) \,.\, \mathsf{a}\rightarrow\mathsf{c}{:}\mathtt{Get} \,.\, \mathsf{c}\rightarrow\mathsf{a}{:}\mathtt{Val}(\mathtt{Nat}) \,.\, \mathbf{end} & (\textit{pull}) \end{cases}$$

Global type $\mathsf{p}\rightarrow\mathsf{q}{:}\{\ell_i(t_i).G_i\}_{i\in I}$ specifies the communication of a message labelled ℓ_j, with a payload of type t_j, from role p to role q, followed by G_j, for some $j \in I$.[1] We omit braces when I is a singleton, and we write “ℓ” instead of “$\ell(\mathtt{Unit})$”. The following **local types, projected from the global type,** specify Alice and Bob. Let $\ell_1 = \mathtt{AppThenGet}$ and $\ell_2 = \mathtt{App}$:

$$L_{\mathsf{a}} = \mathsf{b}{\oplus}\begin{cases} \ell_1(\mathtt{Nat}) \,.\, \mathsf{c}\&\mathtt{Val}(\mathtt{Nat}) \,.\, \mathbf{end} \\ \ell_2(\mathtt{Nat}) \,.\, \mathsf{c}{\oplus}\mathtt{Get} \,.\, \mathsf{c}\&\mathtt{Val}(\mathtt{Nat}) \,.\, \mathbf{end} \end{cases} \qquad L_{\mathsf{b}} = \mathsf{a}\&\begin{cases} \ell_1(\mathtt{Nat}) \,.\, \mathsf{c}{\oplus}\ell_1(\mathtt{Nat}) \,.\, \mathbf{end} \\ \ell_2(\mathtt{Nat}) \,.\, \mathsf{c}{\oplus}\ell_2(\mathtt{Nat}) \,.\, \mathbf{end} \end{cases}$$

Local types $\mathsf{q}{\oplus}\{\ell_i(t_i).L_i\}_{i\in I}$ and $\mathsf{p}\&\{\ell_i(t_i).L_i\}_{i\in I}$ specify the send and receive of a message labelled ℓ_j, with a payload of type t_j, from role p to role q, followed by L_j, for some $j \in I$. The following **processes, well-typed by the local types,** implement Alice and Bob in Figure 2a:

$$P^{\text{Ring}}_{\mathsf{a}} = \mathsf{b}!\ell_1(5) \,.\, \mathsf{c}?\mathtt{Val}(\mathtt{z}) \,.\, \mathbf{end} \qquad P^{\text{Ring}}_{\mathsf{b}} = \mathsf{a}?\begin{cases} \ell_1(\mathtt{x}) \,.\, \mathsf{c}!\ell_1(\mathtt{x{+}1}) \,.\, \mathbf{end} \\ \ell_2(\mathtt{x}) \,.\, \mathsf{c}!\ell_2(\mathtt{x{+}1}) \,.\, \mathbf{end} \end{cases}$$

Process $\mathsf{q}!\ell(e).P$ implements the send of a message labelled ℓ, with (the value of) expression e as the payload, to role q, followed by P. Process $\mathsf{p}?\{\ell_i(x_i).P_i\}_{i\in I}$ implements the receive of the payload of a message labelled ℓ_j, from role p, into variable x_j, followed by P_j, for some $j \in I$. Communication is *synchronous* in this paper: a send blocks the sender until the receiver is ready to perform a corresponding receive. For instance, if Alice is ready to send a $\mathtt{Get}$ message before Carol has finished her computation, then Alice needs to wait until Carol is ready to receive. □

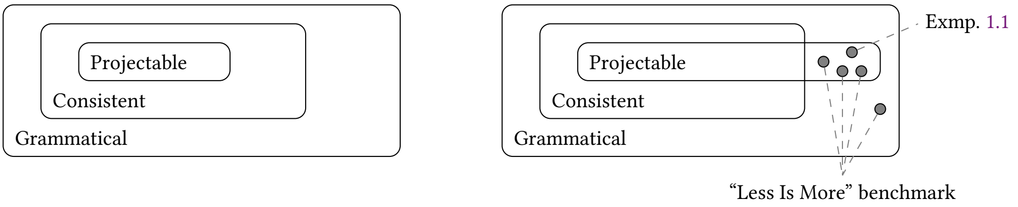
1.2 State of the Art – *“Less Is More” Approach* [29]

For well-typedness to imply safety and liveness, **a family of local types needs to be *consistent***. Intuitively, consistency means that if the local type of Alice specifies a send to Bob, then the local type of Bob should specify a corresponding receive from Alice. That is, consistency is the multiparty generalisation of binary *duality* [13, 14].

In the original paper in the MPST literature [15], **projection implies consistency**: if a family of local types is projected from a global type, then that family is consistent. Thus, well-typedness implies safety and liveness. The trouble with the original paper, though, is that only few global types can be projected. Formally, projection is a function from global type–role pairs, but its domain in the original paper is small. Figure 3a visualises the issue. The following example demonstrates that it affects the Ring protocol in Example 1.1.

Example 1.2. The projections onto Alice and Bob of G^{Ring} are defined as $L^{\text{Ring}}_{\mathsf{a}}$ and $L^{\text{Ring}}_{\mathsf{b}}$ in Example 1.1, but the projection onto Carol is undefined. Intuitively, this is because the basic “plain projection” of the original paper demands that Carol has exactly the same behaviour in each of the

[1] We adopt the same notation to represent a collection of branches—including the usage of *index set* I—as in the original MPST paper [15]. Similar notation for branching dates back at least as far as Milner’s work on CCS (e.g., [27]) and remains standard in recent work (e.g., [33]). We stipulate that there is a one-to-one correspondence between I and $\{\ell_i \mid i \in I\}$.



(a) Plain projection

(b) Full projection

Fig. 3. (Sub)sets of global types in the classical approach: “Grammatical” indicates the set of all global types; “Consistent” indicates the subset of global types for which consistent families of local types exist; “Projectable” indicates the subset of global types for which families of local types can be constructed through projection.

branches (i.e., even though Carol can actually learn which branch is taken based on the label of the message she receives, the plain projection does not leverage this additional information). As a result, in the absence of a local type for Carol, the implementation cannot be fully type-checked, so safety and liveness cannot be proved. Thus, the Ring protocol is not actually supported. □

To address this issue, instead of using the basic “plain projection” of the original paper, a more advanced “full projection” is used in many later papers in the MPST literature.[2] The key benefit of using full projection instead of plain projection is that many more global types become projectable. Formally, the domain of the function is larger. Against conventional wisdom at the time, though, **projection *no longer* implies consistency**. Thus, well-typedness no longer implies safety and liveness: whether or not it does, depends on whether or not the family of local types happens to be consistent, which needs to be proved separately. This surprising discovery was made by Scalas and Yoshida in an influential paper, colloquially called the *“Less Is More” paper* [29]. Figure 3b visualises the issue. The following example demonstrates that it, too, affects the Ring protocol of Example 1.1.

Example 1.3. The following local types, projected from G^{Ring} in Example 1.1 using full projection instead of plain projection, specify Alice, Bob, and Carol in the Ring protocol:

$$L^{\mathsf{Ring}}_{\mathsf{a}} = \ldots\ (\text{Example } 1.1) \qquad L^{\mathsf{Ring}}_{\mathsf{c}} = \mathsf{b}\,\&\begin{cases}\mathtt{AppThenGet}(\mathtt{Nat})\,.\,\mathsf{a}\oplus\mathtt{Val}(\mathtt{Nat})\,.\,\mathbf{end}\\ \mathtt{App}(\mathtt{Nat})\,.\,\mathsf{a}\&\mathtt{Get}\,.\,\mathsf{a}\oplus\mathtt{Val}(\mathtt{Nat})\,.\,\mathbf{end}\end{cases}$$
$$L^{\mathsf{Ring}}_{\mathsf{b}} = \ldots\ (\text{Example } 1.1)$$

The following processes, well-typed by the local types, implement Alice, Bob, and Carol in Figure 2a:

$$P^{\mathsf{Ring}}_{\mathsf{a}} = \ldots\ (\text{Example } 1.1) \qquad P^{\mathsf{Ring}}_{\mathsf{c}} = \mathsf{b}?\begin{cases}\mathtt{AppThenGet}(y)\,.\,\mathsf{a}!\mathtt{Val}(y{*}2)\,.\,\mathbf{end}\\ \mathtt{App}(y)\,.\,\mathbf{let}\ z{=}y{*}2\ \mathbf{in}\ \mathsf{a}?\mathtt{Get}(\_)\,.\,\mathsf{a}!\mathtt{Val}(z)\,.\,\mathbf{end}\end{cases}$$
$$P^{\mathsf{Ring}}_{\mathsf{b}} = \ldots\ (\text{Example } 1.1)$$

Now, not only the projections onto Alice and Bob are defined (cf. Example 1.2), but also the projection onto Carol. As a result, in the presence of a local type for each of Alice, Bob, and Carol, the implementation can be fully type-checked: $P^{\mathsf{Ring}}_{\mathsf{a}}$, $P^{\mathsf{Ring}}_{\mathsf{b}}$, $P^{\mathsf{Ring}}_{\mathsf{c}}$ are, in fact, well-typed by $L^{\mathsf{Ring}}_{\mathsf{a}}$, $L^{\mathsf{Ring}}_{\mathsf{b}}$, $L^{\mathsf{Ring}}_{\mathsf{c}}$. However, projection no longer implies consistency,[3] so well-typedness no longer implies safety and liveness, so safety and liveness cannot be proved. (The parallel composition of the family of processes *is* safe and live, though.) Thus, the Ring protocol is still not actually supported. □

Essentially, well-typedness is meaningless until consistency has been proved separately. Scalas and Yoshida propose a new approach to MPST based on this observation in the “Less Is More” paper,

[2] Plain projection is based on *plain merge*. Full projection is based on *full merge*. The details do not matter in this paper.

[3] Technically, the reason why the family of local types is inconsistent, is that an auxiliary partial function on local types (roughly: a second-order projection that takes the projection of a local type), which is used to compute consistency, is undefined for $L^{\mathsf{Ring}}_{\mathsf{a}}$ and $L^{\mathsf{Ring}}_{\mathsf{c}}$; this function, its effect on (in)consistency, and more examples, appear elsewhere [29].

independent of global types and projection [29]. The idea is to model consistency as a temporal logic formula φ such that the family of local types is consistent if, and only if, its *operational semantics* in the form of a *labelled transition system* (LTS) satisfies φ. Figure 1b visualises the idea:

(1) First, a protocol among roles $\mathsf{r}_1, \ldots, \mathsf{r}_n$ is implemented as a family of **processes** $P_1, \ldots, P_n$ (like the classical approach), while it is specified as a family of **local types** $L_1, \ldots, L_n$, but without a global type and projection (unlike the classical approach).
(2) Next, the family of local types is verified by **model checking** the operational semantics of $L_1, \ldots, L_n$ for satisfaction of a **consistency property** φ.
(3) Last, the family of processes is verified by **type-checking** P_i against L_i for each role. The main result is that consistency and well-typedness imply safety and liveness.

To demonstrate the effectiveness of the "Less Is More" approach, Scalas and Yoshida introduce a set of four challenging example protocols: whereas the classical approach cannot be used to prove the safety and liveness of implementations of these protocols, the "Less Is More" approach can. We call this set the *"Less Is More" benchmark*. Figure 3b visualises that three protocols in the benchmark are projectable (using full projection) but not consistent, while one is not even projectable.

1.3 This Paper: "Less Is More", Compositionally – *Synthetic Approach*

The main strength of the "Less Is More" approach is that it supports many more protocols than the classical approach does: to date, it remains the only approach in the MPST literature that passes the "Less Is More" benchmark. The main weakness, though, is that **the "Less Is More" approach is *non-compositional***: as part of the model checking step, multiple "small" LTSs $[\![L_1]\!], \ldots, [\![L_n]\!]$ (operational semantics of local types $L_1, \ldots, L_n$) need to be composed into a single "large" LTS $[\![L_1]\!] \times \cdots \times [\![L_n]\!]$ (operational semantics of the family). This is computationally expensive [32]: in the worst case, the size of the large LTS is exponential in the sizes of the small LTSs. Moreover, it goes against the compositional nature of concurrent systems programming in general.

For several years now, it has been an open problem to develop an approach that passes the "Less Is More" benchmark compositionally. **This paper presents the first solution to this open problem**: the *synthetic approach*. It leverages a recent style of behavioural type systems, called *synthetic behavioural typing*, in which the operational semantics of behavioural types is used not only to prove type soundness (as usual), but also to define the typing rules [19]. Concretely, we make the following novel contributions:

- **The special case of the synthetic approach:** We present a new MPST-specific type system to type-check processes against *global types, without local types and projection*. In this way, a key innovation of the synthetic approach is that it is, essentially, the opposite of the "Less Is More" approach (in which processes are type-checked against *local types, without global types and projection*).
  The main theoretical result is that well-typedness implies safety and liveness, *without the need to prove consistency separately*. The main practical result is that the synthetic approach is the first one to pass the "Less Is More" benchmark compositionally.
- **The general case:** We present a new generic type system—beyond MPST—to type-check processes against arbitrary *well-behaved* LTSs instead of only global types.
  The main theoretical results are that: (a) well-behavedness and well-typedness imply safety and liveness; (b) the LTSs of all global types are well-behaved. The key advantage of the general case is that it is strictly more expressive than the special case. For instance, beyond "Less Is More", protocols are supported in which a sender chooses between different receivers.

Furthermore:

- We formalised all definitions/theorems/examples, and mechanised all proofs, in Agda.

- We developed a prototype language and tooling as an extension to VS Code.

In Section 2, we present a detailed overview of our contributions. In Section 3, we recall some preliminaries from the MPST literature to set the stage for our theoretical development in later sections. Then, in Section 4, we discuss the details of typing with global types first (the special case), while in Section 5, we generalise the session classifiers from global types to LTSs (the general case). In Section 6 and Section 7, we discuss the mechanisation and implementation of our theory, respectively. We finish with a discussion of related work and a conclusion in Sections 8 and 9.

Throughout the paper, we continue to use colours to emphasise syntactic categories of objects: shades of red for data/process expressions, shades of blue for types, and shades of magenta for objects common to both expressions and types (e.g., role names, message labels). The colours are just syntax highlighting: they do not have additional meaning. Furthermore, all lemmas and theorems that have been formalised in Agda are explicitly marked with the Agda logo: .

2 Overview of the Contributions

Figure 1c visualises the idea behind the synthetic approach of this paper.

(1) First, a protocol among roles $\mathsf{r}_1, \ldots, \mathsf{r}_\mathsf{n}$ is implemented as a family of **processes** $P_1, \ldots, P_n$, while it is specified as a **global type** G.

(2) Next, the family of processes is verified by **type-checking** P_i against G for each role. The main result is that well-typedness implies safety and liveness.

Thus, the synthetic approach to MPST works **without local types and projection** (cf. the "Less Is More" approach) and **without the need to prove consistency separately** (cf. the classical approach). In fact, the synthetic approach can be further generalised to work **without global types**: processes can be type-checked directly against well-behaved *labelled transition systems* (LTS), regardless of any particular syntax to express such LTSs. Global types are just one instantiation (i.e., type-checking against global types is a special case of type-checking against well-behaved LTSs). We clarify the special case and the general case in the remaining subsections.

2.1 The Special Case: Type Checking against Global Types

Our technique to type-check processes against global types consists of two parts:

- First, we associate each global type G with *operational semantics* in the form of an LTS. Each state models a continuation of the protocol specified by G, while each transition models a possible communication.
- Second, we use LTSs to define the typing rules. For instance:

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash \mathsf{p} \triangleleft P : G' \qquad G \xrightarrow{\mathsf{p} \rightarrowtriangle \mathsf{q} : \ell(t)} G'}{\Gamma \vdash \mathsf{p} \triangleleft \mathsf{q}!\ell(e).P : G}$$

This simplified typing rule—we present the actual one later in this paper—states that, as an implementation of role p, process $\mathsf{q}!\ell(e).P$ is well-typed by global type G in environment Γ when: (1) expression e is well-typed by payload type t; (2) process P is well-typed by global type G'; **(3) G has a transition to G'**. That is, G and G' are treated as states of an LTS. We note that G' occurs in the premise of the rule, but not in the conclusion. Thus, from a bottom-up perspective, G' is a free meta-variable that needs to be *synthesised* to apply the rule. In philosophical logic, rules with free meta-variables in the premises are called "synthetic" [3, 12]; this is where the name "synthetic approach" comes from.[4]

[4] Unrelated to "synthetic approaches" in dependent type theory

Table 1. Principles of the typing rules

	Principle	**Exmp.**
Sending	P1: Each send implemented needs to be specified	2.2
	P2: Not each send specified needs to be implemented (at least one, though)	2.2
Receiving	P3: Not each receive implemented needs to be specified	2.4
	P4: Each receive specified needs to be implemented	2.3
Skipping	P5: Each communication specified needs to be skipped by each “third party”	2.4

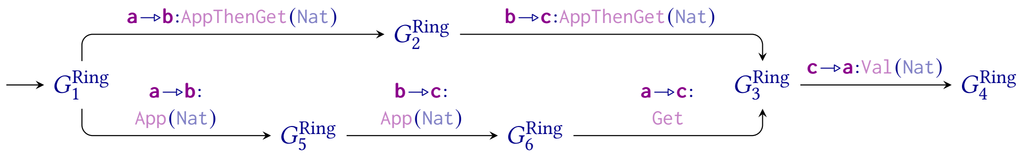


Fig. 4. Operational semantics of G^{Ring} in Example 1.1. Let $G_1^{\mathrm{Ring}} = G^{\mathrm{Ring}}$.

To further clarify our technique, we present a series of examples that revisit the Ring protocol of Example 1.1. The first example in the series demonstrates an LTS; the remaining examples demonstrate the typing rules. In each of the latter examples, we construct a typing derivation for one of the processes in the Ring protocol. Different examples highlight different principles enforced by the typing rules. Table 1 summarises the principles. The asymmetry between the principles for sending and receiving is further explained in the examples.

Example 2.1. Figure 4 visualises the operational semantics of G^{Ring} in Example 1.1 as an LTS with six states. Global action $\mathsf{p} \twoheadrightarrow \mathsf{q} : \ell(t)$ specifies the communication of a message labelled ℓ, with a payload of type t, from role p to role q. □

Example 2.2. We prove that $P_{\mathbf{a}}^{\mathrm{Ring}}$ in Example 1.1 is well-typed by the LTS of G^{Ring} in Figure 4.

First, the following derivation states that, as an implementation of Alice, the empty process is well-typed by “state” G_4^{Ring} in type environments where variable z is a number:

$$\frac{G_4^{\mathrm{Ring}} \not\to}{\varnothing, \mathsf{z} : \mathsf{Nat}; \varnothing \vdash \mathbf{a} \triangleleft \mathbf{end} : G_4^{\mathrm{Ring}}} \tag{1}$$

The intuition is that, as G_4^{Ring} does not have any transitions, **it allows Alice to terminate**. Generally, a global type allows a role to terminate when none of the reachable successor “states” of that global type—after zero-or-more transitions—has a transition with that role. For instance, G_6^{Ring} allows Bob to terminate (because neither G_3^{Ring} nor G_4^{Ring} has a transition with Bob), but G_2^{Ring} does not allow Alice to terminate (because G_3^{Ring} has a transition with Alice).

Next, the following derivation states that, as an implementation of Alice, process $\mathbf{c}?\mathsf{Val}(\mathsf{z}).\mathbf{end}$ is well-typed by “state” G_3^{Ring} in empty type environments:

$$\frac{\left\{\ G_3^{\mathrm{Ring}} \xrightarrow{\mathbf{c} \twoheadrightarrow \mathbf{a} : \mathsf{Val}(\mathsf{Nat})} G_4^{\mathrm{Ring}} \quad \mapsto \quad \varnothing, \mathsf{z} : \mathsf{Nat}; \varnothing \vdash \mathbf{a} \triangleleft \mathbf{end} : G_4^{\mathrm{Ring}}\ (1)\ \right\}}{\varnothing; \varnothing \vdash \mathbf{a} \triangleleft \mathbf{c}?\mathsf{Val}(\mathsf{z}).\mathbf{end} : G_3^{\mathrm{Ring}}} \tag{2}$$

The intuition is that, as G_3^{Ring} has a transition that models a communication from Carol to Alice of a message labelled Val, with a payload of type Nat, **it allows Alice to perform such a receive**. As the payload is received into variable $\mathtt{z}$, the successor process must be well-typed in type environments that map $\mathtt{z}$ to Nat; this was proved by Equation (1).

Next, the following derivation states that, as an implementation of Alice, process $\mathbf{c}?\mathtt{Val}(\mathtt{z}).\mathbf{end}$—the same as in the previous derivation—is well-typed by "state" G_2^{Ring} in empty type environments:

$$\frac{G_2^{\mathsf{Ring}} \overset{\{\mathbf{a}\}}{\not\to} \qquad \left\{ G_2^{\mathsf{Ring}} \xrightarrow{\mathbf{b} \rightarrowtriangle \mathbf{c}:\mathtt{AppThenGet}(\mathsf{Nat})} G_3^{\mathsf{Ring}} \quad \mapsto \quad \varnothing;\varnothing \vdash \mathbf{a} \triangleleft \mathbf{c}?\mathtt{Val}(\mathtt{z}).\mathbf{end} : G_3^{\mathsf{Ring}} \ (2) \right\}}{\varnothing;\varnothing \vdash \mathbf{a} \triangleleft \mathbf{c}?\mathtt{Val}(\mathtt{z}).\mathbf{end} : G_2^{\mathsf{Ring}}} \tag{3}$$

The intuition is that, as G_2^{Ring} has one transition, but none with Alice, **it allows Alice to skip the communication modelled by that transition**. Skipping communications in this way subsumes the concept of *merging*—a key ingredient of projection—in the classical approach to MPST [29].

Last, the following derivation states that, as an implementation of Alice, process $\mathbf{b}!\mathtt{AppThenGet}(5).\mathbf{c}?\mathtt{Val}(\mathtt{z}).\mathbf{end}$ is well-typed by "state" G_1^{Ring} in empty type environments (henceforth omitted):

$$\frac{\vdash 5 : \mathsf{Nat} \qquad \vdash \mathbf{a} \triangleleft \mathbf{c}?\mathtt{Val}(\mathtt{z}).\mathbf{end} : G_2^{\mathsf{Ring}} \ (3) \qquad G_1^{\mathsf{Ring}} \xrightarrow{\mathbf{a} \rightarrowtriangle \mathbf{b}:\mathtt{AppThenGet}(\mathsf{Nat})} G_2^{\mathsf{Ring}}}{\vdash \mathbf{a} \triangleleft \mathbf{b}!\mathtt{AppThenGet}(5).\mathbf{c}?\mathtt{Val}(\mathtt{z}).\mathbf{end} : G_1^{\mathsf{Ring}}} \tag{4}$$

The intuition is that, as G_1^{Ring} has a transition that models a communication from Alice to Bob of a message labelled $\mathtt{AppThenGet}$, with a payload of type Nat, **it allows Alice to perform such a send.**

We note that G_1^{Ring} has two transitions, but the well-typed process has only one corresponding output alternative. This demonstrates principles P1/P2 that **each send implemented needs to be specified**, but ***not*** **each send specified needs to be implemented** (at least one, though).

Given the definitions of $P_{\mathbf{a}}^{\mathsf{Ring}}$ in Example 1.1 and G^{Ring} in Figure 4, we conclude from Equation (4):

$$\vdash P_{\mathbf{a}}^{\mathsf{Ring}} : G^{\mathsf{Ring}}$$ □

Example 2.3. We prove that $P_{\mathbf{b}}^{\mathsf{Ring}}$ in Example 1.1 is well-typed by the LTS of G^{Ring} in Figure 4.

First, using similar derivations as in Example 2.2, we can prove:

$$\varnothing, \mathtt{x} : \mathsf{Nat}; \varnothing \vdash \mathbf{b} \triangleleft \mathbf{c}!\mathtt{AppThenGet}(\mathtt{x{+}1}).\mathbf{end} : G_2^{\mathsf{Ring}} \tag{5}$$

$$\varnothing, \mathtt{x} : \mathsf{Nat}; \varnothing \vdash \mathbf{b} \triangleleft \mathbf{c}!\mathtt{App}(\mathtt{x{+}1}).\mathbf{end} : G_5^{\mathsf{Ring}} \tag{6}$$

Next, the following derivation states that, as an implementation of Bob, his input process $P_{\mathbf{b}}^{\mathsf{Ring}}$ in Example 1.1 is well-typed by "state" G_1^{Ring}. Let $\ell_1 = \mathtt{AppThenGet}$ and $\ell_2 = \mathtt{App}$:

$$\frac{\left\{ \begin{array}{lcl} G_1^{\mathsf{Ring}} \xrightarrow{\mathbf{a} \rightarrowtriangle \mathbf{b}:\mathtt{AppThenGet}(\mathsf{Nat})} G_2^{\mathsf{Ring}} & \mapsto & \varnothing, \mathtt{x} : \mathsf{Nat}; \varnothing \vdash \mathbf{b} \triangleleft \mathbf{c}!\mathtt{AppThenGet}(\mathtt{x{+}1}).\mathbf{end} : G_2^{\mathsf{Ring}} \ (5) \\ G_1^{\mathsf{Ring}} \xrightarrow{\mathbf{a} \rightarrowtriangle \mathbf{b}:\mathtt{App}(\mathsf{Nat})} G_5^{\mathsf{Ring}} & \mapsto & \varnothing, \mathtt{x} : \mathsf{Nat}; \varnothing \vdash \mathbf{b} \triangleleft \mathbf{c}!\mathtt{App}(\mathtt{x{+}1}).\mathbf{end} : G_5^{\mathsf{Ring}} \ (6) \end{array} \right\}}{\varnothing;\varnothing \vdash \mathbf{b} \triangleleft \mathbf{a}?\{\ell_1(\mathtt{x}).\mathbf{c}!\ell_1(\mathtt{x{+}1}).\mathbf{end}, \ell_2(\mathtt{x}).\mathbf{c}!\ell_2(\mathtt{x{+}1}).\mathbf{end}\} : G_1^{\mathsf{Ring}}} \tag{7}$$

The intuition is that, as G_1^{Ring} has transitions that model communications from Alice to Bob of a message labelled $\mathtt{AppThenGet}$ or $\mathtt{App}$, with a payload of type Nat, **it allows Bob to perform such receives**. As the payload is received into variable $\mathtt{x}$, the successor process must be well-typed in type environments that map $\mathtt{x}$ to Nat; this was proved by Equations (5) and (6).

We note that G_1^{Ring} has two transitions, and the well-typed process has two corresponding input alternatives. This demonstrates principle P4 that **each receive specified needs to be implemented**. Thus, there is asymmetry between typing input processes (e.g., Bob in this example) and typing output processes (e.g., Alice in Example 2.2): a sender must be able to offer at least one message label specified in the LTS, while the receiver must be able to accept all of them.

Given the definitions of $P_{\mathbf{b}}^{\mathsf{Ring}}$ in Example 1.1 and G^{Ring} in Figure 4, we conclude from Equation (7):

$$\vdash P_{\mathbf{b}}^{\mathsf{Ring}} : G^{\mathsf{Ring}}$$ □

Example 2.4. We prove that $P_{\mathsf{c}}^{\text{Ring}}$ in Example 1.3 is well-typed by the LTS of G^{Ring} in Figure 4. First, using a similar derivation as in Example 2.2, we can prove:

$$\varnothing, \mathtt{y} : \mathtt{Nat}; \varnothing \vdash \mathsf{c} \triangleleft \mathsf{a}!\mathtt{Val}(\mathtt{y*2}).\mathbf{end} : G_3^{\text{Ring}} \tag{8}$$

$$\varnothing, \mathtt{y} : \mathtt{Nat}; \varnothing \vdash \mathsf{c} \triangleleft \mathbf{let}\ \mathtt{z=y*2}\ \mathbf{in}\ \mathsf{a}?\mathtt{Get}(\_).\mathsf{a}!\mathtt{Val}(\mathtt{z}).\mathbf{end} : G_6^{\text{Ring}} \tag{9}$$

Next, the following derivations state that, as an implementation of Carol, her input process $P_{\mathsf{c}}^{\text{Ring}}$ in Example 1.3 is well-typed by both "state" G_2^{Ring} and "state" G_5^{Ring}. Let $\ell_1 = \mathtt{AppThenGet}$ and $\ell_2 = \mathtt{App}$. Also, let $P_1 = \mathsf{a}!\mathtt{Val}(\mathtt{y*2}).\mathbf{end}$ and $P_2 = \mathbf{let}\ \mathtt{z=y*2}\ \mathbf{in}\ \mathsf{a}?\mathtt{Get}(\_).\mathsf{a}!\mathtt{Val}(\mathtt{z}).\mathbf{end}$:

$$\frac{\left\{\ G_2^{\text{Ring}} \xrightarrow{\mathsf{b}\twoheadrightarrow\mathsf{c}:\ell_1(\mathtt{Nat})} G_3^{\text{Ring}} \quad\mapsto\quad \varnothing, \mathtt{y} : \mathtt{Nat}; \varnothing \vdash \mathsf{c} \triangleleft P_1 : G_3^{\text{Ring}}\ (8)\ \right\}}{\vdash \mathsf{c} \triangleleft \mathsf{b}?\{\ell_1(\mathtt{y}).P_1, \ell_2(\mathtt{y}).P_2\} : G_2^{\text{Ring}}} \tag{10}$$

$$\frac{\left\{\ G_5^{\text{Ring}} \xrightarrow{\mathsf{b}\twoheadrightarrow\mathsf{c}:\ell_2(\mathtt{Nat})} G_6^{\text{Ring}} \quad\mapsto\quad \varnothing, \mathtt{y} : \mathtt{Nat}; \varnothing \vdash \mathsf{c} \triangleleft P_2 : G_6^{\text{Ring}}\ (9)\ \right\}}{\vdash \mathsf{c} \triangleleft \mathsf{b}?\{\ell_1(\mathtt{y}).P_1, \ell_2(\mathtt{y}).P_2\} : G_5^{\text{Ring}}} \tag{11}$$

The intuition is that, as G_2^{Ring} (resp. G_5^{Ring}) has a transition that models a communication from Bob to Carol of a message labelled AppThenGet (resp. App), **it allows Carol to perform such a receive.**

We note that the well-typed process has two input alternatives, but G_2^{Ring} (resp. G_5^{Ring}) has only one corresponding transition. This demonstrates principle P3 that ***not* each receive implemented needs to be specified**: a receiver may be able to accept more message labels than just those specified in the LTS. Reminiscent of *subtyping* in the MPST literature [10], this is fine because the sender—assuming it is well-typed—is guaranteed to offer only message labels specified in the LTS.

Last, the following derivation states that, as an implementation of Carol, her input process $P_{\mathsf{c}}^{\text{Ring}}$ in Example 1.3 is well-typed by "state" G_1^{Ring}:

$$\frac{G_1^{\text{Ring}} \overset{\{\mathsf{c}\}}{\not\longrightarrow} \qquad \left\{\begin{array}{lll} G_1^{\text{Ring}} \xrightarrow{\mathsf{a}\twoheadrightarrow\mathsf{b}:\ell_1(\mathtt{Nat})} G_2^{\text{Ring}} & \mapsto & \vdash \mathsf{c} \triangleleft \mathsf{b}?\{\ell_1(\mathtt{y}).P_1, \ell_2(\mathtt{y}).P_2\} : G_2^{\text{Ring}}\ (10) \\ G_1^{\text{Ring}} \xrightarrow{\mathsf{a}\twoheadrightarrow\mathsf{b}:\ell_2(\mathtt{Nat})} G_5^{\text{Ring}} & \mapsto & \vdash \mathsf{c} \triangleleft \mathsf{b}?\{\ell_1(\mathtt{y}).P_1, \ell_2(\mathtt{y}).P_2\} : G_5^{\text{Ring}}\ (11) \end{array}\right\}}{\vdash \mathsf{c} \triangleleft \mathsf{b}?\{\ell_1(\mathtt{y}).P_1, \ell_2(\mathtt{y}).P_2\} : G_1^{\text{Ring}}} \tag{12}$$

The intuition is that, as G_1^{Ring} has two transitions, but none with Carol, **it allows Carol to skip the communications modelled by those transitions**.

We note that G_1^{Ring} has two successor "states", and the process is well-typed by each of them. This demonstrates principle P5 that **each communication specified needs to be skipped by each "third party"** that does not participate in that communication: regardless of which communications happen between whichever senders and receivers, third parties that do not participate in those communications must be able to behave as specified in any continuation.

Given the definitions of $P_{\mathsf{c}}^{\text{Ring}}$ in Example 1.3 and G^{Ring} in Figure 4, we conclude from Equation (12):

$$\vdash P_{\mathsf{c}}^{\text{Ring}} : G^{\text{Ring}}$$ □

The main theoretical result for the special case of our synthetic approach to MPST is that **well-typedness implies safety and liveness**.

Example 2.5. Examples 2.2 to 2.4 demonstrated that $P_{\mathsf{a}}^{\text{Ring}}, P_{\mathsf{b}}^{\text{Ring}}, P_{\mathsf{c}}^{\text{Ring}}$ are well-typed by G^{Ring}. Thus, by Theorems 4.3 and 4.4, we conclude that the parallel composition of this family of processes—the *session*—is safe and live. Safety means that each communication that happens in the session is allowed by the global type. Liveness means that after any number of communications, either the session has successfully terminated, or another communication can happen. It is the first time in the MPST literature that this is proved compositionally for Example 1.1. □

The main practical result is that the synthetic approach is **the first one to pass the "Less Is More" benchmark compositionally**, as we demonstrate fully in Section 4.4.

2.2 The General Case: Type Checking against LTSs

A key observation of the previous examples is that **the syntax of global types does *not* matter** at all in the typing rules; only **the operational semantics does**. That is, the syntactic structure of global types is never inspected in the typing rules. Instead, global types are treated as opaque states of an LTS, whose transitions are the only objects of significance. This observation is pushed forward in the general case of our synthetic approach to MPST.

The idea of the general case is to define a predicate to judge whether or not an LTS is *well-behaved*. Processes can subsequently be type-checked against well-behaved LTSs, regardless of how those LTSs are generated, and independent of the syntactic structure of states—if any. Intuitively, an LTS is well-behaved when it fulfils the following requirements:

- **Sender determinacy:** If a state has multiple transitions, then these transitions model communications either with different senders and different receivers, or with the same sender but different receivers, or with the same sender and the same receiver—but never with different senders and the same receiver.
- **Determinism:** If a source state has multiple transitions that model the same communication, then they have the same target state.
- **Conditional commutativity and confluence (diamond):** Subject to additional conditions (see Definition 5.3 for details), if a state has multiple transitions that model independent communications, then those communications commute (i.e., the transitions form a diamond).

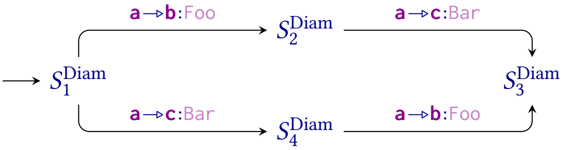
The following example demonstrates these requirements.

Example 2.6. We argue that the LTS in Figure 4 is well-behaved. State G_1^{Ring} is the only state that has multiple transitions. These transitions have the same sender and the same receiver, so sender determinacy is fulfilled. Moreover, these transitions do not model the same communication (i.e., the message labels are different), nor are they independent, so determinism, conditional commutativity, and confluence are fulfilled, too. The remaining states have only a single transition, so they trivially fulfil the well-behavedness requirements. □

The main theoretical results for the general case of our synthetic approach are that: **(a) well-behavedness and well-typedness imply safety and liveness; (b) the LTSs of all global types are well-behaved.** Thus, when processes are type-checked against LTSs of global types, well-behavedness of those LTSs does not need to be proved separately; it is already implied. This makes type checking against global types really a special case of type checking against LTSs.

The key advantage is that **the general case is strictly more expressive than the special case**: more families of processes can be successfully type-checked by well-behaved LTSs than by global types. In particular, there exist well-behaved LTSs that cannot be expressed as a global type, but they are *inhabited* in our type system. An LTS is inhabited when there exists a family of processes each of which is well-typed by that LTS. The following example demonstrates a protocol that is not supported in the "Less Is More" paper, nor is it supported by the special case in this paper, but it is supported by the general case.

Example 2.7. The following well-behaved LTS specifies a protocol in which a `Foo` message is communicated from Alice to Bob, and a `Bar` message from Alice to Carol, in any order:


S_1^{Diam}
S_2^{Diam}
S_3^{Diam}
S_4^{Diam}
$\mathsf{a}{\rightarrow}\!\!\triangleright\mathsf{b}{:}\mathsf{Foo}$
$\mathsf{a}{\rightarrow}\!\!\triangleright\mathsf{c}{:}\mathsf{Bar}$
$\mathsf{a}{\rightarrow}\!\!\triangleright\mathsf{c}{:}\mathsf{Bar}$
$\mathsf{a}{\rightarrow}\!\!\triangleright\mathsf{b}{:}\mathsf{Foo}$

The following processes, well-typed by the LTS, implement Alice, Bob, and Carol:

$$P_{\mathsf{a}}^{\text{Diam}} = \mathsf{b}!\mathsf{Foo}\,.\,\mathsf{c}!\mathsf{Bar}\,.\,\mathbf{end} \qquad P_{\mathsf{b}}^{\text{Diam}} = \mathsf{a}?\mathsf{Foo}(\_)\,.\,\mathbf{end} \qquad P_{\mathsf{c}}^{\text{Diam}} = \mathsf{a}?\mathsf{Bar}(\_)\,.\,\mathbf{end}$$ □

2.3 Formalisation and Mechanisation in Agda

We formalised all the theorems presented in this paper, as well as many examples, using the Agda proof assistant. The formalisation follows the definitions given here directly, without requiring any modifications or simplifications to facilitate the proofs. This reinforces the claim that the synthetic approach to multiparty session types, as introduced above, is particularly amenable to formalisation and mechanisation – a point we substantiate further in Section 6.

2.4 Prototype Language and Tooling in VS Code

We developed a prototype language and tooling as an extension of *VS Code*, including a dedicated *LSP server*. The language consists of textual versions of global types and processes, while the tooling consists of a parser, a syntax highlighter, and a type checker—all running in the LSP server—that leverage the synthetic approach to MPST of this paper. The prototype shows that there exists an algorithm to apply our typing rules in practice, opening up the door towards integration of our type system in mainstream languages. The following example demonstrates the prototype.

Example 2.8. Figure 5 shows three VS Code screenshots of the Ring specification and implementation, as a global type (lines 1-11) and as processes (lines 13-26), in our prototype language; they correspond with G^{Ring} and $P_{\mathsf{a}}^{\text{Ring}}, P_{\mathsf{b}}^{\text{Ring}}, P_{\mathsf{c}}^{\text{Ring}}$ in Examples 1.1 and 1.3. The processes in Figure 5a are well-typed, while the process for Alice in Figures 5b and 5c is ill-typed. The error messages give the programmer actionable feedback about how/why the protocol is violated. □

3 Preliminaries

We first recall the existing syntax and operational semantics of global types and processes. The definitions in this section are standard in the MPST literature (e.g., [10, 29]). For instance, as commonly done, we stipulate that each protocol implementation consists of a fixed set of processes (one for every role) and channels (two between every pair of roles; one in every direction).

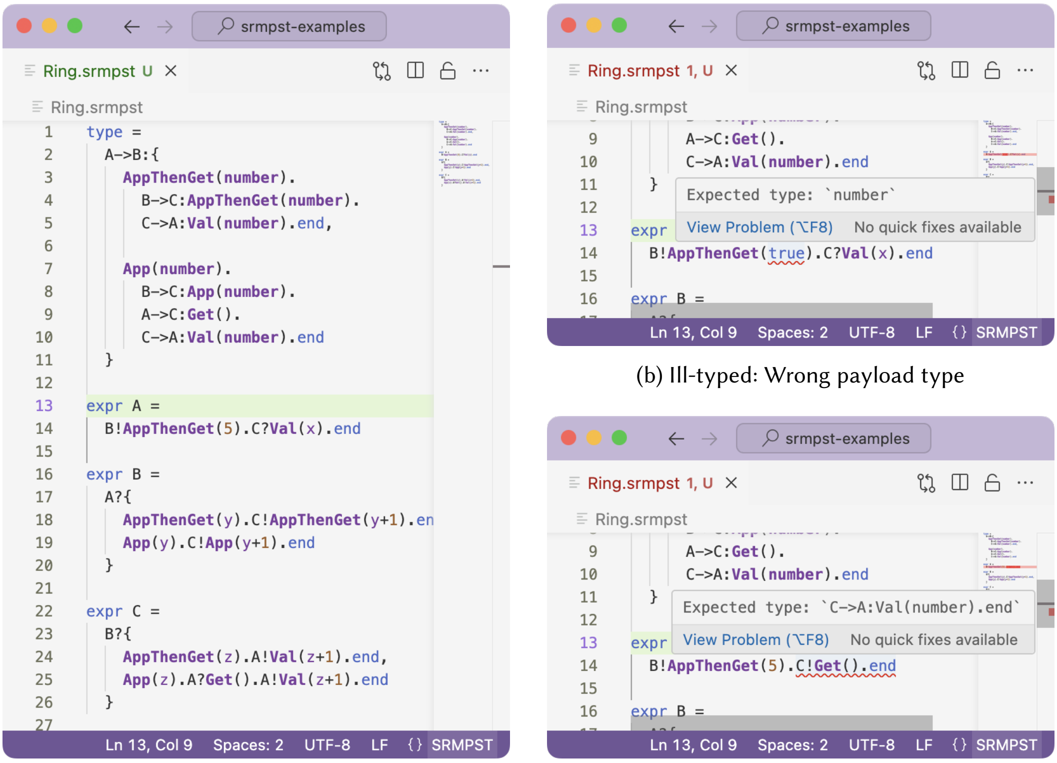
3.1 Global Types

Syntax. Regarding the syntax of global types:

- Let $\mathbb{R} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots\}$ denote the set of *roles*, ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}$.
- Let $\mathcal{L} = \{\mathsf{App}, \mathsf{Get}, \mathsf{Val}, \ldots\}$ denote the set of *message labels*, ranged over by ℓ.
- Let $\mathbb{T} = \{\mathsf{Unit}, \mathsf{Bool}, \ldots\}$ denote the set of *payload types*, ranged over by t.
- Let $\{\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots\}$ denote the set of *recursion variables*, ranged over by X, Y, Z.
- Let $\mathbb{G}$ denote the set of *global types*, ranged over by G. It is induced by the following grammar:

$$G \;::=\; \mathsf{p}{\rightarrow}\!\!\triangleright\mathsf{q}{:}\{\ell_i(t_i).G_i\}_{i \in I} \;\Big|\; \mu X.G \;\Big|\; X \;\Big|\; \mathbf{end} \;\Big|\; G_1 \parallel G_2$$

Global type $\mathsf{p}{\rightarrow}\!\!\triangleright\mathsf{q}{:}\{\ell_i(t_i).G_i\}_{i \in I}$ specifies the communication of a message labelled ℓ_j, with a payload of type t_j, from role p to role q, followed by G_j, for some $j \in I$. Each G_i is called "a branch". Global types $\mu X.G$ and X specify recursion. As usual, recursion must be guarded in the sense of Yoshida and Gheri [35]. Global type $\mathbf{end}$ specifies the empty protocol. Global


srmpst-examples
Ring.srmpst U
Ring.srmpst
Ln 13, Col 9 Spaces: 2 UTF-8 LF {} SRMPST

```
type =
  A->B:{
    AppThenGet(number).
      B->C:AppThenGet(number).
      C->A:Val(number).end,

    App(number).
      B->C:App(number).
      A->C:Get().
      C->A:Val(number).end
  }

expr A =
  B!AppThenGet(5).C?Val(x).end

expr B =
  A?{
    AppThenGet(y).C!AppThenGet(y+1).en
    App(y).C!App(y+1).end
  }

expr C =
  B?{
    AppThenGet(z).A!Val(z+1).end,
    App(z).A?Get().A!Val(z+1).end
  }
```

(a) Well-typed

srmpst-examples
Ring.srmpst 1, U
Ring.srmpst
Ln 13, Col 9 Spaces: 2 UTF-8 LF {} SRMPST

```
      A->C:Get().
      C->A:Val(number).end
  }

expr
  B!AppThenGet(true).C?Val(x).end

expr B =
```

```
Expected type: `number`
View Problem (⌥F8)   No quick fixes available
```

(b) Ill-typed: Wrong payload type

```
      A->C:Get().
      C->A:Val(number).end
  }

expr
  B!AppThenGet(5).C!Get().end

expr B =
```

```
Expected type: `C->A:Val(number).end`
View Problem (⌥F8)   No quick fixes available
```

(c) Ill-typed: Wrong action

Fig. 5. Screenshots of the prototype language and tooling in VS Code

$$\frac{j \in I}{\mathsf{p} \rightarrow \mathsf{q}:\{\ell_i(t_i).G_i\}_{i\in I} \xrightarrow{\mathsf{p}\rightarrow\mathsf{q}:\ell_j(t_j)} G_j}\ [\to\text{G-COM1}]$$

$$\frac{\{\mathsf{p},\mathsf{q}\} \cap \{\mathsf{r},\mathsf{s}\} = \varnothing \qquad G_i \xrightarrow{\mathsf{r}\rightarrow\mathsf{s}:\ell(t)} G_i' \text{ for each } i \in I}{\mathsf{p} \rightarrow \mathsf{q}:\{\ell_i(t_i).G_i\}_{i\in I} \xrightarrow{\mathsf{r}\rightarrow\mathsf{s}:\ell(t)} \mathsf{p} \rightarrow \mathsf{q}:\{\ell_i(t_i).G_i'\}_{i\in I}}\ [\to\text{G-COM2}]$$

$$\frac{G[X := \mu X.G] \xrightarrow{\alpha} G'}{\mu X.G \xrightarrow{\alpha} G'}\ [\to\text{G-REC}] \qquad \frac{G_1 \xrightarrow{\alpha} G_1'}{G_1 \parallel G_2 \xrightarrow{\alpha} G_1' \parallel G_2}\ [\to\text{G-PAR1}] \qquad \frac{G_2 \xrightarrow{\alpha} G_2'}{G_1 \parallel G_2 \xrightarrow{\alpha} G_1 \parallel G_2'}\ [\to\text{G-PAR2}]$$

Fig. 6. Transition rules for global types

type $G_1 \parallel G_2$ specifies the interleaving of G_1 and G_2. As in the original MPST paper [15], we stipulate that the roles in G_1 and G_2 are disjoint (straightforward to syntactically check).

- Let $\mathbb{A} = \{\mathsf{p}\rightarrow\mathsf{q}:\ell(t) \mid \mathsf{p},\mathsf{q} \in \mathbb{R} \text{ and } \ell \in \mathcal{L} \text{ and } t \in \mathbb{T}\}$ denote the set of *global actions*, ranged over by α. Global action $\mathsf{p}\rightarrow\mathsf{q}:\ell(t)$ specifies the communication of a message labelled ℓ, with a payload of type t, from role p to role q.

$$\frac{R \subseteq \{\mathsf{p}, \mathsf{q}\} \qquad G \xrightarrow{\mathsf{p} \to \mathsf{q} : \ell(t)} G'}{G \xrightarrow{R} G'} \qquad \frac{R \cap \{\mathsf{p}, \mathsf{q}\} = \varnothing \qquad G \xrightarrow{\mathsf{p} \to \mathsf{q} : \ell(t)} G'}{G \xrightarrow{\overline{R}} G'} \qquad \frac{G \overset{R}{\not\to} \qquad G \xrightarrow{\overline{R}} G'}{G \overset{\overline{R}}{\Longrightarrow} G'}$$

Fig. 7. Derived transition rules for global types

Operational semantics. Regarding the operational semantics of global types, let $G \xrightarrow{\alpha} G'$ denote the transition from global type G to global type G' through global action α. It is the smallest relation induced by the rules in Figure 6:

- Rule [→G-Com1] states that a communication global type can make a transition to any one of its branches through the corresponding global action.
- Rule [→G-Com2] states that a communication global type can also make a transition when: each of its branches can make a transition through the same global action (second premise); this "lexically next" global action is *independent* of the "lexically first" global action (first premise). Thus, independent global actions may happen out-of-order. This feature [15] is needed to ensure that global types are not unnecessarily restrictive. Without allowing out-of-order execution of independent global actions, for instance, there would exist no global type for the following processes (trailing **end** omitted to save space):

  $$P^{\mathrm{Com2}}_{\mathbf{a}} = \mathbf{b1}!\mathsf{Foo}\,.\,\mathbf{b2}!\mathsf{Foo} \qquad P^{\mathrm{Com2}}_{\mathbf{b1},\mathbf{b2}} = \mathbf{a}?\mathsf{Foo}(\_)\,.\,\mathbf{c}!\mathsf{Bar} \qquad P^{\mathrm{Com2}}_{\mathbf{c}} = \mathbf{b1}?\mathsf{Bar}(\_)\,.\,\mathbf{b2}?\mathsf{Bar}(\_)$$

  Using rule [→G-Com2], though, the following global type precisely specifies the protocol:

  $$G^{\mathrm{Com2}} = \mathbf{a} \to \mathbf{b1} : \mathsf{Foo}\,.\,\mathbf{a} \to \mathbf{b2} : \mathsf{Foo}\,.\,\mathbf{b1} \to \mathbf{c} : \mathsf{Bar}\,.\,\mathbf{b2} \to \mathbf{c} : \mathsf{Bar}\,.\,\mathbf{end}$$

  Crucially, the two middle communications can happen out-of-order.
- Rule [→G-Rec] states that a recursive global type can make a transition when its unfolding can. In this rule, $G[X := \mu X.G]$ denotes the substitution of X by $\mu X.G$ in G.
- Rules [→G-Par1] and [→G-Par2] state that an interleaving global type can make a transition when one of its operands can.

Furthermore, let $G \xrightarrow{R} G'$ (resp. $G \xrightarrow{\overline{R}} G'$) denote the existence of a transition from G to G' in which each (resp. none) of the roles in R participate. Let $G \overset{\overline{R}}{\Longrightarrow} G'$ denote that: (1) none of the roles in R participate in none of the transitions of G; (2) G has a transition to G'. They are the smallest relations induced by the rules in Figure 7.

Given a fixed set of roles, for each derived transition relation $\dashrightarrow\ \in \bigcup\{\{\xrightarrow{R}, \xrightarrow{\overline{R}}, \overset{R}{\Longrightarrow}\} \mid R \subseteq \mathbb{R}\}$, we write "$G \dashrightarrow$" instead of "$G \dashrightarrow G'$ for some G'", we write "$G \not\dashrightarrow$" instead of "$G \not\dashrightarrow G'$ for each G'", and we write $\dashrightarrow^*$ for the reflexive transitive closure.

If $G \xrightarrow{\{r\}}$, then r is *enabled*. If $G \overset{\{r\}}{\not\longrightarrow}$, then r is *disabled*. If there exist global actions $\alpha_1, \ldots, \alpha_n$ such that $G \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \xrightarrow{\{r\}}$, then r is *active* in G; otherwise, r is *inactive*. Let $\mathsf{r} \in G$ and $\mathsf{r} \notin G$ denote the activeness and inactiveness of r in G.

3.2 Processes

Syntax. Regarding the syntax of processes:

- Let $\mathbb{X}$ denote the set of *variables*, ranged over by x.
- Let $\mathbb{V} = \{\mathtt{unit}, \mathtt{false}, \mathtt{true}, \mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots\}$ denote the set of *values*, ranged over by v.
- Let $\mathbb{E} = \mathbb{X} \cup \mathbb{V} \cup \{\mathtt{2==3}, \mathtt{x+1}, \ldots\}$ denote the set of *expressions*, ranged over by e.
- Let $\mathbb{P}$ denote the set of *processes*, ranged over by P. It is induced by the following grammar:

$$P \ ::= \ \mathsf{q}!\ell(e).P \ \big|\ \mathsf{p}?\{\ell_i(x_i{:}t_i).P_i\}_{i\in I} \ \big|\ \mathbf{let}\ x{=}e\ \mathbf{in}\ P \ \big|\ \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \ \big|\ \mathbf{rec}\ X.P \ \big|\ X \ \big|\ \mathbf{end}$$

$$\frac{}{v \Downarrow v} \qquad \frac{e_1 \Downarrow v \qquad e_2 \Downarrow v}{e_1\texttt{==}e_2 \Downarrow \texttt{true}} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \neq v_2}{e_1\texttt{==}e_2 \Downarrow \texttt{false}} \qquad \cdots$$

Fig. 8. Evaluation rules for expressions (excerpt)

$$\frac{e \Downarrow v \qquad j \in I}{\mathsf{p} \triangleleft \mathsf{q}!\ell_j(e).P \mid \mathsf{q} \triangleleft \mathsf{p}?\{\ell_i(x_i{:}t_i).P_i\}_{i\in I} \xrightarrow{\mathsf{p}\twoheadrightarrow\mathsf{q}:\ell_j(t_j)} \mathsf{p} \triangleleft P \mid \mathsf{q} \triangleleft P_j[x_j := v]}\,[\to\text{P-Com}]$$

$$\frac{e \Downarrow v}{\mathsf{r} \triangleleft \textbf{let } x\texttt{=}e \textbf{ in } P \xrightarrow{\tau} \mathsf{r} \triangleleft P[x := v]}\,[\to\text{P-Let}]$$

$$\frac{e \Downarrow \texttt{true}}{\mathsf{r} \triangleleft \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2 \xrightarrow{\tau} \mathsf{r} \triangleleft P_1}\,[\to\text{P-If1}] \qquad \frac{e \Downarrow \texttt{false}}{\mathsf{r} \triangleleft \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2 \xrightarrow{\tau} \mathsf{r} \triangleleft P_2}\,[\to\text{P-If2}]$$

$$\frac{}{\mathsf{r} \triangleleft \textbf{rec } X.P \xrightarrow{\tau} \mathsf{r} \triangleleft P[X := \textbf{rec } X.P]}\,[\to\text{P-Rec}] \qquad \frac{C_1 \xrightarrow{\alpha} C_1'}{C_1 \mid C_2 \xrightarrow{\alpha} C_1' \mid C_2}$$

$$\frac{C_2 \mid C_1 \xrightarrow{\alpha} C'}{C_1 \mid C_2 \xrightarrow{\alpha} C'} \qquad \frac{(C_1 \mid C_2) \mid C_3 \xrightarrow{\alpha} C'}{C_1 \mid (C_2 \mid C_3) \xrightarrow{\alpha} C'} \qquad \frac{C_1 \mid (C_2 \mid C_3) \xrightarrow{\alpha} C'}{(C_1 \mid C_2) \mid C_3 \xrightarrow{\alpha} C'}$$

Fig. 9. Transition rules for sessions

Output process $\mathsf{q}!\ell(e).P$ implements the send of a message labelled ℓ, with (the value of) expression e as the payload, to role q, followed by P. Input process $\mathsf{p}?\{\ell_i(x_i{:}t_i).P_i\}_{i\in I}$ implements the receive of the payload of a message labelled ℓ_j, from role p, into variable x_j of type t_j, followed by P_j, for some $j \in I$. Process $\textbf{let } x\texttt{=}e \textbf{ in } P$ implements the binding of variable x to the value of expression e in P. Process $\textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2$ implements a conditional choice. Processes $\textbf{rec } X.P$ and X implement recursion. As usual, we stipulate that each process is *message-guarded* (i.e., recursion variables occur only under sends/receives) and *closed* (i.e., recursion variables are bound); these are simple syntactic checks.

- Let $\mathbb{C}$ denote the set of *families of processes*—“sessions”—ranged over by C. It is induced by the following grammar:

$$C \;::=\; \mathsf{r} \triangleleft P \;\big|\; C_1 \mid C_2$$

Session $\mathsf{r} \triangleleft P$ implements role r as process P. Session $C_1 \mid C_2$ implements the parallel composition. As usual, we stipulate that each session implements each role at most once (e.g., $\mathsf{r} \triangleleft P_1 \mid \mathsf{r} \triangleleft P_2$ is ruled out); this is a simple syntactic check.
We note that process creation and session creation are orthogonal to the contributions of this paper and thus we omit them.

Furthermore, let $\mathrm{obj}(P)$ denote the *object* of P: it is the receiver if P is an output process, the sender if P is an input process, and undefined otherwise. It is induced by the following equations:

$$\mathrm{obj}(\mathsf{q}!\ell(e).P) = \mathsf{q} \qquad \mathrm{obj}(\mathsf{p}?\{\ell_i(x_i).P_i\}_{i\in I}) = \mathsf{p}$$

Operational semantics. Regarding the operational semantics of processes:

$$\frac{}{\Gamma, x : t \vdash x : t} \qquad \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{Bool}} \qquad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\mathtt{==}e_2 : \mathtt{Bool}} \qquad \cdots$$

Fig. 10. Typing rules for expressions (excerpt)

- Let $e \Downarrow v$ denote the evaluation of expression e to value v. It is induced by the rules in Figure 8 (excerpt); the rules are standard.
- Let $C \xrightarrow{\alpha} C'$ and $C \xrightarrow{\tau} C'$ denote the transition from session C to session C' through global action α or *internal action* τ; we use these transition labels to formally relate the behaviour of sessions to that of global types when proving safety. It is induced by the rules in Figure 9:
  - Rule [→P-Com] states that an output process and a corresponding input process can make a transition by sending and receiving a message. We note that the communication is synchronous. In this rule, $P[x := v]$ denotes the substitution of x by v in P.
  - The remaining rules are standard. We note that parallel composition of sessions is explicitly commutative and associative (bottom three rules), similar to both the original MPST paper and the "Less Is More" paper [15, 29] (which rely on an auxiliary structural congruence relation that includes commutativity and associativity axioms to define the transition rules; we omitted such a relation for simplicity, as we do not need its full power). Equivalently, a session is a partial function from roles to processes.

4 The Special Case: Typing with Global Types

In this section, we present our results for the special case of our synthetic approach to MPST.

4.1 Type System

Let Γ and Δ denote the sets of *data type environments* and *session type environments*, ranged over by Γ and Δ. They are induced by the following grammar:

$$\Gamma ::= \varnothing \mid \Gamma, x : t \qquad \Delta ::= \varnothing \mid \Delta, X : G$$

As usual, we consider type environments up to reordering of entries for different variables (e.g., we can reorder $\Gamma, \mathtt{x} : \mathtt{Unit}, \mathtt{y} : \mathtt{Bool}$ to $\Gamma, \mathtt{y} : \mathtt{Bool}, \mathtt{x} : \mathtt{Unit}$, but we cannot reorder $\Gamma, \mathtt{x} : \mathtt{Unit}, \mathtt{x} : \mathtt{Bool}$).

Let $\Gamma \vdash E : S$ denote well-typedness of expression E by payload type S in data type environment Γ. It is the smallest relation induced by the rules in Figure 10 (excerpt); the rules are standard.

Let $\Gamma; \Delta \vdash C : G$ denote well-typedness of session C by global type G in type environments Γ and Δ. We write $\vdash C : G$ instead of $\varnothing; \varnothing \vdash C : G$. It is the smallest relation induced by the rules in Figure 11. We first explain how the core, non-standard rules should be read and what they informally mean. In Section 4.2, we provide a more in-depth discussion.

- Rule [⊢-Send] states that, as an implementation of role p (sender), an output process is well-typed when: the payload is well-typed (first premise); the successor is well-typed (second premise); there exists a corresponding transition (third premise). More intuitively, this rule means that each send implemented needs to be specified, but not each send specified needs to be implemented.
- Rule [⊢-Recv] states that, as an implementation of role q (receiver), an input process is well-typed when, for each transition (at least one), a corresponding branch exists (i.e., it has the specified message label) and is well-typed in an extended type environment (i.e., the variable has the specified payload type). More intuitively, this rule means that each receive specified needs to be implemented, but not each receive implemented needs to be specified.
- Rule [⊢-Skip] states that, as an implementation of role r, a process is well-typed when:

$$\dfrac{\Gamma \vdash e : t \qquad \Gamma;\Delta \vdash \mathsf{p} \triangleleft P : G' \qquad G \xrightarrow{\mathsf{p}\rightarrowtriangle\mathsf{q}:\ell(t)} G'}{\Gamma;\Delta \vdash \mathsf{p} \triangleleft \mathsf{q}!\ell(e).P : G}\ [\vdash\text{-Send}]$$

$$\dfrac{G \xrightarrow{\mathsf{p}\rightarrowtriangle\mathsf{q}:\ell'(t')} \qquad \forall\, G \xrightarrow{\mathsf{p}\rightarrowtriangle\mathsf{q}:\ell(t)} G' \,.\, \big[\exists\, j \in I \,.\, \big[\ell_j = \ell \ \wedge\ \Gamma, x_j : t;\Delta \vdash \mathsf{q} \triangleleft P_j : G'\big]\big]}{\Gamma;\Delta \vdash \mathsf{q} \triangleleft \mathsf{p}?\{\ell_i(x_i).P_i\}_{i\in I} : G}\ [\vdash\text{-Recv}]$$

$$\dfrac{\begin{array}{c}(1)\ G \overset{\{\mathsf{r}\}}{\not\longrightarrow} \qquad (2)\ \forall\, G \xrightarrow{\overline{\{\mathsf{r}\}}}{}^{*}\, G' \,.\, \big[\exists\, G'' \,.\, \big[G' \xRightarrow{\overline{\{\mathsf{r}\}}}{}^{*}\, G'' \xrightarrow{\{\mathsf{r}\}}\big]\big] \\ (3)\ \forall\, G = G' \xRightarrow{\overline{\{\mathsf{r}\}}}{}^{*}\, G'' \xrightarrow{\{\mathsf{r}\}} .\ \big[\Gamma;\Delta \vdash \mathsf{r} \triangleleft P : G''\big] \\ (4)\ \forall\, G \xrightarrow{\overline{\{\mathsf{r}\}}}{}^{*}\, G' \,.\, \big[G' \xrightarrow{\{\mathsf{r}\}} \ \vee\ G' \xrightarrow{\overline{\{\mathsf{r},\mathrm{obj}(P)\}}}{}^{*} \xrightarrow{\{\mathsf{r},\mathrm{obj}(P)\}}\big]\end{array}}{\Gamma;\Delta \vdash \mathsf{r} \triangleleft P : G}\ [\vdash\text{-Skip}]$$

$$\dfrac{\Gamma \vdash e : t \quad \Gamma, x : t;\Delta \vdash \mathsf{r} \triangleleft P : G}{\Gamma;\Delta \vdash \mathsf{r} \triangleleft \mathbf{let}\ x{=}e\ \mathbf{in}\ P : G}\ [\vdash\text{-Let}] \qquad \dfrac{\Gamma;\Delta, X : G \vdash \mathsf{r} \triangleleft P : G \quad P \text{ is message-guarded}}{\Gamma;\Delta \vdash \mathsf{r} \triangleleft \mathbf{rec}\ X.P : G}\ [\vdash\text{-Rec}]$$

$$\dfrac{\Gamma \vdash e : \mathtt{Bool} \qquad \Gamma;\Delta \vdash \mathsf{r} \triangleleft P_1 : G \qquad \Gamma;\Delta \vdash \mathsf{r} \triangleleft P_2 : G}{\Gamma;\Delta \vdash \mathsf{r} \triangleleft \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 : G}\ [\vdash\text{-If}]$$

$$\dfrac{G \xrightarrow{\overline{\{\mathsf{r}\}}}{}^{*}\, G'}{\Gamma;\Delta, X : G \vdash \mathsf{r} \triangleleft X : G'}\ [\vdash\text{-Var}] \qquad \dfrac{\forall\, G \xrightarrow{\overline{\{\mathsf{r}\}}}{}^{*}\, G' \,.\, \big[G' \overset{\{\mathsf{r}\}}{\not\longrightarrow}\big]}{\Gamma;\Delta \vdash \mathsf{r} \triangleleft \mathbf{end} : G}\ [\vdash\text{-End}]$$

$$\dfrac{\mathrm{dom}(C_1) \cap \mathrm{dom}(C_2) = \varnothing \qquad \vdash C_1 : G \qquad \vdash C_2 : G}{\vdash C_1 \mid C_2 : G}\ [\vdash\text{-Comp}]$$

Fig. 11. Typing rules for processes

(1) For the present G, a send or receive by r *is not* specified.

(2) For the future, a send or receive by r *is* specified.
That is, for each "near future" G'—reachable through zero-or-more transitions without r, but r may have been enabled—there exists a "distant future" G''—reachable through another zero-or-more transitions without r, and r must have been disabled—for which a send or receive by r is specified. At least one distant future exists (when $G = G'$).

(3) In each distant future G'', the process is well-typed. We note that the "$G =$" part in this premise is technically redundant; we included it so that meta-variables G' and G'' are bound in the same way as in premise (2).
More intuitively, this premise means that an implementation of r ignores all communications in which r does not participate. However, regardless of which other communications other processes engage in (ignored by r), an implementation of r must behave in compliance with any possible future that may arise.

(4) In each near future, either r is enabled, or r and $\mathrm{obj}(P)$ (i.e., the next communication partner of r) cannot communicate with each other until either one of them has communicated with another process.
More intuitively, this premise means that there needs to be some kind of causality: implementations of r and $\mathrm{obj}(P)$ cannot start communicating with each other spontaneously:

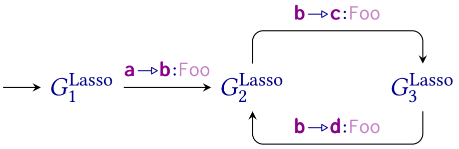
either it must already be possible, or it must happen in response to a communication of one of them with another process.

We note that rule [⊢-Skip] is not syntax-directed. This is different from existing type systems in the MPST literature, including in the classical approach and the “Less Is More” approach.

- Rule [⊢-Rec] and rule [⊢-Var] state that, as an implementation of role r, a recursive process is well-typed when: the body is well-typed (premise of rule [⊢-Rec]); the global types upon starting and finishing the body, G and G', are reachable through transitions without r (premise of rule [⊢-Var]). The latter is a generalisation of the usual equality condition on G and G' in typing rules for recursive processes in the MPST literature. Our relaxation enables typing more recursive processes. The following example demonstrates the usefulness.

  Example 4.1. The following global type and its LTS specify a protocol in which a Foo message is communicated first from Alice to Bob and next, *ad infinitum*, from Bob to Carol and Dave:

$$G^{\text{Lasso}} = \mathbf{a} \to \mathbf{b}{:}\mathtt{Foo}\,.\,\mu\mathtt{X}\,.\ \mathbf{b} \to \mathbf{c}{:}\mathtt{Foo}\,.\ \mathbf{b} \to \mathbf{d}{:}\mathtt{Foo}\,.\ \mathtt{X}$$


G_1^{Lasso}
$\mathbf{a} \to \mathbf{b}{:}\mathtt{Foo}$
G_2^{Lasso}
$\mathbf{b} \to \mathbf{c}{:}\mathtt{Foo}$
G_3^{Lasso}
$\mathbf{b} \to \mathbf{d}{:}\mathtt{Foo}$

  The following derivation (excerpt for simplicity) states that, as an implementation of Dave, process **rec** $\mathtt{X}.\mathbf{b}?\mathtt{Foo}(\mathtt{x}).\mathtt{X}$ is well-typed by G_1^{Lasso}:

$$\dfrac{\dfrac{\dfrac{\dfrac{G_1^{\text{Lasso}} \xrightarrow{\overline{\{\mathbf{d}\}}}{}^{*}\ G_2^{\text{Lasso}}}{\emptyset, \mathtt{x} : \mathtt{Unit}; \emptyset, \mathtt{X} : G_1^{\text{Lasso}} \vdash \mathbf{d} \triangleleft \mathtt{X} : G_2^{\text{Lasso}}}\,[\vdash\text{-Var}] \qquad \cdots}{\emptyset; \emptyset, \mathtt{X} : G_1^{\text{Lasso}} \vdash \mathbf{d} \triangleleft \mathbf{b}?\mathtt{Foo}(\mathtt{x}).\mathtt{X} : G_3^{\text{Lasso}}}\,[\vdash\text{-Recv}] \qquad \cdots}{\emptyset; \emptyset, \mathtt{X} : G_1^{\text{Lasso}} \vdash \mathbf{d} \triangleleft \mathbf{b}?\mathtt{Foo}(\mathtt{x}).\mathtt{X} : G_1^{\text{Lasso}}}\,[\vdash\text{-Skip}]}{\vdash \mathbf{d} \triangleleft \mathbf{rec}\ \mathtt{X}.\mathbf{b}?\mathtt{Foo}(\mathtt{x}).\mathtt{X} : G_1^{\text{Lasso}}}\,[\vdash\text{-Rec}]$$

  This derivation crucially takes advantage of our relaxation: rule [⊢-Var] does not require equality of the global types on the left-hand side and on the right-hand side of the turnstile, but the existence of a sequence of transitions between them is sufficient. □

  In general, typing recursive processes is a non-trivial problem. We are currently working on further generalisations of rule [⊢-Var]. For the purpose of this paper (notably: passing the “Less Is More” benchmark), the current version of rule [⊢-Var] already provides enough expressive power.

For the top-level session, as a well-formedness requirement, the type system also checks that each role that occurs in the global type is implemented as a process in the session.

Just as in the classical approach to MPST, it is possible in our approach to write global types that fundamentally cannot be implemented as well-typed sessions; they are inherently “unrealisable” as distributed systems. In the classical approach, such global types are ruled out by leaving the projection onto at least one role undefined; thus, there are not enough local types to check processes against. In contrast, in our approach, unrealisability manifests through the standard notion of *type inhabitation*. In particular, global types that specify protocols that violate the *Knowledge of Choice* (KC) principle are uninhabited. Intuitively, KC demands that if the future of a protocol depends on choices made in the past, then each role needs to be(come) aware of those choices in a timely fashion. The following example demonstrates an uninhabited global type.

Example 4.2. The *Confusion* protocol consists of roles *Alice*, *Bob*, and *Carol*. First, a Foo message or a Bar message is communicated from Alice to Bob. Next, a Confusion message is communicated from Bob to Carol. Last, a message with the same label as the one that was communicated from Alice to Bob is communicated from Carol to Alice. While Alice and Bob are aware of the choice between Foo and Bar, Carol is not: regardless of the choice, she always receives a Confusion message.

The following global type specifies the Confusion protocol:

$$G^{\text{Conf}} = \mathbf{a}\twoheadrightarrow\mathbf{b}{:}\begin{cases}\mathtt{Foo}\,.\,\mathbf{b}\twoheadrightarrow\mathbf{c}{:}\mathtt{Confusion}\,.\,\mathbf{c}\twoheadrightarrow\mathbf{a}{:}\mathtt{Foo}\,.\,\mathbf{end}\\ \mathtt{Bar}\,.\,\mathbf{b}\twoheadrightarrow\mathbf{c}{:}\mathtt{Confusion}\,.\,\mathbf{c}\twoheadrightarrow\mathbf{a}{:}\mathtt{Bar}\,.\,\mathbf{end}\end{cases}$$

This global type is uninhabited. The problem is that any well-typed implementation of Carol would have to start with receiving a Confusion message:

$$P^{\text{Conf}}_{\mathbf{c}} = \mathbf{b}?\mathtt{Confusion}(\_)\,.\,P'$$

However, P' must now be well-typed by both $\mathbf{c}\twoheadrightarrow\mathbf{a}{:}\mathtt{Foo}.\mathbf{end}$ and $\mathbf{c}\twoheadrightarrow\mathbf{a}{:}\mathtt{Bar}.\mathbf{end}$, so it has to be both $\mathbf{a}!\mathtt{Foo}.\mathbf{end}$ and $\mathbf{a}!\mathtt{Bar}.\mathbf{end}$, which is a contradiction. □

We note that unrealisability implies unprojectability and uninhabitation, but not the other way around: projectability and inhabitation are conservative in the sense that they reject more global types than just the unrealisable ones. For now, the exact relation between projectability and inhabitation is unknown, but we conjecture that the former is strictly subsumed by the latter.

4.2 In-Depth Discussion of the Design of the Typing Rules

4.2.1 Rule [⊢-Skip]. The most complicated rule of the type system is rule [⊢-Skip]. One apparent complication is that it looks ahead multiple transitions of the global type instead of only a single one. The reason why we chose the multi-transition design is that it makes dealing with cycles significantly easier. The key insight is that, ultimately, we need to reason about the *reachable successors* of a global type ("near futures" and "distant futures"), subject to additional conditions along the way. This can be directly expressed by looking ahead multiple transitions, but only indirectly—using substantial additional bookkeeping as part of the typing judgment—by looking ahead a single transition at a time. Essentially, the complexity of dealing with cycles is pushed into the computation of the transitive closure, for which existing algorithms can be straightforwardly adapted (as is done in our prototype language and tooling in VS Code).

When quantifying over reachable successors by looking ahead multiple transitions, a subtle point that needs to be addressed is that the domain is non-empty: at least one reachable successor needs to exist. This is the purpose of premise 2. It ensures that the universal quantification in premise 3 has a non-empty domain *in every possible successor that is reachable after unrelated communications.* This is essential for soundness (i.e., if an empty domain were allowed, then premise 3 would be vacuously true, which would erroneously mean that P could be, or do, anything).

More generally, regarding the efficacy of the four premises of [⊢-Skip], the theorems later on in this paper establish that they are *sufficient* (in the technical sense) to prove type soundness (as also confirmed by our Agda formalisation). We also show that the premises are sufficiently liberal to pass the "Less Is More" benchmark (i.e., for each example in that benchmark, an inhabited global type exists). Whether or not the premises are also *necessary* remains for now an open question.

4.2.2 Rule [⊢-Var]. According to rule [⊢-Var], process variable X is typable by global type G' when G' is reachable from the global type G stored for X in the session type environment. As X can stand for any process, one might expect that a generalisation for any process is sound as well:

$$\frac{\Gamma;\Delta \vdash \mathsf{r} \triangleleft P : G \qquad G \xrightarrow{\overline{\{\mathsf{r}\}}}{}^{*} G'}{\Gamma;\Delta \vdash \mathsf{r} \triangleleft P : G'}\ [\vdash\text{-Forward}]$$

Indeed, this rule is admissible: it is a direct consequence of Lemma 5.6, which we present in the next section. Informally, that lemma states that the well-typedness of a process that implements role r is preserved by any global type transition that does not invole r. Rule [⊢-Forward] is then established by a straightforward inductive argument on a sequence of transitions that do not involve r.

4.2.3 Rule [⊢-End]. In the operational semantics of global types, we do not distinguish between successful and abnormal termination: any global type that has no outgoing transitions is considered successfully terminated. If all recursion is guarded (as stipulated), then the only global type without outgoing transitions is **end**, which specifies succesful termination, so we do not need additional expressive power to represent abnormal termination. In general, however, it can be useful.

Adding an explicit notion of successful termination—independent of the presence/absence of outgoing transitions—is a non-trivial problem to solve, though. In particular, there does not seem to be an obvious way to define a notion of "global final state" that can be used in rule [⊢-End]. For instance, reconsider global type G^{Lasso} from Example 4.1:

$$G^{\text{Lasso}} = G_1^{\text{Lasso}} = \mathbf{a} \twoheadrightarrow \mathbf{b}{:}\mathsf{Foo} \,.\, G_2^{\text{Lasso}} \qquad G_2^{\text{Lasso}} = \mu X \,.\, \mathbf{b} \twoheadrightarrow \mathbf{c}{:}\mathsf{Foo} \,.\, \mathbf{b} \twoheadrightarrow \mathbf{d}{:}\mathsf{Foo} \,.\, X$$

A well-typed implementation of $\mathbf{a}$ is $\mathbf{b}!\mathsf{Foo}.\mathbf{end}$. But, after applying rule [⊢-Send], we would need to type-check **end** against global type G_2^{Lasso}, which cannot be a "global final state". In other words, participants may finish their communications in a protocol before the protocol as a whole terminates (if ever). A possible solution could be to define a separate set of "local final states" for each role, but doing so might have deep consequences that require careful study in future work.

4.3 Main Theoretical Result: Type Soundness

The main theoretical result of the special case of our synthetic approach to MPST is *type soundness*: well-typedness implies safety and liveness. Formally, we prove type soundness in terms of *progress* and *preservation*. Progress means that well-typed sessions eventually either terminate or perform another communication. In particular, it is impossible for well-typed sessions to diverge into an infinite sequence of internal transitions, as all recursion variables in well-typed processes must be message-guarded: at least one send or receive must happen before each recursive call. Thus, well-typed sessions are live (i.e., progress is exactly strong enough to formally define liveness). Preservation means that well-typedness is preserved by transitions of sessions, and that these transitions are allowed by the global types. In particular, if a well-typed session makes a transition through a communication, then the global type can make a transition with exactly the same communication. Thus, well-typed sessions are safe (i.e., preservation is exactly strong enough to formally define safety). We show the proofs of these theorems in Section 5.

Theorem 4.3 (Progress). *If* $\vdash C : G$, *then:* (1) *not* $C \xrightarrow{\tau}\xrightarrow{\tau}\cdots$; (2) $C \xrightarrow{\tau} \cdots \xrightarrow{\tau} \mathbf{end} \mid \cdots \mid \mathbf{end}$, *or* $C \xrightarrow{\tau} \cdots \xrightarrow{\tau}\xrightarrow{\alpha} C'$, *for some* C'.

Theorem 4.4 (Preservation). *Suppose* $\vdash C : G$*:*

- *If* $C \xrightarrow{\alpha} C'$, *then* $\vdash C' : G'$ *and* $G \xrightarrow{\alpha} G'$, *for some* G'.
- *If* $C \xrightarrow{\tau} C'$, *then* $\vdash C' : G$.

We note that our notions of progress and preservation do not prevent *starvation*: while a non-terminating session as-a-whole is guaranteed to alway eventually perform another communication, without *fairness*, individual processes might get stuck waiting for a message that is never sent.

Roles: Server (s), Client (c), Authorisation Service (a)

Protocol: Server tells Client it can continue the session by logging in, or it cancels the session. In the former case, Client tells Authorisation Service its password, after which Authorisation Service tells Server whether the login succeeded. In the latter case, Client tells Authorisation Service to quit.

Global type:

$$\mathsf{s} \rightarrow \mathsf{c}{:}\begin{cases} \mathtt{Login} \,.\, \mathsf{c} \rightarrow \mathsf{a}{:}\mathtt{Passwd}(\mathtt{Str}) \,.\, \mathsf{a} \rightarrow \mathsf{s}{:}\mathtt{Auth}(\mathtt{Bool}) \,.\, \mathbf{end} \\ \mathtt{Cancel} \,.\, \mathsf{c} \rightarrow \mathsf{a}{:}\mathtt{Quit} \,.\, \mathbf{end} \end{cases}$$

Well-typed session:

$$\begin{aligned} & \mathsf{s} \triangleleft \mathsf{c}!\mathtt{Cancel} \,.\, \mathbf{end} \\ \mid\ & \mathsf{c} \triangleleft \mathsf{s}?\{\mathtt{Login}(\_) \,.\, \mathsf{a}!\mathtt{Passwd}(\texttt{"asdf"}) \,.\, \mathbf{end} \,,\, \mathtt{Cancel}(\_) \,.\, \mathsf{a}!\mathtt{Quit} \,.\, \mathbf{end}\} \\ \mid\ & \mathsf{a} \triangleleft \mathsf{c}?\{\mathtt{Passwd}(\mathtt{x}) \,.\, \mathsf{s}!\mathtt{Auth}(\texttt{x=="asdf"}) \,.\, \mathbf{end} \,,\, \mathtt{Quit}(\_) \,.\, \mathbf{end}\} \end{aligned}$$

(a) OAuth2 Fragment

Roles: Alice (a), Store (s), Bob (b)

Protocol: Alice asks Store for a quote of an item. Store tells Alice the price. Alice asks Bob to split the price or to cancel the session. In the former case, Bob tells Alice whether or not he is willing to split. If he is, then Alice tells Store that the purchase goes through, but if not, then she asks Bob again to split the price or to cancel the session. In the latter case, Alice tells Store that no purchase will be made.

Global type:

$$\mathsf{a} \rightarrow \mathsf{s}{:}\mathtt{Query}(\mathtt{Str}) \,.\, \mathsf{a} \rightarrow \mathsf{s}{:}\mathtt{Price}(\mathtt{Int}) \,.\, \mu\mathtt{X} \,.\, \mathsf{a} \rightarrow \mathsf{b}{:}\begin{cases} \mathtt{Split}(\mathtt{Int}) \,.\, \mathsf{b} \rightarrow \mathsf{a}{:}\begin{cases} \mathtt{Yes} \,.\, \mathsf{a} \rightarrow \mathsf{b}{:}\mathtt{Buy} \,.\, \mathbf{end} \\ \mathtt{No} \,.\, \mathtt{X} \end{cases} \\ \mathtt{Cancel} \,.\, \mathsf{a} \rightarrow \mathsf{s}{:}\mathtt{No} \,.\, \mathbf{end} \end{cases}$$

Well-typed session:

$$\begin{aligned} & \mathsf{a} \triangleleft \mathsf{s}!\mathtt{Item}(\texttt{"tapl"}) \,.\, \mathsf{s}?\mathtt{Price}(\mathtt{x}) \,.\, \mathsf{b}!\mathtt{Cancel} \,.\, \mathsf{s}!\mathtt{No} \,.\, \mathbf{end} \\ \mid\ & \mathsf{s} \triangleleft \mathsf{a}?\mathtt{Item}(\mathtt{y}) \,.\, \mathsf{a}!\mathtt{Price}(20) \,.\, \mathsf{s}?\{\mathtt{Buy}(\_) \,.\, \mathbf{end} \,,\, \mathtt{No}(\_) \,.\, \mathbf{end}\} \\ \mid\ & \mathsf{b} \triangleleft \mathsf{a}?\{\mathtt{Split}(\mathtt{z}) \,.\, \mathsf{a}!\mathtt{Yes} \,.\, \mathbf{end} \,,\, \mathtt{Cancel}(\_) \,.\, \mathbf{end}\} \end{aligned}$$

(b) Recursive Two-Buyers

Fig. 12. “Less Is More” benchmark [29, Fig. 4] – spread over two pages

4.4 Main Practical Result: Passing the “Less Is More” Benchmark

The main practical result is that the synthetic approach of this paper passes the “Less Is More” benchmark of Scalas and Yoshida [29]. This is a set of four challenging example protocols that demonstrate limitations of the classical approach to MPST (Figure 1a); it served as a motivation for the “Less Is More” approach (Figure 1b). The type system of this section is the first one to support the example protocols in a fully compositional manner. This means that processes are all individually type-checked, without the need for whole-system reconstruction and analysis (e.g., the model checking step in the “Less Is More” approach).

Figure 12 (spread over two pages) defines, for each example protocol in the “Less Is More” benchmark, a global type and a well-typed session. There are two kinds of example protocols:

- *OAuth2 Fragment* (Figure 12a), *Recursive Map/Reduce* (Figure 12c), and *Independent Multiparty Workers* (Figure 12d) are protocols that can be specified by a projectable global type, but the resulting family of local types is inconsistent.

Roles: Mapper ($\mathtt{m}$), Worker 1 ($\mathtt{w1}$), Worker 2 ($\mathtt{w2}$), Reducer ($\mathtt{r}$)

Protocol: Mapper tells Worker 1 and Worker 2 to each process a datum. Worker 1 and Worker 2 tell Reducer the results of their processing. Reducer tells Master to enter another iteration of mapping/reducing or to stop. In the latter case, Mapper tells Worker 1 and Worker 2 to stop, too.

Global type:

$$\mu \mathtt{X}.\,\mathtt{m}\rightarrow[\mathtt{w1},\mathtt{w2}]{:}\mathtt{Datum}(\mathtt{Int}).\,[\mathtt{w1},\mathtt{w2}]\rightarrow\mathtt{r}{:}\mathtt{Result}(\mathtt{Int}).\,\mathtt{r}\rightarrow\mathtt{m}{:}\begin{cases}\mathtt{Continue}(\mathtt{Int}).\,\mathtt{X}\\ \mathtt{Stop}.\,\mathtt{m}\rightarrow[\mathtt{w1},\mathtt{w2}]{:}\mathtt{Stop}.\,\mathbf{end}\end{cases}$$

We write "$\mathtt{p}\rightarrow[\mathtt{q}_1,\mathtt{q}_2]{:}\ell(t).G$" and "$[\mathtt{p}_1,\mathtt{p}_2]\rightarrow\mathtt{q}{:}\ell(t).G$" instead of "$\mathtt{p}\rightarrow\mathtt{q}_1{:}\ell(t).\mathtt{p}\rightarrow\mathtt{q}_2{:}\ell(t).G$" and "$\mathtt{p}_1\rightarrow\mathtt{q}{:}\ell(t).\mathtt{p}_2\rightarrow\mathtt{q}{:}\ell(t).G$".

Well-typed session:

$$\mathtt{m}\triangleleft\mathbf{rec}\ \mathtt{X}.\,[\mathtt{w1},\mathtt{w2}]!\mathtt{Datum}(123).\,\mathtt{r}?\{\mathtt{Continue}(z).\,\mathtt{X},\ \mathtt{Stop}(\_).\,[\mathtt{w1},\mathtt{w2}]!\mathtt{Stop}.\,\mathbf{end}\}$$
$$\mid \mathtt{w1}\triangleleft P_{\mathtt{w1}} \mid \mathtt{w2}\triangleleft P_{\mathtt{w2}} \mid \mathtt{r}\triangleleft\mathtt{w1}?\mathtt{Result}(y1).\,\mathtt{w2}?\mathtt{Result}(y1).\,\mathtt{m}!\mathtt{Stop}.\,\mathbf{end}$$

where:

$$P_{\mathtt{w}i} = \mathtt{m}?\begin{cases}\mathtt{Datum}(x).\,\mathtt{r}!\mathtt{Result}(x).\,\mathbf{rec}\ \mathtt{X}.\,\mathtt{m}?\{\mathtt{Datum}(x).\,\mathtt{r}!\mathtt{Result}(x).\,\mathtt{X},\ \mathtt{Stop}(\_).\,\mathbf{end}\}\\ \mathtt{Stop}(\_).\,\mathbf{end}\end{cases}$$

We write "$[\mathtt{q}_1,\mathtt{q}_2]!\ell(e).P$" instead of "$\mathtt{q}_1!\ell(e).\mathtt{q}_2!\ell(e).P$".

(c) Recursive Map/Reduce ($n=2$)

Roles: Starter ($\mathtt{s}$), Workers A1, B1, C1 ($\mathtt{wa1}$, $\mathtt{wb1}$, $\mathtt{wc1}$), Workers A2, B2, C2 ($\mathtt{wa2}$, $\mathtt{wb2}$, $\mathtt{wc2}$)

Protocol: Starter tells Worker A1 and Worker A2 to each process a datum. In parallel:

- Worker A1 tells Worker B1 to process the datum or to stop. In the former case, Worker B1 tells Worker C1 to process the datum, after which Worker C1 tells Worker A1 the result, after which Worker A1 again tells Worker B1 to process the datum or to stop. In the latter case, Worker B1 tells Worker C1 to stop, too.
- Workers A2, B2, C2 follow the same sub-protocol as Workers A1, B1, C1, independently.

Global type:

$$\mathtt{s}\rightarrow\mathtt{wa1}{:}\mathtt{Datum}(\mathtt{Int}).\,\mathtt{s}\rightarrow\mathtt{wa2}{:}\mathtt{Datum}(\mathtt{Int}).\,(G_1 \parallel G_2)$$

where:

$$G_i = \mu X.\,\mathtt{wa}i\rightarrow\mathtt{wb}i{:}\begin{cases}\mathtt{Datum}(\mathtt{Int}).\,\mathtt{wb}i\rightarrow\mathtt{wc}i{:}\mathtt{Datum}(\mathtt{Int}).\,\mathtt{wc}i\rightarrow\mathtt{wa}i{:}\mathtt{Result}(\mathtt{Int}).\,\mathtt{X}\\ \mathtt{Stop}.\,\mathtt{wb}i\rightarrow\mathtt{wc}i{:}\mathtt{Stop}.\,\mathbf{end}\end{cases}$$

Well-typed session:

$$\mathtt{s}\triangleleft\mathtt{wa1}!\mathtt{Datum}(123).\,\mathtt{wa2}!\mathtt{Datum}(456).\,\mathbf{end} \mid C_1 \mid C_2 \qquad C_i = \mathtt{wa}i\triangleleft P_{\mathtt{wa}i} \mid \mathtt{wb}i\triangleleft P_{\mathtt{wb}i} \mid \mathtt{wc}i\triangleleft P_{\mathtt{wc}i}$$

where:

$$P_{\mathtt{wa}i} = \mathtt{s}?\mathtt{Datum}(x).\,\mathtt{wb}i!\mathtt{Stop}.\,\mathbf{end}$$

$$P_{\mathtt{wb}i} = \mathtt{wa}i?\begin{cases}\mathtt{Datum}(x).\,\mathtt{wc}i!\mathtt{Datum}(x).\,\mathbf{rec}\ \mathtt{X}.\,\mathtt{wa}i?\{\mathtt{Datum}(x).\,\mathtt{wc}i!\mathtt{Datum}(x).\,\mathtt{X},\ \mathtt{Stop}(\_).\,P'_{\mathtt{wb}i}\}\\ \mathtt{Stop}(\_).\,P'_{\mathtt{wb}i} \qquad P'_{\mathtt{wb}i} = \mathtt{wc}i!\mathtt{Stop}.\,\mathbf{end}\end{cases}$$

$$P_{\mathtt{wc}i} = \mathtt{wb}i?\begin{cases}\mathtt{Datum}(x).\,\mathtt{wc}i!\mathtt{Result}(x).\,\mathbf{rec}\ \mathtt{X}.\,\mathtt{wb}i?\{\mathtt{Datum}(x).\,\mathtt{wc}i!\mathtt{Result}(x).\,\mathtt{X},\ \mathtt{Stop}(\_).\,\mathbf{end}\}\\ \mathtt{Stop}(\_).\,\mathbf{end}\end{cases}$$

(d) Independent Multiparty Workers ($n=2$)

Fig. 12. "Less Is More" benchmark [29, Fig. 4] – spread over two pages

- *Recursive Two-Buyers* (Figure 12b) is a protocol that can be specified by a global type, but it is not projectable (neither using plain projection, nor using full projection). Thus, this is the first time that the safety and liveness of implementations of Recursive Two-Buyers can be proved using a global type.

5 The General Case: Typing with LTSs

As demonstrated in Section 2.4, the synthetic approach can be generalised—beyond global types—to define the typing of sessions even without discussing the syntax of the types themselves. After all, an important observation from the typing rules of Figure 11 is that *no rule relies on the syntactic structure of global types*. This is the essence of our synthetic approach to MPST. Two questions arise naturally from this observation:

(1) Can we consider that the rules in Figure 11 refer to a generic, semantic notion of behaviour that does not depend on a particular syntactic structure?
(2) What are the properties that are required so these semantic objects still allow our type system to guarantee safety and liveness?

Regarding the first question, as shown in the previous sections, we see an LTS as a classifier for a session. This LTS must model the communications among all processes that participate in the session. Regarding the second question, Section 5.1 defines a *well-behaved multiparty LTS* as an LTS that exhibits the shape and properties required for well-typedness to imply safety and liveness in our type system.

Naturally, global types from MPST presentations in the literature can be seen as syntactic objects that support all the requirements of well-behaved multiparty LTSs. Section 5.2 describes how global types in Section 4 (following [35]) intrinsically constitute well-behaved MLTSs in synthetic MPST.

5.1 Well-Behaved Multiparty LTSs (WB-MLTS)

Well-behaved multiparty LTSs consist of transitions of the form $B \xrightarrow{\alpha} B'$ that satisfy the set of properties below. First, we introduce the definition of MLTSs.

Definition 5.1 (Multiparty Labelled Transition System (MLTS)). An MLTS is an LTS $(\mathcal{B}, A, \longrightarrow)$, with a set of states $B, \ldots \in \mathcal{B}$, and global action labels $\alpha \in A$ of the form $\mathsf{p} \to \mathsf{q}{:}\ell(t)$, with $\mathsf{p} \neq \mathsf{q}$.

The typing judgement and typing rules of Figure 11 are then parameterised by MLTSs: $\Gamma; \Delta \vdash \mathsf{r} \triangleleft P : B$. However, simply type-checking against an arbitrary MLTS does not guarantee progress and preservation. To provide these stronger guarantees, we need to restrict to MLTSs that satisfy a set of *well-behavedness conditions*, that specify the criteria that transitions of MLTSs must satisfy. This relies on the notion of receiver disjointness.

Definition 5.2 (Receiver Disjointness). Two global actions $\alpha_1 = \mathsf{p} \to \mathsf{q}{:}\ell_1(t_1)$ and $\alpha_2 = \mathsf{r} \to \mathsf{s}{:}\ell_2(t_2)$ are receiver-disjoint, $\alpha_1 \diamond \alpha_2$, iff $\mathsf{q} \notin \{\mathsf{r}, \mathsf{s}\}$ and $\mathsf{s} \notin \{\mathsf{p}, \mathsf{q}\}$.

Intuitively, if two global actions are receiver-disjoint, then they should be able to be reordered. The idea is that, in a concurrent system, receivers are by definition independent from each other, so the order in which a sender sends messages to them does not matter. We are now ready to define well-behaved multiparty LTSs.

Definition 5.3 (Well-Behaved Multiparty LTS (WB-MLTS)). A WB-MLTS is an MLTS $(\mathcal{B}, A, \longrightarrow)$ that satisfies **all** of the following conditions for any state B:

(1) **Sender determinacy:** For all $B \xrightarrow{\alpha_1} B_1$ and $B \xrightarrow{\alpha_2} B_2$, then either $\alpha_1 \diamond \alpha_2$, or there exists p, q, ℓ_1, ℓ_2, t_1, and t_2 such that $\alpha_1 = \mathsf{p} \to \mathsf{q}{:}\ell_1(t_1)$ and $\alpha_2 = \mathsf{p} \to \mathsf{q}{:}\ell_2(t_2)$.

(2) **Determinism:** For all $B \xrightarrow{\alpha} B_1$ and $B \xrightarrow{\alpha} B_2$, then $B_1 = B_2$.

(3) **Conditional commutativity:** For all $B \xrightarrow{\mathsf{r} \rightarrowtriangle \mathsf{s}:\ell_1(t_1)} B_1 \xrightarrow{\mathsf{p} \rightarrowtriangle \mathsf{q}:\ell_2(t_2)} B'$, if there exist ℓ and t such that $B \xrightarrow{\mathsf{p} \rightarrowtriangle \mathsf{q}:\ell(t)}$ and $\{\mathsf{p}, \mathsf{q}\} \cap \{\mathsf{r}, \mathsf{s}\} = \varnothing$, then there exists a B_2 such that $B \xrightarrow{\mathsf{p} \rightarrowtriangle \mathsf{q}:\ell_2(t_2)} B_2 \xrightarrow{\mathsf{r} \rightarrowtriangle \mathsf{s}:\ell_1(t_1)} B'$.

(4) **Diamond (confluence for reorderable global actions):** For all $B \xrightarrow{\alpha_1} B_1$, and $B \xrightarrow{\alpha_2} B_2$, if $\alpha_1 \diamond \alpha_2$, then there exists a B' such that $B_1 \xrightarrow{\alpha_2} B'$ and $B_2 \xrightarrow{\alpha_1} B'$.

Intuitively, sender determinacy states that, if two global actions are possible in a state, then these actions cannot have different senders but the same receiver. That is, the sender is fixed. Conditional commutativity states that an alternative in a choice cannot become available for two roles p and q after unrelated communications. In other words, for a global action $\mathsf{p} \rightarrowtriangle \mathsf{q}:\ell_2(t_2)$ to become available, either p or q must have received a message enabling this choice. This condition ensures that well-formed MLTSs do not specify "bad" protocols in which actions at one process can enable actions at another process without any interaction between those two processes. After all, in the absence of covert communication between them, it is impossible to implement such processes. Thus, such protocols are ruled out by conditional commutativity.

From WB-MLTS's well-behavedness criteria, we prove a series of lemmas that are then used to establish the standard **progress** and **preservation** properties. The most important of these lemmas are: (1) if we have two processes P_p and P_q well-typed with regards to B as roles p and q, and P_p is ready to send ℓ_j to q, and P_q is ready to receive from p, then the state B must accept global action $\mathsf{p} \rightarrowtriangle \mathsf{q}:\ell_j(t_j)$; (2) the continuations of the output/input processes are still well typed; and (3) a well-typed process is still well-typed after an unrelated global action.

Lemma 5.4. *If* $\vdash \mathsf{p} \triangleleft \mathsf{q}!\ell_j(e).P : B$, *and* $\vdash \mathsf{q} \triangleleft \mathsf{p}?\{\ell_i(x_i).P_i\}_{i \in I} : B$, *then there exists a* B' *such that* $B \xrightarrow{\mathsf{p} \rightarrowtriangle \mathsf{q}:\ell_j(t_j)} B'$.

Lemma 5.5 (Inversions of [$\vdash$-Send] and [$\vdash$-Recv]). *Suppose a state* B *such that* $B \xrightarrow{\mathsf{p} \rightarrowtriangle \mathsf{q}:\ell_j(t_j)} B'$*:*

(1) *If* $\vdash \mathsf{p} \triangleleft \mathsf{q}!\ell_j(e).P : B$, *then* $\vdash \mathsf{p} \triangleleft P : B'$

(2) *If* $\vdash \mathsf{q} \triangleleft \mathsf{p}?\{\ell_i(x_i).P_i\}_{i \in I} : B$, *then* $\varnothing, x_j : t_j; \varnothing \vdash \mathsf{q} \triangleleft P_j : B'$.

Lemma 5.6. *Suppose a state* B, *a role* r, *and a global action* α *such that* r *does not occur in* α. *If* $B \xrightarrow{\alpha} B'$, *and* $\Gamma; \Delta \vdash \mathsf{r} \triangleleft P : B$, *then* $\Gamma; \Delta \vdash \mathsf{r} \triangleleft P : B'$

We now state the main theorems.

Theorem 5.7 (Progress). *Suppose a state* B *of a WB-MLTS. If* $\vdash C : B$, *then:* (1) *not* $C \xrightarrow{\tau}\xrightarrow{\tau} \cdots$; (2) $C \xrightarrow{\tau} \cdots \xrightarrow{\tau} \mathbf{end} \mid \cdots \mid \mathbf{end}$, *or* $C \xrightarrow{\tau} \cdots \xrightarrow{\tau}\xrightarrow{\alpha} C'$, *for some* C'.

Theorem 5.8 (Preservation). *Suppose a state* B *of a WB-MLTS and* $; \vdash C : B$*:*

- *If* $C \xrightarrow{\alpha} C'$, *then* $\vdash C' : B'$ *and* $B \xrightarrow{\alpha} B'$, *for some* B' *(i.e.,* B' *is also a state of the same WB-MLTS).*
- *If* $C \xrightarrow{\tau} C'$, *then* $\vdash C' : B$.

Finally, we note that as long as the MLTSs have finitely many states, the type system is decidable. First, all the typing rules in Figure 11 are structural except [$\vdash$-Skip] and, as long as the MLTS has finitely many states, all of their premises are decidable, and the domain of any universal quantification is finite. In rule [$\vdash$-Skip], the size of the process in the premises does not grow; [$\vdash$-Skip]'s premises (1), (2), and (4) are also decidable for any finite-state MLTS; and the universal quantification of (3) is also finite. Note, also, that [$\vdash$-Skip] cannot be applied twice in a row: premise (1) becomes false after one use of this rule, after which, one structural rule must be used. Therefore, type-checking must terminate, and our VS Code extension is implemented following this approach. Details about algorithmic and performance aspects are covered in Section A.

5.2 Global Types as Multiparty Behaviours

We prove that the global types of Section 3.1 satisfy all of the conditions of Definition 5.3, as a consequence of the operational semantics of global types. Corollary 5.10 below is then a consequence of Theorems 5.7 to 5.9.

Theorem 5.9. *Any global type G satisfies the well-behavedness conditions of Definition 5.3.*

Corollary 5.10 (Progress and Preservation of Well-typed Sessions with Global Types). *A well-typed session with a grammatical global type satisfies progress and preservation.*

6 Formalisation of Synthetic MPST in Agda

A big advantage of the synthetic approach to MPST is that it leads to a simpler formalisation in proof assistants. We justify this claim by presenting a formalisation of the type system in Sections 4 and 5, and a comparison with respect to similar formalisations of MPST in the literature. While our formalisation was done in Agda, it should be straightforward to port it to other proof assistants.

6.1 An Agda Encoding of the Synthetic Approach

The core part of our formalisation is the encoding of well-behaved MLTSs (Definition 5.3). In Agda, we encode them in terms of records parameterised by the number of participants in the protocol: (1) record `BTheory` encodes MLTSs, and (2) record `BT-Prop` encodes the well-behavedness properties. Their encoding in Agda is straightforward, in that it relies on a direct encoding of an LTS as a relation between two states and an action. The remaining definitions are also encoded as expected, and they can be encoded in a similar fashion in other proof assistants.

The encoding of the type system is done within a module that is parameterised by well-behaved MLTSs, i.e., an Agda module that takes as a parameter a record of type `B : BTheory`, and a record that proves that it is well-behaved `BP : BT-Prop B`. The type system itself is defined as a relation between process names, process terms, and well-behaved MLTSs.

The full proofs of progress and preservation are done for arbitrary well-behaved MLTSs in about 650 LOC of Agda code. In general, the key lemmas are also a direct encoding of the ones presented in Section 5. The most difficult proof in our system is showing that any global type has a well-behaved MLTS (roughly 2000 LOC), and even in this case, the majority of the proof is about dealing with binders, renaming, etc., which is tedious but not intellectually complicated.

In our experience, with the synthetic approach to MPST, there is no need to massage the definitions to make the proofs simpler/more natural, unlike when mechanising the classical approach to MPST. We further elaborate this point in the next subsection.

6.2 Comparison with the Mechanisation of Classical MPST

The biggest difficulty in mechanising classical MPST is dealing with the *projection* operator and *recursion*. Specifically, the hardest part is showing that if a (possibly) recursive global type is projectable, then the corresponding family *of local types* is indeed safe and live. Once this is proved, though, it tends to be straightforward to show that type-checking against a safe and live family of local types entails safety and liveness of the well-typed family of processes.

In contrast, the synthetic approach avoids dealing with projection altogether, but we need to deal with a more complex typing relation, where the most complex rule is [⊢-Skip]. The question that we address in this section is: why is it the case that dealing with [⊢-Skip] leads to much simpler proofs than showing that projectability leads to safety and liveness of local types? We will review the most representative mechanisations to illustrate this. There are two main approaches to discuss.

Zooid and related approaches. Zooid [7] relies on a notion of coinductive projection and unrolling of global/local types to guarantee the correspondence between a global type, and the projection of all of the roles in the global type. This is similar to other work that subsumes the proofs in Zooid, e.g., by Tirore et al. [31]. There are several challenges in using a coinductive notion of projection. Firstly, the use of coinductive relations often results in cumbersome proofs within proof assistants. Second, deciding a coinductive projection relation is not straightforward. The most common approach in the literature is to use a more restrictive syntactic projection function, and then show that this syntactic projection is contained within the coinductive projection. Note, however, that the more complex syntactic projection is, the harder it is to mechanise.

For example, Castro-Perez et al. [7] only formalise syntactic projection that uses *plain merge*. This restriction rules out all of the examples in this paper, as well a the majority of the examples in our Agda formalisation. Similarly, Tirore et al. [30] mechanise in Rocq a *sound and complete* projection, that handles μ-binders correctly, so that their syntactic projection exactly corresponds to a notion of coinductive projection. However, it uses *plain merge* as well, as does the authors' follow-up work [31]. This further illustrates the difficulty of dealing with syntactic projection, while it also indicates the complexities of supporting the more expressive notion of *full merge*. To our knowledge, full merge has never been mechanised yet.

Li et al. [24] showed that a *complete* projection relation – where every implementable global type is projectable – can be obtained via automata-theoretic methods, with their notion of implementability later formalised in Rocq [25]. Their framework separates synthesis from implementability checking, the latter decided by a set of *Coherence Conditions*. In essence, their synthesis and implementability together correspond to (a more expressive form of) the classical projection. Thus, their approach does not avoid the challenges associated with defining and reasoning about a projection relation. An interesting open problem is clarifying the precise relationship between Li et al.'s Coherence Conditions and our notions of Well-Behavedness and type checking: this could lead to a unified and more expressive projection-less approach.

Multiparty GV. MPGV [18] allows for multiple sessions, and the mechanisation builds on top of a complex mechanisation of *connectivity graphs* [17] to deal with session interleaving, and guarantee that a series of invariants are preserved. Global types in MPGV are restricted to plain merge, and still rely on a notion of coinductive projection, which causes the same difficulties as the Zooid approach. MPGV also offers a coinductive, global-type-free (i.e. bottom-up) formulation of consistency, and the authors prove that projectability implies consistency. It provides a compositional and mechanised framework supporting multiple interleaved sessions. In contrast, our work develops a top-down form of compositionality, where processes are type-checked directly against richer global specifications within a single-session setting.

The synthetic approach completely abstracts away the syntax used to encode recursive protocols, and avoids completely the need to deal with folding/unfolding, and projection. Instead, MLTSs can have cycles, but the presence or absence of such cycles does not complicate the formalisation.

One might expect that the ability of [⊢-Skip] to postpone type checking in the synthetic approach would increase proof complexity. The main reason this increase does not arise lies in the conditions under which a state is considered "skippable." First, all conditions of [⊢-Skip] must hold in any "near-future" state reachable without involving the role currently being type-checked. This guarantees that [⊢-Skip] can be reapplied in each such state, thereby simplifying the proof of Lemma 5.6. Another potential source of complexity is the need to reason about permutations of actions; however, this does not arise in our formalisation. The rule [⊢-Skip] cannot be applied to two processes that are, respectively, ready to send and to receive from each other. In this sense, its conditions impose a form of determinism on the application of typing rules, effectively eliminating many potentially cumbersome proof cases. This simplification is illustrated in the proof of Lemma 5.4.

Thus, with the synthetic approach, *the absence* of the need to reason about the projectability of (possibly) recursive global types *does* make the formalisation significantly simpler, while *the presence* of the need to reason about rule [⊢-Skip] *does not* make it significantly more complex.

7 Prototype Language and Tooling of Synthetic MPST in VS Code

As demonstrated in Section 2.4, we developed a prototype language and tooling of the synthetic approach to MPST as an extension of *VS Code*, including a dedicated *LSP server*.

To develop this prototype, we use the *Rascal* meta-programming language [21]. Among other features, Rascal has core support to write context-free grammars (for defining concrete syntax), algebraic data types (for defining abstract syntax), and advanced pattern matching on grammar rules and ADT constructors. Together with standard programming abstractions, these features aim to simplify the implementation of parsers, type checkers, interpreters, and code generators.

Leveraging Rascal, the implementation of the type-checking algorithm is done in about 200 LOC, and it relies on a graph representation of protocols. The key insight is that, as long as the protocol can be represented as a finite-state MLTS, then the conditions for our rules are decidable.

8 Additional Related Work

In addition to the related work discussed in Sections 1.2 and 6.2, the following contributions in the literature are relevant to this paper, too. In particular, starting from the introduction of MPST [15] there is a substantial lineage of papers that seek to improve the expressiveness of the MPST method. Below, we focus on two main aspects: first, using the synthetic approach to behavioural typing [19] to simplify MPST theory by removing projection and merge. And second, enabling the use of more powerful classifiers (types) for sessions (i.e., WB-MLTSs) to be able to type more protocols.

Using the operational semantics of types is the key ingredient of the synthetic approach. This was first studied in the context of *multiparty compatibility* (MC) [8] and extensions [4, 22, 23]. The idea is to interpret local types as *communicating finite state machines* (CFSM) [5]. Multiparty compatibility, then, is a predicate on the joint state space of the CFSMs to ensure safety and liveness. As such, MC is a *bottom-up technique* (from local view to a global view), whereas the synthetic approach in this paper is a *top-down technique*.

A different, but related, technique is the notion of well-behaved local types as studied by Jongmans and Ferreira [19], of which a rudimentary version (without processes and type checking) was studied by Jongmans and Yoshida [20]. The key difference between their work and ours is that they rely on local types against which processes are type-checked. In contrast, in this paper, we type processes directly against global types. As a result, for the first time in the MPST literature, we obtain a notion of type inhabitation that is independent of auxiliary concepts such as projectability and/or well-formedness. In particular, global types that specify *unrealisable protocols* are uninhabited.

Another version of well-behaved global types was studied by Gheri et al. [9], in the context of choreography automata [2], but it is limited to projection (no type checking).

Our approach avoids issues with merge by avoiding projection altogether. However, there are several non-traditional techniques for projection in the MPST literature. Lopez et al. [26] capture projection in a decidable type equivalence. Castellani et al. [6] and Hamers et al. [11] do not use projection at all, but type-check families of processes against global types (non-compositional).

Scalas and Yoshida [29], van Glabbeek et al. [34], and Peters and Yoshida [28] presented examples of safe and live families of processes that are unsupported by the MPST method due to limited expressiveness of global types. Scalas and Yoshida address the issue by developing an MPST theory without global types (only local types), while van Glabbeek et al. address the issue by developing improved merging. In contrast, in this paper, we address the issue by proposing an expressive type system to verify protocol implementations purely against global specifications of behaviour (i.e., global types and WB-MLTSs). We believe that having a singular global specification has intrinsic value as a programming artefact that comprehensively defines protocols from a system-wide perspective. Finally, Peters and Yoshida study the expressivity of a session calculus typable by a collection of local types with mixed choice, that we do not address directly in this paper. To support this, the main challenge is to relax the sender determinacy condition without breaking soundness.

9 Conclusion and Future Work

Summary. We have presented the synthetic approach to MPST. The main theoretical result is that well-typedness implies safety and liveness. The main practical result is that our type system is expressive enough to pass the "Less Is More" benchmark compositionally (i.e., we support at least all the challenging examples of Scalas and Yoshida [29]). This has been an open problem for several years. Our complete formalisation in Agda, together with examples, demonstrates that the synthetic approach leads to simpler formalisations in proof assistants. Furthermore, a key practical advantage of the synthetic approach is its ability to extend the expressiveness of MPST by purely relying on global protocol specifications: well-behaved multiparty LTSs in general, and global types as a special case. That is, we showed that a simple form of classical MPST satisfies the necessary well-behavedness conditions to ensure safety and liveness within the synthetic approach.

Discussion. For simplicity, our approach uses a synchronous communication semantics. The main complication with asynchronous communication semantics is that multiparty LTSs may no longer be finite, which may affect decidability. We believe that a careful application of *run-time global types* in the Zooid approach [7] might be adapted to a synthetic setting to address this issue.

In principle, a top-down approach like ours inhabits all process systems provable with a bottom-up approach like "Less Is More". The compared expressivity is difficult to establish at a theoretical level, though. For example, "Less Is More" typing contexts and their reduction semantics can be used as multiparty LTSs. The issue is to determine if well-behavedness together with type inhabitation is equivalent to consistency property φ in the "Less Is More" approach. Since well-behavedness is weaker than the typing context properties in that approach, our intuition is that both approaches are equally expressive.

Sharing a global view among all processes may not always be acceptable. For instance, in a ring protocol, it may be desirable to hide the size of the ring from each of the processes. To address this, we are currently studying ways to avoid exploring unrelated transitions, by relying on strong confluence. This may allow a form of lightweight projection – only used for type checking – where we remove irrelevant transitions from the LTS to hide information. The lightweight projection may also enable potential optimisations to the type checking algorithm.

Future work. These results open the door to several promising extensions. One direction is to identify the largest class of syntactic protocol descriptions that satisfy our well-behavedness criteria, potentially removing the need to prove well-behavedness when implementing protocols. Another is to generalise these criteria further, increasing the expressiveness of our type system (e.g studying global types with mixed choice in the style of [19] or [28].) Finally, we aim to extend our multiparty LTSs with verification conditions that allow properties typically established via model checking to be verified directly through type checking.

References

[1] Davide Ancona et al. 2016. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages* 3, 2-3 (2016), 95–230.

[2] Franco Barbanera, Ivan Lanese, and Emilio Tuosto. 2020. Choreography Automata. In *COORDINATION (LNCS, Vol. 12134)*. Springer, 86–106.

[3] J. F. A. K. Van Benthem. 1974. Hintikka on Analyticity. *Journal of Philosophical Logic* 3, 4 (1974), 419–431. doi:10.1007/bf00257484

[4] Laura Bocchi, Julien Lange, and Nobuko Yoshida. 2015. Meeting Deadlines Together. In *CONCUR (LIPIcs, Vol. 42)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 283–296.

[5] Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (1983), 323–342.

[6] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2022. Asynchronous Sessions with Input Races. *CoRR* abs/2203.12876 (2022).

[7] David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Stephen N. Freund and Eran Yahav (Eds.). ACM, Virtual Event, Canada, June 20-25, 2021, 237–251. doi:10.1145/3453483.3454041

[8] Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *ICALP (2) (LNCS, Vol. 7966)*. Springer, 174–186.

[9] Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. 2022. Design-By-Contract for Flexible Multiparty Session Protocols. In *ECOOP (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:28.

[10] Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. 2019. Precise subtyping for synchronous multiparty sessions. *J. Log. Algebraic Methods Program.* 104 (2019), 127–173.

[11] Ruben Hamers and Sung-Shik Jongmans. 2020. Discourje: Runtime Verification of Communication Protocols in Clojure. In *TACAS (1) (LNCS, Vol. 12078)*. Springer, 266–284.

[12] Jaakko Hintikka. 1973. *Logic, Language-Games and Information: Kantian Themes in the Philosophy of Logic.* Oxford, England: Oxford, Clarendon Press.

[13] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR (Lecture Notes in Computer Science, Vol. 715)*. Springer, 509–523.

[14] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP (Lecture Notes in Computer Science, Vol. 1381)*. Springer, 122–138.

[15] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 273–284. doi:10.1145/1328438.1328472

[16] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016), 3:1–3:36.

[17] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–33. doi:10.1145/3498662

[18] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.* 6, ICFP (2022), 466–495.

[19] Sung-Shik Jongmans and Francisco Ferreira. 2023. Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types. In *ECOOP (LIPIcs, Vol. 263)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 42:1–42:30.

[20] Sung-Shik Jongmans and N. Yoshida. 2020. Exploring Type-Level Bisimilarity towards More Expressive Multiparty Session Types. In *ESOP (LNCS, Vol. 12075)*. Springer, 251–279.

[21] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009.* IEEE Computer Society, 168–177. doi:10.1109/SCAM.2009.28

[22] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. 2015. From Communicating Machines to Graphical Choreographies. In *POPL*. ACM, 221–232.

[23] Julien Lange and Nobuko Yoshida. 2019. Verifying Asynchronous Interactions via Communicating Session Automata. In *CAV (1) (LNCS, Vol. 11561)*. Springer, 97–117.

[24] Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. 2023. Complete Multiparty Session Type Projection with Automata. In *Computer Aided Verification - 35th International Conference, CAV 2023, July 17-22, 2023, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 13966)*, Constantin Enea and Akash Lal (Eds.). Springer, Paris, France, 350–373. doi:10.1007/978-3-031-37709-9_17

[25] Elaine Li and Thomas Wies. 2025. Certified Implementability of Global Multiparty Protocols. In *16th International Conference on Interactive Theorem Proving, ITP 2025, September 28 to October 1, 2025, Reykjavik, Iceland (LIPIcs, Vol. 352)*, Yannick Forster and Chantal Keller (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:20. doi:10.4230/LIPICS.ITP.2025.15

[26] Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2015. Protocol-based verification of message-passing parallel programs. In *OOPSLA*. ACM, 280–298.

[27] Robin Milner. 1982. Four Combinators for Concurrency. In *PODC*. ACM, 104–110.

[28] Kirstin Peters and Nobuko Yoshida. 2024. Separation and Encodability in Mixed Choice Multiparty Sessions. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science* (Tallinn, Estonia) *(LICS '24)*. Association for Computing Machinery, New York, NY, USA, Article 62, 15 pages. doi:10.1145/3661814.3662085

[29] Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* 3, POPL (2019), 30:1–30:29.

[30] Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. 2023. A Sound and Complete Projection for Global Types. In *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland (LIPIcs, Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:19. doi:10.4230/LIPICS.ITP.2023.28

[31] Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. 2025. Multiparty Asynchronous Session Types: A Mechanised Proof of Subject Reduction. In *39th European Conference on Object-Oriented Programming, ECOOP 2025 (LIPIcs, Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, June 30 to July 2, 2025, Bergen, Norway, 31:1–31:30. doi:10.4230/LIPICS.ECOOP.2025.31

[32] Thien Udomsrirungruang and Nobuko Yoshida. 2025. Top-Down or Bottom-Up? Complexity Analyses of Synchronous Multiparty Session Types. *Proc. ACM Program. Lang.* 9, POPL (2025), 1040–1071.

[33] Rob van Glabbeek. 2024. Comparing the Expressiveness of the π-calculus and CCS. *ACM Trans. Comput. Log.* 25, 1 (2024), 1:1–1:58.

[34] Rob van Glabbeek, Peter Höfner, and Ross Horne. 2021. Assuming Just Enough Fairness to make Session Types Complete for Lock-freedom. In *LICS*. IEEE, 1–13.

[35] Nobuko Yoshida and Lorenzo Gheri. 2020. A Very Gentle Introduction to Multiparty Session Types. In *Distributed Computing and Internet Technology*, Dang Van Hung and Meenakshi D´Souza (Eds.). Springer International Publishing, Cham, 73–93.

A On Decidability and Efficiency

A.1 Algorithm

As long as MLTSs have finitely many states, the type system is decidable. To substantiate this claim, we describe a type checking algorithm that succesfully applies typing rules to all the processes of the session-under-analysis or fails. Either way, it terminates. The type checker in our VS Code extension uses this algorithm.

First, we note that the type system in Figure 11 is *almost* syntax-directed: all typing rules are, except rule [$\vdash$-Skip]. However, premise 4 of rule [$\vdash$-Skip] demands that P is an output or input process (because $\mathrm{obj}(P)$ must be defined), while premise 1 demands that the role implemented by P cannot send or receive (which is the opposite of rules [$\vdash$-Send] and [$\vdash$-Recv]). Thus, it is impossible for rule [$\vdash$-Skip] to be applicable at the same time as the other rules (i.e., the rules are mutually exclusive). The type checking algorithm leverages this insight by proceeding in two phases:

- **Phase 1:** Apply all rules except [$\vdash$-Skip] in a bottom-up, syntax-directed fashion until a process is reached for which none of these rules are applicable.
- **Phase 2:** When no other rules apply, then check if [$\vdash$-Skip] is applicable. If so, then type checking continues with phase 1. Otherwise, type checking fails.

To see that each phase 1 and each phase 2 individually terminates, we first note that all the premises of all the rules—including [$\vdash$-Skip]—that require us to "query" an MLTS for the presence/absence of particular transitions are decidable. This is a direct consequence of the assumption that MLTSs have finitely many states (hence, finitely many transitions to exhaustively consider). Algorithmically, the type checker may compute the state space of an MLTS on-the-fly, by need, as it tries to apply typing rules. There are two situations: a rule requires the computation of a single transition, or multiple transitions in sequence. Rules [⊢-Send] and [⊢-Recv] are an example of the former. In this situation, it suffices to check whether the current state admits the transition (which is the leitmotif of synthetic typing). In the case of rule [⊢-Skip], sequences of transitions are computed incrementally until a state is reached that allows the current process to perform an action. With rule [⊢-End], the stop condition when computing sequences of transitions is when a state in which the current process is allowed to perform an action, is no longer reachable.

What remains to be shown, then, is that the type checking algorithm does not diverge in an infinite sequence of phases 1 and 2. First, we note that phase 1 is *structural* in the sense that with each rule application, the process becomes smaller. Thus, it is impossible to have an infinite sequence of phase 1. The key insight, now, is that it is never possible to apply rule [⊢-Skip] twice in a row. This is because after applying rule [⊢-Skip] once, there must be a remaining action, so one of the structural rules must apply (i.e., as part of premise 3, due to the way G'' is constructed, it is impossible to have another application of rule [⊢-Skip], as premise 1 is false for G''). Thus, after each phase 2, there must be a phase 1: it is also impossible to have an infinite sequence of phase 2. So, all in all, we apply the structural rules eagerly, until we type check the whole process, with the occasional [⊢-Skip] when no structural rules apply, after which another structural rule must apply.[5]

A.2 Performance

Regarding performance, first, we note that using the algorithm of Section A.1, type checking against an arbitrary MLTS is polynomial in the number of states of that LTS.

In the special case, when the MLTS arises from a global type **without parallel composition**, the size of the MLTS itself is linear in the size of that global type. Thus, the complexity of type checking against a global type is polynomial. In particular, such global types do not pose a particular computational challenge when it comes to rule [⊢-Skip]. To further illustrate this, consider the following global type:

$$\mathbf{a} \rightarrow \mathbf{b}{:}\begin{cases} \mathtt{Foo1}(\mathtt{Nat})\,.\,\mathbf{b} \rightarrow \mathbf{c}{:}\mathtt{Bar}(\mathtt{Bool})\,.\,\mathbf{c} \rightarrow \mathbf{d}{:}\mathtt{Baz}(\mathtt{Bool})\,.\,\mathbf{end} \\ \vdots \\ \mathtt{Foo}N(\mathtt{Nat})\,.\,\mathbf{b} \rightarrow \mathbf{c}{:}\mathtt{Bar}(\mathtt{Bool})\,.\,\mathbf{c} \rightarrow \mathbf{d}{:}\mathtt{Baz}(\mathtt{Bool})\,.\,\mathbf{end} \end{cases}$$

In this example, each of the N branches has the same continuation, so the MLTS is a DAG. Only this one continuation needs to be explored by rule [⊢-Skip] to type-check implementations of roles c and d (instead of N continuations). If the continuations were different, in contrast, then these processes would need to be checked against each of those continuations. This is the same, though, with the classical approach and the "Less Is More" approach. However, when the MLTS arises from a global type **with parallel composition**, the size of the MLTS can be exponential (due to the interleavings of the branches), so type checking becomes more computationally costly. Most of the global types in the literature lack parallel composition, though.

In general, compared to model checking in the "Less Is More" approach, querying an MLTS in the synthetic approach comes with more control of any potential exponential growth, in the sense that it is syntactically confined (i.e., in our case, only parallel composition may trigger it, while in the "Less Is More" approach, it may happen during model checking at any point). So from the start, we are in a better position already. Moreover, given our requirement that the types have the diamond property and that parallel composition requires disjoint participants in the branches, we

[5] Morally, [⊢-Skip] could be removed and added to the rules for sending and receiving. This would reify the notion of applying all the structural rules first and then "skipping". We chose to separate "skipping" from rules [⊢-Send] and [⊢-Recv] since this leads to a simpler presentation.

also expect that we could use this information to cull the state space for performance (which would bring a form of de facto projection to this work).

A.3 Outlook: Beyond Finite MLTSs

We see several opportunities to support MLTSs with infinitely many states. In the most general case, we could admit behavioural specifications that generate such MLTSs and then generate additional proof obligations (e.g., the fact that certain states are reachable when applying [⊢-Skip]).

Furthermore, going from a well-behaved formalism, these proof obligations could be automated in the type checker. For instance, consider the use of pushdown automata (or a form of context-free global types) as protocol specifications with infinite state space, but for which reachability is still decidable. In that case, the type checker can perform the reachability tests for us—possibly using external tools that are optimised for this kind of analysis—so the proof obligations could be automatically discharged. Depending on the nature of the extension, the type system may remain decidable or become undecidable (but still sound).

We note that a similar approach can be adopted when the formalism is unknown to generate only well-behaved MLTSs (unlike with global types, for which Theorem 5.9 establishes that well-behavedness is guaranteed). In that case, it needs to be proved separately for each MLTS that it is well-behaved. The usage of external tools, such as model checkers, could be useful in this case, too. However, depending on additional restrictions that we might place on MLTSs, one may be able to internalise and automate the well-behavedness checking and avoid depending on an external tool.