Hybrid Multiparty Session Types

Compositionality for Protocol Specification through Endpoint Projection

LORENZO GHERI, University of Oxford, United Kingdom
NOBUKO YOSHIDA, University of Oxford, United Kingdom

Multiparty session types (MPST) are a specification and verification framework for distributed message-passing systems. The communication protocol of the system is specified as a *global type*, from which a collection of *local types* (local process implementations) is obtained by *endpoint projection*. A global type is a single disciplining entity for the whole system, specified by *one designer* that has full knowledge of the communication protocol. On the other hand, distributed systems are often described in terms of their *components*: a different designer is in charge of providing a subprotocol for each component. The problem of modular specification of global protocols has been addressed in the literature, but the state of the art focuses only on dual input/output compatibility. Our work overcomes this limitation. We propose the first MPST theory of *multiparty compositionality for distributed protocol specification* that is semantics-preserving, allows the composition of two or more components, and retains full MPST expressiveness. We introduce *hybrid types* for describing subprotocols interacting with each other, define a novel *compatibility relation*, explicitly describe an algorithm for composing multiple subprotocols into a *well-formed global type*, and prove that compositionality preserves projection, thus retaining semantic guarantees, such as liveness and deadlock freedom. Finally, we test our work against real-world case studies and we smoothly extend our novel compatibility to MPST with delegation and explicit connections.

CCS Concepts: • **Theory of computation → Distributed computing models**; **Type theory**.

Additional Key Words and Phrases: multiparty session types, compositionality, protocol design, concurrency

ACM Reference Format:
Lorenzo Gheri and Nobuko Yoshida. 2018. Hybrid Multiparty Session Types: Compositionality for Protocol Specification through Endpoint Projection. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 31 pages.

1 INTRODUCTION

With the current growth in scale and complexity of systems, their *design* has become of central importance for industry and society in general. Choreographies for interactions among multiple participants, or *(communication) protocols*, arise naturally in numerous fields: authorisation standards [Hardt 2012; MIT 2022], the BPMN graphical specification for business processes [OMG 2022], or smart contracts for financial transactions [Ethereum 2022].

The literature on programming languages offers a variety of formal frameworks for protocol description [Barbanera et al. 2020a; Honda et al. 2016; Montesi 2013], aimed at the verification of behavioural properties of distributed implementations that comply with the communication discipline prescribed by the protocol. Such theories focus on distributed implementations of participants, but rarely feature modularity in the design of protocols, which are instead seen as standalone, monolithic entities. Mostly, when modularity is considered, it is either conceived in terms of nesting [Demangeon and Honda 2012; Tabareau et al. 2014] or it substantially modifies protocol description, by adding additional structure [Carbone et al. 2018; Montesi and Yoshida 2013; Savanovic et al. 2020]. To the best of our knowledge, only in [Barbanera et al. 2021] and [Stolze et al. 2021] the result of composition is a well-formed protocol.

Authors' addresses: Lorenzo Gheri, University of Oxford, United Kingdom, lorenzo.gheri@cs.ox.ac.uk; Nobuko Yoshida, University of Oxford, United Kingdom, nobuko.yoshida@cs.ox.ac.uk.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

This paper presents hybrid multiparty session types: a novel, general theory that offers compositionality for distributed protocol specification, improves on the state of the art, and is immediately compatible with existing multiparty session types systems.

Multiparty session types (MPST) [Coppo et al. 2015; Honda et al. 2016; Yoshida and Gheri 2020] provide a typing discipline for message-passing concurrency, ensuring deadlock freedom for two or more distributed processes. A *global type* or *protocol*, which describes an entire interaction scenario, is projected into a collection of *local types* onto the respective participants (endpoint projection). MPST cater for the safe implementation of distributed processes: as long as the process for each participant is independently type-checked against its local type, its communication behaviour is disciplined by the semantics of the global type, and its execution does not get stuck.

Although alternatives to the top-down approach (i.e., endpoint projection from a global type) have been proposed [Deniélou and Yoshida 2013; Lange et al. 2015; Lange and Yoshida 2019; Scalas and Yoshida 2019], the benefits of an explicit, concise design of the communication protocol for the whole system have been recognised by the research community, since the first appearance of MPST [Honda et al. 2008], until more recent times, e.g., see [Cledou et al. 2022; Glabbeek et al. 2021]. Furthermore, the top-down approach has been extended, e.g., to fault tolerance [Viering et al. 2021], timed specification [Bocchi et al. 2014], refinements [Gheri et al. 2022; Zhou et al. 2020], cost awareness [Castro-Perez and Yoshida 2020], exception handling [Lagaillardie et al. 2022], or explicit connections and delegation [Castellani et al. 2020; Hu and Yoshida 2017].

Concretely, the underlying assumption to top-down MPST systems is that a single designer has *full knowledge* of the communication protocol and can give its formal specification in terms of a global type. Distributed systems, however, are designed modularly, by multiple designers. Recently, the literature has addressed the problem of obtaining a single coherent global type from independently specified subprotocols (components of a protocol) and some solutions have been offered: Barbanera et al. [2021] achieve direct composition of *two* global types, through a dual compatibility relation that matches inputs and outputs, based on gateways [Barbanera et al. 2018, 2019, 2020b]. Stolze et al. [2021] describe a dual methodology beyond gateways, but severely restrict the syntax for global types. In contrast to this approach, our theory substitutes dual compatibility, based only on input/output matching, with the notion of *compatibility through projection*. Thus, we improve on the state of the art: (1) we can compose more than two subprotocols into a well-formed global type and (2) we retain the full expressiveness of MPST (including recursion and parallel composition). See §6 for a broader, in-detail discussion. Moreover, metathoretical results about the semantics of traditional MPST systems [Deniélou and Yoshida 2013; Honda et al. 2016] immediately translate to ours (*semantics preservation*): from distributed specifications in terms of subprotocols, our theory synthesises a global protocol for the whole system; we prove once and for all, as a metatheoretical result, that such global protocol is a traditionally well-formed global type.

Contributions. This paper develops *a theory of compositionality for distributed protocol description in MPST systems* and introduces the following novel MPST concepts:

- *hybrid types*, a generalisation of both global and local types, for the specification of communicating subprotocols (Definition 3.3);
- *generalised projection* onto sets of roles (Definition 3.6), which well-behaves with respect to set inclusion (Theorem 4.9);
- *localiser* (Definition 3.8), a novel operator that isolates, in a subprotocol, the inter-component communication from the intra-component one;
- *compatibility* based on projection and localiser (Equation **C**, §4.1);
- *build-back*, an explicit algorithm to compose *two or more* subprotocols into a more general one (Definitions 4.1 and 4.6 and Theorems 4.4 and 4.7).

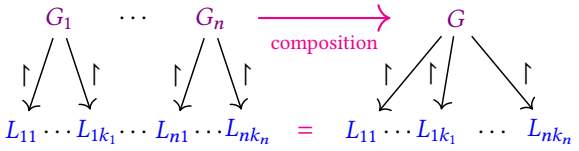


Fig. 1. Distributed Protocol Specification, Naively

To the best of our knowledge, our approach is the first that:

- enables the correct composition of *two or more* subprotocols into a global type, while capturing *full MPST expressiveness*: branching, parallel composition, and recursion (Corollary 4.10);
- operates at a purely syntactic level, thus retaining previously developed MPST semantics results (*semantics preservation*); correctness is guaranteed by compositionality resulting in a traditionally well-formed global type and preserving endpoint projection (Corollary 4.10);
- provides a notion of compatibility that is *more expressive than dual input/output matching* and hence suitable for extension to more sophisticated MPST systems (Example 5.7).

We discuss the applicability and generality of our work, through *case studies*. (1) We give a distributed specification of the real-world protocol OAuth 2.0 [Hardt 2012], which showcases modularity features of our theory (§5.2) and leads to an optimisation (§5.3, Corollary 5.1). (2) We extend our theory beyond traditional MPST, to delegation and explicit connections (§5.4).

Outline. §2 gives an overview of our development, with a simple, but realistic, application scenario. §3 and §4 are dedicated to our technical contributions. §5 tests the strengths of our theory with case studies. §6 discusses in detail, with examples, related work. §8 concludes with future work. Further detail for definitions and proofs can be found in Appendix B of [Gheri and Yoshida 2023].

2 OVERVIEW OF OUR DEVELOPMENT

This work achieves *distributed protocol specification* for MPST: different *(protocol) designers* specify protocols (naively as global types, Figure 1) for different components of the communicating system; then, these compose into a single global type for the whole system. Composition must *preserve (endpoint) projection* (indicated with ↾): local types, for the distributed implementation of *roles* (or *participants*), need to be obtained by projection of each separate component, but, also, they need to be projections of the same global type (obtained by composition), if we want semantic guarantees (e.g., deadlock freedom) to hold. In other words, our protocol-compositionality theory relies on multiparty compatibility, guaranteed by a well-formed global type, and on semantics proofs from previous work (e.g., [Deniélou and Yoshida 2013]). This approach makes our development *semantics-preserving*: it endows existing MPST systems with distributed protocol specification.

Traditionally, a global type is a "closed" standalone entity that describes a one-component communication protocol: all interactions among participants are *internal* to such component. We consider instead the distributed specification of a system, in terms of multiple components (disjoint sets of participants). Each participant can send both internal messages, within its component, or *external*, to other components. Therefore, we "open" the syntax of global types, so that it allows not only for intra-component communication, but also for inter-component communication. By extending the syntax of global types with an interface for inter-component communication, we obtain *hybrid types*. The communication protocol of each component of the system is specified as a hybrid type; multiple components can be composed into a well-formed global type thanks to a novel notion of *compatibility, based on projection.*

$$G_{\mathsf{str}} = \mathtt{d} \to \mathtt{ad} : prod(\mathsf{nat}).\mathtt{d} \to \mathtt{ad} : \{ok.\mathsf{end}, stop.\mathsf{end}\}$$
$$G_{\mathsf{sales}} = \mathtt{s} \to \mathtt{w} : \{publish.\mathsf{end}, stop.\mathsf{end}\}$$
$$G_{\mathsf{fin}} = \mathtt{f}_1 \to \mathtt{f}_2 : prod(\mathsf{nat}).\mathtt{f}_2 \to \mathtt{f}_1 : \{price(\mathsf{nat}).\mathsf{end}, stop.\mathsf{end}\}$$

(a) Global Types for Internal Communication

$$H'_{\mathsf{str}} = \mathtt{d} \to \mathtt{ad} : prod(\mathsf{nat}).\ \mathtt{d}!\mathtt{s};prod(\mathsf{nat})\ .\ \mathtt{d}!\mathtt{f}_1;prod(\mathsf{nat})\ .\mathtt{d} \to \mathtt{ad} : \{ok.\mathsf{end}, stop.\mathsf{end}\}$$
$$H'_{\mathsf{sales}} = \ \mathtt{d}?\mathtt{s};prod(\mathsf{nat})\ .\mathtt{s} \to \mathtt{w} : \{publish.\mathsf{end}, stop.\mathsf{end}\}$$
$$H'_{\mathsf{fin}} = \ \mathtt{d}?\mathtt{f}_1;prod(\mathsf{nat})\ .\mathtt{f}_1 \to \mathtt{f}_2 : prod(\mathsf{nat}).\mathtt{f}_2 \to \mathtt{f}_1 : \{price(\mathsf{nat}).\mathsf{end}, stop.\mathsf{end}\}$$

(b) Hybrid Types for Basic Inter-Department Interactions

$$H_{\mathsf{str}} = \mathtt{d} \to \mathtt{ad} : prod(\mathsf{nat}).\mathtt{d}!\mathtt{s};prod(\mathsf{nat}).\mathtt{d}!\mathtt{f}_1;prod(\mathsf{nat}).\mu X.\mathtt{f}_1?\mathtt{d};\{ok.\mathtt{d} \to \mathtt{ad} : go.\mathsf{end}, wait.\mathtt{d} \to \mathtt{ad} : wait.X\}$$
$$H_{\mathsf{sales}} = \mathtt{d}?\mathtt{s};prod(\mathsf{nat}).\mu X.\mathtt{f}_1?\mathtt{s};\{price(\mathsf{nat}).\mathtt{s} \to \mathtt{w} : publish.\mathsf{end}, wait.\mathtt{s} \to \mathtt{w} : wait.X\}$$
$$H_{\mathsf{fin}} = \mathtt{d}?\mathtt{f}_1;prod(\mathsf{nat}).\mathtt{f}_1 \to \mathtt{f}_2 : prod(\mathsf{nat}).\mu X.\mathtt{f}_2 \to \mathtt{f}_1 : \left\{ \begin{array}{l} price(\mathsf{nat}).\mathtt{f}_1!\mathtt{d};ok.\mathtt{f}_1!\mathtt{s};price(\mathsf{nat}).\mathsf{end}, \\ wait.\mathtt{f}_1!\mathtt{d};wait.\mathtt{f}_1!\mathtt{s};wait.X \end{array} \right\}$$

(c) More Expressive Hybrid Types

Fig. 2. Types for Interactions in the Company

In what follows: we consider a three-component system: a company with three departments, for each of which, a different *(protocol) designer* is in charge of describing the communication protocol. The departments, with respective (internal) roles, are the following: (a) the *strategy team*, the roles of which are the director **d** of the company and the advertisement team **ad**; (b) the *sales department*, with a salesman **s** and the website administrator **w**; (c) the *finance department*, with two employees, $\mathtt{f}_1$ and $\mathtt{f}_2$. We assume that internal roles of different components are *distinct*.

Global Types for Intra-Component Communication. When no inter-component communication happens, each protocol designer gives a global type for the *internal* communication of their department (Figure 2a). In G_{str}, the global type for the strategy department, the director **d** sends the product ID to the responsible for advertisement **ad**; then, **d** gives an *ok* or asks **ad** to *stop*. For the sales department (G_{sales}) **s** decides whether **w** can publish some content on the company website. In the financial department (G_{fin}), $\mathtt{f}_1$ sends the product ID to $\mathtt{f}_2$ and gets back either a *price* or a *stop*.

Hybrid Types for Inter-Component Interactions. The components of a distributed system are expected to communicate with each other. Therefore, we introduce a *hybrid syntax of global and local constructs* (and we call *hybrid types* the terms of this syntax): to the global-type syntax (e.g., $\mathtt{d} \to \mathtt{ad}$), for intra-component communication, we add local send and receive constructs (e.g., $\mathtt{d}!\mathtt{s}$ and $\mathtt{d}?\mathtt{f}_1$), as the interface for inter-component communication. In our example, a first message is sent by **d**, with a product ID *prod*, externally, to the other two departments (Figure 2b): $\mathtt{d}!\mathtt{s}$ and $\mathtt{d}!\mathtt{f}_1$. These are *dually* received by the sales team $\mathtt{d}?\mathtt{s}$ and by the finance team $\mathtt{d}?\mathtt{f}_1$, respectively (as highlighted in Figure 2b).

Remark 2.1 (Generalising Global and Local Types). We observe that hybrid types are a generalisation of both global and local types. A global type is a "closed" hybrid type, where only internal messages are exchanged. The intuition for local types is more subtle: a local type can be interpreted as a basic, *one-participant component* of a communicating system, which communicates only externally, with participants of other components. E.g., the local type $?\mathtt{p};\ell_1.!\mathtt{r};\ell_2.\mathsf{end}$, for the participant **q**, can be written as the hybrid type $\mathtt{p}?\mathtt{q};\ell_1.\mathtt{q}!\mathtt{r};\ell_2.\mathsf{end}$: **q** is the only internal participant that first receives from **p** and then sends to **r** (both **p** and **r** are of other components). Being able to express

global and local types as hybrid types is fundamental: it makes our results correct and compatible with existing MPST theories (see Remark 2.2 below, and Corollary 4.10 in §4).

Expressiveness and Compatibility. We describe a more expressive version of the protocols (Figure 2c) that combines inter-component messages with branching and recursion. Figure 3 shows the communication for each component of the system, as described by the protocol designer of each department. We imagine that the price of the product *prod* is decided within the finance department: the finance expert $\mathtt{f}_2$ either gives a *price* or asks all processes to *wait* in a recursive loop; then, the decision is communicated to the other departments. Figure 3c shows the execution of the protocol for the finance department, where $\mathtt{f}_2$ makes such choice. Figure 2c shows the formal specifications, as hybrid types, of the three protocols. We observe that, to compose H_{str}, H_{sales}, and H_{fin} (and, in general, to compose, more than two communicating protocols), dual relations are not sufficient for compatibility (for a broader discussion see §6). Our proposal is to give separately the specification of a *communication discipline for inter-component interactions only*: intra-component interactions are left to the designer of each respective component and some *chief designer* gives the description of one more protocol, for global guidance of inter-component communication. For our example, we collect all the interactions between any two different departments in the protocol in Figure 4a, and we formalise it with a *compatibility (global) type*:

$$G^\dagger := \mathtt{d} \to \mathtt{s} : \mathit{prod}(\mathsf{nat}).\mathtt{d} \to \mathtt{f}_1 : \mathit{prod}(\mathsf{nat}).\mu X.\mathtt{f}_1 \to \mathtt{d} : \left\{ \begin{array}{l} \mathit{ok}.\mathtt{f}_1 \to \mathtt{s} : \mathit{price}(\mathsf{nat}).\mathsf{end}, \\ \mathit{wait}.\mathtt{f}_1 \to \mathtt{s} : \mathit{wait}.X \end{array} \right\}$$

Compatibility of subprotocols H_{str}, H_{sales}, and H_{fin}, with $G^\dagger$, is achieved by asking that the *(generalised) projection* $\upharpoonright$, of $G^\dagger$, with respect to the internal participants of each subprotocol, is equal to the *localisation* loc, of that subprotocol, where "localising a protocol" means isolating its inter-component communication (by retaining only its *local* constructs). E.g., we consider H_{str} and its internal participants $\{\mathtt{d}, \mathtt{ad}\}$.

$$G^\dagger \upharpoonright_{\{\mathtt{d},\mathtt{ad}\}} = \mathsf{loc}\ H_{\mathsf{str}} = \mathtt{d}!\mathtt{s};\mathit{prod}(\mathsf{nat}).\mathtt{d}!\mathtt{f}_1;\mathit{prod}(\mathsf{nat}).\mu X.\mathtt{f}_1?\mathtt{d};\{\mathit{ok}.\mathsf{end}, \mathit{wait}.X\}$$

Analogously we require that $G^\dagger \upharpoonright_{\{\mathtt{s},\mathtt{w}\}} = \mathsf{loc}\ H_{\mathsf{sales}}$ and $G^\dagger \upharpoonright_{\{\mathtt{f}_1,\mathtt{f}_2\}} = \mathsf{loc}\ H_{\mathsf{str}}$.

We observe that not only we have enriched the syntax of global types with local constructs to get hybrid types, but also we have *generalised projection* to *sets* of participants, introduced a new operator (*localiser*) to isolate external communication, and, based on these, defined compatibility.

Compositionality and Correctness. Our theory (§3 and §4) provides an explicit function $\mathcal{B}$ that *builds back* a single global type for the communication in the company, from the distributed specification above: $G = \mathcal{B}\ (G^\dagger)\ ([H_{\mathsf{str}}, H_{\mathsf{sales}}, H_{\mathsf{fin}}])$. It holds that $G\upharpoonright_{\mathtt{d}} = H_{\mathsf{str}}\upharpoonright_{\mathtt{d}}$, $G\upharpoonright_{\mathtt{f}_1} = H_{\mathsf{fin}}\upharpoonright_{\mathtt{f}_1}$, and analogously for all participants: *projection is preserved*. A figure, representing such G for our example, can be found in Appendix A of [Gheri and Yoshida 2023].

More generally (see Figure 4b), from a compatibility type $G^\dagger$ and hybrid types H_i for each component (with set of internal participants E_i), such that they are compatible $G^\dagger \upharpoonright_{E_i} = \mathsf{loc}\ H_i$, our theory synthesises a global type $G = \mathcal{B}\ (G^\dagger)\ ([H_1, \ldots, H_n])$ (Definition 4.6 and Theorem 4.7). Correctness of our theory is given by Corollary 4.10. Formally, this result guarantees that *the local types, projections of G on each participant, are the same as if obtained by the respective subprotocol H_i*: $G\upharpoonright_{\mathtt{p}} = H_i\upharpoonright_{\mathtt{p}}$ if $\mathtt{p}$ is a participant of the i-th subprotocol H_i. We have achieved *distributed protocol specification*: we can both obtain local types for implementation in a distributed fashion, by projection of the respective component H_i (no designer or programmer needs the full knowledge of G), and rest assured that all local types (for all participants, in all components) are projections of a single, well-formed global type. This makes our development compatible with existing MPST theories, with no need for developing new semantics (*semantics preservation*): a well-formed global type projecting on all participants gives traditional *multiparty compatibility*,

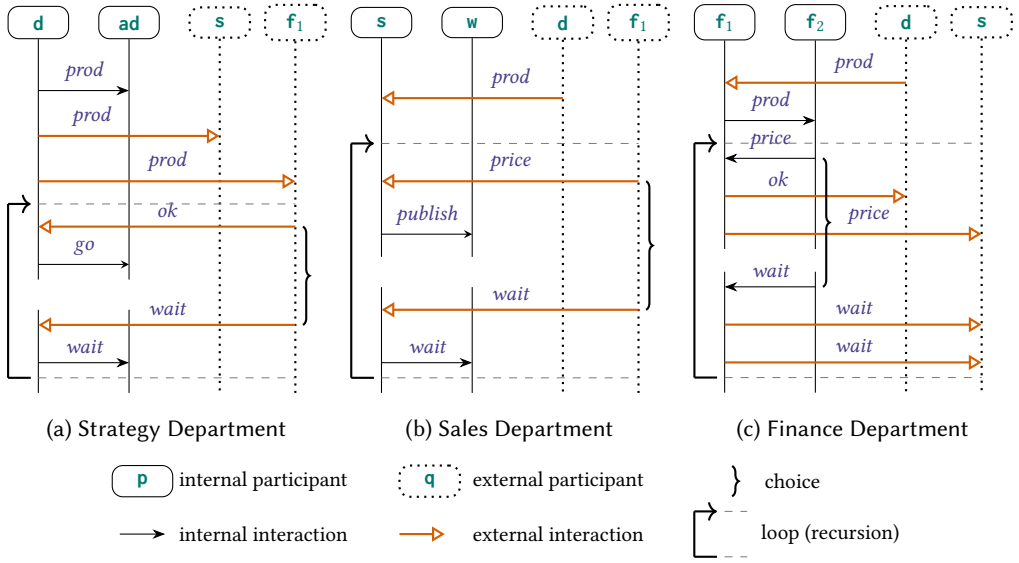


Fig. 3. Communication for Each Department in the Company

which, thanks to the semantics results in the literature [Coppo et al. 2015; Deniélou and Yoshida 2013; Honda et al. 2016], leads to guarantees, such as liveness and deadlock freedom.

Remark 2.2 (Hybrid Types and Generalised Projection). With reference to Figure 4b, Let us consider the set of participants E_i of the generic H_i, and $\mathtt{p} \in E_i$. Generalised projection takes a hybrid type and returns a hybrid type; since *global and local types are hybrid types* (Remark 2.1), e.g., we can project $G^\dagger$ onto E_i for compatibility ($G^\dagger \upharpoonright_{E_i} = \mathrm{loc}\ H_i$), or G onto E_i and verify that it is equal to H_i (see Theorem 4.7, §4). Most importantly, Theorem 4.9 in §4 guarantees that projection composes over set inclusion $G\upharpoonright_{\mathtt{p}} = (G\upharpoonright_{E_i})\upharpoonright_{\mathtt{p}} = H_i\upharpoonright_{\mathtt{p}}$: by projecting H_i and G onto the participant $\mathtt{p}$, we obtain the same local type for $\mathtt{p}$. Namely, we can obtain local types from the specific component H_i and then *implement them in a distributed fashion*, but also they all are projections of a well-formed global type G.

To summarise (see Figure 4b), our proposal for distributed protocol specification is the following:

(1) a different designer specifies, for each component of the system, a hybrid type H_i;
(2) a chief designer gives the compatibility type $G^\dagger$ to discipline inter-component interactions;
(3) compatibility is a simple equality check: $G^\dagger \upharpoonright_{E_i} = \mathrm{loc}\ H_i$ (E_i is the set of participants for H_i).

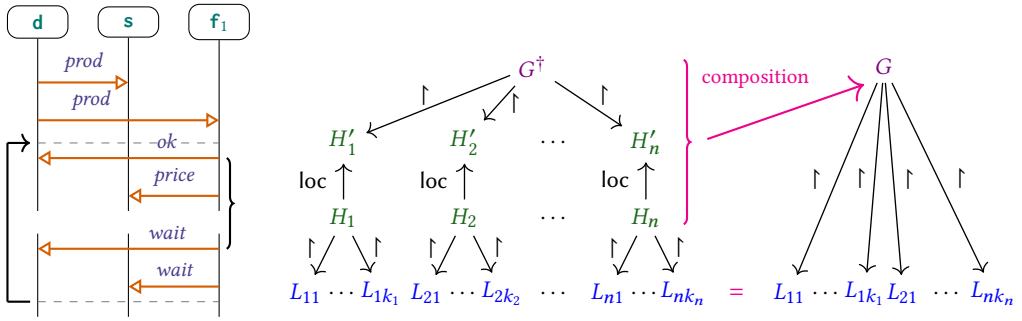
In return, as a metatheoretical result proved once and for all by our theory, the designers obtain $\mathcal{B}\ (G^\dagger)\ ([H_1, \ldots, H_n])$, *a global type for the whole communication*, for which *projections are preserved* (and, hence, MPST semantic guarantees hold).

In §3 and §4 we detail our compositionality theory, including generalised projection, localiser, compatibility, build-back, and correctness results.

3 HYBRID TYPES FOR PROTOCOL SPECIFICATION

3.1 Background: Preliminaries of Multiparty Session Types

We give a short summary of *multiparty session types* [Coppo et al. 2015; Honda et al. 2016; Scalas et al. 2019; Yoshida and Gheri 2020]; specifically, our theory is based on the formulation in [Deniélou



(a) Inter-Component Communication

(b) Distributed Protocol Specification, Compatibility through Projection

Fig. 4. Inter-Component Communication and Compatibility via Projection

and Yoshida 2013], extended to parallel composition of global types. The notation for our MPST system is standard (directly adapted from [Castro-Perez et al. 2021]).

Atoms of our syntax are: a set of *roles* (or *participants*), ranged over by $\mathtt{p}, \mathtt{q}, \ldots$, a set of *(type) variables*, ranged over by $X, Y, \ldots$; and a set of labels, ranged over by $\ell_0, \ell_1, \ldots, \ell_i, \ell_j, \ldots$.

Definition 3.1 (Sorts, Global Types, and Local Types). *Sorts*, *global types*, and *local types*, ranged over by $\mathtt{S}$, G, and L respectively, are inductive datatypes generated by:

$$\mathtt{S} ::= \mathtt{unit} \mid \mathtt{nat} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{S{+}S} \mid \mathtt{S{*}S} \qquad G ::= \mathtt{end} \mid X \mid \mu X.G \mid \mathtt{p} \to \mathtt{q} : \{\ell_i(\mathtt{S}_i).G_i\}_{i\in I} \mid G_1 \mid G_2$$
$$L ::= \mathtt{end} \mid X \mid \mu X.L \mid \mathtt{!q};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I} \mid \mathtt{?p};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I}$$

where, $\mathtt{p} \neq \mathtt{q}$, $I \neq \emptyset$, and $\ell_i \neq \ell_j$ when $i \neq j$, for all $i, j \in I$, in $\mathtt{p} \to \mathtt{q} : \{\ell_i(\mathtt{S}_i).G_i\}_{i\in I}$, $\mathtt{!q};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I}$, and $\mathtt{?p};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I}$.

The *global message* $\mathtt{p} \to \mathtt{q} : \{\ell_i(\mathtt{S}_i).G_i\}_{i\in I}$ describes a protocol where participant $\mathtt{p}$ sends to $\mathtt{q}$ one message with label ℓ_i and a value of sort $\mathtt{S}_i$ as payload, for some $i \in I$; then, depending on which ℓ_i was sent by $\mathtt{p}$, the protocol continues as G_i. The type $\mathtt{end}$ represents a *terminated protocol*. *Recursive protocol* is modelled as $\mu X.G$, where recursion *variable* X is bound. The *parallel* construct $G_1 \mid G_2$ describes a protocol composed by two independent ones. The participants of G_1 and G_2 are required to be disjoint: no communication happens between G_1 and G_2, but only internally in each one of them (for a broader discussion, see §5.1). The intuition for local types $\mathtt{end}$, X and $\mu X.L$ is the same as for global types. The *send type* $\mathtt{!q};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I}$ says that the participant implementing the type must choose a labelled message to send to $\mathtt{q}$; if the participant chooses the label ℓ_i, it must include in the message to $\mathtt{q}$ a payload value of sort $\mathtt{S}_i$, and continue as prescribed by L_i. The *receive type* $\mathtt{?p};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I}$ requires to wait to receive a value of sort $\mathtt{S}_i$ (for some $i \in I$) from the participant $\mathtt{p}$, via a message with label ℓ_i; then the process continues as prescribed by L_i.

We are interested in types that are (1) *guarded*—e.g., $\mu X.\mathtt{p} \to \mathtt{q} : \ell(\mathtt{nat}).X$ is a valid global type, whereas $\mu X.X$ is not—(detail in Appendix B.1 of [Gheri and Yoshida 2023], Definition B.1) and (2) *closed*, i.e., all variables are bound by μX. In messages, sends and receives, the payload type can be omitted (e.g., $\mathtt{p} \to \mathtt{q} : \ell.\ldots$), when only a label is exchanged. We assume that global and local types are always guarded, and that, in $G_1 \mid G_2$, G_1 and G_2 are closed.

Projection plays a central role in MPST theories: it connects the protocol discipline, provided by the global type, with the local types that separately describe the behaviour of each participant.

Definition 3.2 (Projection for Global Types). The *projection of a global type onto a role* $\mathtt{r}$ is a partial function defined by recursion on G, whenever the recursive call is defined:

$$
\begin{array}{ll}
\text{[PROJ-END]} & \mathsf{end}\upharpoonright\mathtt{r} = \mathsf{end} \qquad \text{[PROJ-VAR]} \quad X\upharpoonright\mathtt{r} = X \\
\text{[PROJ-REC]} & (\mu X.G)\upharpoonright\mathtt{r} = \mu X.(G\upharpoonright\mathtt{r}) \text{ if } \mu X.(G\upharpoonright\mathtt{r}) \text{ is guarded, else, if } (\mu X.G) \text{ is closed, } (\mu X.G)\upharpoonright\mathtt{r} = \mathsf{end} \\
\text{[PROJ-PAR]} & G_1 \mid G_2\upharpoonright\mathtt{r} = G_i\upharpoonright\mathtt{r} \text{ if } \mathtt{r} \text{ is a participant of } G_i, \mathsf{end} \text{ otherwise} \\
\text{[PROJ-SEND]} & \mathtt{p}\to\mathtt{q}:\{\ell_i(\mathtt{S}_i).G_i\}_{i\in I}\upharpoonright\mathtt{r} = \mathtt{!q};\{\ell_i(\mathtt{S}_i).(G_i\upharpoonright\mathtt{r})\}_{i\in I} \text{ if } \mathtt{r}=\mathtt{p} \\
\text{[PROJ-RECV]} & \mathtt{p}\to\mathtt{q}:\{\ell_i(\mathtt{S}_i).G_i\}_{i\in I}\upharpoonright\mathtt{r} = \mathtt{?p};\{\ell_i(\mathtt{S}_i).(G_i\upharpoonright\mathtt{r})\}_{i\in I} \text{ if } \mathtt{r}=\mathtt{q} \\
\text{[PROJ-MERGE]} & \mathtt{p}\to\mathtt{q}:\{\ell_i(\mathtt{S}_i).G_i\}_{i\in I}\upharpoonright\mathtt{r} = \bigsqcup_{i\in I}(G_i\upharpoonright\mathtt{r}) \text{ if } \mathtt{r}\neq\mathtt{p} \text{ and } \mathtt{r}\neq\mathtt{q}
\end{array}
$$

$G\upharpoonright\mathtt{r}$ is undefined if none of the above applies. *Merging* ($\sqcup$) is defined as a partial operator over two local types such that: $L\sqcup L = L$ for every type, it delves inductively inside all constructors (e.g., $\mathtt{!q};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I}\sqcup\mathtt{!q};\{\ell_i(\mathtt{S}_i).L'_i\}_{i\in I} = \mathtt{!q};\{\ell_i(\mathtt{S}_i).(L_i\sqcup L'_i)\}_{i\in I}$), and $\mathtt{?p};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I}\sqcup\mathtt{?p};\{\ell_j(\mathtt{S}_j).L'_j\}_{j\in J} = \mathtt{?p};\{\ell_k(\mathtt{S}_k).L_k\sqcup L'_k\}_{k\in I\cap J}\cup\{\ell_k(\mathtt{S}_k).L_k\}_{k\in I\setminus J}\cup\{\ell_k(\mathtt{S}_k).L'_k\}_{k\in J\setminus I}$.

We describe the clauses of Definition 3.2. [PROJ-END], [PROJ-VAR], and [PROJ-REC] are standard. [PROJ-SEND] (resp. [PROJ-RECV]) states that a global type starting with a message from $\mathtt{r}$ to $\mathtt{q}$ (resp. from $\mathtt{p}$ to $\mathtt{r}$) projects onto a sending (resp. receiving) local type $\mathtt{!q};\{\ell_i(\mathtt{S}_i).G_i\upharpoonright\mathtt{r}\}$ (resp. $\mathtt{?p};\{\ell_i(\mathtt{S}_i).G_i\upharpoonright\mathtt{r}\}$), provided that the continuations $G_i\upharpoonright\mathtt{r}$ are also projections of the corresponding global-type continuations G_i. [PROJ-MERGE] states that, if the projected global type starts with an interaction between $\mathtt{p}$ and $\mathtt{q}$, and if we are projecting it onto a third participant $\mathtt{r}$, then the projection is defined (and we can skip the message $\mathtt{p}\to\mathtt{q}$:) if all the continuations project onto *mergeable* types (according to the merge operator $\sqcup$ defined above). [PROJ-PAR] states that projecting a parallel type $G_1 \mid G_2$ on $\mathtt{r}$ is the same as projecting G_1 or G_2 onto $\mathtt{r}$, depending on whether $\mathtt{r}$ is a participant of one or the other type.

By projecting a global type G onto all participants ($G\upharpoonright_{\mathtt{r}} = L_{\mathtt{r}}$, for the generic role $\mathtt{r}$), we obtain a collection of local types $\{L_{\mathtt{r}}\}$, where $L_{\mathtt{r}}$ is the behavioural type for $\mathtt{r}$. Existing MPST theories (e.g., [Deniélou and Yoshida 2013]) guarantee that a session of well-typed implementations of the participants inherits semantic guarantees for its communication, from G. Our development in §4 is compatible with such theories: result of composition is a well-formed G, thus, implementations of projected local types benefit from well-established semantics results from the literature.

3.2 Hybrid Types

To allow the specification of interacting subprotocols, we enrich the syntax of global types with local constructs. We thus obtain "hybrid" types, which use global messages for intra-protocol communication and local sends and receives as openings for inter-protocol communication.

Definition 3.3 (Hybrid Types). *Hybrid types* are defined inductively by:

$$H \quad ::= \quad \mathsf{end} \mid X \mid \mu X.H \mid H_1|H_2 \mid \mathtt{p!q};\{\ell_i(\mathtt{S}_i).H_i\}_{i\in I} \mid \mathtt{p?q};\{\ell_i(\mathtt{S}_i).H_i\}_{i\in I} \mid \mathtt{p}\to\mathtt{q}:\{\ell_i(S_i).H_i\}_{i\in I}$$

where, in $\mathtt{p}\to\mathtt{q}:\{\ell_i(\mathtt{S}_i).H_i\}_{i\in I}$, $\mathtt{p!q};\{\ell_i(\mathtt{S}_i).H_i\}_{i\in I}$, and $\mathtt{p?q};\{\ell_i(\mathtt{S}_i).H_i\}_{i\in I}$, $\mathtt{p}\neq\mathtt{q}$, $I\neq\emptyset$, and $\ell_i\neq\ell_j$ when $i\neq j$, for all $i,j\in I$. We indicate the datatype of hybrid types with the notation $\mathcal{H}$.

The intuition behind each construct is the same as in Definition 3.1, but we write $\mathtt{p!q};\{\ell_i(\mathtt{S}_i).H_i\}_{i\in I}$ in place of $\mathtt{!q};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I}$, and $\mathtt{p?q};\{\ell_i(\mathtt{S}_i).H_i\}_{i\in I}$ in place of $\mathtt{?p};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I}$. For local types, $\mathtt{!q};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I}$ (resp. $\mathtt{?p};\{\ell_i(\mathtt{S}_i).L_i\}_{i\in I}$) describes the communication of the participant $\mathtt{p}$ sending a message to $\mathtt{q}$ (resp. $\mathtt{q}$ receiving a message from $\mathtt{p}$), and then continuing with interactions *all involving* $\mathtt{p}$ *(resp.* $\mathtt{q}$*) as a subject.* Therefore, such subject can be left implicit. For hybrid types, instead, different (internal) subjects interact both with internal and external participants. For instance:

$$^{(a)}\mathtt{p!q};\ell_1.\ \ ^{(b)}\mathtt{p}\to\mathtt{r}:\ell_2.\ \ ^{(c)}\mathtt{q?r};\ell_3.\ \ ^{(d)}\mathsf{end}$$

(a) $\mathtt{p}$ sends an external message to $\mathtt{q}$; (b) $\mathtt{p}$ exchanges a message internally with $\mathtt{r}$; (c) $\mathtt{r}$ receives an external message from $\mathtt{q}$; and (d) the protocol terminates.

Definition 3.4 (Internal and External Participants). We define the sets of *internal participants* and *external participants* of a hybrid type H by recursion:

$$\begin{array}{ll}
\mathsf{ipart}(\mathsf{end}) = \emptyset \quad \mathsf{ipart}(X) = \emptyset & \mathsf{epart}(\mathsf{end}) = \emptyset \quad \mathsf{epart}(X) = \emptyset \\
\mathsf{ipart}(\mu X.H) = \mathsf{ipart}(H) & \mathsf{epart}(\mu X.H) = \mathsf{epart}(H) \\
\mathsf{ipart}(H_1|H_2) = \mathsf{ipart}(H_1) \cup \mathsf{ipart}(H_2) & \mathsf{epart}(H_1|H_2) = \mathsf{epart}(H_1) \cup \mathsf{epart}(H_2) \\
\mathsf{ipart}(\mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}) = \{\mathtt{p}\} \cup \bigcup_{i\in I} \mathsf{ipart}(H_i) & \mathsf{epart}(\mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}) = \{\mathtt{q}\} \cup \bigcup_{i\in I} \mathsf{epart}(H_i) \\
\mathsf{ipart}(\mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}) = \{\mathtt{q}\} \cup \bigcup_{i\in I} \mathsf{ipart}(H_i) & \mathsf{epart}(\mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}) = \{\mathtt{p}\} \cup \bigcup_{i\in I} \mathsf{epart}(H_i) \\
\mathsf{ipart}(\mathtt{p} \to \mathtt{q} : \{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}) = \{\mathtt{p},\mathtt{q}\} \cup \bigcup_{i\in I} \mathsf{ipart}(H_i) & \mathsf{epart}(\mathtt{p} \to \mathtt{q} : \{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}) = \bigcup_{i\in I} \mathsf{epart}(H_i)
\end{array}$$

We define, for hybrid types, *guardedness* and *closedness*, as for global and local types (Definition 3.1). We require that all hybrid types in this paper are guarded (detail in Appendix B.1 of [Gheri and Yoshida 2023], Definition B.1) and that, for all H, $\mathsf{ipart}(H) \cap \mathsf{epart}(H) = \emptyset$. Also, we require well-formedness for parallel constructs: for $H_1|H_2$, H_1 and H_2 must be closed and $(\mathsf{ipart}(H_1) \cup \mathsf{epart}(H_1)) \cap (\mathsf{ipart}(H_2) \cup \mathsf{epart}(H_2)) = \emptyset$. Namely, the parallel construct describes communication that happens independently, within two separate groups of participants. We express *global and local types in terms of hybrid types*, with two predicates on $\mathcal{H}$: $\mathit{isGlobal}(H)$ holds iff H is formed only by global constructs (global type syntax in Definition 3.1); and $\mathit{isLocal}(H)$ holds iff $\mathsf{ipart}(H)$ contains at most one element (hence H contains only local constructs, see local types in Definition 3.1).

Example 3.5. We use as a recurring example the company from §2. A designer, D_{str}, describes the protocol H_{str} for the strategic team, as in Figure 2c:

$$H_{\mathsf{str}} := \mathtt{d} \to \mathtt{ad} : \mathit{prod}(\mathsf{nat}).\mathtt{d}!\mathtt{s};\mathit{prod}(\mathsf{nat}).\mathtt{d}!\mathtt{f}_1;\mathit{prod}(\mathsf{nat}).\mu X.\mathtt{f}_1?\mathtt{d};\left\{\begin{array}{l} \mathit{ok}.\mathtt{d} \to \mathtt{ad} : \mathit{go}.\mathsf{end}, \\ \mathit{wait}.\mathtt{d} \to \mathtt{ad} : \mathit{wait}.X \end{array}\right\}$$

First, $\mathtt{d}$ sends internally a product ID to $\mathtt{ad}$, then a similar external message to $\mathtt{s}$, of the sales department, and to $\mathtt{f}_1$, of the finance department. $\mathtt{d}$ waits in a recursive loop for $\mathtt{f}_1$ to give the *ok*. When this happens, $\mathtt{d}$ internally communicates to $\mathtt{ad}$ that they can proceed with the product advertisement. For H_{str}, the sets of internal and external participants are $\mathsf{ipart}(H_{\mathsf{str}}) = \{\mathtt{d}, \mathtt{ad}\}$ and $\mathsf{epart}(H_{\mathsf{str}}) = \{\mathtt{s}, \mathtt{f}_1\}$. We observe that D_{str} is not concerned with the communication that happens internally to the sales department or the financial one, nor with the communication between these two. Designers D_{sales} and D_{fin} independently give protocols for the sales and financial departments respectively (as in Figure 2c):

$$H_{\mathsf{sales}} = \mathtt{d}?\mathtt{s};\mathit{prod}(\mathsf{nat}).\mu X.\mathtt{f}_1?\mathtt{s};\left\{\begin{array}{l} \mathit{price}(\mathsf{nat}).\mathtt{s} \to \mathtt{w} : \mathit{publish}(\mathsf{nat}).\mathsf{end}, \\ \mathit{wait}.\mathtt{s} \to \mathtt{w} : \mathit{wait}.X \end{array}\right\}$$

$$H_{\mathsf{fin}} = \mathtt{d}?\mathtt{f}_1;\mathit{prod}(\mathsf{nat}).\mathtt{f}_1 \to \mathtt{f}_2 : \mathit{prod}(\mathsf{nat}).\mu X.\mathtt{f}_2 \to \mathtt{f}_1 : \left\{\begin{array}{l} \mathit{price}(\mathsf{nat}).\mathtt{f}_1!\mathtt{d};\mathit{ok}.\mathtt{f}_1!\mathtt{s};\mathit{price}(\mathsf{nat}).\mathsf{end}, \\ \mathit{wait}.\mathtt{f}_1!\mathtt{d};\mathit{wait}.\mathtt{f}_1!\mathtt{s};\mathit{wait}.X \end{array}\right\}$$

In the sales department, once $\mathtt{d}$ has communicated the product, $\mathtt{s}$ waits in a loop for the decision about the price from the financial department, then gives to the website administrator $\mathtt{w}$ the command to publish. We have that $\mathsf{ipart}(H_{\mathsf{sales}}) = \{\mathtt{s}, \mathtt{w}\}$ and $\mathsf{epart}(H_{\mathsf{sales}}) = \{\mathtt{d}, \mathtt{f}_1\}$. The decision about the price of the product is taken by $\mathtt{f}_2$, and communicated internally to the financial department with $\mathtt{f}_2 \to \mathtt{f}_1 : \ldots$; then $\mathtt{f}_1$ communicates the decision to the other departments, which can continue with their internal communication. We have that $\mathsf{ipart}(H_{\mathsf{fin}}) = \{\mathtt{f}_1, \mathtt{f}_2\}$ and $\mathsf{epart}(H_{\mathsf{fin}}) = \{\mathtt{d}, \mathtt{s}\}$.

In §4, we prove the above types *compatible* and *compose* them into a single global type.

3.3 Projection and Localiser

We introduce *projection* and *localiser* for hybrid types. These operators play are fundamental for defining compatibility and, ultimately, achieving compositionality.

Definition 3.6 (Projection). The (generalised) projection on the set of participants E, is a partial operator, $\_ \upharpoonright_E : \mathcal{H} \to \mathcal{H}$, recursively defined by the following clauses (whenever the

recursive call is defined):

[PROJ-END] $\mathsf{end}\upharpoonright_E = \mathsf{end}$

[PROJ-REC]
$\mu X.H\upharpoonright_E = \mu X.(H\upharpoonright_E)$ if $\mu X.(H\upharpoonright_E)$ is guarded, else, if $(\mu X.H)$ is closed, $(\mu X.H)\upharpoonright_E = \mathsf{end}$

[PROJ-SEND]
$\mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}\upharpoonright_E =$
$\begin{cases}\mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}).(H_i\upharpoonright_E)\}_{i\in I} & \text{if } \mathtt{p}\in E \text{ and } \mathtt{q}\notin E\\ \bigsqcup_{i\in I}(H_i\upharpoonright_E) & \text{if } \mathtt{p},\mathtt{q}\notin E\end{cases}$

[PROJ-RECV]
$\mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}\upharpoonright_E =$
$\begin{cases}\mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}).(H_i\upharpoonright_E)\}_{i\in I} & \text{if } \mathtt{p}\notin E \text{ and } \mathtt{q}\in E\\ \bigsqcup_{i\in I}(H_i\upharpoonright_E) & \text{if } \mathtt{p},\mathtt{q}\notin E\end{cases}$

[PROJ-VAR] $X\upharpoonright_E = X$

[PROJ-PAR]
$H_1|H_2\upharpoonright_E = H_i\upharpoonright_E$
if $E\cap \mathsf{ipart}(H_j)=\emptyset$, with $i,j\in\{1,2\}$ and $i\neq j$

[PROJ-MSG]
$\mathtt{p}\to\mathtt{q}:\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}\upharpoonright_E =$
$\begin{cases}\mathtt{p}\to\mathtt{q}:\{\ell_i(\mathsf{S}).(H_i\upharpoonright_E)\}_{i\in I} & \text{if } \mathtt{p},\mathtt{q}\in E\\ \mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}).(H_i\upharpoonright_E)\}_{i\in I} & \text{if } \mathtt{p}\in E \text{ and } \mathtt{q}\notin E\\ \mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}).(H_i\upharpoonright_E)\}_{i\in I} & \text{if } \mathtt{p}\notin E \text{ and } \mathtt{q}\in E\\ \bigsqcup_{i\in I}(H_i\upharpoonright_E) & \text{if } \mathtt{p},\mathtt{q}\notin E\end{cases}$

and undefined otherwise.

Merging ($\sqcup$) is defined as a partial commutative operator over two hybrid types such that: for all H, $H\sqcup H = H$, it delves inductively inside all constructors (e.g., $\mathtt{p}\to\mathtt{q}:\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}\sqcup \mathtt{p}\to\mathtt{q}:\{\ell_i(\mathsf{S}_i).H'_i\}_{i\in I} = \mathtt{p}\to\mathtt{q}:\{\ell_i(\mathsf{S}_i).(H_i\sqcup H'_i)\}_{i\in I}$), and $\mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}\sqcup\mathtt{p}?\mathtt{q};\{\ell_j(\mathsf{S}_j).H'_j\}_{j\in J} = \mathtt{p}?\mathtt{q};\{\ell_k(\mathsf{S}_k).H_k\sqcup H'_k\}_{k\in I\cap J}\cup\{\ell_k(\mathsf{S}_k).H_k\}_{k\in I\setminus J}\cup\{\ell_k(\mathsf{S}_k).H'_k\}_{k\in J\setminus I}$

With respect to Definition 3.2, we now allow projection onto a *set* of participants, and we introduce rules for projecting send and receive constructs. We highlight the differences below:

- [PROJ-MSG] defines $\mathtt{p}\to\mathtt{q}:\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}$ to be projected onto E, if *both* $\mathtt{p}\in E$ *and* $\mathtt{q}\in E$; in this case the structure of the global message $\mathtt{p}\to\mathtt{q}:$ is maintained in the projected type;
- [PROJ-SEND] defines projection when the sender $\mathtt{p}$ is in E and $\mathtt{q}$ is not, and when both $\mathtt{p},\mathtt{q}\notin E$;
- [PROJ-RECV] defines projection when the receiver $\mathtt{q}$ is in E and $\mathtt{p}$ is not, and when both $\mathtt{p},\mathtt{q}\notin E$.

Remark 3.7. Projection is defined only onto sets of *internal* participants; e.g., $\mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}$ can be projected onto $\mathtt{q}$, but not onto $\mathtt{p}$; also, $\mathsf{ipart}(H\upharpoonright_E)\subseteq\mathsf{ipart}(H)$. If we project a hybrid type onto a singleton, we obtain a local type: $\mathit{isLocal}(H\upharpoonright_{\{\mathtt{p}\}})$. Furthermore, if $\mathit{isGlobal}(H)$, then $H\upharpoonright_{\{\mathtt{p}\}}$ is exactly the traditional MPST projection of the global type H onto $\mathtt{p}$, $H\upharpoonright_{\mathtt{p}}$ (Definition 3.2).

Definition 3.8 (Localiser). The localiser of a hybrid type is a partial operator, $\mathsf{loc}\ \_ : \mathcal{H}\to\mathcal{H}$, recursively defined by the following clauses (whenever the recursive call is defined):

[LOC-END] $\mathsf{loc}\ \mathsf{end} = \mathsf{end}$ [LOC-VAR] $\mathsf{loc}\ X = X$ [LOC-PAR] $\mathsf{loc}\ (H_1|H_2) = \mathsf{loc}\ H_1|\mathsf{loc}\ H_2$
[LOC-REC] $\mathsf{loc}\ \mu X.H = \mu X.(\mathsf{loc}\ H)$ if $\mu X.(\mathsf{loc}\ H)$ is guarded, else, if $\mu X.H$ is closed, $\mathsf{loc}\ \mu X.H = \mathsf{end}$
[LOC-SEND] $\mathsf{loc}\ (\mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}) = \mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).\mathsf{loc}\ H_i\}_{i\in I}$
[LOC-RECV] $\mathsf{loc}\ (\mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}) = \mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).\mathsf{loc}\ H_i\}_{i\in I}$
[LOC-MSG] $\mathsf{loc}\ (\mathtt{p}\to\mathtt{q}:\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}) = \bigsqcup^{\mathsf{L}}{}_{i\in I}\mathsf{loc}\ (H_i)$
and undefined otherwise.

Merging for the localiser ($\sqcup^{\mathsf{L}}$) is a partial commutative operator over two hybrid types such that: for all H, $H\sqcup^{\mathsf{L}}H = H$, it delves inductively inside all constructors (e.g., $\mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}\sqcup^{\mathsf{L}}\mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H'_i\}_{i\in I} = \mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).(H_i\sqcup^{\mathsf{L}}H'_i)\}_{i\in I}$), and $\mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H_i\}_{i\in I}\sqcup^{\mathsf{L}}\mathtt{p}!\mathtt{q};\{\ell_j(\mathsf{S}_j).H'_j\}_{j\in J} = \mathtt{p}!\mathtt{q};\{\ell_k(\mathsf{S}_k).H_k\sqcup^{\mathsf{L}}H'_k\}_{k\in I\cap J}\cup\{\ell_k(\mathsf{S}_k).H_k\}_{k\in I\setminus J}\cup\{\ell_k(\mathsf{S}_k).H'_k\}_{k\in J\setminus I}$

The localiser is a forgetful operator that *preserves local constructs* and discards global messages. [LOC-END], [LOC-VAR], [LOC-REC], and [LOC-PAR] preserve the non-message structure of the type, into its localisation. [LOC-SEND] and [LOC-RECV] state that send construct (an internal participant $\mathtt{p}$ sends to an external participant $\mathtt{q}$) and receive construct (an internal participant $\mathtt{q}$ receives from an external participant $\mathtt{p}$) are to be maintained and their continuations H_i localised into $\mathsf{loc}\ H_i$. [LOC-MSG] is the central rule: each global message has to be skipped and its continuations need to be merged.

Remark 3.9. The merge operator for the localiser, $\sqcup^L$, is *dual to the merge for projection*, $\sqcup$. To build the intuition behind this, let us consider the following hybrid type: $H = \mathtt{p} \to \mathtt{q} : \{\ell_1.\mathtt{p}!\mathtt{r};\ell_3.\mathsf{end}, \ell_2.\mathtt{p}!\mathtt{r};\ell_4.\mathsf{end}\}$. First, $\mathtt{p}$ chooses on which branch to take, by *internally* sending either ℓ_1 or ℓ_2 to $\mathtt{q}$; then according to the chosen branch, $\mathtt{p}$ itself sends a different *external* message to $\mathtt{r}$. When we localise the above type we obtain $\mathsf{loc}\ H = \mathtt{p}!\mathtt{r};\{\ell_3.\mathsf{end}, \ell_4.\mathsf{end}\}$, namely we have merged send constructs with different labels, $\mathtt{p}!\mathtt{r}\ell_3$ and $\mathtt{p}!\mathtt{r}\ell_4$. From the point of view of the external receiver $\mathtt{r}$, it makes no difference whether such choice has been taken by $\mathtt{p}$ at the time $\mathtt{p}$ sends to $\mathtt{r}$ (with $\mathtt{p}!\mathtt{r};$), or at a precedent stage of communication, internal to H (with $\mathtt{p} \to \mathtt{q}$). This intuition is proven correct by the results from the next section, when we define a compatibility notion, based on localiser and projection, and we prove compositionality.

4 COMPOSITIONALITY FOR DISTRIBUTED SPECIFICATION

In §3, we have set definitions in place to compose subprotocols. In particular, following the overview of Figure 4b, §2, what we need is:

- hybrid types $H_1, H_2, \ldots, H_N$ for the multiple components of the communicating system;
- a compatibility hybrid type $H^\dagger$ (we sometimes use the notation $G^\dagger$, when $H^\dagger$ is a global type, namely when $\mathit{isGlobal}(H^\dagger)$) that disciplines the inter-component communication; and
- the property that $H^\dagger$ projects onto the localisations of $H_1, H_2, \ldots, H_N$ (compatibility).

In this section, we present our journey to multiparty compositional specification in three steps:

(1) we focus on a single hybrid type H_i, for which compatibility holds: $H^\dagger \upharpoonright_{E_i} = \mathsf{loc}\ H_i$; we build a new type $\mathcal{B}^1_{E_i}\ (H^\dagger)\ (H_i)$, whose projection on E_i coincides with H_i, and which contains the information for external communication from $H^\dagger$ (Theorem 4.4);
(2) we show how Step 1 is a base case for composing multiple compatible protocols: from $\mathcal{B}^1$, we recursively define $\mathcal{B}\ (H^\dagger)\ ([H_1, \ldots, H_N])$, which projects onto H_i *for all* $i = 1, \ldots, N$ (compositionality, Theorem 4.7);
(3) we prove that projection composes over the subset relation (Theorem 4.9); this guarantees the applicability and correctness of our result: if $H^\dagger$ is a *global type*, we obtain a well-formed global type $G = \mathcal{B}\ (H^\dagger)\ ([H_1, \ldots, H_N])$ for the whole system, the projections of which, onto every participant, are the same as the projections of the subprotocols H_i (Corollary 4.10).

4.1 Step 1: Building Back a Single Subprotocol

Our first step towards compositionality is also the most technical of the three. In this section we present the main design choices, both in constructions and in proofs, that make our theory sound. For more details, we refer to Appendix B.1 of [Gheri and Yoshida 2023].

We are given $H^\dagger$, the compatibility type disciplining communication happening among subprotocols, and with one of these subprotocols H^E, describing the communication from the point of view of its internal participants, contained in the set E. The local constructs of H^E are *compatible* with what prescribed by $H^\dagger$ for communicating externally, formally:

$$H^\dagger \upharpoonright_E = \mathsf{loc}\ H^E \tag{C}$$

The above notion is designed for the direct composition of multiple subprotocols: the hybrid type H^E for one component is checked compatible, not against other components, but against $H^\dagger$, which gives global guidance for inter-component communication. This design choice differentiates our theory from previous work, where compatibility is checked by directly matching the inputs and outputs of two separate components (see §6 for further discussion). With such compatible types, we build $H = \mathcal{B}^1_E\ (H^\dagger)\ (H^E)$ that retains the information about external communication of $H^\dagger$ and about internal communication in the component H^E.

$$\mathcal{B}^1_E\ (\mathsf{end})\ (H^E) = H^E \qquad \mathcal{B}^1_E\ (X)\ (H^E) = H^E$$

$$\mathcal{B}^1_E\ (\mu X.H^{\dagger\prime})\ (H^E) = \begin{cases} (\mu X.H^{\dagger\prime})|H^E \text{ if } \mathsf{ipart}(H^{\dagger\prime}) \cap E = \emptyset \text{ and both } \mu X.H^{\dagger\prime} \text{ and } H^E \text{ are closed} \\ \mathtt{s} \to \mathtt{r} :\{\ell_i(\mathsf{S}).(\mathcal{B}^1_E\ (\mu X.H^{\dagger\prime})\ (H^E_i))\}_{i\in I} \text{ if } \mathsf{ipart}(H^{\dagger\prime}) \cap E = \emptyset, \\ \quad H^E = \mathtt{s} \to \mathtt{r} :\{\ell_i(\mathsf{S}_i)H^E_i\}_{i\in I}, \text{ and one of } \mu X.H^{\dagger\prime} \text{ and } H^E \text{ is not closed} \\ \mu X.(\mathcal{B}^1_E\ (H^{\dagger\prime})\ (H^{E\prime})) \text{ if } \mathsf{ipart}(H^{\dagger\prime}) \cap E \neq \emptyset \text{ and } H^E = \mu X.H^{E\prime} \end{cases}$$

$$\mathcal{B}^1_E\ (H^\dagger_1|H^\dagger_2)\ (H^E) = \begin{cases} (\mathcal{B}^1_E\ (H^\dagger_1)\ (H^E))|H^\dagger_2 \text{ if } \mathsf{ipart}(H^\dagger_2) \cap E = \emptyset \\ H^\dagger_1|(\mathcal{B}^1_E\ (H^\dagger_2)\ (H^E)) \text{ if } \mathsf{ipart}(H^\dagger_2) \cap E \neq \emptyset \text{ and } \mathsf{ipart}(H^\dagger_1) \cap E = \emptyset \end{cases}$$

$$\mathcal{B}^1_E\ (\mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H^\dagger_i\}_{i\in I})\ (H^E) = \begin{cases} \mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).(\mathcal{B}^1_E\ (H^\dagger_i)\ (H^E_i))\}_{i\in I} \\ \quad \text{if } H^E = \mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H^E{}_i\}_{i\in I} \\ \mathtt{s} \to \mathtt{r} :\{m_j(\mathsf{S}'_j).(\mathcal{B}^1_E\ (\mathsf{UL}_j)\ (H^E_j))\}_{j\in J} \\ \quad \text{with } \mathsf{UL}_j = \pi_j(\mathsf{unmL}\ H^\dagger\ [\mathsf{loc}\ H^E_j]_{j\in J}), \\ \quad \text{if } H^E = \mathtt{s} \to \mathtt{r} :\{m_j(\mathsf{S}'_j)H^E_j\}_{j\in J} \text{ and } \mathtt{p} \in E \\ \mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).(\mathcal{B}^1_E\ (H^\dagger_i)\ (\mathsf{UP}_i))\}_{i\in I} \\ \quad \text{with } \mathsf{UP}_i = \pi_i(\mathsf{unmP}\ H^E\ [H^\dagger_i \upharpoonright_E]_{i\in I}), \text{ if } \mathtt{p} \notin E \end{cases}$$

$$\mathcal{B}^1_E\ (\mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).H^\dagger_i\}_{i\in I})\ (H^E) = \begin{cases} \mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).(\mathcal{B}^1_E\ (H^\dagger_i)\ (H^E_i))\}_{i\in I} \\ \quad \text{if } H^E = \mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).H^E{}_i\}_{i\in I} \\ \mathtt{s} \to \mathtt{r} :\{m_j(\mathsf{S}'_j).(\mathcal{B}^1_E\ (\mathsf{UL}_j)\ (H^E_j))\}_{j\in J} \\ \quad \text{with } \mathsf{UL}_j = \pi_j(\mathsf{unmL}\ H^\dagger\ [\mathsf{loc}\ H^E_j]_{j\in J}), \\ \quad \text{if } H^E = \mathtt{s} \to \mathtt{r} :\{m_j(\mathsf{S}'_j)H^E_j\}_{j\in J} \text{ and } \mathtt{q} \in E \\ \mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).(\mathcal{B}^1_E\ (H^\dagger_i)\ (\mathsf{UP}_i))\}_{i\in I} \\ \quad \text{with } \mathsf{UP}_i = \pi_i(\mathsf{unmP}\ H^E\ [H^\dagger_i \upharpoonright_E]_{i\in I}), \text{ if } \mathtt{q} \notin E \end{cases}$$

$$\mathcal{B}^1_E\ (\mathtt{p} \to \mathtt{q} :\{\ell_i(\mathsf{S}_i).H^\dagger_i\}_{i\in I})\ (H^E) = \begin{cases} \mathtt{p} \to \mathtt{q} :\{\ell_i(\mathsf{S}_i).(\mathcal{B}^1_E\ (H^\dagger_i)\ (H^E_i))\}_{i\in I} \\ \quad \text{if } H^E = \mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H^E_i\}_{i\in I} \text{ or } H^E = \mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).H^E_i\}_{i\in I} \\ \mathtt{s} \to \mathtt{r} :\{m_j(\mathsf{S}'_j).(\mathcal{B}^1_E\ (\mathsf{UL}_j)\ (H^E_j))\}_{j\in J} \\ \quad \text{with } \mathsf{UL}_j = \pi_j(\mathsf{unmL}\ H^\dagger\ [\mathsf{loc}\ H^E_j]_{j\in J}), \\ \quad \text{if } H^E = \mathtt{s} \to \mathtt{r} :\{m_j(\mathsf{S}'_j).H^E_j\}_{j\in J} \text{ and } \mathtt{p} \in E \text{ or } \mathtt{q} \in E \\ \mathtt{p} \to \mathtt{q} :\{\ell_i(\mathsf{S}_i).(\mathcal{B}^1_E\ (H^\dagger_i)\ (\mathsf{UP}_i))\}_{i\in I} \\ \quad \text{with } \mathsf{UP}_i = \pi_i(\mathsf{unmP}\ H^E\ [H^\dagger_i \upharpoonright_E]_{i\in I}), \text{ if } \{\mathtt{p},\mathtt{q}\} \cap E = \emptyset \end{cases}$$

Fig. 5. Build-Back of a Single Component: Equations

Definition 4.1 (Build-Back of a Single Component). Given a set of participants E, we define the *build-back of a single component* as the partial recursive function $\mathcal{B}^1_E\ (H^\dagger)\ (H^E)$. The recursive equations are given in Figure 5; if none of those apply or if $\mathsf{ipart}(H^E) \not\subseteq E$, $\mathcal{B}^1_E\ (H^\dagger)\ (H^E)$ is undefined.

The rest of this subsection is dedicated to discussing the intuition behind the function $\mathcal{B}^1_E$ and its correctness (Theorem 4.4). First, let us consider the following equations from Definition 4.1:

$$\mathcal{B}^1_E\ (\mathtt{p} \to \mathtt{q} : \{\ell_i(\mathsf{S}_i).H^\dagger{}_i\}_{i\in I})\ (\mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H^E{}_i\}_{i\in I}) = \mathtt{p} \to \mathtt{q} :\{\ell_i(\mathsf{S}_i).(\mathcal{B}^1_E\ (H^\dagger_i)\ (H^E_i)\}_{i\in I}$$
$$\mathcal{B}^1_E\ (\mathtt{p} \to \mathtt{q} : \{\ell_i(\mathsf{S}_i).H^\dagger{}_i\}_{i\in I})\ (\mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).H^E{}_i\}_{i\in I}) = \mathtt{p} \to \mathtt{q} :\{\ell_i(\mathsf{S}_i).(\mathcal{B}^1_E\ (H^\dagger_i)\ (H^E_i)\}_{i\in I}$$

The two equations above show how our compatibility (Equation **C**) comes into play when building back a more general type. E.g., when the projection of $H^\dagger = \mathtt{p} \to \mathtt{q} : \{\ell_i(\mathsf{S}_i).H^\dagger{}_i\}_{i\in I}$ onto E is equal to the localisation of a send type $H^E = \mathtt{p}!\mathtt{q};\{\ell_i(\mathsf{S}_i).H^E{}_i\}_{i\in I}$ (with $\mathtt{p} \in E$) then their build-back is $\mathtt{p} \to \mathtt{q} :\{\ell_i(\mathsf{S}_i).(\mathcal{B}^1_E\ (H^\dagger_i)\ (H^E_i)\}_{i\in I}$. The case where $H^E = \mathtt{p}?\mathtt{q};\{\ell_i(\mathsf{S}_i).H^E{}_i\}_{i\in I}$ is analogous.

The next example shows how the build-back retains the information both (1) internal to the component of H^E and (2) about the inter-component communication of the system, given by $H^\dagger$.

Example 4.2 (Build-Back, Intuition). We are given $H^E = \mathtt{s} \to \mathtt{r} :\ell_1.\mathtt{s}!\mathtt{p};\ell_2.\mathsf{end}$, with internal participants $E = \{\mathtt{s},\mathtt{r}\}$, $H^\dagger = \mathtt{p} \to \mathtt{q} :\ell_0.\mathtt{s} \to \mathtt{p} :\ell_2.\mathsf{end}$, for compatibility, describing inter-component communication in the system: compatibility **C** holds. Following the equations in Figure 5, we first *build*

back the prefix $\mathbf{p} \to \mathbf{q} : \ell_0$, then recursively the prefix $\mathbf{s} \to \mathbf{r} : \ell_1$, and ultimately we exploit compatibility and $\mathbf{s}!\mathbf{p};\ell_2$ gets absorbed into $\mathbf{s} \to \mathbf{p} : \ell_2$. Namely, $\mathcal{B}^1_E\ (H^\dagger)\ (H^E) = \mathbf{p} \to \mathbf{q} : \ell_0.\mathbf{s} \to \mathbf{r} : \ell_1.\mathbf{s} \to \mathbf{p} : \ell_2.\ \mathsf{end}$. We observe that $\mathcal{B}^1_E\ (H^\dagger)\ (H^E)$ contains *all the interactions*, both intra-component (in H^E) and inter-component (in $H^\dagger$); in other words, $\mathcal{B}^1_E\ (H^\dagger)\ (H^E)$ carries the information both in H^E and in $H^\dagger$. This property of the build-back is formalised by conclusions (1) and (2) of Theorem 4.4.

Some detail from the previous example is hidden in the auxiliary "unmerge" functions unmP and unmL, for projection and localiser respectively. They reproduce in $\mathcal{B}^1_E\ (H^\dagger)\ (H^E)$ a branching structure that is faithful to the branching both in $H^\dagger$ and in H^E, where such branching may have been merged when projecting $H^\dagger$ on $E = \{\mathbf{s}, \mathbf{r}\}$ (with $\sqcup$, see Definition 3.6) or when localising H^E (with $\sqcup^L$, see Definition 3.8). We present the unmerge mechanism with the next example, while, for formal details, we refer the interested reader to Appendix B.1 of [Gheri and Yoshida 2023].

Example 4.3 (Unmerge). We focus on the merge for the localiser $\sqcup^L$; the case for projection is analogous. We are given $H^E = \mathbf{s} \to \mathbf{r} : \{\ell_{11}.\mathbf{s}!\mathbf{p};\ell_{21}.\mathsf{end}, \ell_{12}.\mathbf{s}!\mathbf{p};\ell_{22}.\mathsf{end}\}$ and $H^\dagger = \mathbf{p} \to \mathbf{q} : \ell_0.\mathbf{s} \to \mathbf{p} : \{\ell_{21}.\mathsf{end}, \ell_{22}.\mathsf{end}\}$ In this case, $H^\dagger \upharpoonright_E = \mathsf{loc}\ H^E = \mathbf{s}!\mathbf{p};\{\ell_{21}.\mathsf{end}, \ell_{22}.\mathsf{end}\}$. In particular, when computing $\mathsf{loc}\ H^E$, a merge of branches happens: $\mathsf{loc}\ H^E = \mathbf{s}!\mathbf{p};\{\ell_{21}.\mathsf{end}\} \sqcup^L \mathbf{s}!\mathbf{p};\{\ell_{22}.\mathsf{end}\}$. When building back, we need to unmerge and reproduce the original branching from H^E. In particular

$$\mathbf{s} \to \mathbf{p} : \{\ell_{21}.\mathsf{end}, \ell_{22}.\mathsf{end}\} \upharpoonright_E = (\mathbf{s}!\mathbf{p};\{\ell_{21}.\mathsf{end}\}) \sqcup^L (\mathbf{s}!\mathbf{p};\{\ell_{22}.\mathsf{end}\})$$

Under this hypothesis, unmL returns suitable branches for the build-back:

$$\mathsf{unmL}\ (\mathbf{s} \to \mathbf{p} : \{\ell_{21}.\mathsf{end}, \ell_{22}.\mathsf{end}\})\ [\mathbf{s}!\mathbf{p};\{\ell_{21}.\mathsf{end}\}, \mathbf{s}!\mathbf{p};\{\ell_{22}.\mathsf{end}\}] = [\ \mathbf{s} \to \mathbf{p} : \{\ell_{21}.\mathsf{end}\}\ ,\ \mathbf{s} \to \mathbf{p} : \{\ell_{22}.\mathsf{end}\}\ ]$$

Then, by following the build-back algorithm we obtain:

$$\mathcal{B}^1_E\ (H^\dagger)\ (H^E) = \mathbf{p} \to \mathbf{q} : \ell_0.\mathbf{s} \to \mathbf{r} : \{\ell_{11}.\mathbf{s} \to \mathbf{p} : \ell_{21}.\mathsf{end}, \ell_{12}.\mathbf{s} \to \mathbf{p} : \ell_{22}.\mathsf{end}\}$$

We observe that, above, the output of unmL, with arguments $H^{\dagger\prime} = \mathbf{s} \to \mathbf{p} : \{\ell_{21}.\mathsf{end}, \ell_{22}.\mathsf{end}\}$ $[H^L_1, H^L_2] = [\mathbf{s}!\mathbf{p};\{\ell_{21}.\mathsf{end}\}, \mathbf{s}!\mathbf{p};\{\ell_{21}.\mathsf{end}\}]$, is a list of two types $[H^\dagger_1, H^\dagger_2] = [\mathbf{s} \to \mathbf{p} : \{\ell_{21}.\mathsf{end}\}, \mathbf{s} \to \mathbf{p} : \{\ell_{21}.\mathsf{end}\}]$; for these, in particular, the following properties hold: (*a*) $H^\dagger_i \upharpoonright_E = H^L_i$, for $i = 1, 2$ (*b*) $H^{\dagger\prime} \upharpoonright_{E'} = H^\dagger_1 \upharpoonright_{E'} \sqcup^L H^\dagger_2 \upharpoonright_{E'}$, for any E', set of participants such that $E \cap E' = \emptyset$. The nesting of branching for general $H^\dagger$ and H^E may be intricate and tedious; the auxiliary functions unmP and unmL take care of the detail (see Appendix B.1 of [Gheri and Yoshida 2023]), in a way that properties (*a*) and (*b*) as above hold for unmL, and similar ones for unmP (Lemmas B.11 and B.15, Appendix B.1, of [Gheri and Yoshida 2023]). Generally, such properties ensure that both conclusions (1) and (2) (essential for composing multiple subprotocols), of Theorem 4.4, hold.

Finally, we can state our first compositionality result, which certifies the definition of $\mathcal{B}^1$.

Theorem 4.4 (Building Back a Single Component). *We fix a set of participants E, and we are given hybrid types $H^\dagger$ and H^E, such that:* (a) $\mathit{ipart}(H^E) \subseteq E$, (b) $\mathit{epart}(H^E) \cap E = \emptyset$, *and* (c) $H^\dagger \upharpoonright_E = \mathsf{loc}\ H^E$, *(compatibility C). We set $H = \mathcal{B}^1_E\ (H^\dagger)\ (H^E)$ and we have:*

(1) $H \upharpoonright_E = H^E$ and
(2) for all E', such that $E' \cap E = \emptyset$, $H \upharpoonright_{E'} = H^\dagger \upharpoonright_{E'}$.

Moreover if $\mathit{isGlobal}(H^\dagger)$ then $\mathit{isGlobal}(H)$.

The proof of Theorem 4.4 proceeds by induction on $\mathsf{depth}(H^\dagger) + \mathsf{depth}(H^E)$. Its inductive structure follows the defining equations of $\mathcal{B}^1$ (Figure 5) and it is non-trivial; the full detail can be found in Appendix B.1 of [Gheri and Yoshida 2023] (Theorem B.17), together with the auxiliary lemmas for merging. Theorem 4.4 ensures that the result of building back (backwards, with respect to the usual direction of projection) $H = \mathcal{B}^1_E\ (H^\dagger)\ (H^E)$ contains both (1) the information for

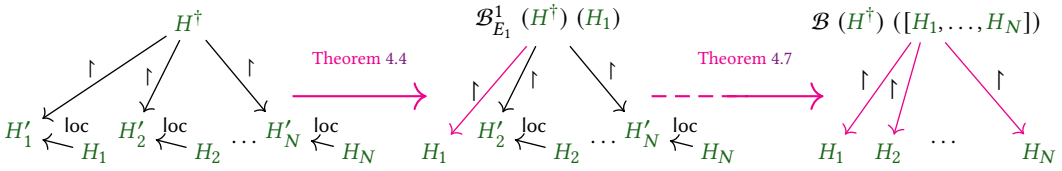


Fig. 6. Composing Multiple Protocols: Theorem 4.4 as the base case, Theorem 4.7 as the inductive step.

the internal communication in H^E (i.e., $H\restriction_E = H^E$) and (2) the information for the external communication prescribed by $H^\dagger$ (i.e., for all E', such that $E' \cap E = \emptyset$, $H\restriction_{E'} = H^\dagger\restriction_{E'}$). We describe the algorithm of $\mathcal{B}^1$ and the proof outline of Theorem 4.4 below, via example.

Example 4.5 (Definition 4.1 and Theorem 4.4). From Example 3.5, we consider the subprotocol for the strategy department.

$$H_{\mathsf{str}} = \mathbf{d} \to \mathbf{ad} : \mathit{prod}(\mathsf{nat}).\mathbf{d}!\mathbf{s};\mathit{prod}(\mathsf{nat}).\mathbf{d}!\mathbf{f}_1;\mathit{prod}(\mathsf{nat}).\mu X.\mathbf{f}_1?\mathbf{d};\{\mathit{ok}.\mathbf{d} \to \mathbf{ad} : \mathit{go}.\mathsf{end},\ \mathit{wait}.\mathbf{d} \to \mathbf{ad} : \mathit{wait}.X\}$$

The following protocol, described by the chief designer of the company D, coordinates the communication among the three departments (and ignores their internal one).

$$G^\dagger = \mathbf{d} \to \mathbf{s} : \mathit{prod}(\mathsf{nat}).\mathbf{d} \to \mathbf{f}_1 : \mathit{prod}(\mathsf{nat}).\mu X.\mathbf{f}_1 \to \mathbf{d} : \{\mathit{ok}.\mathbf{f}_1 \to \mathbf{s} : \mathit{price}(\mathsf{nat}).\mathsf{end},\ \mathit{wait}.\mathbf{f}_1 \to \mathbf{s} : \mathit{wait}.X\}$$

We observe that $\mathsf{ipart}(H_{\mathsf{str}}) = \{\mathbf{d}, \mathbf{ad}\}$, and that compatibility **C** holds:

$$\mathsf{loc}\ H_{\mathsf{str}} = G^\dagger\restriction_{\{\mathbf{d},\mathbf{ad}\}} = \mathbf{d}!\mathbf{s};\mathit{prod}(\mathsf{nat}).\mathbf{d}!\mathbf{f}_1;\mathit{prod}(\mathsf{nat}).\mu X.\mathbf{f}_1?\mathbf{d};\{\mathit{ok}.\mathsf{end},\ \mathit{wait}.X\}$$

To obtain $H_1^\dagger = \mathcal{B}^1_{\{\mathbf{d},\mathbf{ad}\}}\ (G^\dagger)\ (H_{\mathsf{str}})$, we first build back the internal global prefix in H_{str}:

$$H_1^\dagger = \mathbf{d} \to \mathbf{ad} : \mathit{prod}(\mathsf{nat}).H_1^{\dagger\prime}$$

We then proceed by induction, namely $H_1^{\dagger\prime}$ is built by composing $H^\dagger$ and the smaller hybrid type obtained from H_{str}, by removing this first prefix: $\mathbf{d}!\mathbf{s};\mathit{prod}(\mathsf{nat}).\mathbf{d}!\mathbf{f}_1;\mathit{prod}(\mathsf{nat}).\ldots$. We observe that the first two send constructs correspond to the projection of the two initial messages of $H^\dagger$ (this is guaranteed by the compatibility condition **C**); we take:

$$H_1^\dagger = \mathbf{d} \to \mathbf{ad} : \mathit{prod}(\mathsf{nat}).\mathbf{d} \to \mathbf{s} : \mathit{prod}(\mathsf{nat}).\mathbf{d} \to \mathbf{f}_1 : \mathit{prod}(\mathsf{nat}).H_1^{\dagger\prime\prime}$$

To obtain $H_1^{\dagger\prime\prime}$ we observe that the compatibility condition takes care of the recursive construct $\mu X.\ldots$ and of the first message $\mathbf{f}_1?\mathbf{d};\ldots$. After that, in each branch, we need to add first the internal message in H_{str} and then the external messages given by $G^\dagger$. We obtain

$$H_1^{\dagger\prime\prime} = \mu X.\mathbf{f}_1 \to \mathbf{d} : \{\mathit{ok}.\mathbf{d} \to \mathbf{ad} : \mathit{go}.\mathbf{f}_1 \to \mathbf{s} : \mathit{price}(\mathsf{nat}).\mathsf{end},\ \mathit{wait}.\mathbf{d} \to \mathbf{ad} : \mathit{wait}.\mathbf{f}_1 \to \mathbf{s} : \mathit{wait}.X\}$$

and, ultimately,

$$H_1^\dagger = \mathbf{d} \to \mathbf{ad} : \mathit{prod}(\mathsf{nat}).\mathbf{d} \to \mathbf{s} : \mathit{prod}(\mathsf{nat}).\mathbf{d} \to \mathbf{f}_1 : \mathit{prod}(\mathsf{nat}).\ \mu X.\mathbf{f}_1 \to \mathbf{d} : \left\{ \begin{array}{l} \mathit{ok}.\mathbf{d} \to \mathbf{ad} : \mathit{go}.\mathbf{f}_1 \to \mathbf{s} : \mathit{price}(\mathsf{nat}).\mathsf{end}, \\ \mathit{wait}.\mathbf{d} \to \mathbf{ad} : \mathit{wait}.\mathbf{f}_1 \to \mathbf{s} : \mathit{wait}.X \end{array} \right\}$$

Indeed, $H_1^\dagger$ contains all interactions from both $G^\dagger$ and H_{str}. The recursive definition of $\mathcal{B}^1$ (Definition 4.1) and the inductive proof of Theorem 4.4 follow the procedure presented in this example (Theorem B.17, Appendix B.1 of [Gheri and Yoshida 2023]).

4.2 Step 2: Multiparty Compositionality

$\mathcal{B}^1_E\ (H^\dagger)\ (H^E)$ composes the subprotocol H^E with the compatibility type $H^\dagger$. Here, we iterate this process for an arbitrary number of subprotocols $H_1, \ldots, H_N$, whenever compatible with respect to $H^\dagger$ (Equation **C**): we achieve full *multiparty compositionality* of subprotocols. The overview is given in Figure 6.

Definition 4.6 (Build-Back). Given a list of (disjoint) sets of participants $L = [E_1, \ldots, E_N]$, we define the partial recursive function $\mathcal{B}_L\ (H^\dagger)\ ([H_1, \ldots, H_N])$ as follows:

$$\mathcal{B}_{[E_1]}\ (H^\dagger)\ ([H_1]) = \mathcal{B}^1_{E_1}\ (H^\dagger)\ (H_1)$$
$$\mathcal{B}_{[E_1,\ldots,E_{n+1}]}\ (H^\dagger)\ ([H_1, \ldots, H_{n+1}]) = \mathcal{B}_{[E_2,\ldots,E_{n+1}]}\ (\mathcal{B}^1_{E_1}\ H^\dagger\ H_1)\ ([H_2, \ldots, H_{n+1}])$$

From now on, we leave implicit the first list argument (of sets of participants): we write $\mathcal{B}\ (H^\dagger)\ ([H_1, \ldots, H_N])$ for $\mathcal{B}_L\ (H^\dagger)\ ([H_1, \ldots, H_N])$.

Theorem 4.7 (Compositionality for Multiple Protocols). *We are given $E_1, \ldots, E_N$ sets of roles, and the hybrid types $H_1, H_2, \ldots, H_N$, and $H^\dagger$, such that: (a) $E_i \cap E_j = \emptyset$ for all $i \neq j$, (b) ipart$(H_i) \subseteq E_i$ for all i, (c) epart$(H_i) \cap E_i = \emptyset$ for all i, and (d) $H^\dagger \upharpoonright_{E_i} = \mathsf{loc}\ H_i$ (compatibility **C**). We set $H = \mathcal{B}\ (H^\dagger)\ ([H_1, \ldots, H_N])$ and we have that, for all i, $H \upharpoonright_{E_i} = H_i$. Moreover if isGlobal$(H^\dagger)$ then isGlobal(H).*

Proof. By induction on N. We add to the thesis: (*dsj*) for all E', $E' \cap \bigcup_{i=1,\ldots,N} E_i = \emptyset$, $H \upharpoonright_{E'} = H^\dagger \upharpoonright_{E'}$, since we need (*dsj*) within the induction hypothesis. The case $N = 1$ is Theorem 4.4. For $N = n + 1$, we set $H^{\dagger\prime} = \mathcal{B}^1_{E_1}\ (H^\dagger)\ (H_1)$ and we apply the induction hypothesis to $H = \mathcal{B}_{[E_2,\ldots,E_{n+1}]}\ (H^{\dagger\prime})\ ([H_2, \ldots, H_{n+1}])$: we obtain that for $i = 2, \ldots, n$, $H' \upharpoonright_{E_i} = H_i$. For E_1, since $E_1 \cap \bigcup_{i=2,\ldots,n+1} E_i = \emptyset$, thanks to (*dsj*), we have $H \upharpoonright_{E_1} = H^{\dagger\prime} \upharpoonright_{E_1} = H_1$ (by Theorem 4.4). For E', $E' \cap \bigcup_{i=1,\ldots,n+1} E_i = \emptyset$, we have that $E' \cap \bigcup_{i=2,\ldots,n+1} E_i = \emptyset$, and hence for (*dsj*), $H \upharpoonright_{E'} = H' \upharpoonright_{E'}$. We conclude by observing that $E' \cap E_1 = \emptyset$ and thus, by Theorem 4.4, $H' \upharpoonright_{E'} = H^\dagger \upharpoonright_{E'}$. □

Example 4.8 (Theorem 4.7). In Example 4.5, we have seen how to build back $G^\dagger$ and H_{str} into $H_1^\dagger$, a new hybrid type containing the information both for the inter-protocol communication (from $G^\dagger$) and for the communication inside the strategy department (from H_{str}). We observe that compatibility **C** holds not only for H_{str}, but also for H_{sales} and H_{fin}, namely:

$$\mathsf{loc}\ H_{\mathsf{str}} = G^\dagger \upharpoonright_{\mathtt{d},\mathtt{ad}} = \mathtt{d}!\mathtt{s};\mathit{prod}(\mathsf{nat}).\mathtt{d}!\mathtt{f}_1;\mathit{prod}(\mathsf{nat}).\mu X.\mathtt{f}_1?\mathtt{d};\{\mathit{ok}.\mathsf{end}, \mathit{wait}.X\}$$
$$\mathsf{loc}\ H_{\mathsf{sales}} = G^\dagger \upharpoonright_{\mathtt{s},\mathtt{w}} = \mathtt{d}?\mathtt{s};\mathit{prod}(\mathsf{nat}).\mu X.\mathtt{f}_1?\mathtt{s};\{\mathit{price}(\mathsf{nat}).\mathsf{end}, \mathit{wait}.X\}$$
$$\mathsf{loc}\ H_{\mathsf{fin}} = G^\dagger \upharpoonright_{\mathtt{f}_1,\mathtt{f}_2} = \mathtt{d}?\mathtt{f}_1;\mathit{prod}(\mathsf{nat}).\mu X.\mathtt{f}_1!\mathtt{d};\{\mathit{ok}.\mathtt{f}_1!\mathtt{s};\mathit{price}(\mathsf{nat}).\mathsf{end}, \mathit{wait}.\mathtt{f}_1!\mathtt{s};\mathit{wait}.X\}$$

The hypothesis of Theorem 4.7 holds and thus we can build $G = \mathcal{B}\ (G^\dagger)\ ([H_{\mathsf{str}}, H_{\mathsf{sales}}, H_{\mathsf{fin}}])$, such that $G \upharpoonright_{\mathtt{d},\mathtt{ad}} = H_{\mathsf{str}}$, $G \upharpoonright_{\mathtt{s},\mathtt{w}} = H_{\mathsf{sales}}$, and $G \upharpoonright_{\mathtt{f}_1,\mathtt{f}_2} = H_{\mathsf{fin}}$. To make the construction of G explicit, we follow the inductive proof structure of Theorem 4.7. The base case is taken care of in Example 4.5, where we apply Theorem 4.4 and build $H_1^\dagger$, by composition of $G^\dagger$ and H_{str}, we obtain :

$$H_1^\dagger = \mathtt{d} \to \mathtt{ad}:\mathit{prod}(\mathsf{nat}).\mathtt{d} \to \mathtt{s}:\mathit{prod}(\mathsf{nat}).\mathtt{d} \to \mathtt{f}_1:\mathit{prod}(\mathsf{nat}).\ \mu X.\mathtt{f}_1 \to \mathtt{d}:\left\{\begin{array}{l}\mathit{ok}.\mathtt{d} \to \mathtt{ad}:\mathit{go}.\mathtt{f}_1 \to \mathtt{s}:\mathit{price}(\mathsf{nat}).\mathsf{end},\\ \mathit{wait}.\mathtt{d} \to \mathtt{ad}:\mathit{wait}.\mathtt{f}_1 \to \mathtt{s}:\mathit{wait}.X\end{array}\right\}$$

We observe that $H_1^\dagger \upharpoonright_{\mathtt{s},\mathtt{w}} = G^\dagger \upharpoonright_{\mathtt{s},\mathtt{w}}$, hence, since **C** still holds, we can apply again the build-back procedure and obtain:

$$H_2^\dagger = \mathtt{d} \to \mathtt{ad}:\mathit{prod}(\mathsf{nat}).\mathtt{d} \to \mathtt{s}:\mathit{prod}(\mathsf{nat}).\mathtt{d} \to \mathtt{f}_1:\mathit{prod}(\mathsf{nat}).\ \mu X.\mathtt{f}_1 \to \mathtt{d}:\left\{\begin{array}{l}\mathit{ok}.\mathtt{d} \to \mathtt{ad}:\mathit{go}.\mathtt{f}_1 \to \mathtt{s}:\mathit{price}(\mathsf{nat}).\mathtt{s} \to \mathtt{w}:\mathit{publish}(\mathsf{nat}).\mathsf{end},\\ \mathit{wait}.\mathtt{d} \to \mathtt{ad}:\mathit{wait}.\mathtt{f}_1 \to \mathtt{s}:\mathit{wait}.\mathtt{s} \to \mathtt{w}:\mathit{wait}.X\end{array}\right\}$$

$H_2^\dagger$ collects all the interactions from $G^\dagger$, H_{str}, and H_{sales}. To obtain a type G that also includes the internal interactions of H_{fin}, we perform one more induction step, building back from $H_2^\dagger$ and H_{fin}.

$$G = \mathtt{d} \to \mathtt{ad}:\mathit{prod}(\mathsf{nat}).\mathtt{d} \to \mathtt{s}:\mathit{prod}(\mathsf{nat}).\mathtt{d} \to \mathtt{f}_1:\mathit{prod}(\mathsf{nat}).\mathtt{f}_1 \to \mathtt{f}_2:\mathit{prod}(\mathsf{nat}).\ \mu X.\mathtt{f}_2 \to \mathtt{f}_1:\left\{\begin{array}{l}\mathit{price}(\mathsf{nat}).\mathtt{f}_1 \to \mathtt{d}:\mathit{ok}.\mathtt{d} \to \mathtt{ad}:\mathit{go}.\mathtt{f}_1 \to \mathtt{s}:\mathit{price}(\mathsf{nat}).\mathtt{s} \to \mathtt{w}:\mathit{publish}(\mathsf{nat}).\mathsf{end},\\ \mathit{wait}.\mathtt{f}_1 \to \mathtt{d}:\mathit{wait}.\mathtt{d} \to \mathtt{ad}:\mathit{wait}.\mathtt{f}_1 \to \mathtt{s}:\mathit{wait}.\mathtt{s} \to \mathtt{w}:\mathit{wait}.X\end{array}\right\}$$

A graphical representation of G can be found in Appendix A of [Gheri and Yoshida 2023].

Theorem 4.7 gives a technique for composing multiple subprotocols into a more general one. The next, conclusive step proves that compositionality well-behaves with respect to MPST projection.

4.3 Step 3: Compositionality through Projection

With Definition 3.6, we have generalised the MPST projection to hybrid types. We prove that generalised projection well-behaves with respect to set inclusion.

Theorem 4.9 (Projection Composes over Set Inclusion). *Given H, E_1 and E_2, $E_2 \subseteq E_1$, if $H \upharpoonright_{E_1}$ is defined, then $(H \upharpoonright_{E_1}) \upharpoonright_{E_2} = H \upharpoonright_{E_2}$.*

Proof. By structural induction on H (see Appendix B.2 of [Gheri and Yoshida 2023]). □

Theorem 4.9 is the last fundamental ingredient to achieve *distributed protocol specification.*

Corollary 4.10 (Distributed Protocol Specification). *Given $E_1, \ldots, E_N$ disjoint sets of participants, a global type $G^\dagger$, and $H_1, \ldots, H_N$ hybrid types, such that:* (a) *$\mathit{ipart}(H_i) = E_i$ for all i,* (b) *$\mathit{epart}(H_i) \cap E_i = \emptyset$ for all i,* (c) *$\mathit{ipart}(G^\dagger) \subseteq \bigcup_{i=1,\ldots,N} E_i$, and* (d) *$G^\dagger \upharpoonright_{E_i} = \mathsf{loc}\ H_i$ (compatibility **C**); there exists G such that, for all i, for all $\mathtt{r} \in E_i$, $G \upharpoonright_{\{\mathtt{r}\}} = H_i \upharpoonright_{\{\mathtt{r}\}}$.*

Proof. We set $G = \mathcal{B}\ (G^\dagger)\ ([H_1, \ldots, H_N])$; by Theorem 4.7, G is global and such that $G \upharpoonright_{E_i} = H_i$. Then, we apply Theorem 4.9 and we obtain, for $\mathtt{r} \in E_i$, $G \upharpoonright_{\{\mathtt{r}\}} = G \upharpoonright_{E_i} \upharpoonright_{\{\mathtt{r}\}} = H_i \upharpoonright_{\{\mathtt{r}\}}$. We observe that $G \upharpoonright_{\{\mathtt{r}\}}$ is a local type, for all $\mathtt{r}$, (Remark 3.7). □

Example 4.11. In Examples 4.5 and 4.8, the protocol designer for each department has given their hybrid type, H_{str}, H_{sales}, and H_{fin}, disciplining internal communication (with messages $\mathtt{p} \to \mathtt{q}$) and specifying the communication with other departments (with sends/receives, $\mathtt{p}!\mathtt{q}/\mathtt{p}?\mathtt{q}$). Compatibility (Equation **C**) has been verified against $G^\dagger$, as described by the chief designer D. In (Example 4.8) we build back G (we observe that is $\mathit{isGlobal}(G^\dagger)$ implies $\mathit{isGlobal}(G)$). Corollary 4.10 holds and projections of components H_{str}, H_{sales}, and H_{fin} onto single participants are local types, also projections of G (Theorem 4.9), the global type disciplining the whole system. For instance, if we wanted to get the local type for $\mathtt{d}$, traditionally, we would do so by projecting G. With our distributed protocol specification, it is enough to project H_{str} onto $\mathtt{d}$.

$$G \upharpoonright_{\{\mathtt{d}\}} = (G \upharpoonright_{\{\mathtt{d},\mathtt{ad}\}}) \upharpoonright_{\{\mathtt{d}\}} = H_{\mathsf{str}} \upharpoonright_{\{\mathtt{d}\}} = \mathtt{d}!\mathtt{ad};\mathit{prod}(\mathsf{nat}).\mathtt{d}!\mathtt{s};\mathit{prod}(\mathsf{nat}).\mathtt{d}!\mathtt{f}_1;\mathit{prod}(\mathsf{nat}).\mu X.\mathtt{f}_1?\mathtt{d};\{\mathit{ok}.\mathtt{d}!\mathtt{ad};\mathit{go}.\mathsf{end},\ \mathit{wait}.\mathtt{d}!\mathtt{ad};\mathit{wait}.X\}$$

If instead want to implement processes for $\mathtt{f}_1$ and $\mathtt{f}_2$, we can obtain the local types from H_{fin}.

$$G \upharpoonright_{\{\mathtt{f}_1\}} = H_{\mathsf{fin}} \upharpoonright_{\{\mathtt{f}_1\}} = \mathtt{d}?\mathtt{f}_1;\mathit{prod}(\mathsf{nat}).\mathtt{f}_1!\mathtt{f}_2;\mathit{prod}(\mathsf{nat}).\mu X.\mathtt{f}_2?\mathtt{f}_1;\left\{\begin{array}{l}\mathit{price}(\mathsf{nat}).\mathtt{f}_1!\mathtt{d};\mathit{ok}.\mathtt{f}_1!\mathtt{s};\mathit{price}(\mathsf{nat}).\mathsf{end},\\ \mathit{wait}.\mathtt{f}_1!\mathtt{d};\mathit{wait}.\mathtt{f}_1!\mathtt{s};\mathit{wait}.X\end{array}\right\}$$

$$G \upharpoonright_{\{\mathtt{f}_2\}} = H_{\mathsf{fin}} \upharpoonright_{\{\mathtt{f}_2\}} = \mathtt{f}_1?\mathtt{f}_2;\mathit{prod}(\mathsf{nat}).\mu X.\mathtt{f}_2!\mathtt{f}_1;\{\mathit{price}(\mathsf{nat}).\mathsf{end},\ \mathit{wait}.X\}$$

We can proceed analogously for all participants.

Remark 4.12 (Applicability and Preservation of Semantics). Example 4.11 displays the essence of of our theory, formally captured by Corollary 4.10: for a system specified in a distributed way, with components $H_1, \ldots, H_N$ and compatibility type $G^\dagger$, there is *no need for an explicit description of G*. After compatibility checks (Equation **C**), our theory builds back a well-formed global type G for the whole system, and the session of local types, projections of G, can be obtained, in a distributed fashion, by directly projecting subprotocols, since, for $\mathtt{p} \in \mathsf{ipart}(H_i)$, $H_i \upharpoonright_{\{\mathtt{p}\}} = G \upharpoonright_{\{\mathtt{p}\}}$.

At the design stage, each designer D_i gives the subprotocol H_i and, with the simple equality check **C**, they make sure that their protocol is compatible with $G^\dagger$ (described by the chief designer D). At the type-checking/implementation stage, the designer D_i *independently* obtains local types for well-behaved implementations directly from their specification H_i. D_i is never concerned with the communication happening internally to, or among, other components. What guarantees

global well-behaviour is the existence of G, proved once and for all by our theory; no designer or programmer needs its explicit description. We have achieved *distributed protocol specification*.

Desirable MPST semantic guarantees, such as liveness and deadlock freedom, are preserved by our theory, thanks to its *semantics preservation*. Our compositionality-through-projection technique can explicitly build back a protocol as a global type that is traditionally well-formed (see, e.g., [Deniélou and Yoshida 2013; Honda et al. 2016]). Thus, our theory brings modularity to the protocol design phase, but, after such distributed specification, the result is a traditional MPST system, with a *single* global type that projects on local types for *all* participants, which benefits from existing semantics results from the MPST literature.

Remark 4.13 (On Hybrid Types). Central to this work is Definition 3.3 of hybrid types. Our theory shows how "open" subprotocols, interacting with other components of the system, can be specified as *hybrid types*, safely composed into a *global type*, and projected onto *local types*.

The syntax of hybrid types is simply the combination of the syntaxes of global and local types and, through the predicates $\mathit{isGlobal}(H)$ and $\mathit{isLocal}(H)$, we can isolate global and local types respectively, from the rest of hybrid types. This choice makes our development compatible with existing MPST systems and is key to semantics preservation: in our compositionality theory, well-formed global types guarantee semantics properties and local types are used for participant implementation, exactly as in traditional MPST (see Corollary 4.10 and Remark 4.12).

At the same time, dealing with a single syntax (hybrid types) simplifies our theory significantly. This paper proposes an approach to protocol compositionality that heavily relies on projection. Traditional MPST projection operates on global types and returns local types for implementation. Our generalised projection, instead, takes a hybrid type and returns a hybrid type, but, since global and local types are hybrid types, our projection maintains and extends the functionalities of the traditional operator. The main gain is flexibility: a function with the same domain as its codomain can be composed with itself and this is central in our proofs (see Theorem 4.9 and its role in the proof of Corollary 4.10). In other words, instead of working with multiple operators (which would have very similar definitions) and proving them compatible, we rely on a single one: generalised projection. In particular, we apply projection in the following key steps of our development:

- projecting the component H^E of a system onto one of its internal participants $\mathtt{p}$, gives—as it is customary in the literature—the local type $L_{\mathtt{p}}$ for implementing $\mathtt{p}$;
- projecting the type $G^\dagger$ is necessary for compatibility $\mathbf{C}$: $G^\dagger \upharpoonright_E = \mathrm{loc}\ H^E$; and
- the preservation of projection (Theorem 4.9) ensures applicability and correctness: local types for implementation, obtained by separately projecting the distributed components, are also projections of a single, well-formed global type for the whole system (equation $G \upharpoonright_{\{\mathtt{r}\}} = H_i \upharpoonright_{\{\mathtt{r}\}}$, Corollary 4.10).

5 CASE STUDIES

In this section, we evaluate our development with case studies. In §5.1, we discuss the role of the *parallel construct* $H_1|H_2$ in our protocol composition. Then, we consider the industry-standard protocol for authorisation, OAuth 2.0, [Hardt 2012; Horne 2020]: in §5.2, we observe the *modularity* benefits of our theory, and, in §5.3, we reach an *optimisation* for it. In §5.4, we show how hybrid types can be smoothly extended to feature *delegation and explicit connections*.

5.1 On the Parallel Construct and Compositionality

Our work enriches the type system of MPST with compositionality at the protocol-description level, while retaining the traditional syntax of local types. Consequently, the targeted process language is standard [Castro-Perez et al. 2021; Glabbeek et al. 2021]. In particular, local types and processes

$$H_{\mathsf{auth}} = \mathbf{ua}?\mathbf{oa};\{\mathit{init}(\mathtt{id},\mathtt{scp}).\mathbf{oa} \to \mathbf{ow} :\{\mathit{login}(\mathtt{id},\mathtt{scp}).\mathbf{ow} \to \mathbf{oa} : \left\{\begin{array}{l} \mathit{deny}.\mathbf{oa}!\mathbf{ua};\mathit{close}.\mathbf{oa}!\mathbf{res};\{\mathit{release}.\mathsf{end}\}, \\ \mathit{auth}(\mathtt{name},\mathtt{pwd}).\mathbf{oa}!\mathbf{ua};\left\{\begin{array}{l} \mathit{close}.\mathbf{oa}!\mathbf{res};\{\mathit{release}.\mathsf{end}\}, \\ \mathit{code}(\mathtt{code}).\mathbf{ua}?\mathbf{oa};\{\mathit{exchange}(\mathtt{id},\mathtt{secret},\mathtt{code}).\mathbf{oa}!\mathbf{ua}; \\ \{\mathit{close}.\mathbf{oa}!\mathbf{res};\{\mathit{release}.\mathsf{end}\},\ \mathit{token}(\mathtt{token}).\mathbf{oa}!\mathbf{res};\{\mathit{pass}.\mathsf{end}\}\}\} \end{array}\right\} \end{array}\right\}\}\}$$

$$H_{\mathsf{res}} = \mathbf{ua}!\mathbf{oa};\{\mathit{init}(\mathtt{id},\mathtt{scp}).\mathbf{oa}?\mathbf{ua};\left\{\begin{array}{l} \mathit{close}.\mathbf{oa}?\mathbf{res};\{\mathit{release}.\mathsf{end}\}, \\ \mathit{code}(\mathtt{code}).\mathbf{ua}!\mathbf{oa};\{\mathit{exchange}(\mathtt{id},\mathtt{secret},\mathtt{code}).\mathbf{oa}?\mathbf{ua}; \\ \left\{\begin{array}{l} \mathit{close}.\mathbf{oa}?\mathbf{res};\{\mathit{release}.\mathsf{end}\}, \\ \mathit{token}(\mathtt{token}).\mathbf{oa}?\mathbf{res};\{\mathit{pass}.H_{\mathsf{res\_acc}}\} \end{array}\right\}\} \end{array}\right\}\}$$

$$H_{\mathsf{res\_acc}} = \mu X.\mathbf{ua} \to \mathbf{res} :\{\mathit{request}(\mathtt{token}).\mathbf{res} \to \mathbf{ua} :\{\mathit{revoke}.\mathsf{end},\ \mathit{response}(\mathtt{data}).X\}\}$$

$$G^{\dagger}_{\mathsf{oa}} = \mathbf{ua} \to \mathbf{oa} :\{\mathit{init}(\mathtt{id},\mathtt{scp}).\mathbf{oa} \to \mathbf{ua} : \left\{\begin{array}{l} \mathit{close}.\mathbf{oa} \to \mathbf{res} :\{\mathit{release}.\mathsf{end}\}, \\ \mathit{code}(\mathtt{code}).\mathbf{ua} \to \mathbf{oa} :\{\mathit{exchange}(\mathtt{id},\mathtt{secret},\mathtt{code}).\mathbf{oa} \to \mathbf{ua} : \\ \{\mathit{close}.\mathbf{oa} \to \mathbf{res} :\{\mathit{release}.\mathsf{end}\},\ \mathit{token}(\mathtt{token}).\mathbf{oa} \to \mathbf{res} :\{\mathit{pass}.\mathsf{end}\}\}\} \end{array}\right\}\}$$

Fig. 7. Distributed Specification for OAuth 2.0

are *single-threaded*. On the other hand, to allow the description of protocols where two different components execute independently, without exchanging messages with each other, we have added the parallel construct to the syntax of hybrid (and global) types.

The *parallel construct* (or *parallel composition*) for global types appears in the first presentation of MPST [Honda et al. 2008], but dismissed in subsequent literature [Bettini et al. 2008; Castro-Perez et al. 2021; Coppo et al. 2015]. For achieving compositionality, without requiring further well-formedness restrictions on global types, we need to explicitly add parallel composition $H_1|H_2$. Let us consider the following distributed specification:

$$G^{\dagger}_{\mathsf{par}} = \mathbf{p} \to \mathbf{r} :\ell.\mathsf{end} \qquad H^1_{\mathsf{par}} = \mathbf{p}!\mathbf{r};\ell.\mu X.\mathbf{p} \to \mathbf{q} :\{\ell_1.X,\ell_2.\mathsf{end}\} \qquad H^2_{\mathsf{par}} = \mathbf{p}?\mathbf{r};\ell.\mu Y.\mathbf{r} \to \mathbf{s} :\{\ell_3.Y,\ell_4.\mathsf{end}\}$$

Compatibility holds: $G^{\dagger}_{\mathsf{par}}\upharpoonright_{\{\mathbf{p},\mathbf{q}\}} = \mathbf{p}!\mathbf{r};\ell.\mathsf{end} = \mathsf{loc}\ H^1_{\mathsf{par}}$ and $G^{\dagger}_{\mathsf{par}}\upharpoonright_{\{\mathbf{r},\mathbf{s}\}} = \mathbf{p}?\mathbf{r};\ell.\mathsf{end} = \mathsf{loc}\ H^1_{\mathsf{par}}$. Our theory guarantees the existence of a well-formed global type for the whole system, which entails a deadlock-free session: $G_{\mathsf{par}} = \mathbf{p} \to \mathbf{r} :\ell.\ \big(\mu X.\mathbf{p} \to \mathbf{q} :\{\ell_1.X,\ell_2.\mathsf{end}\}\big)\ |\ \big(\mu Y.\mathbf{r} \to \mathbf{s} :\{\ell_3.Y,\ell_4.\mathsf{end}\}\big)$. Without parallel composition, it is not clear how to compose H^1_{par} and H^2_{par}, even if they are compatible with respect to $G^{\dagger}_{\mathsf{par}}$. [1] Indeed, recent work [Glabbeek et al. 2021] has drawn attention to the role of the parallel construct in traditional MPST: by exploiting a similar example to the G_{par} above, the authors show that, if the syntax does not include parallel composition, there are non-deadlocked sessions that do not have a global type.

5.2 Distributed Specification for OAuth 2.0

We consider the industry-standard protocol for authorisation, OAuth 2.0, [Hardt 2012; Horne 2020]. In such protocol, the owner of a resource gives approval (through the OAuth server) for an external application to access some resource; the OAuth server ensures, by means of tokens, that the sensitive data of the owner are not shared with the external application.

We present a specification for the OAuth 2.0 protocol in two components (Figure 7): a first designer D_{auth} gives the hybrid type H_{auth} for the *OAuth server* **oa** and the resource *owner* **ow**, while a second D_{res} is in charge of the interactions involving an *untrusted app* **ua** and the *resource service* **res**, H_{res}. Separately, a chief designer describes the compatibility type $G^{\dagger}_{\mathsf{oa}}$. Compatibility holds,

[1] For more details on the role of parallel composition in the inductive proof of Theorem 4.4, see the full proof in Appendix B.1 of [Gheri and Yoshida 2023], Theorem B.17

$$G^{\dagger}_{\mathsf{op}} = \mathtt{ua} \to \mathtt{oa} : \{ \mathit{init}(\mathsf{id}, \mathsf{scp}).\mathtt{oa} \to \mathtt{ow} : \{ \mathit{login}(\mathsf{id}, \mathsf{scp}).\mathtt{ow} \to \mathtt{oa} : \left\{ \begin{array}{l} \mathit{deny}.\mathtt{oa} \to \mathtt{ua} : \mathit{close}.\mathtt{oa} \to \mathtt{res} : \{\mathit{release}.\mathsf{end}\}, \\ \mathit{auth}(\mathsf{name}, \mathsf{pwd}).\mathtt{oa}!\mathtt{ua}; \left\{ \begin{array}{l} \mathit{close}.\mathtt{oa} \to \mathtt{res} : \{\mathit{release}.\mathsf{end}\}, \\ \mathit{code}(\mathsf{code}).\mathtt{ua} \to \mathtt{oa} : \{\mathit{exchange}(\mathsf{id}, \mathsf{secret},).\mathtt{oa} \to \mathtt{ua} : \\ \{\mathit{close}.\mathtt{oa} \to \mathtt{res} : \{\mathit{release}.\mathsf{end}\}, \mathit{token}(\mathsf{token}).\mathtt{oa} \to \mathtt{res} : \{\mathit{pass}.\mathsf{end}\}\}\} \end{array} \right. \end{array} \right. \}\}$$

H_{res} (and $H_{\mathsf{res\_acc}}$) the same as in Figure 7

Fig. 8. Distributed for OAuth 2.0, Optimised

$G^{\dagger}_{\mathsf{oa}} \upharpoonright_{\{\mathtt{oa},\mathtt{ow}\}} = \mathsf{loc}\ H_{\mathsf{auth}}$ and $G^{\dagger}_{\mathsf{oa}} \upharpoonright_{\{\mathtt{ua},\mathtt{res}\}} = \mathsf{loc}\ H_{\mathsf{res}}$, and Corollary 4.10 guarantees the existence of a well-formed global type for the whole protocol.

Let us focus on the component H_{res}, and in particular on its subcomponent $H_{\mathsf{res\_acc}}$, which contains exclusively internal interactions: once authorisation is granted, the untrusted app communicates directly with the resource service. We observe then that $H_{\mathsf{res\_acc}}$ can be specified *modularly*: the designer D_{res} could, at a later stage, *specify a different protocol*, $H'_{\mathsf{res\_acc}}$, for the interaction between **ua** and **res**, *without affecting compatibility*. We recognise one extra benefit of our theory in *modular specification for intra-component communication*: a designer (D_{res}) can modify the specification of their protocol (H_{res}) over time, in its internal interactions ($H_{\mathsf{res\_acc}}, H'_{\mathsf{res\_acc}}$), as long as its external communication ($\mathsf{loc}\ H_{\mathsf{res}}$)—and hence compatibility with respect to the prescription of the chief designer ($G^{\dagger}_{\mathsf{oa}}$)—is preserved.

5.3 Optimisation of OAuth 2.0 Specification

By inspecting the types in Figure 7, we notice that all inter-component interactions (between H_{auth} and H_{res}) go through the participant **oa** and, hence, they are all documented by $G^{\dagger}_{\mathsf{oa}}$, but also in H_{auth}. Here, describing explicitly $G^{\dagger}_{\mathsf{oa}}$ looks redundant and we ask ourselves whether a more efficient specification is possible. It turns out that we can omit the specification of $G^{\dagger}_{\mathsf{oa}}$ for the OAuth 2.0 protocol and *optimise distributed specification* (Corollary 4.10) in general.

Corollary 5.1 (Distributed MPST Specification, Optimisation). *Given $E, E_1, \ldots, E_N$ disjoint sets of participants, a global type $G^{\dagger}$, and $H_1, \ldots, H_N$ hybrid types, such that: (a) $\mathit{ipart}(H_i) = E_i$, for all $i = 1, \ldots, N$, (b) $\mathit{epart}(H_i) \cap E_i = \emptyset$, for all i, (c) $\mathit{ipart}(G^{\dagger}) \subseteq (\bigcup_{i=1,\ldots,N} E_i) \cup E$, and (d) $G^{\dagger} \upharpoonright_{E_i} = \mathsf{loc}\ H_i$; there exists G such that, for all $i = 1, \ldots, N$, for all $\mathtt{r} \in E_i$, $G \upharpoonright_{\mathtt{r}} = H_i \upharpoonright_{\{\mathtt{r}\}}$, and for all $\mathtt{r} \in E$, $G \upharpoonright_{\mathtt{r}} = G^{\dagger} \upharpoonright_{\{\mathtt{r}\}}$.*

The proof of the above corollary is analogous to the proof of Corollary 4.10 and entails the same semantic guarantees, since it leads to the existence of a global type for the whole system. However, in Corollary 5.1, $G^{\dagger}$ plays a twofold role: (a) it is the hybrid type, communication subprotocol for the component with set of participants E, and (b) it is the compatibility protocol, carrying all the inter-component interactions of the system. For OAuth 2.0, by exploiting Corollary 5.1, we obtain a more efficient distributed specification (Figure 8). Only two hybrid types, one for each component, are specified, provided that one contains also the compatibility information. We observe, with notations from Figures 7 and 8 that $H_{\mathsf{auth}} \upharpoonright_{\{\mathtt{p}\}} = G^{\dagger}_{\mathsf{op}} \upharpoonright_{\{\mathtt{p}\}}$, for $\mathtt{p} \in \{\mathtt{oa}, \mathtt{ow}\}$. Hence, for all participants, in both specifications we obtain the same local types for implementation. However, in the optimised case of Figure 8, for compatibility, we only need to check one equality: $G^{\dagger}_{\mathsf{op}} \upharpoonright_{\{\mathtt{ua},\mathtt{res}\}} = \mathsf{loc}\ H_{\mathsf{res}}$.

Remark 5.2 (Local Types from Partial Protocols). We observe that Corollaries 4.10 and 5.1 prescribe differently how to obtain local types for implementation. In Corollary 4.10, we project each H_i onto its internal participants (in E_i) to get the right local types. $G^{\dagger}$ has the sole role of disciplining inter-component communication for compatibility, hence all its projections onto single participants

(even if well-defined) are *not* meant for implementation. In Corollary 5.1, instead, the global type $G^\dagger$ not only provides for compatibility, but also describes the internal communication of the component that is concerned with the participants in E. Thus, if $\mathtt{r} \in E_i$, the respective local type is obtained by $H_i \upharpoonright_{\{\mathtt{r}\}}$, while, if $\mathtt{r} \in E$, the local type for implementing $\mathtt{r}$ is $G^\dagger \upharpoonright_{\{\mathtt{r}\}}$.

5.4 An Extension to Delegation and Explicit Connections

The MPST literature offers a variety of formalisms that enrich the type system with expressive features [Bocchi et al. 2014; Castro-Perez and Yoshida 2020; Lagaillardie et al. 2022; Viering et al. 2021; Zhou et al. 2020], while maintaining the central mechanism of projecting global types onto local types for distributed implementation. This suggests that our compositional methodology is general enough to capture more sophisticated formalisms than core MPST. In this section, we support this intuition with a case study: we extend hybrid types to include *delegation* and *explicit connections*. At the end of the section (Example 5.7), we show that this suitability for extensions is a prerogative of our compositionality-through-projection, differently from other compositionality approaches based on input/output matching [Barbanera et al. 2021; Stolze et al. 2021].

Both delegation and explicit connections are relevant and practical features, which have been extensively studied by the literature on concurrency. Delegation—the mechanism in which a participant appoints a different participant to act on their behalf—first appears in the context of object-oriented concurrency [Agha 1990; Ungar and Smith 1987; Yonezawa and Tokoro 1986] and, naturally, it has been implemented in mainstream object-oriented languages [Hu et al. 2008; Imai et al. 2020; Scalas and Yoshida 2016]. Over the years, the session-type literature has investigated the verification of concurrency in the presence of delegation [Bettini et al. 2008; Honda et al. 1998, 2008; Scalas et al. 2017]. In particular, we take the approach of Castellani et al. [2020], who treat delegation as internal to the session, in contrast with channel-passing delegation, which requires the interleaving of sessions. Thus, the authors can model (internal) delegation simply by adding specific constructs to the syntax of global types. Moreover, global types from this paper benefit from the flexibility of explicit connections: some participant may or may not take part in the communication, depending on the choice made by some other participant at a previous stage of execution. Explicit connections are common in the design of real-world protocols [Hardt 2012; MIT 2022] and have been significantly addressed by the literature [Castellani et al. 2019; Gheri et al. 2022; Harvey et al. 2021; Hu and Yoshida 2017]. Specifically, our approach is immediately compatible with the type system and the semantics in [Castellani et al. 2020] (from which we take most notation): we can compose subprotocols into a well-formed and *well-delegated* global type, projecting on local session types for the whole communicating system. Thus, no new semantics is needed, but MPST semantics guarantees (subject reduction, session fidelity, and progress) hold.

For this case study, we focus on our novel notion of compatibility through projection (Equation **C**). Developing the full theory goes beyond the scope of this paper: the structure of the proofs would be exactly the same as in our core theory (Section 4). Instead, we (a) extend hybrid types to delegation and explicit connections, (b) generalise projection, (c) define the localiser, and (d) state compatibility.

Definition 5.3 (Hybrid Types with Delegation). We define a set of *prefixes* for global and local messages; when $\mathtt{p}$ establishes an *explicit connection* to $\mathtt{q}$, we use the superscript ^{e}.

$$\alpha ::= \mathtt{p} \to \mathtt{q} \mid \mathtt{p} \to^e \mathtt{q} \quad \pi^{\mathsf{out}} ::= \mathtt{p}!\mathtt{q} \mid \mathtt{p}!^e\mathtt{q} \quad \pi^{\mathsf{in}} ::= \mathtt{p}?\mathtt{q} \mid \mathtt{p}?^e\mathtt{q} \quad \sigma ::= \alpha \mid \pi^{\mathsf{out}}$$

Hybrid types with delegation are defined inductively by:

$$\begin{aligned} H ::= \quad & \mathsf{end} \mid X \mid \mu X.H \mid H_1 | H_2 \mid \boxplus_{i\in I} \sigma_i^{\mathtt{p}}(\ell_i); H_i \mid \bigwedge_{i\in I} \pi_i^{\mathsf{in}}(\ell_i); H_i \mid \\ & \mathtt{p}\circ\langle\!\langle \bullet\mathtt{q}; H \mid \mathtt{q}\bullet\rangle\!\rangle\circ\mathtt{p}; H \mid \mathtt{p}\circ\langle\!\langle \bullet[\mathtt{q}]; H \mid [\mathtt{q}]\bullet\rangle\!\rangle\circ\mathtt{p}; H \mid [\mathtt{p}]\circ\langle\!\langle \bullet\mathtt{q}; H \mid \mathtt{q}\bullet\rangle\!\rangle\circ[\mathtt{p}]; H \end{aligned}$$

Without loss of generality, we omit payload types (sorts) from the syntax above: formally, participants only exchange labels ℓ. For degenerate branchings (with a single branch, where no actual choice happens) we omit the operators $\boxplus$ and $\bigwedge$ and we simply write the message (starting with a prefix α, π^{out}, π^{in}).

Hybrid types with delegation (Definition 5.3) endow with local send/receive constructs the global types from [Castellani et al. 2020]. In particular, *global choices* and *union types* from [Castellani et al. 2020] are particular cases of the above syntax. The global choice, $\boxplus_{i\in I}\alpha_i^{\mathsf{p}}(\ell_i);H_i$, is obtained by asking that, in $\boxplus_{i\in I}\sigma_i^{\mathsf{p}}(\ell_i);H_i$, all σ_i^{p} are global messages, α_i^{p}. Local union (send) types, $\bigvee_{i\in I}\pi_i^{\mathsf{out}}(\ell_i);H_i$, are now written as $\boxplus_{i\in I}\sigma_i^{\mathsf{p}}(\ell_i);H_i$, where all σ_i^{p} are send constructs π_i^{out}. In what follows we discuss how Definition 5.3 is a direct extension of Definition 3.3.

We observe that branching is more permissive than in Definition 3.3.

- In a single choice $\boxplus_{i\in I}\sigma_i^{\mathsf{p}}(\ell_i);H_i$ the sender $\mathbf{p}$ is unique, but receivers may be different, internal or external: in choices we allow the mixing of global messages α and send constructs π^{out}.
- Intersection types $\bigwedge_{i\in I}\pi_i^{\mathsf{in}}(\ell_i);H_i$ combine receive constructs π^{in}, possibly with different senders (and receivers).

Remark 5.4 (Generalising Hybrid Types to Delegation). The syntax of hybrid types with delegation (Definition 5.3) is a generalisation of the core syntax of hybrid types (Definition 3.3). [2] For each construct in Definition 3.3, we show how it can be expressed in the formalism of 5.3.

- end, X, $\mu X.H$, and $H_1|H_2$ are preserved.
- $\mathbf{p}!\mathbf{q};\{\ell_i.H_i\}_{i\in I}$ can be written as $\boxplus_{i\in I}\sigma_i^{\mathbf{p}}(\ell_i);H_i$, with $\sigma_i^{\mathbf{p}}=\mathbf{p}!\mathbf{q}$ for all $i\in I$.
- $\mathbf{p}?\mathbf{q};\{\ell_i.H_i\}_{i\in I}$ can be written as $\bigwedge_{i\in I}\pi_i^{\mathsf{in}}(\ell_i);H_i$, with $\pi_i^{\mathsf{in}}=\mathbf{p}?\mathbf{q}$ for all $i\in I$.
- $\mathbf{p}\to\mathbf{q};\{\ell_i.H_i\}_{i\in I}$ can be written as $\boxplus_{i\in I}\sigma_i^{\mathbf{p}}(\ell_i);H_i$, with $\sigma_i^{\mathbf{p}}=\mathbf{p}\to\mathbf{q}$ for all $i\in I$.

The global construct $\mathbf{p}\circ\langle\!\langle\bullet\mathbf{q};H$ models a *forward delegation* where $\mathbf{p}$ delegates their behaviour to $\mathbf{q}$; then, such behaviour is given back with the global construct for *backward delegation*, $\mathbf{q}\bullet\rangle\!\rangle\circ\mathbf{p};H$. The notation for local constructs is analogous: *active forward delegation* $\mathbf{p}\circ\langle\!\langle\bullet[\mathbf{q}];H$, *passive backward delegation* $[\mathbf{q}]\bullet\rangle\!\rangle\circ\mathbf{p};H$, *passive forward delegation* $[\mathbf{p}]\circ\langle\!\langle\bullet\mathbf{q};H$, and *active backward delegation* $\mathbf{q}\bullet\rangle\!\rangle\circ[\mathbf{p}];H$. As in our core theory (see Definition 3.3), local constructs carry both internal participants and external ones (in square brackets): e.g., in $\mathbf{p}\circ\langle\!\langle\bullet[\mathbf{q}];H$, $\mathbf{p}$ (internal) delegates their behaviour to the participant $\mathbf{q}$ of a different component (external).

Example 5.5 (Global Types with Delegation and Explicit Connections). Here, we consider two simple examples that display in isolation the features of delegation and explicit connections. Let us consider the following global protocols, written with the syntax of hybrid types with delegation (Definition 5.3).

$$G_d = \mathbf{customer}\to\mathbf{seller}(\mathit{ok});\mathbf{seller}\circ\langle\!\langle\bullet\mathbf{bank};\mathbf{customer}\to\mathbf{seller}(\mathit{card});\mathbf{bank}\bullet\rangle\!\rangle\circ\mathbf{seller};\mathsf{end}$$

In G_d, after the **customer** has given the *ok*, the **seller** *delegates* to the **bank** their role in the communication ($\mathbf{seller}\circ\langle\!\langle\bullet\mathbf{bank}$). Namely, when the **customer** sends their *card* number to the **seller**, they are in fact sharing that information with the **bank**. With the construct $\mathbf{bank}\bullet\rangle\!\rangle\circ\mathbf{seller}$, the **bank** delegates back their role to the **seller**.

$$G_{ec} = \mathbf{user}\to\mathbf{website}(\mathit{location});\boxplus\left\{\begin{array}{l}\mathbf{website}\to^{e}\mathbf{EUshop}(\mathit{open});\mathbf{EUshop}\to\mathbf{user}(\mathit{EUfrontpage});\mathsf{end}\\ \mathbf{website}\to^{e}\mathbf{UKshop}(\mathit{open});\mathbf{UKshop}\to\mathbf{user}(\mathit{UKfrontpage});\mathsf{end}\end{array}\right\}$$

In G_{ec}, after the **website** receives the **user**'s location, it decides whether to *(explicitly) connect* them to the **EUshop** or to the **UKshop**. Then the **EUshop** (resp. the **UKshop**) sends the *EUfrontpage* (resp. the *UKfrontpage*) to the **user**. Here in the first branch (resp. the second) of the choice, there is

[2] In order to make the notation lighter, we ignore sorts in both syntaxes: only labels ℓ are sent.

an explicit connection $\mathbf{website} \to^e \mathbf{EUshop}$ (resp. $\mathbf{website} \to^e \mathbf{UKshop}$) to the participant $\mathbf{EUshop}$ (resp. $\mathbf{UKshop}$), which does not appear in the other branch.

We adopt the definition of *well-delegated* type from [Castellani et al. 2020] (Definition 4.11); in particular a forward delegation (e.g., $\mathbf{p} \circ \langle\!\langle \bullet \mathbf{q};$) is always followed by a corresponding backwards one (e.g., $\mathbf{p}\bullet \rangle\!\rangle \circ \mathbf{q};$); also, choices must not appear between corresponding forward and backward delegation.

We define *projection* and *localiser* for hybrid types with delegation, by extending Definitions 3.6 and 3.8, §3.2. The main differences are listed below.

- Projection behaves on *delegation constructs* following the same intuition as for messages. E.g., $(\mathbf{p} \circ \langle\!\langle \bullet \mathbf{q}; H) \upharpoonright_E = \mathbf{p} \circ \langle\!\langle \bullet \mathbf{q}; (H \upharpoonright_E)$ if $\{\mathbf{p}, \mathbf{q}\} \subseteq E$, or $(\mathbf{p} \circ \langle\!\langle \bullet \mathbf{q}; H) \upharpoonright_E = \mathbf{p} \circ \langle\!\langle \bullet [\mathbf{q}]; (H \upharpoonright^d_{\{(\mathbf{p},\mathbf{q})\},E})$ if $\mathbf{p} \in E$ and $\mathbf{q} \notin E$, where projection needs to keep track of $\mathbf{p}$ delegating to $\mathbf{q}$, as indicated by the notation $H \upharpoonright^d_{\{(\mathbf{p},\mathbf{q})\},E}$.
- Branches with explicit connections of a participant $\mathbf{s}$ can be merged with branches where $\mathbf{s}$ does not appear; e.g., if $H = \boxplus\{\mathbf{p} \to \mathbf{q}(\ell); \mathsf{end}, \mathbf{p} \to^e \mathbf{s}(\ell^e); \mathbf{s} \to \mathbf{q}(\ell'); \mathsf{end}\}$, then $H \upharpoonright_{\{\mathbf{s}\}} = \mathbf{p}?^e\mathbf{s}(\ell^e); \mathbf{s}!\mathbf{q}(\ell'); \mathsf{end}$.
- We allow merging of receive constructs with different senders; e.g., with H as in the bullet point above, $H \upharpoonright_{\{\mathbf{q}\}} = \bigwedge\{\mathbf{p}?^e\mathbf{q}(\ell); \mathsf{end}, \mathbf{s}?^e\mathbf{q}(\ell'); \mathsf{end}\}$
- As in our core theory §3.2, the localiser skips global constructs and retains local ones; e.g., $\mathsf{loc}\ \mathbf{p} \circ \langle\!\langle \bullet \mathbf{q}; H = \mathsf{loc}\ H$ and $\mathsf{loc}\ \mathbf{p} \circ \langle\!\langle \bullet [\mathbf{q}]; H = \mathbf{p} \circ \langle\!\langle \bullet [\mathbf{q}]; (\mathsf{loc}\ H)$.

The partial function $\_\upharpoonright^d_{dE,E}$ generalises the delegation projection functions $\_\upharpoonright^1_{(\mathbf{p},\mathbf{q})}$ and $\_\upharpoonright^2_{(\mathbf{p},\mathbf{q})}$ from [Castellani et al. 2020] (Figure 6); this function is of a sequential nature: it is not defined for branching, recursion, and parallel constructs. Full definitions of projection $\upharpoonright$, the auxiliary delegation projection $\upharpoonright^d$, and localiser loc are in Appendix B.3 of [Gheri and Yoshida 2023] and compatible with [Castellani et al. 2020].

Example 5.6 (Projection with Delegation and Explicit Connections). Let us consider G_d and G_{ec} from Example 5.5 and, in particular, their projections onto single participants

The intuition behind the delegation constructs in G_d is well displayed by its projections.

$$
\begin{aligned}
G_d \upharpoonright_{\mathbf{seller}} &= \mathbf{seller}!\mathbf{customer}(ok); \mathbf{seller} \circ \langle\!\langle \bullet [\mathbf{bank}]; [\mathbf{bank}] \bullet \rangle\!\rangle \circ \mathbf{seller}; \mathsf{end} \\
G_d \upharpoonright_{\mathbf{customer}} &= \mathbf{seller}?\mathbf{customer}(ok); \mathbf{customer}!\mathbf{seller}(card); \mathsf{end} \\
G_d \upharpoonright_{\mathbf{bank}} &= [\mathbf{seller}] \circ \langle\!\langle \bullet \mathbf{bank}; \mathbf{customer}?\mathbf{bank}(card); \mathbf{bank} \bullet \rangle\!\rangle \circ [\mathbf{seller}]; \mathsf{end}
\end{aligned}
$$

After *actively* delegating their role to the $\mathbf{bank}$ ($\mathbf{seller} \circ \langle\!\langle \bullet [\mathbf{bank}]$) and before *passively* being delegated their role back ($[\mathbf{bank}] \bullet \rangle\!\rangle \circ \mathbf{seller}$), the $\mathbf{seller}$ is not involved in any communication ($G_d \upharpoonright_{\mathbf{seller}}$). At the same time, the $\mathbf{bank}$ plays the opposite role: is passive in receiving the delegation from $\mathbf{seller}$ ($[\mathbf{seller}] \circ \langle\!\langle \bullet \mathbf{bank}$) and active in delegating the role back ($\mathbf{bank} \bullet \rangle\!\rangle \circ [\mathbf{seller}]$). Furthermore, we observe that, while the $\mathbf{bank}$ knows that they are receiving the *card* number from the $\mathbf{customer}$, the $\mathbf{customer}$ acts as if they are sending it directly to the $\mathbf{seller}$: as expected, the $\mathbf{customer}$ is not involved in the delegation process and hence ignores it.

When projecting G_{ec} onto its participants, we obtain the following types.

$$
\begin{aligned}
G_{ec} \upharpoonright_{\mathbf{user}} &= \mathbf{user}!\mathbf{website}(location); \bigwedge \left\{ \begin{array}{l} \mathbf{EUshop}?\mathbf{user}(EUfrontpage); \mathsf{end} \\ \mathbf{UKshop}?\mathbf{user}(UKfrontpage); \mathsf{end} \end{array} \right\} \\
G_{ec} \upharpoonright_{\mathbf{website}} &= \mathbf{user}?\mathbf{website}(location); \boxplus \left\{ \begin{array}{l} \mathbf{website}!^e\mathbf{EUshop}(open); \mathsf{end} \\ \mathbf{website}!^e\mathbf{UKshop}(open); \mathsf{end} \end{array} \right\} \\
G_{ec} \upharpoonright_{\mathbf{EUshop}} &= \mathbf{website}?^e\mathbf{EUshop}(open); \mathbf{EUshop}!\mathbf{user}(EUfrontpage); \mathsf{end} \\
G_{ec} \upharpoonright_{\mathbf{UKshop}} &= \mathbf{website}?^e\mathbf{UKshop}(open); \mathbf{UKshop}!\mathbf{user}(UKfrontpage); \mathsf{end}
\end{aligned}
$$

We observe that the participants **EUshop** and **UKshop** are concerned only with the interactions that happen after their explicit connection (**website**$?^e$**EUshop** and **website**$?^e$**UKshop** respectively) and not with the communication in the other branch, where they are not connected.

We now define *compatibility*. In the usual multi-component scenario, we give hybrid types (with delegation and explicit connections) $H_1, H_2, \ldots, H_N$, for each component, and a compatibility type $G^\dagger$. As in §4, we consider the generic $H^E \in \{H_1, H_2, \ldots, H_N\}$ (with E being the set of its internal participants) in isolation and we state compatibility:

$$G^\dagger \upharpoonright_E = \mathsf{loc}\ H^E \tag{D}$$

Equation **D** is exactly the same as Equation **C**. This captures the *generality* of our technique: given an existing top-down MPST system, first, we extend its syntax to hybrid types and generalise projection to sets of participants; then we isolate the inter-component communication of each protocol with the localiser; and, finally, we can state compatibility, prove compositionality, and achieve distributed protocol specification. Such general design gives a clear advantage to our theory, with respect to the dual approach from previous work [Barbanera et al. 2021; Stolze et al. 2021].

Example 5.7 (Compatibility Through Projection VS Input/Output Matching). We consider the following three-component system H_1, H_2, H_3, with $G^\dagger$ for compatibility.

$$G^\dagger = \boxplus\{\mathbf{p} \to \mathbf{q}(\ell); \mathsf{end}, \mathbf{p} \to^e \mathbf{s}(\ell^e); \mathbf{s} \to \mathbf{q}(\ell'); \mathsf{end}\} \qquad H_2 = \mathbf{q} \to \mathbf{q}_1(\ell_1); \bigwedge\{\mathbf{p}?\mathbf{q}(\ell); \mathsf{end}, \mathbf{s}?\mathbf{q}(\ell'); \mathsf{end}\}$$
$$H_1 = \boxplus\{\mathbf{p} \to \mathbf{p}_0(\ell_0); \mathbf{p}!\mathbf{q}(\ell); \mathsf{end}, \mathbf{p}!^e\mathbf{s}(\ell^e); \mathbf{p} \to \mathbf{p}_0(\ell_0); \mathsf{end}\} \qquad H_3 = \mathbf{p}?^e\mathbf{s}(\ell^e); \mathbf{s} \to^e \mathbf{r}(\ell_0^e); \mathbf{s}!\mathbf{q}(\ell'); \mathsf{end}$$

In H_1, **p** makes a choice: on the first branch, first it sends an internal message and then an external one to **q** in H_2; on the other branch **p** first makes an explicit connection to **s** in H_3 and then sends an internal message. The component H_2 (after an internal interaction) is waiting to receive either from **p** in H_1 or from **s** in H_2. The third component H_3 is concerned only with the second branch, in case it is chosen by **p**. All types are compatible with respect to $G^\dagger$: $G^\dagger \upharpoonright_{E_i} = \mathsf{loc}\ H_i$, with $E_1 = \{\mathbf{p}, \mathbf{p}_0\}$, $E_2 = \{\mathbf{q}, \mathbf{q}_1\}$, and $E_3 = \{\mathbf{s}, \mathbf{r}\}$. Indeed, we can build a well-formed global type for the whole system:

$$G = \mathbf{q} \to \mathbf{q}_1(\ell_1); \boxplus\{\mathbf{p} \to \mathbf{p}_0(\ell_0); \mathbf{p} \to \mathbf{q}(\ell); \mathsf{end}, \mathbf{p} \to^e \mathbf{s}(\ell^e); \mathbf{s} \to^e \mathbf{r}(\ell_0^e); \mathbf{s} \to \mathbf{q}(\ell'); \mathbf{p} \to \mathbf{p}_0(\ell_0); \mathsf{end}\}$$

We focus on H_2 and we observe that, in the intersection of inputs, **q** is waiting to receive from **p** in H_1, on the first branch, and from **s** in H_3, on the second. Therefore, a match for inputs in H_2 cannot happen solely with outputs of H_1, nor solely with outputs of H_3. In this case, dual compatibility relations would fail, while, with our approach based on projection, compatibility can be simply stated with respect to the global guidance of $G^\dagger$, through Equation **D**.

6 RELATED WORK

Our work achieves *protocol compositionality* in MPST top-down systems, namely those systems *where the communication protocol is explicitly described* as a global type and, subsequently, from the projection of it, local types are obtained for implementation. We organise this section as a progressive discussion of related work, with respect to our paper, from more distant to closer.

MPST Alternatives to the Top-Down Approach. Since their first appearance [Honda et al. 2008], MPST have evolved into a variety of frameworks for the specification and the verification of concurrency, often beyond the original top-down approach. The works of Lange and Tuosto [2012] and Deniélou and Yoshida [2013] explicitly give algorithms for the synthesis of global types from communicating finite-state machines, while Lange et al. [2015] propose a similar method to build graphical choreographies, expressed as global graphs. Scalas and Yoshida [2019] develop a framework where global types are not necessary, relying instead on model- and type-checking techniques for verifying safety properties of collections of local types. The advantage of such approaches is that they offer analysis for pre-existent systems. However, before the need to

hierarchically design a new communicating system, the top-down approach enables a high-level specification of the system that guarantees safe interactions of distributed implementations. The top-down approach has seen a variety of tools and implementations, e.g., [Carbone and Montesi 2013; Castro-Perez et al. 2021; Cruz-Filipe et al. 2021; Gheri et al. 2022; Honda et al. 2011; Neykova and Yoshida 2019; Yoshida et al. 2013, 2021] and it has been investigated beyond MPST [Barbanera et al. 2020a; Montesi 2013]. Recent research (e.g., [Cledou et al. 2022; Glabbeek et al. 2021; Jacobs et al. 2022; Lagaillardie et al. 2022]) has kept exploring the possibilities offered by an explicit design of systems through protocol specification. Our work adds compositionality to top-down protocol specification.

Protocol Flexibility and Modularity beyond MPST. Caires and Vieira [2009]; Padovani et al. [2014] do not address protocol compositionality, but, in defining conversation types, the authors combine global and local constructs in the same syntax, for a flexible specification: messages can be scheduled, while participants can be at first left unspecified, thus allowing for interleaving of sessions. In the context of reactive programming, Carbone et al. [2018]; Savanovic et al. [2020] propose a technique for modular design: a communicating system is specified in terms of components, each (*composite*) component contains a choreography (or, *protocol*) and for each role in the protocol a new implementation (component) is specified. Intuitively speaking, each component comes with an input/output interface allowed by the protocol, so to keep track of data-flow dependencies. Montesi and Yoshida [2013] develop a compositional technique for choreographic programming: the specification of partial choreographies is allowed, i.e., the implementation of some of the roles can be left unspecified. These roles can be implemented by a different choreography at a later time; *compatibility* is achieved through an additional typing relation on choreographies, which relies both on global and local types. All the above work modifies the essence of the protocol structure. Our theory differs from it, first, because our compatibility condition (Equation **C**) relies on projection, instead of dual input/output matching. Then, compositionality based on hybrid types retains the simplicity of traditional MPST protocol structure: the result of composition is a well-formed global type and no new semantics need to be developed. Thanks to such semantics preservation and our novel compatibility, our techniques are general enough to be applied to traditional MPST (§3 and §4), as well as to more expressive specifications (§5.4).

Modular Global Types through Nesting. Global types are choreographic objects and, hence, originally [Honda et al. 2008] intended as standalone entities. Given their monolithic nature, techniques for making them more flexible have been proposed very early on; e.g., Demangeon and Honda [2012] describe a methodology for nesting global types, via calls to a subprotocol from a parent protocol. Tabareau et al. [2014] propose an alternative approach to nesting protocols, by extending the syntax of global and local types with *aspects* [Kiczales et al. 1997]. Demangeon et al. [2015] also explore nesting techniques for global types, and apply these to extend the Scribble protocol description language [Honda et al. 2011; Neykova and Yoshida 2019; Yoshida et al. 2021] with interruptible interactions. Our work does not establish a parent/offspring relation, but it explores direct composition of subprotocols treated as peers.

State of The Art of Direct Composition of MPST Protocols. To the best of our knowledge, only Barbanera et al. [2021] and Stolze et al. [2021] investigate the direct composition of protocols for MPST, so that *the result of composition is a well-formed global type.* Our development differs from this work, in primis, because of its semantics preservation: in our theory, global types are exactly as in traditional MPST theories (e.g., [Deniélou and Yoshida 2013]); thus, our compositionality is immediately compatible with those and benefits from their semantics results. Furthermore, our theory can compose *more than two* protocols (missing in [Barbanera et al. 2021]) and captures the full expressiveness of MPST, including *parallel composition and recursion* (missing in [Stolze et al. 2021]). Ultimately, our compatibility (Equation **C**) *overcomes the limitations of dual input/output*

matching, on which both Barbanera et al. [2021] and Stolze et al. [2021] rely: our theory is general enough to be applied to more sophisticated MPST systems (§5.4, Example 5.7). Below we expand on the significance of our contribution, with respect to this related work, with detailed examples.

In [Barbanera et al. 2021], the composition of *two* global types is achieved with *gateways* [Barbanera et al. 2018, 2019, 2020b]: two participants, one for each subprotocol (global type), are selected as forwarders (gateways) for communicating with the other; if the subprotocols are compatible, with respect to the gateway choice, they can be composed into a more general global type. The central difference between our design and [Barbanera et al. 2021] is the "interface" through which subprotocols communicate. Instead of gateways, hybrid types use local constructs for inter-component interactions and, instead of a dual input/output matching, they rely on projection for compatibility. Thus, we can safely combine two or more protocols at once. Concretely, we consider the following example involving three subprotocols, described as global types with gateways.

$$G_1 = \mathtt{p} \to \mathtt{h}_2 : \ell.\mathtt{p} \to \mathtt{q} : \ell.\mathtt{h}_3 \to \mathtt{q} : \ell.\mathsf{end} \qquad G_2 = \mathtt{k}_1 \to \mathtt{r} : \ell.\mathtt{s} \to \mathtt{r} : \ell.\mathtt{r} \to \mathtt{k}_3 : \ell.\mathsf{end}$$
$$G_3 = \mathtt{t} \to \mathtt{u} : \ell.\mathtt{u} \to \mathtt{l}_1 : \ell.\mathtt{l}_2 \to \mathtt{u} : \ell.\mathsf{end}$$

We identify $(\mathtt{h}_2, \mathtt{k}_1)$, $(\mathtt{h}_3, \mathtt{l}_1)$, and $(\mathtt{k}_3, \mathtt{l}_2)$ as gateways; we choose the first one and compose.

$$G_1 {}^{\mathtt{h}_2 \leftrightarrow \mathtt{k}_1} G_2 = \mathtt{p} \to \mathtt{h}_2 : \ell.\mathtt{h}_2 \to \mathtt{k}_1 : \ell.\mathtt{k}_1 \to \mathtt{r} : \ell.\mathtt{p} \to \mathtt{q} : \ell.\mathtt{h}_3 \to \mathtt{q} : \ell.\mathtt{s} \to \mathtt{r} : \ell.\mathtt{r} \to \mathtt{k}_3 : \ell.\mathsf{end}$$

The next steps of composition should happen between $\mathtt{h}_3$ and $\mathtt{l}_1$, and between $\mathtt{k}_3$ and $\mathtt{l}_2$. Applying again gateway-composition does not work: after a first step (e.g., using $\mathtt{h}_3$ and $\mathtt{l}_1$) we would be left with a single global type, i.e., no two types left to compose (e.g., using $\mathtt{k}_3$ and $\mathtt{l}_2$). While we could try simultaneous composition for more than one pair of gateways at once, the simple example in [Barbanera et al. 2021], Section 11, shows how this extension is unsound and could lead to deadlocked systems. Another naïve attempt to improve on gateway composition is the following: we force G_1 and G_2 to have the same participant for communicating with G_3 (relaxing the condition from [Barbanera et al. 2021], where participants in two composing global types must be distinct):

$$G_1' = \mathtt{p} \to \mathtt{h} : \ell.\mathtt{l} \to \mathtt{p} : \ell.\mathsf{end} \qquad G_2' = \mathtt{k} \to \mathtt{q} : \ell.\mathtt{q} \to \mathtt{l} : \ell.\mathsf{end}$$
$$G_3' = \mathtt{t} \to \mathtt{r} : \ell.\mathtt{r} \to \mathtt{t} : \ell.\mathsf{end}$$

The communication is supposed to happen between $\mathtt{h}$ and $\mathtt{k}$, and between $\mathtt{l}$ and $\mathtt{t}$. If we allow non-distinct participants in distinct global types, however, it is not immediate to achieve a well-defined gateways compositionality. For example, the order in which we compose plays a key role: while $(G_2 {}^{\mathtt{k} \leftrightarrow \mathtt{h}} G_1) {}^{\mathtt{l} \leftrightarrow \mathtt{t}} G_3$ is well defined, $(G_1 {}^{\mathtt{h} \leftrightarrow \mathtt{k}} G_2) {}^{\mathtt{l} \leftrightarrow \mathtt{t}} G_3$ is not. Also, associativity would not hold: we cannot compose G_3 with any of G_1 and G_2 and then finish composing with the remaining one.

We specify the system above in terms of hybrid types: in place of gateways, we use local constructs for inter-component communication (highlighted), and we give the type $G^\dagger$ for compatibility.

$$H_1 = \mathtt{p}!\mathtt{r};\ \ell.\mathtt{p} \to \mathtt{q} : \ell.\ \mathtt{u}?\mathtt{q};\ \ell.\mathsf{end} \qquad H_2 = \mathtt{p}?\mathtt{r};\ \ell.\mathtt{s} \to \mathtt{r} : \ell.\ \mathtt{r}!\mathtt{u};\ \ell.\mathsf{end}$$
$$H_3 = \mathtt{t} \to \mathtt{u} : \ell.\ \mathtt{u}!\mathtt{q};\ \ell.\ \mathtt{r}?\mathtt{u};\ \ell.\mathsf{end} \qquad G^\dagger = \mathtt{p} \to \mathtt{r} : \ell.\ \mathtt{u} \to \mathtt{q} : \ell.\ \mathtt{r} \to \mathtt{u} : \ell.\mathsf{end}$$

For all components, compatibility $\mathbf{C}$ holds and our theory can compose the three protocols into a well-formed global type (Corollary 4.10), thus overcoming the *binary* limitation of gateways.

Stolze et al. [2021], develop a binary compatibility relation that partially improves on [Barbanera et al. 2021]: they can compose more than two types, but put severe restrictions on the syntax. Global types in [Stolze et al. 2021] are inductive, but with *no recursion and no parallel construct*, thus making our work (and traditional MPST in general) strictly more expressive. The key role of parallel composition for an inductive syntax is discussed in [Glabbeek et al. 2021] (Example 13 and following paragraph) and in §5.1. Recursion is paramount for expressiveness in MPST [Coppo et al. 2015; Honda et al. 2008, 2016; Yoshida and Gheri 2020]: it allows typing processes with loops and it is omnipresent in real-world protocols (e.g., see §5.2 and §5.3). As for semantics, recursion allows for infinite executions and is a key element in the correspondence between local types and communicating finite-state machines [Deniélou and Yoshida 2013], on which practical implementations of the popular Scribble protocol language are based [Honda et al. 2011;

Hu and Yoshida 2016; Neykova and Yoshida 2019; Yoshida et al. 2013, 2021]. Also, recursion (in combination with branching) is among the most delicate aspects of MPST (see, e.g., Observation 3 in [Glabbeek et al. 2021] or Definition B.1 and the following well-formedness lemmas in Appendix B.1 of [Gheri and Yoshida 2023]). E.g., the recursive types $H_1 = \mu X.\mathtt{p} \to \mathtt{q} :\ell_1.\mathtt{p}!\mathtt{r};\ell.X$ and $H_2 = \mu X.\mathtt{p}?\mathtt{r};\ell.\mathtt{s} \to \mathtt{r} :\ell_2.X$, are composable both with gateways [Barbanera et al. 2021] and with our theory ($G^\dagger = \mu X.\mathtt{p} \to \mathtt{r} :\ell.X$ for compatibility), but they cannot be expressed in [Stolze et al. 2021].

In summary, our theory allows composing *two or more* subprotocols into a well-formed global type, while retaining *full expressiveness* of MPST, thus improving on the state of the art [Barbanera et al. 2021; Stolze et al. 2021]. Furthermore, previous work focuses on dual input/output matching, which makes it inapplicable to more expressive MPST systems, where instead, to the best of our knowledge, our novel compatibility **C** is the first notion to succeed (see §5.4, Example 5.7). We observe that, as it happens for the extension of binary session types to multiparty session types, when relying on projection instead of duality, we commit to the specification of one more global object (what we called $G^\dagger$), but, in return, we obtain full multiparty compatibility.

7 FUTURE WORK

The first envisioned application for our compositionality theory is its integration with practical protocol design languages, such as Zooid [Castro-Perez et al. 2021] or Scribble [Honda et al. 2011; Yoshida et al. 2013, 2021]. In what follows we briefly describe how this integration can be realised.

In the past ten years, Scribble has been employed as the protocol language of multiple toolchains, supporting different programming languages and integrating a variety of expressive features [Castro et al. 2019; Gheri et al. 2022; Honda et al. 2011; Hu and Yoshida 2017; Miu et al. 2021; Neykova et al. 2018; Neykova and Yoshida 2019; Scalas et al. 2017; Yoshida et al. 2013, 2021; Zhou et al. 2020]. Independently of the specific implementation, the Scribble toolchain is generally designed as follows.

(1) The designer specifies the communication protocol (a global type G) in Scribble.
(2) G is projected onto local types $L_1, \ldots, L_n$—or, equivalently, their representation as CFSMs [Deniélou and Yoshida 2013].
(3) From local types, APIs for the distributed implementation of all participants are generated (following the approach of Hu and Yoshida [2016]).

Through API implementation, the communication for the multiparty system is MPST certified and does not get stuck (semantic guarantees hold). Thanks to its semantics-preserving features and its backwards compatibility with existing MPST systems, our compositionality framework can be integrated with Scribble toolchains, with minimal effort. First, we will extend the Scribble protocol design language to the syntax of hybrid types (Definition 3.3)—of which global types are a particular case—with constructs $\mathtt{p}!\mathtt{q};$ and $\mathtt{p}?\mathtt{q};$. Then, we will implement the localiser (Definition 3.8) and extend projection to Definition 3.6; here compatibility with previous implementations is guaranteed by the considerations in Remark 3.7: Definition 3.6 generalises traditional MPST projection (Definition 3.2). Finally, we will implement checks for $\mathit{isGlobal}(G^\dagger)$ and compatibility **C** ($G^\dagger \upharpoonright_E = \mathsf{loc}\ H^E$). With these simple changes, we will endow Scribble with compositionality, thus enabling distributed protocol specification. "Compositional Scribble" will look as follows.

(1^c) Multiple designers give $H_1, \ldots, H_N$ in a distributed fashion and a chief designer specifies $G^\dagger$, as *hybrid types*.
(2^c) Internal checks are performed for $\mathit{isGlobal}(G^\dagger)$ and compatibility **C**.
(3^c) Each H_i is projected onto local types $L^i_1, \ldots, L^i_{n_i}$—or, equivalently, their representation as CFSMs [Deniélou and Yoshida 2013].
(4^c) From local types, APIs for the distributed implementation of all participants are generated.

We observe that, thanks to our theory, the checks in (2^c) are enough to guarantee the existence of a well-formed global type for the whole system, without the need for explicitly building back such type (see also Corollary 4.10 and Remark 4.12). Moreover, since projection is preserved (Theorem 4.9 and Corollary 4.10), local types are the same as in traditional MPST: not only do semantic guarantees still hold, but so does the correspondence between local types and CFSMs in [Deniélou and Yoshida 2013].

With respect to traditional Scribble, nothing changes for the user, apart from the added functionality of distributed protocol specification: the safe distributed implementation of all participants is still enabled, but, now, also protocols can be specified in a distributed fashion, in terms of their components (as hybrid types $H_1, \ldots, H_N$).

Beyond its integration with existing protocol design languages, we plan to build on this work in different directions. In Section 5.4, we have shown the potential of our approach to compositionality, by adapting it to an MPST formalism that extends the traditional syntax of global types to include the advanced features of delegation and explicit connections. Similarly, beyond our core compositionality for global types, we envision future applications to the wide variety of MPST systems (e.g., featuring fault tolerance [Viering et al. 2021], timed specification [Bocchi et al. 2014], refinements [Zhou et al. 2020], cost awareness [Castro-Perez and Yoshida 2020], or exception handling [Lagaillardie et al. 2022]) and, orthogonally, future extensions that add flexibility to compositionality itself (e.g., by factoring in renaming mechanisms for participants [Jacobs et al. 2022]). Another promising perspective is the application of our techniques beyond MPST, to other protocol-design formalisms based on projection, e.g., *choreography automata* [Barbanera et al. 2020a; Gheri et al. 2022] or *choreographic programming* [Montesi 2013].

8 CONCLUSION

We have developed a "compositionality-through-projection" technique that allows the distributed specification of MPST protocols, in terms of *hybrid types* (Definition 3.3). Our work neatly improves on the state of the art, by allowing for composition of *more than two protocols*, while retaining the *full expressiveness* of global types. Our results (Corollaries 4.10 and 5.1) guarantee correctness and make our theory compatible with existing MPST systems (*semantics preservation*). Our *novel compatibility* relation (Equation **C**), based on generalised projection and localiser (Definitions 3.6 and 3.8), overcomes the limitations of dual input/output matching and it is general enough to capture extensions beyond traditional MPST (e.g., to delegation and explicit connections §5.4).

ACKNOWLEDGMENTS

We thank the Reviewers for their careful reviews and suggestions. We thank Franco Barbanera, Mariangiola Dezani-Ciancaglini, Francisco Ferreira, and Franco Raimondi for the in-depth conversations and useful comments on the preliminary versions of the paper. This work is supported by EPSRC EP/T006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462/1, EP/X015955/1 and NCSC/EPSRC VeTSS.

REFERENCES

Gul Agha. 1990. Concurrent Object-Oriented Programming. *Commun. ACM* 33, 9 (sep 1990), 125–141. https://doi.org/10.1145/83880.84528

Franco Barbanera, Ugo de'Liguoro, and Rolf Hennicker. 2018. Global Types for Open Systems. In *Proceedings 11th Interaction and Concurrency Experience, ICE 2018, Madrid, Spain, June 20-21, 2018 (EPTCS, Vol. 279)*, Massimo Bartoletti and Sophia Knight (Eds.). 4–20. https://doi.org/10.4204/EPTCS.279.4

Franco Barbanera, Ugo de'Liguoro, and Rolf Hennicker. 2019. Connecting open systems of communicating finite state machines. *Journal of Logical and Algebraic Methods in Programming* 109 (2019), 100476. https://doi.org/10.1016/j.jlamp.2019.07.004

Franco Barbanera, Mariangiola Dezani-Ciancaglini, Ivan Lanese, and Emilio Tuosto. 2021. Composition and decomposition of multiparty sessions. *Journal of Logical and Algebraic Methods in Programming* 119 (2021), 100620. https://doi.org/10.1016/j.jlamp.2020.100620

Franco Barbanera, Ivan Lanese, and Emilio Tuosto. 2020a. Choreography Automata. In *Coordination Models and Languages*, Simon Bliudze and Laura Bocchi (Eds.). Springer International Publishing, Cham, 86–106. https://doi.org/10.1007/978-3-030-50029-0_6

Franco Barbanera, Ivan Lanese, and Emilio Tuosto. 2020b. Composing Communicating Systems, Synchronously. In *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 39–59. https://doi.org/10.1007/978-3-030-61362-4_3

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR 2008 - Concurrency Theory*, Franck van Breugel and Marsha Chechik (Eds.). Springer, Berlin, Heidelberg, 418–433. https://doi.org/10.1007/978-3-540-85361-9_33

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. 2014. Timed Multiparty Session Types. In *25th International Conference on Concurrency Theory (LNCS, Vol. 8704)*. Springer, 419–434. https://doi.org/10.1007/978-3-662-44584-6_29

Luís Caires and Hugo Torres Vieira. 2009. Conversation Types. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 285–300. https://doi.org/10.1007/978-3-642-00590-9_21

Marco Carbone and Fabrizio Montesi. 2013. Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/2429069.2429101

Marco Carbone, Fabrizio Montesi, and Hugo Torres Vieira. 2018. Choreographies for Reactive Programming. https://doi.org/10.48550/ARXIV.1801.08107

Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2019. Reversible sessions with flexible choices. *Acta Informatica* 56, 7-8 (2019), 553–583. https://doi.org/10.1007/s00236-019-00332-y

Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Ross Horne. 2020. Global types with internal delegation. *Theoretical Computer Science* 807 (2020), 128–153. https://doi.org/10.1016/j.tcs.2019.09.027 In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part II.

David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed Programming Using Role-parametric Session Types in Go: Statically-typed Endpoint APIs for Dynamically-instantiated Communication Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 29 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290342

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 *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 237–251. https://doi.org/10.1145/3453483.3454041

David Castro-Perez and Nobuko Yoshida. 2020. CAMP: Cost-Aware Multiparty Session Protocols. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 155 (nov 2020), 30 pages. https://doi.org/10.1145/3428223

Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. 2022. API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.27

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming (LNCS, Vol. 9104)*. Springer, 146–178. https://doi.org/10.1007/978-3-319-18941-3_4

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021. Formalising a Turing-Complete Choreographic Language in Coq. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:18. https://doi.org/10.4230/LIPIcs.ITP.2021.15

Romain Demangeon and Kohei Honda. 2012. Nested Protocols in Session Types. In *CONCUR 2012 – Concurrency Theory*, Maciej Koutny and Irek Ulidowski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 272–286. https://doi.org/10.1007/978-3-642-32940-1_20

Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *FMSD* 46, 3 (2015), 197–225. https://doi.org/10.1007/s10703-014-0218-8

Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *Automata, Languages, and Programming*, Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 174–186. https://doi.org/10.1007/978-3-642-39212-2_18

Ethereum. 2022. Introduction to Smart Contracts. https://ethereum.org/en/developers/docs/smart-contracts/. Accessed: 20/10/2022.

Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. 2022. Design-By-Contract for Flexible Multiparty Session Protocols. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 8:1–8:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.8

Lorenzo Gheri and Nobuko Yoshida. 2023. Hybrid Multiparty Session Types – Full Version. https://doi.org/10.48550/ARXIV.2302.01979

Rob van Glabbeek, Peter Höfner, and Ross Horne. 2021. Assuming Just Enough Fairness to make Session Types Complete for Lock-freedom. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. https://doi.org/10.1109/LICS52264.2021.9470531

Dick Hardt. 2012. The OAuth 2.0 Authorization Framework. RFC 6749. https://doi.org/10.17487/RFC6749

Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. 2021. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:30. https://doi.org/10.4230/LIPIcs.ECOOP.2021.10

Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. Scribbling Interactions with a Formal Foundation. In *Distributed Computing and Internet Technology*, Raja Natarajan and Adegboyega Ojo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–75. https://doi.org/10.1007/978-3-642-19056-8_4

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proc. of 35th Symp. on Princ. of Prog. Lang.* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 273–284. https://doi.org/10.1145/1328897.1328472

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. https://doi.org/10.1145/2827695

Ross Horne. 2020. Session Subtyping and Multiparty Compatibility Using Circular Sequents. In *31st International Conference on Concurrency Theory (CONCUR 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 12:1–12:22. https://doi.org/10.4230/LIPIcs.CONCUR.2020.12

Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016,Eindhoven, The Netherlands (Lecture Notes in Computer Science, Vol. 9633)*, Perdita Stevens and Andrzej Wasowski (Eds.). Springer, 401–418. https://doi.org/10.1007/978-3-662-49665-7_24

Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *FASE (LNCS, Vol. 10202)*. 116–133. https://doi.org/10.1007/978-3-662-54494-5_7

Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-Based Distributed Programming in Java. In *ECOOP 2008 – Object-Oriented Programming*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 516–541. https://doi.org/10.1007/978-3-540-70592-5_22

Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. 2020. Multiparty Session Programming With Global Protocol Combinators. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.9

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom. *Proc. ACM Program. Lang.* 6, ICFP, Article 107 (aug 2022), 30 pages. https://doi.org/10.1145/3547638

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242. https://doi.org/10.1007/BFb0053381

Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2022. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:29. https://doi.org/10.4230/LIPIcs.ECOOP.2022.4

Julien Lange and Emilio Tuosto. 2012. Synthesising Choreographies from Local Session Types. In *CONCUR 2012 – Concurrency Theory*, Maciej Koutny and Irek Ulidowski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 225–239. https://doi.org/10.1007/978-3-642-32940-1_17

Julien Lange, Emilio Tuosto, and Nobuko Yoshida. 2015. From communicating machines to graphical choreographies. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 221–232. https://doi.org/10.1145/2676726.2676964

Julien Lange and Nobuko Yoshida. 2019. Verifying Asynchronous Interactions via Communicating Session Automata. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 97–117. https://doi.org/10.1007/978-3-030-25540-4_6

MIT. 2022. Kerberos: The Network Authentication Protocol. https://web.mit.edu/kerberos/. Accessed: 20/10/2022.

Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. 2021. Communication-Safe Web Programming in TypeScript with Routed Multiparty Session Types. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 94–106. https://doi.org/10.1145/3446804.3446854

Fabrizio Montesi. 2013. *Choreographic Programming*. Ph. D. Dissertation. https://www.fabriziomontesi.com/files/choreographic_programming.pdf

Fabrizio Montesi and Nobuko Yoshida. 2013. Compositional Choreographies. In *CONCUR 2013 – Concurrency Theory*, Pedro R. D'Argenio and Hernán Melgratti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 425–439. https://doi.org/10.1007/978-3-642-40184-8_30

Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#. In *27th International Conference on Compiler Construction*. ACM, 128–138. https://doi.org/10.1145/3178372.3179495

Rumyana Neykova and Nobuko Yoshida. 2019. *Featherweight Scribble*. LNCS, Vol. 11665. Springer, Cham, 236–259. https://doi.org/10.1007/978-3-030-21485-2_14

OMG. 2022. Business Process Model and Notation. https://www.bpmn.org/. Accessed: 20/10/2022.

Luca Padovani, Vasco Thudichum Vasconcelos, and Hugo Torres Vieira. 2014. Typing Liveness in Multiparty Communicating Systems. In *Coordination Models and Languages*, Eva Kühn and Rosario Pugliese (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 147–162. https://doi.org/10.1007/978-3-662-43376-8_10

Zorica Savanovic, Letterio Galletta, and Hugo Torres Vieira. 2020. A type language for message passing component-based systems. In *Proceedings 13th Interaction and Concurrency Experience, ICE 2020, Online, 19 June 2020 (EPTCS, Vol. 324)*, Julien Lange, Anastasia Mavridou, Larisa Safina, and Alceste Scalas (Eds.). 3–24. https://doi.org/10.4204/EPTCS.324.3

Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 21:1–21:28. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

Alceste Scalas and Nobuko Yoshida. 2019. Less Is More: Multiparty Session Types Revisited. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, Vol. 3. ACM, 30:1–30:29. https://doi.org/10.1145/3290343

Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying Message-passing Programs with Dependent Behavioural Types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 502–516. https://doi.org/10.1145/3314221.3322484

Claude Stolze, Marino Miculan, and Pietro Di Gianantonio. 2021. Composable Partial Multiparty Session Types. In *FACS 2021 Conference Proceedings*, Vol. 13077. Springer. https://doi.org/10.1007/978-3-030-90636-8_3

Nicolas Tabareau, Mario Südholt, and Éric Tanter. 2014. Aspectual Session Types. In *Proceedings of the 13th International Conference on Modularity* (Lugano, Switzerland) *(MODULARITY '14)*. Association for Computing Machinery, New York, NY, USA, 193–204. https://doi.org/10.1145/2577080.2577085

David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity *(OOPSLA '87)*. Association for Computing Machinery, New York, NY, USA, 227–242. https://doi.org/10.1145/38765.38828

Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. 2021. A Multiparty Session Typing Discipline for Fault-Tolerant Event-Driven Distributed Programming. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 124 (oct 2021), 30 pages. https://doi.org/10.1145/3485501

A Yonezawa and M Tokoro. 1986. *Object-oriented concurrent programming*. The MIT Press.

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. https://doi.org/10.1007/978-3-030-36987-3_5

Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2013. The Scribble Protocol Language. In *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina (Lecture Notes in Computer Science, Vol. 8358)*, Martín Abadi and Alberto Lluch-Lafuente (Eds.). Springer, 22–41. https://doi.org/10.1007/978-3-319-05119-2_3

Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. 2021. Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types. In *Fundamentals of Computation Theory*, Evripidis Bampis and Aris Pagourtzis (Eds.). Springer International Publishing, Cham, 18–35. https://doi.org/10.1007/978-3-030-86593-1_2

Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically Verified Refinements for Multiparty Protocols. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 148 (nov 2020), 30 pages. https://doi.org/10.1145/3428216