Generalising Projection in Asynchronous Multiparty Session Types

Rupak Majumdar ✉

Max Planck Institute for Software Systems, Kaiserslautern, Germany

Madhavan Mukund ✉

Chennai Mathematical Institute, India
CNRS IRL 2000, ReLaX, Chennai, India

Felix Stutz ✉

Max Planck Institute for Software Systems, Kaiserslautern, Germany

Damien Zufferey ✉

Max Planck Institute for Software Systems, Kaiserslautern, Germany

Abstract

Multiparty session types (MSTs) provide an efficient methodology for specifying and verifying message passing software systems. In the theory of MSTs, a global type specifies the interaction among the roles at the global level. A local specification for each role is generated by projecting from the global type on to the message exchanges it participates in. Whenever a global type can be projected on to each role, the composition of the projections is deadlock free and has exactly the behaviours specified by the global type. The key to the usability of MSTs is the projection operation: a more expressive projection allows more systems to be type-checked but requires a more difficult soundness argument.

In this paper, we generalise the standard projection operation in MSTs. This allows us to model and type-check many design patterns in distributed systems, such as load balancing, that are rejected by the standard projection. The key to the new projection is an analysis that tracks causality between messages. Our soundness proof uses novel graph-theoretic techniques from the theory of message-sequence charts. We demonstrate the efficacy of the new projection operation by showing many global types for common patterns that can be projected under our projection but not under the standard projection operation.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Multiparty session types, Verification, Communicating state machines

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2021.35

Related Version *Extended Version with Proofs*: `https://arxiv.org/abs/2107.03984`

Supplementary Material Tool available at:
Software (Source Code): `https://doi.org/10.5281/zenodo.5144684`

Funding This research was funded in part by the Deutsche Forschungsgemeinschaft project 389792660-TRR 248 and by the European Research Council under ERC Synergy Grant ImPACT (610150).

Acknowledgements The authors would like to thank Nobuko Yoshida, Thomas Wies, Elaine Li, and the anonymous reviewers for their feedback and suggestions.

1 Introduction

Distributed message-passing systems are both widespread and challenging to design and implement. A process tries to implement its role in a protocol with only the partial information received through messages. The unpredictable communication delays mean that messages from different sources can be arbitrarily reordered. Combining concurrency, asynchrony, and message buffering makes the verification problem algorithmically undecidable [10] and principled design and verification of such systems is an important challenge.

32nd International Conference on Concurrency Theory (CONCUR 2021).
Editors: Serge Haddad and Daniele Varacca; Article No. 35; pp. 35:1–35:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Multiparty Session Types (MSTs) [37, 52] provide an appealing type-based approach for formalising and compositionally verifying structured concurrent distributed systems. They have been successfully applied to web services [58], distributed algorithms [41], smart contracts [23], operating systems [26], high performance computing [34], timed systems [8], cyber-physical systems [47], etc. By decomposing the problem of asynchronous verification on to local roles, MSTs provide a clean and modular approach to the verification of distributed systems (see the surveys [4, 39]).

The key step in MSTs is the *projection* from a *global type*, specifying all possible global message exchanges, to *local types* for each role. The soundness theorem of MSTs states that every projectable global type is *implementable*: there is a distributed implementation that is free from communication safety errors such as deadlocks and unexpected messages.

The projection keeps only the operations observable by a given role and yet maintains the invariant that every choice can be distinguished in an unambiguous way. Most current projection operations ensure this invariant by syntactically merging different paths locally for each role. While these projections are syntactic and efficient, they are also very conservative and disallow many common design patterns in distributed systems.

In this paper, we describe a more general projection for MSTs to address the conservatism of existing projections. To motivate our extension, consider a simple load balancing protocol: a client sends a request to a server and the server forwards the request to one of two workers. The workers serve the request and directly reply to the client. (We provide the formal syntax later.) This common protocol is disallowed by existing MST systems, either because they syntactically disallow such messages (the *directed choice* restriction that states the sender and recipient must be the same along every branch of a choice), or because the projection operates only on the global type and disallows inferred choice.

The key difficulty in projection is to manage the interaction between choice and concurrency in a distributed setting. Without choice, all roles would just follow one predetermined sequence of send and receive operations. Introducing choice means a role either decides whom to send which message next, or reacts to the choices of other roles – even if such choices are not locally visible. This is only possible when the outcome of every choice propagates unambiguously. At each point, every role either is agnostic to a prior choice or knows exactly the outcome of the choice, even though it may only receive information about the choice indirectly through subsequent communication with other roles. Unfortunately, computing how choice propagates in a system is undecidable in general [3]; this is the reason why conservative restrictions are used in practice.

The key insight in our projection operation is to manage the interaction of choice and concurrency via a *message causality analysis*, inspired by the theory of communicating state machines (CSMs) and message sequence charts (MSCs), that provides a more global view. We resolve choice based on *available messages* along different branches. The causality analysis provides more information when merging two paths based on expected messages.

We show that our generalised projection subsumes previous approaches that lift the directed choice restriction [16, 38, 17, 40]. Empirically, it allows us to model and verify common distributed programming patterns such as load balancing, replicated data, and caching – where a server needs to choose between different workers – that are not in scope of current MSTs, while preserving the efficiency of projection.

We show type soundness for generalised projection. This generalisation is non-trivial, since soundness depends on subtle arguments about asynchronous messages in the system. We prove the result using an automata-theoretic approach, also inspired by the theory of MSCs, that argues about traces in communicating state machines. Our language-theoretic proof is different from the usual proof-theoretic approaches in soundness proofs of MST systems, and builds upon technical machinery from the theory of MSCs.

We show empirically that generalised choice is key to modelling several interesting instances in distributed systems while maintaining the efficiency of more conservative systems. Our global type specifications go beyond existing examples in the literature of MSTs.

2 Multiparty Session Types with Generalised Choice

In this section, we define global and local types. We explain how multiparty session types (MSTs) work and present a shortcoming of current MSTs. Our MSTs overcome this shortcoming by allowing a role to wait for messages coming from different senders. We define a new projection operation from global to local types: the projection represents global message exchanges from the perspective of a single role. The key to the new projection is a generalised *merge* operator that prevents confusion between messages from different senders.

2.1 Global and Local Types

We describe the syntax of global types following work by Honda et al. [36], Hu and Yoshida [38], and Scalas and Yoshida [52]. We focus on the core message-passing aspects of asynchronous MSTs and do not include features such as delegation.

▶ **Definition 1** (Syntax). Global types for MSTs *are defined by the grammar:*

$$G ::= 0 \mid \sum_{i \in I} \mathtt{p} \to \mathtt{q}_i : m_i . G_i \mid \mu t.\, G \mid t$$

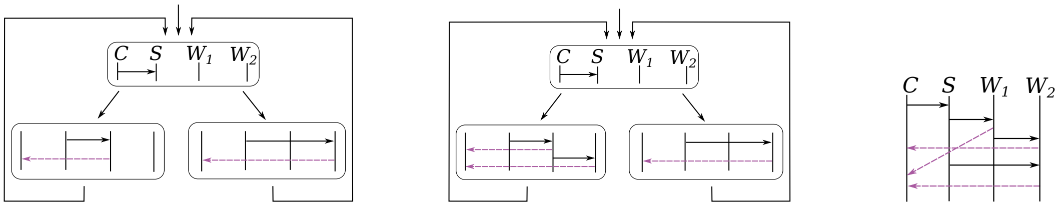
where $\mathtt{p}, \mathtt{q}_i$ *range over a set of roles* $\mathcal{P}$, m_i *over a set of messages* $\mathcal{V}$, *and* t *over type variables.*

Note that our definition of global types extends the standard syntax (see, e.g., [36]), which has a *directed choice* restriction, requiring that a sender must send messages to the same role along different branches of a choice. Our syntax $\sum_{i \in I} \mathtt{p} \to \mathtt{q}_i : m_i . G_i$ allows a sender to send messages to different roles along different branches (as in, e.g., [38]). For readability, we sometimes use the infix operator $+$ for choice, instead of $\sum$. When $|I| = 1$, we omit $\sum$.

In a global type, the send and the receive operations of a message exchange are specified atomically. An expression $\mathtt{p} \to \mathtt{q} : m$ represents two events: a *send* $\mathtt{p} \triangleright \mathtt{q}!m$ and a *receive* $\mathtt{q} \triangleleft \mathtt{p}?m$. We require the sender and receiver processes to be different: $\mathtt{p} \neq \mathtt{q}$. A *choice* ($\sum$) occurs at the sender role. Each branch of a choice needs to be uniquely distinguishable: $\forall i, j \in I.\, i \neq j \Rightarrow (\mathtt{q}_i, m_i) \neq (\mathtt{q}_j, m_j)$. The least fixed point operator encodes loops and we require recursion to be guarded, i.e., in $\mu t.\, G$, there is at least one message between μt and each t in G. Without loss of generality, we assume that all occurrences of t are bound and each bound variable t is distinct. As the recursion is limited to tail recursion, it is memoryless and generates regular sequences, so a global type can be interpreted as a regular language of message exchanges.

▶ **Example 2** (Load balancing). A simple load balancing scenario can be modelled with the global type: $\mu t.\ \mathsf{Client} \to \mathsf{Server} : \mathit{req}. + \begin{cases} \mathsf{Server} \to \mathsf{Worker}_1 : \mathit{req}.\ \mathsf{Worker}_1 \to \mathsf{Client} : \mathit{reply}.\ t \\ \mathsf{Server} \to \mathsf{Worker}_2 : \mathit{req}.\ \mathsf{Worker}_2 \to \mathsf{Client} : \mathit{reply}.\ t \end{cases}$

The least fixed point operator μ encodes a loop in which a client sends a request to a server. The server then non-deterministically forwards the request to one of two workers. The chosen worker handles the request and replies to the client. In this protocol, the server communicates with a different worker in each branch. Figure 1a shows this example as a high-level message sequence chart (HMSC). The timeline of roles is shown with vertical lines and the messages with horizontal arrows. Different message contents are represented by different styles of arrows. ⌟



(a) Load balancing. **(b)** Variant of load balancing. **(c)** Execution.

Figure 1 Load balancing and some variant with potential for confusion exemplified by an execution.

Next, we define local types, which specify a role's view of a protocol.

▶ **Definition 3** (Local types). *The* local types *for a role* p *are defined as:*

$$L ::= 0 \;\mid\; \bigoplus_{i\in I} \mathsf{q}_i!m_i.L_i \;\mid\; \mathop{\&}_{i\in I} \mathsf{q}_i?m_i.L_i \;\mid\; \mu t.L \;\mid\; t$$

where the internal choice ($\oplus$) *and external choice* ($\&$) *both respect* $\forall i,j \in I.\ i \neq j \Rightarrow (\mathsf{q}_i, m_i) \neq (\mathsf{q}_j, m_j)$*. As for global types, we assume every recursion variable is bound, each recursion operator* (μ) *uses a different identifier* t*, and we may omit* $\oplus$ *and* $\&$ *if* $|I| = 1$*.*

Note that a role can send to, resp. receive from, multiple roles in a choice: we generalise $\oplus_{i\in I}\, \mathsf{q}!m_i.L_i$ of standard MSTs to $\oplus_{i\in I}\, \mathsf{q}_i!m_i.L_i$ and $\&_{i\in I}\, \mathsf{q}?m_i.L_i$ to $\&_{i\in I}\, \mathsf{q}_i?m_i.L_i$.

▶ **Example 4.** We can give the following local types for Figure 1a:

$$\begin{aligned} \mathsf{Server}: &\quad \mu t.\, \mathsf{Client}?req.\, (\mathsf{Worker_1}!req.\, t \;\oplus\; \mathsf{Worker_2}!req.\, t) \\ \mathsf{Client}: &\quad \mu t.\, \mathsf{Server}!req.\, (\mathsf{Worker_1}?reply.\, t \;\&\; \mathsf{Worker_2}?reply.\, t) \\ \mathsf{Worker_i}: &\quad \mu t.\, \mathsf{Server}?req.\, \mathsf{Client}!reply.\, t \text{ for } i \in \{1,2\} \end{aligned}$$

Note that their structure, i.e., having a loop with at most two options, resembles the one of the global type in Example 2. ⌟

Our goal is to define a partial *projection* operation from a given global type to a local type for each role. If the projection is defined, we expect that the type is *implementable*. We shall show that the global type of Example 2 projects to the local types in Example 4. As a consequence, the global type is implementable. Intuitively, when each role in the example executes based on its local type, they agree on a unique global path in an unrolling of the global type. We formalise projection and soundness in Section 3. We note that existing projection operations, including the ones by Hu and Yoshida [38] as well as Scalas and Yoshida [52], reject the above global type as not implementable.

Notations and Assumptions. We write $\mathbf{G}$ for the global type we try to project. When traversing the global type $\mathbf{G}$, we use G for the current term (which is a subterm of $\mathbf{G}$). To simplify the notation, we assume that the index i of a choice uniquely determines the sender and the message $\mathsf{q}_i?m_i$. Using this notation, we write $I \cap J$ to select the set of choices with identical sender and message value and $I \setminus J$ to select the alternatives present in I but not in J. When looking at send and receive events in a global setting we write $\mathsf{p} \triangleright \mathsf{q}!m$ for p sending to q and $\mathsf{q} \triangleleft \mathsf{p}?m$ for q receiving from p.

In later definitions, we unfold the recursion in types. We could get the unfolding through a congruence relation. However, this requires dealing with infinite structures, which makes some definitions not effective. Instead, we precompute the map from each recursion variable t to its unfolding. For a given global type, let $get\mu$ be a function that returns a map from t to G for each subterm $\mu t.\, G$. Recall, each t in a type is different. $get\mu$ is defined as follows:

$$get\mu(0) := [\,] \qquad get\mu(t) := [\,] \qquad get\mu(\mu t.G) := [t \mapsto G] \cup get\mu(G)$$

$$get\mu(\sum_{i \in I} \mathtt{p} \to \mathtt{q}_i : m_i.G_i) := \bigcup_{i \in I} get\mu(G_i)$$

We write $get\mu_{\mathbf{G}}$ as shorthand for the map returned by $get\mu(\mathbf{G})$.

2.2 Generalised Projection and Merge

We now define a partial *projection* operation that projects a global type on to each role. The projection on to a role $\mathtt{r}$ is a local type and keeps only $\mathtt{r}$'s actions. Intuitively, it gives the "local view" of message exchanges performed by $\mathtt{r}$. While projecting, non-determinism may arise due to choices that $\mathtt{r}$ does not observe directly. In this case, the different branches are merged using a partial *merge* operator ($\sqcap$). The merge operator checks that a role, which has not yet learned the outcome of a choice, only performs actions that are allowed in all possible branches. The role can perform branch-specific actions after it has received a message that resolves the choice. For a role that is agnostic to the choice, i.e., behaves the same on all the branches, the merge allows the role to proceed as if the choice does not exist.

So far, the idea follows standard asynchronous MSTs. What distinguishes our new projection operator from prior ones (e.g., [38, 52]), is that we allow a role to learn which branch has been taken through messages received from different senders. This generalisation is non-trivial. When limiting the reception to messages from a single role, one can rely on the FIFO order provided by the corresponding channel. However, messages coming from different sources are only partially ordered. Thus, unlike previous approaches, our merge operator looks at the result of a *causality analysis* on the global type to make sure that this partial ordering cannot introduce any confusion.

▶ **Example 5** (Intricacies of generalising projection)**.** We demonstrate that a straightforward generalisation of existing projection operators can lead to unsoundness. Consider a *naive* projection that merges branches with internal choice if they are equal, and for receives, simply always merges external choices – also from different senders. In addition, it removes empty loops. For Figure 1a, this naive projection yields the expected local types presented in Example 4. We show that naive projection can be unsound. Figure 1b shows a variant of load balancing, for which naive projection yields the following local types:

$$\begin{aligned}
\mathsf{Server}: &\quad \mu t.\, \mathsf{Client}?req.\, (\mathsf{Worker}_1!req.\, t \ \oplus\ \mathsf{Worker}_2!req.\, t)\\
\mathsf{Client}: &\quad \mu t.\, \mathsf{Server}!req.\, (\mathsf{Worker}_1?reply.\, \mathsf{Worker}_2?reply.\, t \ \&\ \mathsf{Worker}_2?reply.\, t)\\
\mathsf{Worker}_1: &\quad \mu t.\, \mathsf{Server}?req.\, \mathsf{Worker}_2!req.\, t\\
\mathsf{Worker}_2: &\quad \mu t.\, (\mathsf{Worker}_1?req.\, \mathsf{Client}{\triangleright}!reply.\, t \ \&\ \mathsf{Server}?req.\, \mathsf{Client}{\triangleright}!reply.\, t)
\end{aligned}$$

Unfortunately, the global type is not implementable. The problem is that, for the *Client*, the two messages on its left branch are not causally related. Consider the execution prefix in Figure 1c which is not specified in the global type. The Server decided to first take the left (L) and then the right (R) branch. For Server, the order LR is obvious from its events and the same applies for Worker$_2$. For Worker$_1$, every possible order R^*LR^* is plausible as it does not have events in the right branch. Since LR belongs to the set of plausible orders, there is no confusion. Now, the messages from the two workers to the client are independent and, therefore, can be received in any order. If the client receives *Worker*$_2$*?reply* first, then its local view is consistent with the choice RL as the order of branches. This can lead to confusion and, thus, execution prefixes which are not specified in the global type. ⌟

We shall now define our generalised projection operation. To identify confusion as above, we keep track of causality between messages. We determine what messages a role *could* receive at a given point in the global type through an *available messages* analysis. Tracking causality needs to be done at the level of the global type. We look for chains of dependent messages and we also need to unfold loops. Fortunately, since we only check for the presence or absence of some messages, it is sufficient to unfold each recursion at most once.

Projection and Interferences from Independent Messages. The challenge of projecting a global type lies in resolving the non-determinism introduced by having only the endpoint view. Example 5 shows that in order to decide if a choice is safe, we need to know which messages can arrive at the same time. To enable this, we annotate local types with the messages that could be received at that point in the protocol. We call these *availability annotated local types* and write them as $AL = \langle L, \mathit{Msg}\rangle$ where L is a local type and Msg is a set of messages. This signifies that when a role has reached AL, the messages in Msg can be present in the communication channels. We annotate types using the grammar for local types (Definition 3), where each subterm is annotated. To recover a local type, we erase the annotation, i.e., recursively replace each $AL = \langle L, \mathit{Msg}\rangle$ by L. The projection internally uses annotated types.

The projection of $\mathbf{G}$ on to $\mathtt{r}$, written $\mathbf{G}{\upharpoonright_{\mathtt{r}}}$, traverses $\mathbf{G}$ to erase the operations that do not involve $\mathtt{r}$. During this phase, we also compute the messages that $\mathtt{r}$ may receive. The function $\mathrm{avail}(B, T, G)$ computes the set of messages that other roles can send while $\mathtt{r}$ has not yet learned the outcome of the choice. This set depends on B, the set of *blocked* roles, i.e., the roles which are waiting to receive a message and hence cannot move; T, the set of recursion variables we have already visited; and G, the subterm in $\mathbf{G}$ at which we compute the available messages. We defer the definition of $\mathrm{avail}(B, T, G)$ to later in this section.

Empty Paths Elimination. When projecting, there may be paths and loops where a role neither sends nor receives a message, e.g., the right loop in Example 2 for Worker_1. Such paths can be removed during projection. Even if conceptually simple, the notational overhead impedes understandability of how our message availability analysis is used. Therefore, we first focus on the message availability analysis and define a projection operation that does not account for empty paths elimination. After defining the merge operator $\sqcap$, we give the full definition of our generalised projection operation.

▶ **Definition 6** (Projection without empty paths elimination). *The* projection without empty paths elimination $\mathbf{G}{\upharpoonright_{\mathtt{r}}}$ *of a global type* $\mathbf{G}$ *on to a role* $\mathtt{r} \in \mathcal{P}$ *is an availability annotated local type inductively defined as:*

$$t{\upharpoonright_{\mathtt{r}}} := \langle t, \mathrm{avail}(\{\mathtt{r}\}, \{t\}, \mathit{get}\mu_{\mathbf{G}}(t))\rangle \qquad\qquad 0{\upharpoonright_{\mathtt{r}}} := \langle 0, \emptyset\rangle$$

$$(\mu t.G){\upharpoonright_{\mathtt{r}}} := \begin{cases} \langle \mu t.(G{\upharpoonright_{\mathtt{r}}}), \mathrm{avail}(\{\mathtt{r}\}, \{t\}, G)\rangle & \textit{if } G{\upharpoonright_{\mathtt{r}}} \neq \langle t, \_\rangle \\ \langle 0, \emptyset\rangle & \textit{otherwise} \end{cases}$$

$$\Big(\textstyle\sum_{i\in I} \mathtt{p}\!\to\!\mathtt{q}_i : m_i.G_i\Big){\upharpoonright_{\mathtt{r}}} := \begin{cases} \langle \oplus_{i\in I}\, \mathtt{q}_i!m_i.(G_i{\upharpoonright_{\mathtt{r}}}), \bigcup_{i\in I} \mathrm{avail}(\{\mathtt{q}_i, \mathtt{r}\}, \emptyset, G_i)\rangle & \textit{if } \mathtt{r} = \mathtt{p} \\ \sqcap \left(\begin{array}{l} \langle \&_{i\in I_{[=\mathtt{r}]}}\, \mathtt{p}?m_i.(G_i{\upharpoonright_{\mathtt{r}}}), \bigcup_{i\in I_{[=\mathtt{r}]}} \mathrm{avail}(\{\mathtt{r}\}, \emptyset, G_i)\rangle \\ \sqcap_{i\in I_{[\neq\mathtt{r}]}}\, G_i{\upharpoonright_{\mathtt{r}}} \end{array} \right) & \textit{otherwise} \end{cases}$$
$$\textit{where } I_{[=\mathtt{r}]} := \{i \in I \mid \mathtt{q}_i = \mathtt{r}\} \textit{ and } I_{[\neq\mathtt{r}]} := \{i \in I \mid \mathtt{q}_i \neq \mathtt{r}\}$$

A global type $\mathbf{G}$ *is said to be* projectable *if* $\mathbf{G}{\upharpoonright_{\mathtt{r}}}$ *is defined for every* $\mathtt{r} \in \mathcal{P}$.

Projection erases events not relevant to $\mathtt{r}$ by a recursive traversal of the global type; however, at a choice not involving $\mathtt{r}$, it has to ensure that either $\mathtt{r}$ is indifferent to the outcome of the choice or it indirectly receives enough information to distinguish the outcome.

This is managed by the merge operator $\sqcap$ and the use of available messages. The merge operator takes as arguments a sequence of availability annotated local types. Our merge operator generalises the full merge by Scalas and Yoshida [52]. When faced with choice, it only merges receptions that cannot interfere with each other. For the sake of clarity, we define only the binary merge. As the operator is commutative and associative, it generalises to a set of branches I. When I is a singleton, the merge just returns that one branch.

▶ **Definition 7** (Merge operator $\sqcap$). *Let $\langle L_1, \mathit{Msg}_1\rangle$ and $\langle L_2, \mathit{Msg}_2\rangle$ be availability annotated local types for a role $\mathtt{r}$. $\langle L_1, \mathit{Msg}_1\rangle \sqcap \langle L_2, \mathit{Msg}_2\rangle$ is defined by cases, as follows:*

- $\langle L_1, \mathit{Msg}_1 \cup \mathit{Msg}_2\rangle \quad \text{if } L_1 = L_2$
- $\langle \mu t_1.(AL_1 \sqcap AL_2[t_2/t_1]), \mathit{Msg}_1 \cup \mathit{Msg}_2\rangle \quad \text{if } L_1 = \mu t_1.AL_1,\ L_2 = \mu t_2.AL_2$
- $\langle \oplus_{i\in I}\, \mathtt{q}_i!m_i.(AL_{1,i} \sqcap AL_{2,i}), \mathit{Msg}_1 \cup \mathit{Msg}_2\rangle \quad \text{if } \begin{cases} L_1 = \oplus_{i\in I}\, \mathtt{q}_i!m_i.AL_{1,i}, \\ L_2 = \oplus_{i\in I}\, \mathtt{q}_i!m_i.AL_{2,i} \end{cases}$
- $\begin{array}{ll}\langle & \&_{i\in I\setminus J}\, \mathtt{q}_i?m_i.AL_{1,i} \quad \& \\ & \&_{i\in I\cap J}\, \mathtt{q}_i?m_i.(AL_{1,i} \sqcap AL_{2,i}) \quad \& \\ & \&_{i\in J\setminus I}\, \mathtt{q}_i?m_i.AL_{2,i} \quad , \\ & \mathit{Msg}_1 \cup \mathit{Msg}_2 \quad \rangle \end{array} \quad \text{if } \begin{cases} L_1 = \&_{i\in I}\, \mathtt{q}_i?m_i.AL_{1,i}, \\ L_2 = \&_{i\in J}\, \mathtt{q}_i?m_i.AL_{2,i}, \\ \forall i \in I \setminus J.\, \mathtt{r} \triangleleft \mathtt{q}_i?m_i \notin \mathit{Msg}_2, \\ \forall i \in J \setminus I.\, \mathtt{r} \triangleleft \mathtt{q}_i?m_i \notin \mathit{Msg}_1 \end{cases}$

When no condition applies, the merge and, thus, the projection are undefined.[1]

The important case of the merge is the external choice. Here, when a role can potentially receive a message that is unique to a branch, it checks that the message cannot be available in another branch so actually being able to receive this message uniquely determines which branch was taken by the role to choose. For the other cases, a role can postpone learning the branch as long as the actions on both branches are the same.

Adding Empty Paths Elimination. The preliminary version of projection requires every role to have at least one event in each branch of a loop and, thus, rejects examples where a role has no event in some loop branch. Such paths can be eliminated. However, determining such empty paths cannot be done on the level of the merge operator but only when projecting. To this end, we introduce an additional parameter E for the generalised projection: E contains those variables t for which $\mathtt{r}$ has not observed any message send or receive event since μt.

▶ **Definition 8** (Generalised projection – with empty paths elimination). *The* projection $\mathbf{G}{\upharpoonright}^{E}_{\mathtt{r}}$ *of a global type $\mathbf{G}$ on to a role $\mathtt{r} \in \mathcal{P}$ is an availability annotated local type which is inductively defined as follows:*

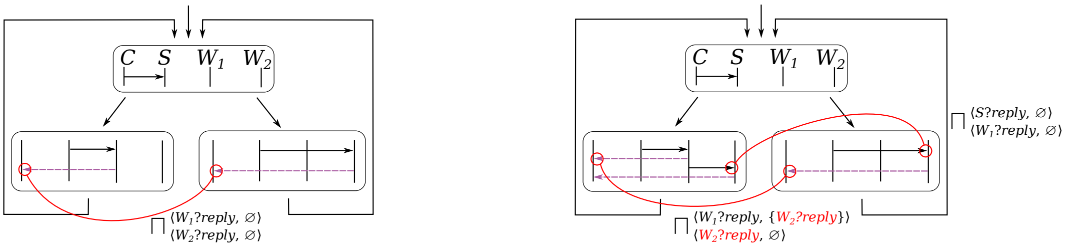
$$t{\upharpoonright}^{E}_{\mathtt{r}} := \langle t, \mathrm{avail}(\{R\}, \{t\}, \mathit{get}\mu_{\mathbf{G}}(t))\rangle \qquad 0{\upharpoonright}^{E}_{\mathtt{r}} := \langle 0, \emptyset\rangle$$

$$(\mu t.G){\upharpoonright}^{E}_{\mathtt{r}} := \begin{cases} \langle \mu t.(G{\upharpoonright}^{E\cup\{t\}}_{\mathtt{r}}), \mathrm{avail}(\{R\}, \{t\}, G)\rangle & \text{if } G{\upharpoonright}^{E\cup\{t\}}_{\mathtt{r}} \neq \langle t, \_\rangle \\ \langle 0, \emptyset\rangle & \text{otherwise} \end{cases}$$

$$\left(\textstyle\sum_{i\in I} \mathtt{p}{\to}\mathtt{q}_i : m_i.G_i\right){\upharpoonright}^{E}_{\mathtt{r}} := \begin{cases} \langle \oplus_{i\in I}\, \mathtt{q}_i!m_i.(G_i{\upharpoonright}^{\emptyset}_{\mathtt{r}}), \bigcup_{i\in I} \mathrm{avail}(\{\mathtt{q}_i, \mathtt{r}\}, \emptyset, G_i)\rangle & \text{if } \mathtt{r} = \mathtt{p} \\ \sqcap\left(\begin{array}{l} \langle \&_{i\in I_{[=\mathtt{r}]}}\, \mathtt{p}?m_i.(G_i{\upharpoonright}^{\emptyset}_{\mathtt{r}}), \bigcup_{i\in I_{[=\mathtt{r}]}} \mathrm{avail}(\{\mathtt{r}\}, \emptyset, G_i)\rangle \\ \sqcap_{i\in I_{[\neq\mathtt{r}]} \wedge \forall t\in E.\, G_i{\upharpoonright}^{E}_{\mathtt{r}} \neq \langle t, \_\rangle}\ G_i{\upharpoonright}^{E}_{\mathtt{r}} \end{array} \right) & \text{otherwise} \end{cases}$$
$$\text{where } I_{[=\mathtt{r}]} := \{i \in I \mid \mathtt{q}_i = \mathtt{r}\} \text{ and } I_{[\neq\mathtt{r}]} := \{i \in I \mid \mathtt{q}_i \neq \mathtt{r}\}$$

Since the merge operator $\sqcap$ is partial, the projection may be undefined. We use $\mathbf{G}{\upharpoonright}_{\mathtt{r}}$ as shorthand for $\mathbf{G}{\upharpoonright}^{\emptyset}_{\mathtt{r}}$ and only consider the generalised projection with empty paths elimination from now on. A global type $\mathbf{G}$ is called projectable *if $\mathbf{G}{\upharpoonright}_{\mathtt{r}}$ is defined for every role $\mathtt{r} \in \mathcal{P}$.*

[1] When we use the n-ary notation $\sqcap_{i\in I}$ and $|I| = 0$, we implicitly omit this part. Note that this can only happen if $\mathtt{r}$ is the receiver among all branches for some choice so there is either another local type to merge with, or the projection is undefined anyway.



(a) Merging for Figure 1a.

(b) Merging for Figure 1b.

Figure 2 Availability annotated types for merging on the two examples. The red lines connect the receptions that get merged during projection. The annotations only show the receiver's messages.

We highlight the differences for the empty paths elimination. Recall that E contains all recursion variables from which the role $\mathtt{r}$ has not encountered any events. To guarantee this, for the case of recursion $\mu t.\, G$, the (unique) variable t is added to the current set E, while the parameter turns to the empty set $\emptyset$ as soon as $\mathtt{r}$ encounters an event. The previous steps basically constitute the necessary bookkeeping. The actual elimination is achieved with the condition $\forall t \in E.\, G_i{\restriction}^{E}_{\mathtt{r}} \neq \langle t, \_ \rangle$ which filters all branches without events of role $\mathtt{r}$.

Other works [38, 17] achieve this with *connecting actions*, marking non-empty paths. Like classical MSTs, we do not include such explicit actions. Still, we can automatically eliminate such paths in contrast to previous work.

Computing Available Messages. Finally, the function avail is computed recursively:

$$\mathrm{avail}(B, T, 0) := \emptyset$$

$$\mathrm{avail}(B, T, \mu t.G) := \mathrm{avail}(B, T \cup \{t\}, G)$$

$$\mathrm{avail}(B, T, t) := \begin{cases} \emptyset & \text{if } t \in T \\ \mathrm{avail}(B, T \cup \{t\}, get\mu_{\mathbf{G}}(t)) & \text{if } t \notin T \end{cases}$$

$$\mathrm{avail}(B, T, \textstyle\sum_{i \in I} \mathtt{p} \to \mathtt{q}_i : m_i.G_i) := \begin{cases} \bigcup_{i \in I, m \in \mathcal{V}} (\mathrm{avail}(B, T, G_i) \setminus \{\mathtt{q}_i \triangleleft \mathtt{p}?m\}) \cup \{\mathtt{q}_i \triangleleft \mathtt{p}?m_i\} & \text{if } \mathtt{p} \notin B \\ \bigcup_{i \in I} \mathrm{avail}(B \cup \{\mathtt{q}_i\}, T, G_i) & \text{if } \mathtt{p} \in B \end{cases}$$

Since all channels are FIFO, we only keep the first possible message in each channel. The fourth case discards messages not at the head of the channel.

Our projection is different from the one of Scalas and Yoshida [52], not just because our syntax is more general. It also represents a shift in paradigm. In their work, the full merge works only on local types. No additional knowledge is required. This is possible because their type system limits the flexibility of communication. Since we allow more flexible communication, we need to keep some information, in form of available messages, about the possible global executions for the merge operator.

▶ **Example 9.** Let us explain how our projection operator catches the problem in $\mathbf{G}{\restriction}_{Client}$ of Figure 1b. Figure 2 shows the function of available messages during the projection for Figure 1a and Figure 1b. In Figure 2a, the messages form chains, i.e., except for the role making the choice, a role only sends in reaction to another message. Therefore, only a single message is available at each reception and the protocol is projectable. On the other hand, in Figure 2b both replies are available and, therefore, the protocol is not projectable.

Here are the details of the projection for Figure 2b. If not needed, we omit the availability annotations for readability. Recall that Client receives *reply* from Worker$_2$ in the left branch, which is also present in the right branch. Let us denote the two branches as follows:

$$G_1 := \mathsf{Server} \to \mathsf{Worker}_1 : req.\ \mathsf{Worker}_1 \to \mathsf{Worker}_2 : req.\ \mathsf{Worker}_2 \to \mathsf{Client} : reply.\ t, \text{ and}$$
$$G_2 := \mathsf{Server} \to \mathsf{Worker}_2 : req.\ \mathsf{Worker}_2 \to \mathsf{Client} : reply.\ t$$

Since the first message in G_1 does not involve Client, the projection descends and we compute:

$$\begin{aligned} \mathbf{G}\restriction_{\mathsf{Client}} &= \langle \mu t.(\langle \mathsf{Worker}_1 ? \mathit{reply}.\,(G'_1\restriction_{\mathsf{Client}}),\ \mathrm{avail}(\{\mathsf{Client}\}, \emptyset, G'_1)\rangle \\ &\qquad \sqcap \langle \mathsf{Worker}_2 ? \mathit{reply}.\,(G'_2\restriction_{\mathsf{Client}}),\ \mathrm{avail}(\{\mathsf{Client}\}, \emptyset, G'_2)\rangle), \_\rangle \\ \text{where} \quad & G'_1 = \mathsf{Worker}_1 \to \mathsf{Worker}_2 : \mathit{req}.\ \mathsf{Worker}_2 \to \mathsf{Client} : \mathit{reply}.\ t \text{ and } G'_2 = t. \end{aligned}$$

For this, we compute $\mathrm{avail}(\{\mathsf{Client}\}, \emptyset, G'_1) = \emptyset$ and $\mathrm{avail}(\{\mathsf{Client}\}, \emptyset, G'_2) = \{\mathsf{Worker}_2 \triangleleft \mathsf{Worker}_1 ? \mathit{req}, \mathsf{Client} \triangleleft \mathsf{Worker}_2 ? \mathit{reply}\}$ and see that the conditions are not satisfied. Indeed, $\mathsf{Client} \triangleleft \mathsf{Worker}_2 ? \mathit{reply} \in \mathrm{avail}(\{\mathsf{Client}\}, \emptyset, G'_2)$. Thus, the projection is undefined. ⌟

3 Type Soundness

We now show a soundness theorem for generalised projection; roughly, a projectable global type can be implemented by communicating state machines in a distributed way. Our proof uses automata-theoretic techniques from the theory of MSCs. We assume familiarity with the basics of formal languages.

As our running example shows, a protocol implementation often cannot enforce the event ordering specified in the type but only a weaker order. In this section, we capture both notions through a type language and an execution language.

3.1 Type Languages

A *state machine* $A = (Q, \Sigma, \delta, q_0, F)$ consists of a finite set Q of states, an alphabet Σ, a transition relation $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$, an initial state $q_0 \in Q$, and a set $F \subseteq Q$ of final states. We write $q \xrightarrow{x} q'$ for $(q, x, q') \in \delta$. We define the runs and traces in the standard way. A run is maximal if it is infinite or if it ends at a final state. The *language* $\mathcal{L}(A)$ is the set of (finite or infinite) maximal traces. The projection $A\Downarrow_\Delta$ of a state machine is its projection to a sub-alphabet $\Delta \subseteq \Sigma$ obtained by replacing all letters in $\Sigma \setminus \Delta$ with ε-transitions. It accepts the language $\mathcal{L}(A)\Downarrow_\Delta = \{w\Downarrow_\Delta \mid w \in \mathcal{L}(A)\}$.

▶ **Definition 10** (Type language for global types). *The semantics of a global type* $\mathbf{G}$ *is given as a regular language. We construct a state machine* $\mathsf{GAut}(\mathbf{G})$ *using an auxiliary state machine* $M(\mathbf{G})$*. First, we define* $M(\mathbf{G}) = (Q_{M(\mathbf{G})}, \Sigma_{sync}, \delta_{M(\mathbf{G})}, q_{0M(\mathbf{G})}, F_{M(\mathbf{G})})$ *where*

- $Q_{M(\mathbf{G})}$ *is the set of all syntactic subterms in* $\mathbf{G}$ *together with the term* 0*,*
- $\Sigma_{sync} = \{\mathsf{p} \to \mathsf{q} : m \mid \mathsf{p}, \mathsf{q} \in \mathcal{P} \text{ and } m \in \mathcal{V}\}$*,*
- $\delta_{M(\mathbf{G})}$ *is the smallest set containing* $(\sum_{i \in I} \mathsf{p} \to \mathsf{q}_i : m_i.G_i, \mathsf{p} \to \mathsf{q}_i : m_i, G_i)$ *for each* $i \in I$*, as well as* $(\mu t.G', \varepsilon, G')$ *and* $(t, \varepsilon, \mu t.G')$ *for each subterm* $\mu t.G'$ *of* $\mathbf{G}$*,*
- $q_{0M(\mathbf{G})} = \mathbf{G}$ *and* $F_{M(\mathbf{G})} = \{0\}$*.*

Next, we expand each message $\mathsf{p} \to \mathsf{q} : m$ *into two events,* $\mathsf{p} \triangleright \mathsf{q} ! m$ *followed by* $\mathsf{q} \triangleleft \mathsf{p} ? m$*. We define* $\mathsf{GAut}(\mathbf{G}) = (Q_{\mathsf{GAut}(\mathbf{G})}, \Sigma_{\mathsf{GAut}(\mathbf{G})}, \delta_{\mathsf{GAut}(\mathbf{G})}, q_{0\mathsf{GAut}(\mathbf{G})}, F_{\mathsf{GAut}(\mathbf{G})})$ *as follows:*

- $Q_{\mathsf{GAut}(\mathbf{G})} = Q_{M(\mathbf{G})} \cup (Q_{M(\mathbf{G})} \times \Sigma_{sync} \times Q_{M(\mathbf{G})})$*,*
- $\Sigma_{\mathsf{GAut}(\mathbf{G})} = \{\mathsf{p} \triangleright \mathsf{q} ! m \mid \mathsf{p}, \mathsf{q} \in \mathcal{P}, m \in \mathcal{V}\} \cup \{\mathsf{q} \triangleleft \mathsf{p} ? m \mid \mathsf{p}, \mathsf{q} \in \mathcal{P}, m \in \mathcal{V}\}$*,*
- $\delta_{\mathsf{GAut}(\mathbf{G})}$ *is the smallest set containing the transitions* $(s, \mathsf{p} \triangleright \mathsf{q} ! m, (s, \mathsf{p} \to \mathsf{q} : m, s'))$ *and* $((s, \mathsf{p} \to \mathsf{q} : m, s'), \mathsf{q} \triangleleft \mathsf{p} ? m, s'))$ *for each transition* $(s, \mathsf{p} \to \mathsf{q} : m, s') \in \delta_{M(\mathbf{G})}$*,*
- $q_{0\mathsf{GAut}(\mathbf{G})} = q_{0M(\mathbf{G})}$ *and* $F_{\mathsf{GAut}(\mathbf{G})} = F_{M(\mathbf{G})}$*.*

The type language $\mathcal{L}(\mathbf{G})$ *of a global type* $\mathbf{G}$ *is given by* $\mathcal{L}(\mathsf{GAut}(\mathbf{G}))$*.*

▶ **Definition 11** (Type language for local types). *Given a local type L for $\mathtt{p}$, we construct a state machine $\mathsf{LAut}(L) = (Q, \Sigma_{\mathtt{p}}, \delta, q_0, F)$ where*

- *Q is the set of all syntactic subterms in L,*
- $\Sigma_{\mathtt{p}} = \{\mathtt{p} \triangleright \mathtt{q}!m \mid \mathtt{q} \in \mathcal{P}, m \in \mathcal{V}\} \cup \{\mathtt{p} \triangleleft \mathtt{q}?m \mid \mathtt{p}, \mathtt{q} \in \mathcal{P}, m \in \mathcal{V}\}$,
- *δ is the smallest set containing $(\oplus_{i \in I}\, \mathtt{q}_i!m_i.L_i, \mathtt{p} \triangleright \mathtt{q}_i!m_i, L_i)$ and $(\&_{i \in I}\, \mathtt{q}_i?m_i.L_i, \mathtt{p} \triangleleft \mathtt{q}_i?m_i, L_i)$ for each $i \in I$, as well as $(\mu t.L', \varepsilon, L')$ and $(t, \varepsilon, \mu t.L')$ for each $\mu t.L'$ in L,*
- *$q_0 = L$ and $F = \{0\}$ if 0 is a subterm of L, and empty otherwise.*

We define the type language *of L as language of this automaton: $\mathcal{L}(L) = \mathcal{L}(\mathsf{LAut}(L))$.*

3.2 Implementability

An *implementation* consists of a set of state machines, one per role, communicating with each other through asynchronous messages and pairwise FIFO channels. We use communicating state machines (CSMs) [10] as our formal model. A CSM $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}}$ consists of a set of state machines $A_{\mathtt{p}}$, one for each $\mathtt{p} \in \mathcal{P}$ over the alphabet of message sends and receives. Communication between machines happens asynchronously through FIFO channels. The semantics of a CSM is a language $\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}})$ of maximal traces over the alphabet of message sends and receives satisfying the FIFO condition on channels. A CSM is *deadlock free* if every trace can be extended to a maximal trace. We omit the (standard) formal definition of CSMs (see Appendix A for details).

Indistinguishability Relation. In the type language of a global type, every send event is always immediately succeeded by its receive event. However, in a CSM, other independent events may occur between the send and the receipt and there is no way to force the order specified by the type language. To capture this phenomenon formally, we define a family of *indistinguishability relations* $\sim_i \subseteq \Sigma^* \times \Sigma^*$, for $i \geq 0$ and $\Sigma = \Sigma_{\mathsf{GAut}(\mathbf{G})}$, as follows. For all $w \in \Sigma^*$, we have $w \sim_0 w$. For $i = 1$, we define:

(1) If $\mathtt{p} \neq \mathtt{r}$, then $w.\mathtt{p} \triangleright \mathtt{q}!m.\mathtt{r} \triangleright \mathtt{s}!m'.u \;\sim_1\; w.\mathtt{r} \triangleright \mathtt{s}!m'.\mathtt{p} \triangleright \mathtt{q}!m.u$.

(2) If $\mathtt{q} \neq \mathtt{s}$, then $w.\mathtt{q} \triangleleft \mathtt{p}?m.\mathtt{s} \triangleleft \mathtt{r}?m'.u \;\sim_1\; w.\mathtt{s} \triangleleft \mathtt{r}?m'.\mathtt{q} \triangleleft \mathtt{p}?m.u$.

(3) If $\mathtt{p} \neq \mathtt{s} \wedge (\mathtt{p} \neq \mathtt{r} \vee \mathtt{q} \neq \mathtt{s})$, then $w.\mathtt{p} \triangleright \mathtt{q}!m.\mathtt{s} \triangleleft \mathtt{r}?m'.u \;\sim_1\; w.\mathtt{s} \triangleleft \mathtt{r}?m'.\mathtt{p} \triangleright \mathtt{q}!m.u$.

(4) If $|w{\Downarrow}_{\mathtt{p} \triangleright \mathtt{q}!\_}| > |w{\Downarrow}_{\mathtt{q} \triangleleft \mathtt{p}?\_}|$, then $w.\mathtt{p} \triangleright \mathtt{q}!m.\mathtt{q} \triangleleft \mathtt{p}?m'.u \;\sim_1\; w.\mathtt{q} \triangleleft \mathtt{p}?m'.\mathtt{p} \triangleright \mathtt{q}!m.u$.

We refer to the proof of Lemma 21 in Appendix B for further details on the conditions for swapping events. Let w, w', w'' be sequences of events s.t. $w \sim_1 w'$ and $w' \sim_i w''$ for some i. Then, $w \sim_{i+1} w''$. We define $w \sim u$ if $w \sim_n u$ for some n. It is straightforward that $\sim$ is an equivalence relation. Define $u \preceq_{\sim} v$ if there is $w \in \Sigma^*$ such that $u.w \sim v$. Observe that $u \sim v$ iff $u \preceq_{\sim} v$ and $v \preceq_{\sim} u$. To extend $\sim$ to infinite words, we follow the approach of Gastin [27]. For infinite words $u, v \in \Sigma^\omega$, we define $u \preceq^{\omega}_{\sim} v$ if for each finite prefix u' of u, there is a finite prefix v' of v such that $u' \preceq_{\sim} v'$. Define $u \sim v$ iff $u \preceq^{\omega}_{\sim} v$ and $v \preceq^{\omega}_{\sim} u$.

We lift the equivalence relation $\sim$ on words to languages:

For a language Λ, we define $\mathcal{C}^{\sim}(\Lambda) = \left\{ w' \mid \bigvee \begin{array}{l} w' \in \Sigma^* \wedge \exists w \in \Sigma^*.\ w \in \Lambda \text{ and } w' \sim w \\ w' \in \Sigma^\omega \wedge \exists w \in \Sigma^\omega.\ w \in \Lambda \text{ and } w' \preceq^{\omega}_{\sim} w \end{array} \right\}$.

For the infinite case, we take the downward closure w.r.t. $\preceq^{\omega}_{\sim}$. Unlike [27, Definition 2.1], our closure operator is asymmetric. Consider the protocol $(\mathtt{p} \triangleright \mathtt{q}!m.\ \mathtt{q} \triangleleft \mathtt{p}?m)^\omega$. Since we do not make any fairness assumption on scheduling, we need to include in the closure the execution where only the sender is scheduled, i.e., $(\mathtt{p} \triangleright \mathtt{q}!m)^\omega \preceq^{\omega}_{\sim} (\mathtt{p} \triangleright \mathtt{q}!m.\ \mathtt{q} \triangleleft \mathtt{p}?m)^\omega$.

▶ **Example 12** (Indistinguishability relation $\sim$ by examples)**.** The four rules for $\sim_1$ present conditions under which two adjacent events in an execution (prefix) can be swapped. These conditions are designed such that they characterise possible changes in an execution (prefix) which cannot be recognised by any CSM. To be precise, if w is recognised by some CSM $\{\!\!\{A_\mathtt{p}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ and $w' \sim_1 w$ holds, then w' is also recognised by $\{\!\!\{A_\mathtt{p}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$. In this example, we illustrate the intuition behind these rules.

For the remainder of this example, the *active role* of an event is the receiver of a receive event and the sender of a send event. Visually, the active role is always the first role in an event. In addition, we assume that variables do not alias, i.e., two roles or messages with different names are different.

Two send events (or two receive events) can be swapped if the active roles are distinct because there cannot be any dependency between two such events which do occur next to each other in an execution. For send events, the 1st rule, thus, admits $\mathtt{p} \triangleright \mathtt{r}!m.\ \mathtt{q} \triangleright \mathtt{r}!m \sim_1 \mathtt{q} \triangleright \mathtt{r}!m.\ \mathtt{p} \triangleright \mathtt{r}!m$ even though the receiver is the same. In contrast, the corresponding receive events cannot be swapped: $\mathtt{r} \triangleleft \mathtt{p}?m.\ \mathtt{r} \triangleleft \mathtt{q}?m \not\sim_1 \mathtt{r} \triangleleft \mathtt{q}?m.\ \mathtt{r} \triangleleft \mathtt{p}?m$. Note that the 1st rule is the only one with which two send events can be swapped while the 2nd rule is the only one for receive events so indeed no rule applies for the last case.

The 3rd rule allows one send and one receive event to be swapped if either both senders or both receivers are different – in addition to the requirement that both active roles are different. For instance, it admits $\mathtt{p} \triangleright \mathtt{r}!m.\ \mathtt{q} \triangleleft \mathtt{r}?m \sim_1 \mathtt{q} \triangleleft \mathtt{r}?m.\ \mathtt{p} \triangleright \mathtt{r}!m$. However, it does not admit two swap $\mathtt{p} \triangleright \mathtt{q}!m.\ \mathtt{q} \triangleleft \mathtt{p}?m \not\sim_1 \mathtt{q} \triangleleft \mathtt{p}?m.\ \mathtt{p} \triangleright \mathtt{q}!m$. This is reasonable since the send event could be the one which emits m in the corresponding channel. In this execution prefix, this is in fact the case since there have been no events before, but in general one needs to incorporate the context to understand whether this is the case. The 4th rule does this and therefore admits swapping the same events when appended to some prefix: $\mathtt{p} \triangleright \mathtt{q}!m.\mathtt{p} \triangleright \mathtt{q}!m.\ \mathtt{q} \triangleleft \mathtt{p}?m \not\sim_1 \mathtt{p} \triangleright \mathtt{q}!m.\mathtt{q} \triangleleft \mathtt{p}?m.\ \mathtt{p} \triangleright \mathtt{q}!m$. Then, the FIFO order of channels ensures that the first message will be received first and the 2nd send event can happen after the reception of the 1st message. ⌟

▶ **Example 13** (Load balancing revisited)**.** Let us consider the execution with confusion in Figure 1c. Compared to a synchronous execution, we need to delay the reception $C \triangleleft W_1?\mathit{reply}$ to come after the first $C \triangleleft W_2?\mathit{reply}$. Using the 2nd and 3rd cases of $\sim$ we can move $C \triangleleft W_1?\mathit{reply}$ across the communications between the two workers. Finally, we use the 3rd case again to swap $C \triangleleft W_1?\mathit{reply}$ and $W_2 \triangleright C!\mathit{reply}$ to get the desired sequence. ⌟

This example shows that $\sim$ does not change the order of send and receive events of a single role. Thus, if $w, w' \in \Sigma_\mathtt{p}^\infty$, then $w \sim w'$ iff $w = w'$. Hence, any language over the message alphabet of a single role is (trivially) closed under $\sim$. Further, two runs of a CSM on w and w' with $w \sim w'$ end in the same configuration.

Execution Languages. For a global type $\mathbf{G}$, the above discussion implies that any implementation $\{\!\!\{A_\mathtt{p}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ can at most achieve that $\mathcal{L}(\{\!\!\{A_\mathtt{p}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{C}^{\sim}(\mathcal{L}(\mathbf{G}))$. This is why we call $\mathcal{C}^{\sim}(\mathcal{L}(\mathbf{G}))$ the *execution language* of $\mathbf{G}$. One might also call $\mathcal{C}^{\sim}(\mathcal{L}(L))$ of a local type L an execution language, however, since $\sim$ does not swap any events on the same role, the type language and execution language are equivalent.

▶ **Definition 14** (Implementability)**.** *A global type* $\mathbf{G}$ *is said to be* implementable *if there exists a CSM* $\{\!\!\{A_\mathtt{p}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ *s.t. (i) [protocol fidelity]* $\mathcal{L}(\{\!\!\{A_\mathtt{p}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{C}^{\sim}(\mathcal{L}(\mathbf{G}))$*, and (ii) [deadlock freedom]* $\{\!\!\{A_\mathtt{p}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ *is deadlock free. We say that* $\{\!\!\{A_\mathtt{p}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ *implements* $\mathbf{G}$*.*

3.3 Type Soundness: Projectability implies Implementability

The projection operator preserves the local order of events for every role and does not remove any possible event. Therefore, we can conclude that, for each role, the projected language of the global type is subsumed by the language of the projection.

▶ **Proposition 15.** *For every projectable* $\mathbf{G}$*, role* $\mathtt{r} \in \mathcal{P}$*, run with trace* w *in* $\mathsf{GAut}(\mathbf{G}){\Downarrow_{\Sigma_\mathtt{r}}}$*, there is a run with trace* w *in* $\mathsf{LAut}(\mathbf{G}{\restriction_\mathtt{r}})$*. Therefore, it holds that* $\mathcal{L}(\mathbf{G}){\Downarrow_{\Sigma_\mathtt{r}}} \subseteq \mathcal{L}(\mathbf{G}{\restriction_\mathtt{r}})$*.*

The previous result shows that the projection does not remove behaviours. Now, we also need to show that it does not add unwanted behaviours. The main result is the following.

▶ **Theorem 16.** *If a global type* $\mathbf{G}$ *is projectable, then* $\mathbf{G}$ *is implementable.*

The complete proof can be found in the extended version [46]. Here, we give a brief summary of the main ideas. To show that a projectable global type is implemented by its projections, we need to show that the projection preserves behaviours, does not add behaviours, and is deadlock free. With Proposition 15, showing that the projections combine to admit at least the behaviour specified by the global protocol is straightforward. For the converse direction, we establish a property of the executions of the local types with respect to the global type: all the projections agree on the run taken by the overall system in the global type. We call this property *control flow agreement.* Executions that satisfy control flow agreement also satisfy protocol fidelity and are deadlock free. The formalisation and proof of this property is complicated by the fact that not all roles learn about a choice at the same time. Some roles can perform actions after the choice has been made and before they learn which branch has been taken. In the extreme case, a role may not learn at all that a choice happened. The key to control flow agreement is in the definition of the merge operator. We can simplify the reasoning to the following two points.

Roles learn choices before performing distinguishing actions. When faced with two branches with different actions, a role that is not making the choice needs to learn the branch by receiving a message. This follows from the definition of the merge operator. Let us call this message the choice message. Merging branches is only allowed as long as the actions are similar for this role. When there is a difference between two (or more) branches, an external choice is the only case that allows a role to continue on distinct branches.

Checking available messages ensures no confusion. From the possible receptions ($\mathtt{q}_i?m_i$) in an external choice, any pair of sender and message is unique among this list for the choice. This follows from two facts. First, the projection computes the available messages along the different branches of the choice. Second, merging uses that information to make sure that the choice message of one branch does not occur in another branch as a message independent of that branch's choice messages.

▶ **Example 17.** Let us use an example to illustrate why this is non-trivial. Consider:

$\mathbf{G} := \big(\mathtt{p}\to\mathtt{q}\!:\!l.\, \mu t.\, \mathtt{r}\to\mathtt{p}\!:\!m.\, t\big) + \big(\mathtt{p}\to\mathtt{q}\!:\!r.\, \mu s.\, \mathtt{r}\to\mathtt{p}\!:\!m.\, s\big)$

with its projections:

$\mathbf{G}{\restriction_\mathtt{p}} = \big(\mathtt{q}!l.\, \mu t.\, \mathtt{r}?m.\, t\big) \oplus \big(\mathtt{q}!r.\, \mu s.\, \mathtt{r}?m.\, s\big) \qquad \mathbf{G}{\restriction_\mathtt{q}} = \mathtt{p}?l.\, 0 \;\&\; \mathtt{p}?r.\, 0 \qquad \mathbf{G}{\restriction_\mathtt{r}} = \mu t.\, \mathtt{p}!m.\, t$

and an execution prefix w of $\{\!\{\mathsf{LAut}(\mathbf{G}{\restriction_\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$: $\mathtt{r}\triangleright\mathtt{p}!m.\ \mathtt{r}\triangleright\mathtt{p}!m.\ \mathtt{p}\triangleright\mathtt{q}!l.\ \mathtt{p}\triangleleft\mathtt{r}?m.\ \mathtt{r}\triangleright\mathtt{p}!m.$ For this execution prefix, we check which runs in $\mathsf{GAut}(\mathbf{G})$ each role could have pursued. In this case, $\mathtt{r}$ is not directly affected by the choice so it can proceed before the $\mathtt{p}$ has even made the choice. As the part of the protocol after the choice is a loop, we cannot bound how far some roles can proceed before the choice gets resolved. ⌟

Table 1 Projecting MSTs. For each example, we report the size as the number of nodes in the AST, the number of roles, whether it uses our extension, the time to compute the projections.

Source	Name	Size	$\|\mathcal{P}\|$	Gen. Proj. needed	Time
[52]	Instrument Contr. Prot. A	16	3	✗	0.50 ms
	Instrument Contr. Prot. B	13	3	✗	0.41 ms
	Multi Party Game	16	3	✗	0.48 ms
	OAuth2	7	3	✗	0.29 ms
	Streaming	7	4	✗	0.33 ms
[16]	Non-Compatible Merge	5	3	✓	0.22 ms
[53]	Spring-Hibernate	44	6	✓	1.97 ms
New	Group Present	43	4	✓	1.62 ms
	Late Learning	12	4	✓	0.56 ms
	Load Balancer ($n = 10$)	32	12	✓	8.18 ms
	Logging ($n = 10$)	56	13	✓	20.96 ms

▶ Remark 18. Our projection balances expressiveness with tractability: it does not unfold recursion, i.e., the merge operator never expands a term $\mu t.G$ to obtain the local type (and we only unfold once to obtain the set of available messages). Recursion variables are only handled by equality. While this restriction may seem arbitrary, unfolding can lead to comparing unbounded sequences of messages and, hence, undecidability [3] or non-effective constructions [16]. Our projection guarantees that a role is either agnostic to a choice or receives a choice message in an horizon bounded by the size of the type.

4 Evaluation

We implemented our generalised projection in a prototype tool which is publicly available [1]. The core functionality is implemented in about 800 lines of Python3 code. Our tool takes as input a global type and outputs its projections (if defined). We run our experiments on a machine with an Intel Xeon E5-2667 v2 CPU. Table 1 presents the results of our evaluation.

Our prototype successfully projects global types from the literature [52], in particular Multi-Party Game, OAuth2, Streaming, and two corrected versions of the Instrument Control Protocol. These existing examples can be projected, but do not require generalised projection.

The Need for Generalised Projection. The remaining examples exemplify when our generalised projection is needed. In fact, if some role can receive from different senders along two paths (immediately or after a sequence of same actions), its projection is only defined for the generalised projection operator. To start with, our generalised projection can project a global type presented by Castagna et al. [16, p. 19] which is not projectable with their effective projection operator (see Section 5 for more details). The Spring-Hibernate example was obtained by translating a UML sequence diagram [53] to a global type. There, `Hibernate Session` can receive from two different senders along two paths. The Group Present example is a variation of the traditional book auction example [37] and describes a protocol where friends organise a birthday present for someone; in the course of the protocol, some people can be contacted by different people. The Late Learning example models a protocol where a role submits a request and the server replies either with `reject` or `wait`, however, the last case applies to two branches where the result is served by different roles. The Load Balancer (Example 2) and Logging examples are simple versions of typical communication patterns in distributed computing. The examples are parameterised by the number of workers, respectively, back-ends that call the logging service, to evaluate the efficiency of projection. For both, we present one instance ($n = 10$) in the table. All new examples are rejected by previous approaches but shown projectable by our new projection.

Overhead. The generalised projection does not incur any overhead for global types that do not need it. Our implementation computes the sets of available messages lazily, i.e., it is only computed if our message causality analysis is needed. These sets are only needed when merging receptions from different senders. We modelled four more parameterised protocols: Mem Cache, Map Reduce, Tree Broadcast, and P2P Broadcast. We tested these examples, which do not need the generalised projection, up to size 1000 and found that our generalised projection does not add any overhead. Thus, while the message causality analysis is crucial for our generalised projection operator and hence applicability of MST verification, it does not affect its efficiency.

5 Related Work

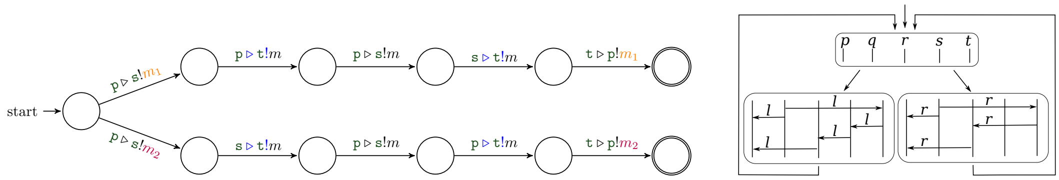
Session Types. MSTs stem from process algebra and they have been proposed for typing communication channels. The seminal work on binary session types by Honda [33] identified channel duality as a condition for safe two party communication. This work was inspired by linear logic [30] and lead to further studies on the connections between session types and linear logic [57, 13]. Moving from binary to multiparty session types, Honda et al. [36] identified consistency as the generalisation of duality for the multiparty setting. The connection between MSTs and linear logic is still ongoing [12, 14, 15]. While we focus solely on communication primitives, the theory is extended with other features such as delegation [35, 36, 18] and dependent types [54, 25, 55]. These extensions have their own intricacies and we leave incorporating such features into our generalised projection for future work.

In this paper, we use local types directly as implementations for roles for simplicity. Subtyping investigates ways to give implementation freedom while preserving the correctness properties. For further details on subtyping, we refer to work by Lange and Yoshida [43], Bravetti et al. [11], and Chen et al. [21, 20].

Generalisations of Choice in MSTs. Castagna et al. [16] consider a generalised choice similar to this work. They present a non-effective general approach for projection, relying on global information, and an algorithmic projection which is limited to local information. Our projection keeps some global information in the form of message availability and, therefore, handles a broader class of protocols. For instance, our generalised projection operator can project the following example [16, p. 19] but their algorithmic version cannot:

$$\big(\mathtt{p}\!\to\!\mathtt{r}\!:\!a.\ \mathtt{r}\!\to\!\mathtt{p}\!:\!a.\ \mathtt{p}\!\to\!\mathtt{q}\!:\!a.\ \mathtt{q}\!\to\!\mathtt{r}\!:\!b.\ 0\big) + \big(\mathtt{p}\!\to\!\mathtt{q}\!:\!a.\ \mathtt{q}\!\to\!\mathtt{r}\!:\!b.\ 0\big)$$

Hu and Yoshida [38] syntactically allow a sender to send to different recipients in global and local types as well as a receiver to receive from different senders in local types. However, their projection is only defined if a receiver receives messages from a single role. From our evaluation, all the examples that needs the generalised projection are rejected by their projection. Recently, Castellani et al. [17] investigated ways to allow local types to specify receptions from multiple senders for reversible computations but only in the synchronous



(a) Conditions for choreography automata are unsound in the asynchronous setting.

(b) Reconstructible HMSC that is not implementable.

Figure 3 These examples show that previous results for the asynchronous setting are flawed.

setting. Similarly, for synchronous communication only, Jongmans and Yoshida [40] discuss generalising choice in MSTs. Because their calculus has an explicit parallel composition, they can emulate some asynchronous communication but their channels have bag semantics instead of FIFO queues. The correctness of the projection also computes causality among messages as in our case and shares the idea of annotating local types.

Choreography Automata. Choreography automata [6] and graphical choreographies [42] model protocols as automata with transitions labelled by message exchanges, e.g., $\mathtt{p} \to \mathtt{q} : m$. Barbanera el al. [6] develop conditions for safely mergeable branches that ensure implementability on synchronous choreography automata. However, when lifting them to the asynchronous setting, they miss the subtle introduction of partial order for messages from different senders. Consider the choreography automaton in Figure 3a. It can also be represented as a global type:

$$+\begin{cases}\mathtt{p}\to\mathtt{s}:m_1.\ \mathtt{p}\to\mathtt{t}:m.\ \mathtt{p}\to\mathtt{s}:m.\ \mathtt{s}\to\mathtt{t}:m.\ \mathtt{t}\to\mathtt{p}:m_1.\ 0\\ \mathtt{p}\to\mathtt{s}:m_2.\ \mathtt{s}\to\mathtt{t}:m.\ \mathtt{p}\to\mathtt{s}:m.\ \mathtt{p}\to\mathtt{t}:m.\ \mathtt{t}\to\mathtt{p}:m_2.\ 0\end{cases}$$

It is *well-formed* according to their conditions. However, $\mathtt{t}$ cannot determine which branch was chosen since the messages m from $\mathtt{p}$ and $\mathtt{s}$ are not ordered when sent asynchronously. As a result, it can send m_2 in the top (resp. left) branch which is not specified as well as m_1 in the bottom (resp. right) branch.

Lange et al. [42] have shown how to obtain graphical choreographies from CSM executions. Unfortunately, they cannot fully handle unbounded FIFO channels as their method internally uses Petri nets. Still, their branching property [42, Def. 3.5] consists of similar – even though more restrictive – conditions as our MST framework: a single role chooses at each branch but roles have to learn with the first received message or do not commit any action until the branches merge back. We allow a role to learn later by recursive application of $\sqcap$.

Implementability in Message Sequence Charts. Projection is studied in hierarchical message sequence charts (HMSCs) as *realisability*. There, variations of the problem like changing the payload of existing messages or even adding new messages in the protocol are also considered [49, 29]. Here, we focus on implementability without altering the protocol. HMSCs are a more general model than MSTs and, unsurprisingly, realisability is undecidable [28, 3]. Thus, restricted models have been studied. Boundedness [3] is one such example: checking safe realisability for bounded HMSCs is EXPSPACE-complete [45]. Boundedness is a very strong restriction. Weaker restrictions, as in MSTs, center on choice. As we explain below, these restrictions are either flawed, overly restrictive, or not effectively checkable.

The first definition of (non-)local choice for HMSCs by Ben-Abdallah and Leue [7] suffers from severely restrictive assumptions and only yields finite-state systems. Given an HMSC specification, research on *implied scenarios*, e.g. by Muccini et al. [50], investigates whether there are behaviours which, due to the asynchronous nature of communication, every implementation must allow in addition to the specified ones. In our setting, an implementable protocol specification must not have any implied scenarios. Mooij et al. [48] point out several contradictions of the observations on implied scenarios and non-local choice. Hence, they propose more variants of non-local choices but allow implied scenarios. In our setting, this corresponds to allowing roles to follow different branches.

Similar to allowing implied scenarios of specifications, Hélouët [31] pointed out that non-local choice has been frequently misunderstood as it actually does not ensure implementability but less ambiguity. Hélouët and Jard proposed the notion of reconstructibility [32] for a quite restrictive setting: first, messages need to be unique in the protocol specification and, second, each node in an HMSC is implicitly terminal. Unfortunately, we show their results are flawed. Consider the HMSC in Figure 3b. (For simplicity, we use the same message identifier in each branch but one can easily index them for uniqueness.) The same protocol can be represented by the following global type:

$$\mu t.\ +\begin{cases} \mathtt{q}\to\mathtt{t}:l.\ \mathtt{q}\to\mathtt{p}:l.\ \mathtt{t}\to\mathtt{s}:l.\ \mathtt{s}\to\mathtt{r}:l.\ \mathtt{r}\to\mathtt{p}:l.\ t \\ \mathtt{q}\to\mathtt{t}:r.\ \mathtt{q}\to\mathtt{p}:r.\ \mathtt{t}\to\mathtt{r}:r.\ \mathtt{r}\to\mathtt{p}:r.\ t \end{cases}$$

Because their notion of reconstructibility [32, Def. 12] only considers loop-free paths, they report that the HMSC is reconstructible. However, the HMSC is not implementable. Suppose that $\mathtt{q}$ first chooses to take the top (resp. left) and then the bottom (resp. right) branch. The message l from $\mathtt{s}$ to $\mathtt{r}$ can be delayed until after $\mathtt{r}$ received r from $\mathtt{t}$. Therefore, $\mathtt{r}$ will first send r to $\mathtt{p}$ and then l which contradicts with the order of branches taken. This counterexample contradicts their result [32, Thm. 15] and shows that reconstructibility is not sufficient for implementability.

Dan et al. [22], improving Baker et al. [5], provide a definition that ensures implementability. They provide a definition which is based on projected words of the HMSC in contrast to the choices. It is unknown how to check their condition for HMSCs.

CSMs and MSTs. The connection between MSTs, CSMs, and automata [16, 24] came shortly after the introduction of MSTs. Denielou and Yoshida [24] use CSMs but they preserve the restrictions on choice from MSTs. It is well-known that CSMs are Turing-powerful [10]. Decidable instances of CSM verification can be obtained by restricting the communication topology [51, 56] or by altering the semantics of communication, e.g. by making channels lossy [2], half-duplex [19], or input-bounded [9]. Lange and Yoshida [44] proposed additional notions that resemble ideas from MSTs.

6 Conclusion

We have presented a generalised projection operator for asynchronous MSTs. The key challenge lies in the generalisation of the external choice to allow roles to receive from more than one sender. We provide a new projectability check and a soundness theorem that shows projectability implies implementability. The key to our results is a message causality analysis and an automata-theoretic soundness proof. With a prototype implementation, we have demonstrated that our MST framework can project examples from the literature as well as new examples, including typical communication patterns in distributed computing, which were not projectable by previous projection operators.

References

1 Prototype Implementation of Generalised Projection for Multiparty Session Types. URL: `https://gitlab.mpi-sws.org/fstutz/async-mpst-gen-choice/`.

2 Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28 – July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 305–318. Springer, 1998. doi:10.1007/BFb0028754.

3 Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of MSC graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005. doi:10.1016/j.tcs.2004.09.034.

4 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.

5 Paul Baker, Paul Bristow, Clive Jervis, David J. King, Robert Thomson, Bill Mitchell, and Simon Burton. Detecting and resolving semantic pathologies in UML sequence diagrams. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 50–59. ACM, 2005. doi:10.1145/1081706.1081716.

6 Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography automata. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages – 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2020. doi:10.1007/978-3-030-50029-0_6.

7 Hanêne Ben-Abdallah and Stefan Leue. Syntactic detection of process divergence and non-local choice inmessage sequence charts. In Ed Brinksma, editor, *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands, April 2-4, 1997, Proceedings*, volume 1217 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 1997. doi:10.1007/BFb0035393.

8 Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Asynchronous timed session types – from duality to time-sensitive processes. In Luís Caires, editor, *Programming Languages and Systems – 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 583–610. Springer, 2019. doi:10.1007/978-3-030-17184-1_21.

9 Benedikt Bollig, Alain Finkel, and Amrita Suresh. Bounded reachability problems are decidable in FIFO machines. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 49:1–49:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CONCUR.2020.49.

10 Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983. doi:10.1145/322374.322380.

11 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018. doi:10.1016/j.tcs.2018.02.010.

12 Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems – 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 74–95. Springer, 2016. `doi:10.1007/978-3-319-39570-8_6`.

13 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Math. Struct. Comput. Sci.*, 26(3):367–423, 2016. `doi:10.1017/S0960129514000218`.

14 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.CONCUR.2016.33`.

15 Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. *Acta Informatica*, 54(3):243–269, 2017. `doi:10.1007/s00236-016-0285-y`.

16 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Log. Methods Comput. Sci.*, 8(1), 2012. `doi:10.2168/LMCS-8(1:24)2012`.

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

18 Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Ross Horne. Global types with internal delegation. *Theor. Comput. Sci.*, 807:128–153, 2020. `doi:10.1016/j.tcs.2019.09.027`.

19 Gérard Cécé and Alain Finkel. Verification of programs with half-duplex communication. *Inf. Comput.*, 202(2):166–190, 2005. `doi:10.1016/j.ic.2005.05.006`.

20 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the preciseness of subtyping in session types. *Log. Methods Comput. Sci.*, 13(2), 2017. `doi:10.23638/LMCS-13(2:12)2017`.

21 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the preciseness of subtyping in session types. In Olaf Chitil, Andy King, and Olivier Danvy, editors, *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 135–146. ACM, 2014. `doi:10.1145/2643135.2643138`.

22 Haitao Dan, Robert M. Hierons, and Steve Counsell. Non-local choice and implied scenarios. In José Luiz Fiadeiro, Stefania Gnesi, and Andrea Maggiolo-Schettini, editors, *8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010, Pisa, Italy, 13-18 September 2010*, pages 53–62. IEEE Computer Society, 2010. `doi:10.1109/SEFM.2010.14`.

23 Ankush Das, Stephanie Balzer, Jan Hoffmann, and Frank Pfenning. Resource-aware session types for digital contracts. *CoRR*, abs/1902.06056, 2019. `arXiv:1902.06056`.

24 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *Programming Languages and Systems – 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 – April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012. `doi:10.1007/978-3-642-28869-2_10`.

25 Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Log. Methods Comput. Sci.*, 8(4), 2012. `doi:10.2168/LMCS-8(4:6)2012`.

26 Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity OS. In Yolande Berbers and Willy Zwaenepoel, editors, *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*, pages 177–190. ACM, 2006. doi:10.1145/1217935.1217953.

27 Paul Gastin. Infinite traces. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings*, volume 469 of *Lecture Notes in Computer Science*, pages 277–308. Springer, 1990. doi:10.1007/3-540-53479-2_12.

28 Blaise Genest, Anca Muscholl, and Doron A. Peled. Message sequence charts. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*, volume 3098 of *Lecture Notes in Computer Science*, pages 537–558. Springer, 2003. doi:10.1007/978-3-540-27755-2_15.

29 Blaise Genest, Anca Muscholl, Helmut Seidl, and Marc Zeitoun. Infinite-state high-level mscs: Model-checking and realizability. *J. Comput. Syst. Sci.*, 72(4):617–647, 2006. doi:10.1016/j.jcss.2005.09.007.

30 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

31 Loïc Hélouët. Some pathological message sequence charts, and how to detect them. In Rick Reed and Jeanne Reed, editors, *SDL 2001: Meeting UML, 10th International SDL Forum Copenhagen, Denmark, June 27-29, 2001, Proceedings*, volume 2078 of *Lecture Notes in Computer Science*, pages 348–364. Springer, 2001. doi:10.1007/3-540-48213-X_22.

32 Loïc Hélouët and Claude Jard. Conditions for synthesis of communicating automata from HMSCs. In *In 5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, 2000.

33 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

34 Kohei Honda, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Verification of MPI programs using session types. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface – 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, volume 7490 of *Lecture Notes in Computer Science*, pages 291–293. Springer, 2012. doi:10.1007/978-3-642-33518-1_37.

35 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems – ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 – April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

36 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

37 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

38 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering – 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10202 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2017. doi:10.1007/978-3-662-54494-5_7.

39 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

40 Sung-Shik Jongmans and Nobuko Yoshida. Exploring type-level bisimilarity towards more expressive multiparty session types. In Peter Müller, editor, *Programming Languages and Systems – 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 251–279. Springer, 2020. doi:10.1007/978-3-030-44914-8_10.

41 Dimitrios Kouzapas, Ramunas Gutkovas, A. Laura Voinea, and Simon J. Gay. A session type system for asynchronous unreliable broadcast communication. *CoRR*, abs/1902.01353, 2019. arXiv:1902.01353.

42 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 221–232. ACM, 2015. doi:10.1145/2676726.2676964.

43 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures – 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017. doi:10.1007/978-3-662-54458-7_26.

44 Julien Lange and Nobuko Yoshida. Verifying asynchronous interactions via communicating session automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification – 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2019. doi:10.1007/978-3-030-25540-4_6.

45 Markus Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theor. Comput. Sci.*, 309(1-3):529–554, 2003. doi:10.1016/j.tcs.2003.08.002.

46 Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. Generalising projection in asynchronous multiparty session types. *CoRR*, abs/2107.03984, 2021. arXiv:2107.03984.

47 Rupak Majumdar, Marcus Pirron, Nobuko Yoshida, and Damien Zufferey. Motion session types for robotic interactions (brave new idea paper). In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPIcs*, pages 28:1–28:27. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ECOOP.2019.28.

48 Arjan J. Mooij, Nicolae Goga, and Judi Romijn. Non-local choice and beyond: Intricacies of MSC choice nodes. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2005. doi:10.1007/978-3-540-31984-9_21.

49 Rémi Morin. Recognizable sets of message sequence charts. In Helmut Alt and Afonso Ferreira, editors, *STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science, Antibes – Juan les Pins, France, March 14-16, 2002, Proceedings*, volume 2285 of *Lecture Notes in Computer Science*, pages 523–534. Springer, 2002. doi:10.1007/3-540-45841-7_43.

50 Henry Muccini. Detecting implied scenarios analyzing non-local branching choices. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering, 6th International Conference, FASE 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2621 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2003. doi:10.1007/3-540-36578-8_26.

51 Wuxu Peng and S. Purushothaman. Analysis of a class of communicating finite state machines. *Acta Informatica*, 29(6/7):499–522, 1992. doi:10.1007/BF01185558.

52 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

53 Spring and Hibernate Transaction in Java. URL: https://www.uml-diagrams.org/examples/spring-hibernate-transaction-sequence-diagram-example.html.

54 Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In Peter Schneider-Kamp and Michael Hanus, editors, *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 161–172. ACM, 2011. doi:10.1145/2003476.2003499.

55 Bernardo Toninho and Nobuko Yoshida. Depending on session-typed processes. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures – 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2018. doi:10.1007/978-3-319-89366-2_7.

56 Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Context-bounded analysis of concurrent queue systems. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2008. doi:10.1007/978-3-540-78800-3_21.

57 Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.

58 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Martín Abadi and Alberto Lluch-Lafuente, editors, *Trustworthy Global Computing – 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi:10.1007/978-3-319-05119-2_3.

A Communicating State Machines

A *communicating state machine* (CSM) $\mathcal{A} = \{\!\{A_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ over $\mathcal{P}$ and $\mathcal{V}$ consists of a state machine A_p over Σ_p for each $\mathsf{p} \in \mathcal{P}$. A state machine for p will be denoted by $(Q_\mathsf{p}, \Sigma_\mathsf{p}, \delta_\mathsf{p}, q_{0,\mathsf{p}}, F_\mathsf{p})$. If a state q has multiple outgoing transitions, all labelled with send actions, then q is called an *internal choice* state. If all the outgoing transitions are labelled with receive actions, q is called an *external choice* state. Otherwise, q is a *mixed choice* state. In this paper, we only consider state machines without mixed choice states.

Intuitively, a CSM represents a set of state machines, one for each role in $\mathcal{P}$, interacting via message sends and receipts. We assume that each pair of roles $\mathsf{p}, \mathsf{q} \in \mathcal{P}$, $\mathsf{p} \neq \mathsf{q}$, is connected by a *message channel*. A transition $q_\mathsf{p} \xrightarrow{\mathsf{p}\triangleright\mathsf{q}!m} q'_\mathsf{p}$ in the state machine of p specifies that, when p is in the state q_p, it sends a message m to q, and updates its local state to q'_p. The message m is appended to the channel $\langle \mathsf{p}, \mathsf{q} \rangle$. Similarly, a transition $q_\mathsf{q} \xrightarrow{\mathsf{q}\triangleleft\mathsf{p}?m} q'_\mathsf{q}$ in the state machine of q specifies that q in state q_q can retrieve the message m from the head of the channel $\langle \mathsf{p}, \mathsf{q} \rangle$ and update its local state to q'_q.

Let $\mathsf{Chan} = \{\langle \mathsf{p}, \mathsf{q} \rangle \mid \mathsf{p}, \mathsf{q} \in \mathcal{P}, \mathsf{p} \neq \mathsf{q}\}$ denote the set of channels. The set of global states of the CSM is given by $\prod_{\mathsf{p}\in\mathcal{P}} Q_\mathsf{p}$. For a global state q, we write q_p for the state of p in q. A *configuration* of $\mathcal{A}$ is a pair (q, ξ), where q is a global state and $\xi : \mathsf{Chan} \to \mathcal{V}^*$ maps each channel to the queue of messages currently in the channel. The initial configuration is (q_0, ξ_ε), where $q_{0,\mathtt{p}}$ is the initial state of $A_\mathtt{p}$ for each $\mathtt{p} \in \mathcal{P}$ and ξ_ε maps each channel to ε. A configuration (q, ξ) is *final* iff $q_\mathtt{p}$ is final for every $\mathtt{p}$ and $\xi = \xi_\varepsilon$.

In a global move of a CSM, a single role executes a local transition to change its state, while all other roles remain stationary. For a send or a receive action, the corresponding channel is updated, while all other channels remain unchanged. Formally, the global transition relation $\rightarrow$ on configurations is defined as follows:

- $(q, \xi) \xrightarrow{\mathtt{p}\triangleright\mathtt{q}!m} (q', \xi')$ if $(q_\mathtt{p}, \mathtt{p}\triangleright\mathtt{q}!m, q'_\mathtt{p}) \in \delta_\mathtt{p}$, $q_\mathtt{r} = q'_\mathtt{r}$ for every $\mathtt{r} \neq \mathtt{p}$, $\xi'(\langle \mathtt{p}, \mathtt{q}\rangle) = \xi(\langle \mathtt{p}, \mathtt{q}\rangle) \cdot m$ and $\xi'(c) = \xi(c)$ for every other channel $c \in \mathsf{Chan}$.
- $(q, \xi) \xrightarrow{\mathtt{q}\triangleleft\mathtt{p}?m} (q', \xi')$ if $(q_\mathtt{q}, \mathtt{q}\triangleleft\mathtt{p}?m, q'_\mathtt{q}) \in \delta_\mathtt{q}$, $q_\mathtt{r} = q'_\mathtt{r}$ for every $\mathtt{r} \neq \mathtt{q}$, $\xi(\langle \mathtt{p}, \mathtt{q}\rangle) = m \cdot \xi'(\langle \mathtt{p}, \mathtt{q}\rangle)$ and $\xi'(c) = \xi(c)$ for every other channel $c \in \mathsf{Chan}$.
- $(q, \xi) \xrightarrow{\tau} (q', \xi)$ if $(q_\mathtt{p}, \varepsilon, q'_\mathtt{p}) \in \delta_\mathtt{p}$ for some role $\mathtt{p}$, and $q_\mathtt{q} = q'_\mathtt{q}$ for every role $\mathtt{q} \neq \mathtt{p}$.

A run of the CSM is a finite or infinite sequence: $(q_0, \xi_0) \xrightarrow{x_0} (q_1, \xi_1) \xrightarrow{x_1} \ldots$, such that (q_0, ξ_0) is the initial configuration and for each $i \geq 0$, we have $(q_i, \xi_i) \xrightarrow{x_i} (q_{i+1}, \xi_{i+1})$. The *trace* of the run is the word $x_0 x_1 \ldots \in \Sigma^\infty$. We also call $x_0 x_1 \ldots$ an *execution prefix*. A run is maximal if it is infinite or if it is finite and ends in a final configuration. A trace is maximal if it is the trace of a maximal run. The language $\mathcal{L}(\mathcal{A})$ of the CSM $\mathcal{A}$ is the set of maximal traces. A CSM is *deadlock free* if every finite run can be extended to a maximal run.

The following lemma summarises some properties of execution prefixes of CSMs. The proofs are by induction on the length of the run.

▶ **Lemma 19.** *Let $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be a CSM. For any run $(q_0, \xi_0) \xrightarrow{x_0} \cdots \xrightarrow{x_n} (q, \xi)$ with trace $w = x_0 \ldots x_n$, it holds that (1) $\xi(\langle \mathtt{p}, \mathtt{q}\rangle) = u$ where $\mathcal{V}(w{\Downarrow}_{\mathtt{p}\triangleright\mathtt{q}!\_}) = \mathcal{V}(w{\Downarrow}_{\mathtt{q}\triangleleft\mathtt{p}?\_}).u$ for every pair of roles $\mathtt{p}, \mathtt{q} \in \mathcal{P}$ and (2) w is channel-compliant. Maximal traces of $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ are channel-compliant and complete.*

Proof. We prove the claims by induction on an execution prefix w. The base case where $w = \varepsilon$ is trivial. For the induction step, we consider wx with the following run in $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$: $(q_0, \xi_0) \xrightarrow{w} (q, \xi) \xrightarrow{x} (q', \xi')$. The induction hypothesis holds for w and (q, ξ) and we prove the claims for (q', ξ') and wx. We do a case analysis on x. If $x = \tau$, the claim trivially follows.

Let $x = \mathtt{q} \triangleleft \mathtt{p}?m$. From the induction hypothesis, we know that $\xi(\langle \mathtt{p}, \mathtt{q}\rangle) = u$ where $\mathcal{V}(w{\Downarrow}_{\mathtt{p}\triangleright\mathtt{q}!\_}) = \mathcal{V}(w{\Downarrow}_{\mathtt{q}\triangleleft\mathtt{p}?\_}).u$. Since x is a possible transition, we know that $u = m.u'$ for some u' and $\xi'(\langle \mathtt{p}, \mathtt{q}\rangle) = u'$. By definition, it holds that $\mathcal{V}(w{\Downarrow}_{\mathtt{q}\triangleleft\mathtt{p}?\_}).m.u' = \mathcal{V}((wx){\Downarrow}_{\mathtt{q}\triangleleft\mathtt{p}?\_}).u'$. For all other pairs of roles, the induction hypothesis applies since the above projections do not change. Hence, wx is channel-compliant.

Let $x = \mathtt{p} \triangleright \mathtt{q}!m$. From the induction hypothesis, we know that $\xi(\langle \mathtt{p}, \mathtt{q}\rangle) = u$ where $\mathcal{V}(w{\Downarrow}_{\mathtt{p}\triangleright\mathtt{q}!\_}) = \mathcal{V}(w{\Downarrow}_{\mathtt{q}\triangleleft\mathtt{p}?\_}).u$. Since x is a possible transition, we know that $\xi'(\langle \mathtt{p}, \mathtt{q}\rangle) = u.m$. By definition and induction hypothesis, we have: $\mathcal{V}((wx){\Downarrow}_{\mathtt{p}\triangleright\mathtt{q}!\_}) = \mathcal{V}(w{\Downarrow}_{\mathtt{p}\triangleright\mathtt{q}!\_}).m = \mathcal{V}(w{\Downarrow}_{\mathtt{q}\triangleleft\mathtt{p}?\_}).u.m$. For all other combinations of roles, the induction hypothesis applies since the above projections do not change. Hence, wx is channel-compliant.

From (1) and (2), it follows directly that maximal traces of $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ are channel-compliant and complete. ◀

B Properties of $\mathcal{C}^\sim$

▶ **Lemma 20.** *Let $L \subseteq \Sigma_\mathtt{p}^\infty$. Then, $L = \mathcal{C}^\sim(L)$.*

Proof. For any $w \in \Sigma_\mathtt{p}^\infty$, none of the rules of $\sim_1$ applies to w, and we have that $w \sim w'$ iff $w = w'$. Thus, $L = \mathcal{C}^\sim(L)$ for any language $L \subseteq \Sigma_\mathtt{p}^\infty$. ◀

▶ **Lemma 21.** *Let $\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ be a CSM. Then, for every finite w with a run in $\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ and every $w' \sim w$, w' has a run that ends with the same configuration. The language $\mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}})$ is closed under $\sim$: $\mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{C}^{\sim}(\mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}))$.*

Proof. Let w be a finite word with a run in $\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ and $w' \sim w$. By definition, $w' \sim_n w$ for some n. We prove that w' has a run that ends in the same configuration by induction on n. The base case for $n = 0$ is trivial. For the induction step, we assume that the claim holds for n and prove it for $n+1$. Suppose that $w \sim_{n+1} w'$. Then, there is w'' such that $w' \sim_1 w''$ and $w'' \sim_n w$. By assumption, we know that $w' = u'u_1u_2u''$ and $w'' = u'u_2u_1u''$ for some $u', u'' \in \Sigma^*$, $u_1, u_2 \in \Sigma$. By induction hypothesis, we know that $w'' \in \mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}})$ so there is run for w'' in $\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$. Let us investigate the run at u_1 and u_2: $\cdots (q_1, \xi_1) \xrightarrow{u_1} (q_2, \xi_2) \xrightarrow{u_2} (q_3, \xi_3) \cdots$. It suffices to show that $\cdots (q_1, \xi_1) \xrightarrow{u_2} (q_2', \xi_2') \xrightarrow{u_1} (q_3, \xi_3) \cdots$ is possible in $\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ for some configuration (q_2', ξ_2'). We do a case analysis on the rule that was applied for $w' \sim_1 w''$.

- $u_1 = \mathtt{p} \triangleright \mathtt{q}!m$, $u_2 = \mathtt{r} \triangleright \mathtt{s}!m'$, and $\mathtt{p} \neq \mathtt{r}$:
  We define q_2' such that $q'_{2,\mathtt{p}} = q_{1,\mathtt{p}}$, $q'_{2,\mathtt{r}} = q_{3,\mathtt{r}}$, and $q'_{2,\mathtt{t}} = q_{3,\mathtt{t}}$ for all $\mathtt{t} \in \mathcal{P}$ with $\mathtt{t} \neq \mathtt{p}$ and $\mathtt{t} \neq \mathtt{r}$. Both transitions are feasible in $\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ because both $\mathtt{p}$ and $\mathtt{r}$ are different and send a message to different channels. They can do this independently from each other.
- $u_1 = \mathtt{q} \triangleleft \mathtt{p}?m$, $u_2 = \mathtt{s} \triangleleft \mathtt{r}?m'$, and $\mathtt{q} \neq \mathtt{s}$:
  We define q_2' such that $q'_{2,\mathtt{q}} = q_{1,\mathtt{q}}$, $q'_{2,\mathtt{s}} = q_{3,\mathtt{s}}$, and $q'_{2,\mathtt{t}} = q_{3,\mathtt{t}}$ for all $\mathtt{t} \in \mathcal{P}$ with $\mathtt{t} \neq \mathtt{p}$ and $\mathtt{t} \neq \mathtt{r}$. Both transitions are feasible in $\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ because both $\mathtt{q}$ and $\mathtt{s}$ are different and receive a message from a different channel. They can do this independently from each other.
- $u_1 = \mathtt{p} \triangleright \mathtt{q}!m$, $u_2 = \mathtt{s} \triangleleft \mathtt{r}?m'$, and and $\mathtt{p} \neq \mathtt{s} \wedge (\mathtt{p} \neq \mathtt{r} \vee \mathtt{q} \neq \mathtt{s})$.
  We define q_2' such that $q'_{2,\mathtt{p}} = q_{1,\mathtt{p}}$, $q'_{2,\mathtt{s}} = q_{3,\mathtt{s}}$, and $q'_{2,\mathtt{t}} = q_{3,\mathtt{t}}$ for all $\mathtt{t} \in \mathcal{P}$ with $\mathtt{t} \neq \mathtt{p}$ and $\mathtt{t} \neq \mathtt{r}$. Let us do a case split according to the side conditions. First, let $\mathtt{p} \neq \mathtt{s}$ and $\mathtt{p} \neq \mathtt{r}$. The channels of u_1 and u_2 are different and $\mathtt{p}$ and $\mathtt{s}$ are different, so both transitions are feasible in $\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$.
  Second, let $\mathtt{p} \neq \mathtt{s}$ and $\mathtt{q} \neq \mathtt{s}$. The channels of u_1 and u_2 are different and $\mathtt{q}$ and $\mathtt{s}$ are different, so both transitions are feasible in $\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$.
- $u_1 = \mathtt{p} \triangleright \mathtt{q}!m$, $u_2 = \mathtt{q} \triangleleft \mathtt{p}?m'$, and $|u' \Downarrow_{\mathtt{p}\triangleright\mathtt{q}!\_}| > |u' \Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?\_}|$:
  We define q_2' such that $q'_{2,\mathtt{p}} = q_{1,\mathtt{p}}$, $q'_{2,\mathtt{q}} = q_{3,\mathtt{q}}$, and $q'_{2,\mathtt{t}} = q_{3,\mathtt{t}}$ for all $\mathtt{t} \neq \mathtt{p}$ and $\mathtt{t} \neq \mathtt{q}$. In this case, the channel of u_1 and u_2 is the same but the side condition ensures that u_2 actually has a different message read since the channel $\xi_1(\langle \mathtt{p}, \mathtt{q} \rangle)$ is not empty by Lemma 19 and, hence, both transitions can act independently and lead to the same configuration.

This proves that w' has a run in $\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ that ends in the same configuration which concludes the proof of the first claim.

For the second claim, we know that $\mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}) \subseteq \mathcal{C}^{\sim}(\mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}))$ by definition. Hence, it suffices to show that $\mathcal{C}^{\sim}(\mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}})) \subseteq \mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}})$.

We show the claim for finite traces first:

$$\mathcal{C}^{\sim}(\mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}})) \cap \Sigma^* \subseteq \mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}) \cap \Sigma^*.$$

Let $w' \in \mathcal{C}^{\sim}(\mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}})) \cap \Sigma^*$. There is $w \in \mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}) \cap \Sigma^*$ such that $w \sim w'$. By definition, w has a run in $\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ which ends in a final configuration. From the first claim, we know that w' also has a run that ends in the same configuration which is final. Therefore, $w \in \mathcal{L}(\{\!\!\{A_{\mathtt{p}}\}\!\!\}_{\mathtt{p}\in\mathcal{P}}) \cap \Sigma^*$. Hence, the claim holds for finite traces.

It remains to show the claim for infinite traces. To this end, we show that for every execution prefix w of $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ such that $w \sim u$ for $u \in \mathrm{pref}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$ and any continuation x of w, i.e., wx is an execution prefix of $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$, it holds that $wx \sim ux$ and $ux \in \mathrm{pref}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$ ($\Box$). We know that $w \sim_n u$ for some n by definition, so $wx \sim_n ux$ since we can mimic the same n swaps when extending both w and u by x. From the first claim, we know that $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is in the same configuration (q, ξ) after processing w and u. Therefore, ux is an execution prefix of $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ because wx is which yields ($\Box$).

We show that

$$\mathcal{C}^{\sim}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) \cap \Sigma^{\omega} \subseteq \mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) \cap \Sigma^{\omega}.$$

Let $w \in \mathcal{C}^{\sim}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) \cap \Sigma^{\omega}$. We show that w has an infinite run in $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$.

Consider a tree $\mathcal{T}$ where each node corresponds to a run ρ on some finite prefix $w' \leq w$ in $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$. The root is labelled by the empty run. The children of a node ρ are runs that extend ρ by a single transition – these exist by ($\Box$). Since our CSM, derived from a global type, is built from a finite number of finite state machines, $\mathcal{T}$ is finitely branching. By König's Lemma, there is an infinite path in $\mathcal{T}$ that corresponds to an infinite run for w in $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$, so $w \in \mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) \cap \Sigma^{\omega}$. ◀

▶ **Lemma 22.** *Let $w \in \Sigma^{\infty}$ be channel-compliant. Then, $w \sim w'$ iff w' is channel-compliant and $w{\Downarrow}_{\Sigma_{\mathtt{p}}} = w'{\Downarrow}_{\Sigma_{\mathtt{p}}}$ for all roles $\mathtt{p} \in \mathcal{P}$.*

Proof. We use the characterisation of $\sim$ using dependence graphs [27]. For a word w and a letter $a \in \Sigma$ that appears in w, let (a, i) denote the ith occurrence of a in w. Define the dependence graph (V, E, λ), where $V = \{(a, i) \mid a \in \Sigma, i \geq 1\}$, $E = \{((a, i), (b, j)) \mid a$ and b cannot be swapped and (a, i) occurs before (b, j) in $w\}$, and $\lambda(a, i) = a$ for all $a \in \Sigma$, $i \geq 1$. A fundamental result of trace theory states that $w \sim w'$ iff they have isomorphic dependence graphs. We observe that for two channel compliant words, the ordering of the letters on each $\Sigma_{\mathtt{p}}$ for $\mathtt{p} \in \mathcal{P}$ ensures isomorphic dependence graphs, since the ordering of receives is thus fixed. ◀